

Programming with *Grace*



Draft of February 7, 2018

Programming with *Grace*

Adapted by Kim B. Bruce, *Pomona College*
from

Java: An Eventful Approach by Kim B. Bruce, Andrea Pohoreckyj Danyluk,
and Thomas P. Murtagh

DRAFT!

DO NOT DISTRIBUTE WITHOUT PRIOR PERMISSION

Website: <http://eventfuljava.cs.williams.edu>

Printed on February 7, 2018

©2018 Kim B. Bruce, Andrea Pohoreckyj Danyluk, and Thomas P. Murtagh ¹

Comments, corrections, and other feedback appreciated

kim@cs.pomona.edu

¹This book is a revision by Kim Bruce of the text **Java:An eventful approach** by Kim B. Bruce, Andrea Pohoreckyj Danyluk, and Thomas P. Murtagh, ©2006 by Pearson Education

Contents

1	What is Programming Anyway?	1
1.1	Without Understanding	2
1.2	The Grace Programming Language	6
1.3	Your First Encounter with Grace	8
1.3.1	Programming Tools	8
1.3.2	Definitions & Methods	10
1.3.3	Objects	13
1.3.4	Classes	14
1.3.5	Program Layout	15
1.4	Generating Graphics with Objectdraw	15
1.4.1	The Graphics Coordinate System	16
1.4.2	A simple example	18
1.4.3	Constructing Graphic Objects	19
1.4.4	Methods	20
1.4.5	startGraphics	20
1.5	Creating other graphics items	21
1.6	Additional Event Handling Methods	26
1.6.1	Mouse Event Handling Methods	26
1.6.2	The Initialization code	27
1.7	To Err is Human	28
1.8	Summary	30
1.9	Chapter Review Problems	30
1.10	Programming Problems	31
2	What's in a name?	33
2.1	Naming and Modifying Objects	33
2.1.1	Mutator Methods	33
2.1.2	Definitions	36
2.1.3	Variable Declarations	38
2.1.4	Comments	39
2.1.5	Labeling identifiers with types	40
2.1.6	Additional Mutator Methods	43
2.1.7	Exercises	45
2.2	Non-graphical Classes of Objects	47

2.2.1	Colors	47
2.2.2	Creating Points	51
2.3	Layering on the Canvas	55
2.4	Accessing the Mouse Position	57
2.5	Sharing Parameter Information between Methods	58
2.6	Summary	65
2.7	Chapter Review Problems	66
2.8	Programming Problems	67
3	Working with Numbers	69
3.1	Writing numbers	69
3.2	Introduction to Accessor Methods	69
3.3	Accessing the Size of the Canvas	71
3.4	Expressions and Statements	74
3.5	Arithmetic Expressions	76
3.5.1	Ordering of Arithmetic Operations	79
3.6	Numeric Instance Variables	81
3.7	Displaying Numeric Information	84
3.7.1	Displaying Numbers as text	84
3.7.2	Using <code>print</code>	87
3.7.3	Mixing Text and Numbers	88
3.8	Random numbers	90
3.9	Summary	93
3.10	Chapter Review Problems	93
3.11	Programming Problems	94
4	Making Choices	99
4.1	A Brief Example: Using the <code>if-then</code> Statement to Count Votes	99
4.2	The <code>if-then</code> Statement	102
4.2.1	Example: Using the <code>if-then</code> Statement with 2-D Objects	104
4.3	Understanding Conditions	105
4.3.1	Using Boolean Values	106
4.4	Selecting Among Many Alternatives	109
4.5	More on Boolean Expressions	112
4.6	Nested Conditionals	116
4.7	Short-circuit boolean operators	120
4.8	<code>if-then-else</code> expressions	121
4.9	Summary	122
4.10	Chapter Review Problems	123
4.11	Programming Problems	125
5	Types, Numbers, and Strings	129
5.1	What is a type?	129
5.2	The type <code>Point</code>	130
5.3	Types from <code>objectdraw</code>	131

5.4	Specializing types	131
5.5	Type compatibility	134
5.6	Numbers	135
5.6.1	More numeric operations	136
5.6.2	Problems using equality between numbers	136
5.6.3	Time flies	140
5.7	Strings	142
5.8	Booleans	145
5.9	Chapter Review Problems	146
5.10	Programming Problems	146
6	Objects & Classes	149
6.1	An Example Without objects	149
6.2	Designing Objects and Classes: A funny face	153
6.2.1	Defining objects	153
6.2.2	Programming with class	156
6.2.3	Programmers create abstractions	157
6.2.4	Putting It All Together	160
6.3	Adding Methods to our face	161
6.3.1	Some Methods Are Like Those Already Written	161
6.3.2	Some Methods Are Trickier	162
6.3.3	Using <code>self</code> as a Parameter	164
6.4	Another Example: Implementing a <code>Timer</code> Class	165
6.5	Local Variables	167
6.6	What about <code>graphicApplication</code>	169
6.7	Summary	172
6.8	Chapter Review Problems	173
6.9	Programming Problems	173
7	Control Structures	177
7.1	Repetition and <code>while</code> Loops	177
7.2	More Examples Using <code>while</code> Loops	184
7.3	Loops that Count	187
7.4	Nested Loops	190
7.5	Style Guidelines for Control Structures	194
7.6	DeMorgan's Laws and Complex Boolean Expressions*	197
7.7	The <code>match</code> Statement	200
7.8	Summary	203
7.9	Chapter Review Problems	203
7.10	Programming Problems	204
8	Declarations and Scope	209
8.1	Access Control: <code>public</code> versus <code>confidential</code>	209
8.2	Using Instance Variables, Parameters, and Local Variables	215
8.3	Scope of Identifiers	217

8.4	Summary	220
8.5	Chapter Review Problems	220
9	Animating Objects	223
9.1	Animation	223
9.2	Animator	223
9.3	Using images in programs	227
9.4	Including audio in programs	228
9.5	Interacting with Animated Objects	230
9.6	Making Animated Objects Affect Other Objects	233
	9.6.1 Interacting with a Non-animated Object	233
	9.6.2 Animated Objects that Construct Other Animated Objects	237
9.7	Animated Objects without Loops	239
9.8	Summary	242
9.9	Chapter Review Problems	242
9.10	Programming Problems	243
10	Graphical User Interfaces in Grace	245
10.1	Text fields	245
	10.1.1 Constructing a text field	246
	10.1.2 Adding a <code>TextField</code> to a Window	246
	10.1.3 Getting Information from a Text Field	247
	10.1.4 Text Boxes	248
	10.1.5 Number Fields	248
10.2	Buttons and Events in Grace	248
	10.2.1 Creating and Adding Buttons	248
	10.2.2 Handling Events	249
10.3	Checklist for Using GUI Components in a Program	251
10.4	Choice menus	252
10.5	Using Containers	254
10.6	Handling Keystroke Events	256
10.7	Summary	257
10.8	Chapter Review Problems	257
10.9	Programming Problems	258
11	Recursion	269
11.1	Recursive Structures	270
	11.1.1 Nested Rectangles	270
	11.1.2 Building and Searching Data Collections with Recursive Structures	279
	11.1.3 Designing Recursive Structures	281
	11.1.4 Why Does this Work?	284
	11.1.5 Broccoli	286
11.2	Recursive Methods	290
	11.2.1 Fast Exponentiation	292
	11.2.2 Towers of Hanoi	295

11.3	Summary	298
11.4	Review Problems	299
11.5	Programming Problems	300
12	General Loops in Grace	303
12.1	Recognizing Patterns with Loops	303
12.2	Counting and for Loops	304
12.2.1	Examples of Using for Loops	306
12.3	More loop constructs	308
12.4	Avoiding Loop Errors	309
12.5	Summary of Loops	312
12.6	Chapter Review Problems	313
12.7	Programming Problems	314
13	Lists	317
13.1	Declaring Lists	318
13.2	Creating a List	319
13.3	Using Lists: A Triangle Class	321
13.4	Gathering Information from a List	325
13.4.1	Counting Speeders	328
13.4.2	Drawing a Histogram	328
13.5	Collections with Variable Sizes	335
13.5.1	Parallel lists vs. lists of Objects	337
13.5.2	Displaying the Results	338
13.6	Adding and Removing Elements	344
13.6.1	Adding an Element to an Ordered list	345
13.6.2	Removing an Element from a list	352
13.7	Summary	355
13.8	Chapter Review Problems	355
13.9	Programming Problems	360
14	Multidimensional lists	365
14.1	General Two-Dimensional Lists	365
14.1.1	Declaring a list of lists	366
14.1.2	Creating a list of lists	366
14.1.3	Indexing a list of lists	367
14.1.4	Traversing a Two-Dimensional list	368
14.1.5	Beyond Two Dimensions: Extending the Calendar Program	371
14.2	Matrices	374
14.2.1	Designing the Matrix class	376
14.2.2	Magic Squares	376
14.2.3	Traversing a Matrix	376
14.2.4	Filling a Magic Square	380
14.3	Summary	382
14.4	Chapter Review Problems	384

14.5	Programming Problems	386
15	Strings	389
15.1	Little Strings and Big Strings	389
15.1.1	The Empty String	389
15.1.2	Long Strings	390
15.1.3	Interpolation in Strings	392
15.2	A Collection of Useful String Methods	393
15.2.1	Building a String of URLs	393
15.2.2	Adding a <code>contains</code> method to <code>URLHistory</code>	394
15.2.3	Finding the Position of a Substring with <code>indexOf</code>	396
15.2.4	Dealing with Lower and Upper Case	396
15.2.5	Cutting and Pasting Strings	399
15.2.6	Trimming Strings	400
15.2.7	Comparing Strings	400
15.3	Characters	402
15.3.1	Extracting Characters from Strings	402
15.3.2	Performing Calculations based on Characters	404
15.4	Summary	409
15.5	Chapter Review Problems	409
15.6	Programming Problems	411
16	Inheritance	415
16.1	Inheritance as Specialization	415
16.2	Using <code>inherit</code>	416
16.3	Confidential features are accessible in subclasses	422
16.4	Inheritance Hierarchies	424
16.5	Overriding Method Definitions	426
16.5.1	Subtyping and Inheritance	427
16.5.2	Overriding the default <code>==</code> , <code>!=</code> , and <code>asString</code> methods	431
16.5.3	Using <code>aliases</code> to access overridden methods	435
16.5.4	Dynamic method invocation	436
16.5.5	Planned obsolescence	438
16.6	Summary	442
16.7	Chapter Review Problems	444
16.8	Programming Problems	450
17	Exceptions	453
17.1	Exception Handlers	454
17.2	Exceptions are Objects	456
17.3	Exception Propagation	457
17.4	Raising Exceptions	459
17.5	Summary	460
17.6	Chapter Review Problems	461
17.7	Programming Problems	462

18 Streams	465
18.1 Bookmark Files: An Example	466
18.2 Opening a file for writing	467
18.3 Writing web pages	469
18.4 Readers	473
18.5 Summary	476
18.6 Chapter Review Problems	476
18.7 Programming Problems	480
19 Searching and Sorting	483
19.1 Searching	483
19.1.1 Linear Searches	484
19.1.2 Recursive Processing of Lists	484
19.1.3 Complexity of Linear Search	486
19.1.4 Binary Searches	487
19.2 Using Preconditions and Postconditions	491
19.3 Sorting	492
19.3.1 Selection Sort	493
19.3.2 Complexity of Selection Sort	496
19.3.3 Insertion Sort	497
19.3.4 Complexity of Insertion Sort	500
19.3.5 Merge Sort	500
19.3.6 Complexity of Merge Sort	504
19.4 Impact of the Complexity of Searching and Sorting	506
19.5 Summary	507
19.6 Chapter Review Problems	507
20 Introduction to Object-Oriented Design	509
20.1 What is Object-Oriented Design?	509
20.2 Properties and Behaviors	510
20.2.1 Modeling Properties of Objects	510
20.2.2 Modeling Behavior	510
20.3 Design Fundamentals	511
20.3.1 Design Step 1: Identify the objects to be modeled	511
20.3.2 Design Step 2: List the properties and behaviors of each type of object	512
20.3.3 Design Step 3: Model properties with instance variables	514
20.3.4 Design Step 4: Model behaviors with methods	516
20.3.5 Design: the Process of Refinement	520
20.3.6 Filling in the Details	520
20.3.7 Summary of the Design Process	528
20.4 Incremental Testing and Debugging	529
20.4.1 Developing and Testing Individual Classes	529
20.4.2 Putting the Pieces Together: Testing the Controller	531
20.5 Classes That Don't Model the Real World	533
20.5.1 An Animated Shell Game	533

20.5.2	Identifying Classes that Don't Represent Entities in the Real World	533
20.5.3	Completing the Design for the Animated Shell Game	534
20.5.4	Filling in the Details for the Animated Shell Game	538
20.6	Guidelines for Writing Comments	540
20.7	Encapsulation and Information Hiding	542
20.7.1	Nested Rectangles: A Graphical Object Class: Nested Rectangles	543
20.7.2	The Important Relationship between Encapsulation and Abstraction	544
20.8	Summary	546
20.9	Chapter Review Problems	548
20.10	Programming Problems	548
A	Style Guidelines	551
A.1	Commenting	551
A.2	Blank Lines	552
A.3	Names	552
A.4	Format	553
B	Built-In Grace Types, Classes, and Libraries	555
B.1	Introduction	555
B.2	Built-in Types	555
B.2.1	Object	555
B.2.2	Number	556
B.2.3	String	558
B.2.4	Boolean	562
B.2.5	Blocks	562
B.2.6	Point	562
B.2.7	Binding	563
B.3	Collection objects	563
B.3.1	Common Abstractions	564
B.3.2	Lineups	565
B.3.3	Sequence	566
B.3.4	Ranges	567
B.3.5	List	567
B.3.6	Sets	568
B.3.7	Dictionary	570
B.3.8	Iterators and for loops	571
B.3.9	Primitive Array	573
B.4	Built-In Libraries	574
B.4.1	Random	574
B.4.2	Option	574
B.4.3	Sys	574

C	Objectdraw API Summary	577
C.1	Application	577
C.2	GraphicApplication	578
C.2.1	Creating a GraphicApplication	578
C.3	Animator	579
C.3.1	Creating an Animator object	579
C.3.2	Types used in methods of type Animator	579
C.3.3	Methods available in objects of type Animator	579
C.4	DrawingCanvas objects	580
C.4.1	Constructing a DrawingCanvas object	580
C.4.2	Methods available in objects of type DrawingCanvas	580
C.5	Graphic objects	580
C.5.1	Methods available for all Graphic objects	580
C.6	Graphic2D Objects	581
C.6.1	Additional methods available for Graphic2D objects	581
C.6.2	Classes generating Graphic2D objects	581
C.7	Line Objects	582
C.7.1	Additional methods available for Line objects	582
C.7.2	Class generating a Line object	582
C.8	Text Objects	582
C.8.1	Additional methods available for Text items	582
C.8.2	Class generating a Text object	583
C.9	Colors	583
C.9.1	Type Color	583
C.10	Sounds	584
C.10.1	Type Audio	584
D	Using GUI components in Grace	585
D.1	Introduction	585
D.2	GUI components	585
D.2.1	Events generated by GUI components	585
D.2.2	Component	586
D.2.3	TextBox	587
D.2.4	Button	587
D.2.5	TextField and NumberField	588
D.2.6	DrawingCanvas	589
D.3	Adding components to a window	590
D.3.1	Application and GraphicApplication	590
D.3.2	Adding Components	591
D.4	Responding to events involving GUI components	592
D.4.1	Button events	593
D.4.2	Input events	593
	Index	594

List of Figures

1.1	Connect dots 1 through 82 (©2000 Monkeying Around)	5
1.2	A Web Page to Run Grace Programs	9
1.3	Program to calculate the square of 7	12
1.4	Defining a second person and a dog	14
1.5	Comparison of computer and Cartesian coordinate systems	17
1.6	Text enlarged to make pixels visible	17
1.7	TouchyWindow program in Grace	19
1.8	Drawing of a single line	21
1.9	A line from (100, 150) to (200, 0)	22
1.10	A program that draws two crossed lines.	24
1.11	A FilledOval nested within a FramedRect	25
2.1	An oval rises over the horizon	35
2.2	Declaring sun in the RisingSun program	37
2.3	Commented code for rising sun example	39
2.4	Adding type annotations to Grace code	40
2.5	Commented code for rising sun example	42
2.6	Rising sun program with reset feature	44
2.7	The types Color and ColorFactory	50
2.8	An application of the + operation on points	51
2.9	Display after one click	52
2.10	Display after second click	53
2.11	Display after many clicks	54
2.12	Today's forecast: Partly cloudy	55
2.13	Tonight's forecast: Partly sunny?	56
2.14	A Program that Uses the Mouse Location	57
2.15	Using a different parameter name	58
2.16	A program to record mouse button changes	59
2.17	Connecting the ends of a mouse motion	60
2.18	A Program to track mouse actions	61
2.19	Scribbling with a computer's mouse	63
2.20	A simple sketching program	65
3.1	Program to make the sun scroll with the mouse	71
3.2	Drawing produced by the DrawGrid program	72

3.3	The sun rises over the horizon	76
3.4	Program to make the sun scroll with the mouse	78
3.5	Drawing desired for Exercise 3.5.1	80
3.6	Using a numeric instance variable	83
3.7	A computer counting program before the first click	85
3.8	A simple counting program.	86
3.9	Counting in the Grace console	88
3.10	Counting using the Grace console	89
3.11	Sample message drawn by dice simulation program	91
3.12	Simulating the rolling of a pair of dice	92
3.13	Display for <code>ICanCountALot</code>	95
3.14	Initial display for <code>GrowMan</code>	96
3.15	Display for <code>GrowMan</code> after ten clicks	97
4.1	Screen shot of <code>Voter</code> program.	100
4.2	Code for <code>Voter</code> program.	101
4.3	Semantics of the if–then–else statement.	103
4.4	Code to display individual and total vote counts	103
4.5	Semantics of the if–then with no else.	104
4.6	Code for dragging a box.	107
4.7	Three stages of dragging a rectangle.	108
4.8	Voting for four candidates.	113
4.9	A summary of the boolean and comparison operators in Grace	115
4.10	<code>Craps</code> program illustrating nested conditionals.	119
4.11	Display for <code>InvisibleBox</code>	125
4.12	Display for <code>Dicey</code>	127
5.1	Point type from <code>objectdraw</code>	130
5.2	The type <code>Graphic</code> from <code>objectdraw</code>	132
5.3	The type <code>Text</code> from the <code>objectdraw</code> library	133
5.4	The type <code>Line</code> from the <code>objectdraw</code> library	133
5.5	The type <code>Graphic2D</code> from the <code>objectdraw</code> library	134
5.6	The type <code>DrawingCanvas</code>	134
5.7	Specification of <code>Number</code> type	137
5.8	Further specification of <code>Number</code> type	138
5.9	Grace program to measure mouse click duration.	141
5.10	Sample output of the <code>ClickTimer</code> program	141
5.11	A program to display Morse code as dots and dashes	143
5.12	Sample of Morse code program display	144
5.13	Methods of type <code>Boolean</code>	146
5.14	Display for <code>DNAGenerator</code>	147
6.1	Funny face generated by <code>FaceDrag</code>	150
6.2	Code for dragging a funny face.	151
6.3	Revised code for dragging a <code>FunnyFace</code>	152

6.4	Definition of the face object	155
6.5	Class for generating funny faces	158
6.6	Timer class	166
6.7	Chase program	168
6.8	Chase game with a local definition <code>newPoint</code>	170
6.9	The type <code>GraphicApplication</code> of graphic programs in <code>objectdraw</code>	171
6.10	A small stick-man	174
6.11	FunnyFace with a TopHat	175
7.1	Grass	178
7.2	Code for drawing the grass scene.	179
7.3	Alternate code for drawing grass. Note the pattern that has emerged in the code.	180
7.4	Code that allows a user to draw the grass in the grass scene by clicking.	181
7.5	Using a while loop to draw blades of grass.	182
7.6	Bricks	184
7.7	A while loop to draw a row of bricks.	185
7.8	Grid	186
7.9	A while loop to draw the grid.	187
7.10	The picture for the AlmostBullseye exercise	188
7.11	A while loop to draw a row of exactly ten bricks.	189
7.12	Pictures of rulers for exercises 7.3.1 and 7.3.2	190
7.13	A simple brick wall	191
7.14	Nested loops to draw a brick wall	192
7.15	Nested loops to draw an alternating brick wall	193
7.16	A better brick wall	195
7.17	Using a match statement to randomly select a color.	201
7.18	Generating an error message for unexpected cases.	202
7.19	Example of scarf for Exercise 7.10.1	205
7.20	Bullseye for Exercise 7.10.3	206
7.21	Fence for Exercise 7.10.4	207
7.22	Railroad tracks for Exercise 7.10.5	208
8.1	Chase class: <code>onMouseClicked</code> method.	210
8.2	Chase class: revised <code>onMouseClicked</code> method using confidential method <code>resetGame</code>	211
8.3	The class <code>color</code> from the <code>objectdraw</code> library	212
8.4	The class in <code>object color</code> as it might have been written in the <code>objectdraw</code> library	213
8.5	Scope example.	218
9.1	A falling ball in motion.	224
9.2	First example of using an animator with a while loop	225
9.3	Program that creates a falling ball where mouse is pressed.	226
9.4	Code for a <code>FallingBall</code> class.	227
9.5	A drawing of a raindrop	228
9.6	Class that creates a falling raindrop picture with each mouse click.	229
9.7	Code for a <code>FallingRainDrop</code> class.	229

9.8	Defining a class that allows a user to drop a ball that changes color when sliced by the mouse.	231
9.9	A falling ball class that responds to slices	232
9.10	A pool filling with rain.	234
9.11	Adding a collector to collect falling raindrops.	235
9.12	Making a droplet fill a collector.	236
9.13	A rain cloud class.	238
9.14	A class of individual movie credit lines.	240
9.15	A class to generate movie credits.	241
9.16	Run of <code>HitTheTarget</code>	244
10.1	<code>textController</code> using a text field	246
10.2	<code>TextField</code> in window.	247
10.3	<code>TextButtonController</code> program using <code>TextField</code>	249
10.4	Drawing program, part 1	259
10.5	Program for drawing, part 2	260
10.6	<code>FallingBall</code> class	261
10.7	Menu constructed by <code>MenuBallController</code>	262
10.8	<code>MenuBallController</code> class	263
10.9	Windows before and after use of containers.	264
10.10	<code>adjustableBallController</code> with multiple buttons and a menu. Part 1.	265
10.11	<code>adjustableBallController</code> with multiple buttons in a panel. Part 2.	266
10.12	Program to print key codes	266
10.13	Controlling ball speed with the keyboard	267
11.1	Nested rectangles	271
11.2	A view of nested rectangles as a recursive structure.	272
11.3	Recursive nested rectangles	273
11.4	Nested rectangles drawn by evaluating <code>nestedRectsAt (50 @ 50) size (19 @ 21) on (canvas)</code>	276
11.5	Objects constructed by evaluating <code>nestedRectsAt (54 @ 54) size (11 @ 13) on (canvas)</code>	277
11.6	Recursive URL lists	280
11.7	A broccoli plant	286
11.8	Movable type	287
11.9	<code>flower</code> class implementing <code>Movable</code>	288
11.10	<code>broccoli</code> class	289
11.11	Program to drag broccoli	291
11.12	Towers of Hanoi puzzle with 8 disks.	296
11.13	Solving tower of Hanoi puzzle	297
11.14	Parsley	301
12.1	Nested for loops for drawing a checkerboard	308
12.2	Craps game using <code>do()</code> / <code>while()</code>	310
13.1	Constructor for the <code>Triangle</code> class	322

13.2	Elements of the edge list refer to components of a Triangle	322
13.3	Radar Trailer	326
13.4	The vehicleReport Method	328
13.5	A graph displaying numbers of speeders detected at different times of the day	329
13.6	The speedersSeen method	330
13.7	A drawing of the essence of a histogram	331
13.8	Method that draws histograms	332
13.9	A bar graph with rather short bars	332
13.10	Method to find largest value in speedersAt list	333
13.11	Method that draws well-scaled histograms	334
13.12	Sample of output expected from the drawLineGraph method	335
13.13	List of finishing times for racers	336
13.14	Definition of racerInfo class	338
13.15	The individualResults method	339
13.16	The placement method	341
13.17	The teamScore method computes the score for one team	342
13.18	The teamStandings method	343
13.19	racer list with a missing element.	345
13.20	State of racer list after adding all 11 racers	346
13.21	racer list with entries after omission shifted right.	346
13.22	racer list after insertion is complete.	347
13.23	racer list after moving element 10.	347
13.24	racer list after moving elements 9 and 10.	348
13.25	racer list after moving 6 elements over by one.	349
13.26	racer list after executing racer.at (6) put (racer.at (5))	350
13.27	list after executing racer.at (7) put (racer.at (6))	351
13.28	Where did all the racers go?	351
13.29	The addRacerAtPosition method	352
13.30	list after putting dummy value in slot 7	353
13.31	racer list with last element duplicated	353
13.32	The removeRacerAtPosition method	354
13.33	The removeRacerAtPosition method with the loop (incorrectly) reversed	354
13.34	A class to represent continents	356
13.35	Method mystery for exercise 13.8.2	357
13.36	Skeleton of a Polynomial class for Exercise 13.8.3	358
13.37	Matching patches program after one patch is selected	362
13.38	Matching patches program after one pair has been identified	362
13.39	Matching patches program after many pairs have been identified	363
14.1	Interface used to enter new calendar events	366
14.2	A 12-element list. Each element in the list has the potential to refer to a list of strings.	367
14.3	Creating a calendar as a list of lists	368

14.4	Daily events organized as a 12-element list (months) of lists (days of each month).	369
14.5	Yearly calendar after two events have been entered.	370
14.6	Another view of daily events as a jagged table	371
14.7	General structure of nested for loops to traverse a two-dimensional list	372
14.8	The YearlyCalendar class	373
14.9	Calendar entries for one day	374
14.10	Sketch of calendar-related class to hold daily events.	375
14.11	Magnified pixels from an image	376
14.12A	chessboard and sliding block puzzle: examples of objects that can be represented by two-dimensional matrices	377
14.13A	class for representing matrices	378
14.14	Magic squares	379
14.15	Wrapping around after falling off the bottom of the square	382
14.16	Moving up if the next cell to be filled is already full	382
14.17A	a method to fill a magic square	383
14.18	Player moving off the right of the game grid re-enters from the left	386
15.1	A program to display Morse code as dots and dashes	391
15.2	Instructions for the Morse code program	392
15.3	Using <code>\n</code> to force line breaks	392
15.4	URLHistory type	394
15.5	uRLHistory class	395
15.6	Case-insensitive uRLHistory class	398
15.7	Adding a URL to a history of URLs maintained in alphabetical order	401
15.8	Inserting a string into a list in alphabetical order	401
15.9	User interface for a medical record system	403
15.10	Method to check whether a String looks like a valid integer	404
15.11	Alternative method to check whether a String looks like a valid integer	404
15.12A	partial Morse code alphabet	405
15.13A	a method to translate a message into Morse code	406
15.14	Useful methods of type String	408
15.15	Useful methods of type String (continued)	413
15.16	Useful methods of type String (continued)	414
16.1	The user interface of an interactive card game.	417
16.2	An empty frame.	417
16.3	Implementation of an empty framed display class	418
16.4	Definition of the framedText subclass.	420
16.5	A framedBarMeter class that references a confidential superclass member	423
16.6	Definition of framedCounter as an extension of framedText	425
16.7	Inheritance hierarchy diagrammed for subclasses of framedDisplay	426
16.8	Definition of framedText with specialized highlight:= method.	428
16.9	Class illustrating redefinition of == method	433
16.10	Version of framedDisplay based on positionContents.	440

16.11	framedText with revised positionContents.	441
16.12	Definition of the framedBarMeter class.	443
16.13	The rubberBall and tennisBall classes for Exercises 16.7.2 and 16.7.3	446
16.14	The SimpleSticky and MessageSticky classes for Exercise 16.7.5	448
17.1	Interface used to enter new calendar events	454
17.2	Exception handling with a try–catch clause	456
17.3	The type of exceptions	457
17.4	Methods for Exercise 17.3.1	458
17.5	Interface for program described in Exercise 17.7.1	462
18.1	A class of objects designed to represent single bookmarks	467
18.2	Creating and printing a list of bookmarks	467
18.3	Opening a file and writing	468
18.4	Browser display of file produced by saveBookmarksFile	470
18.5	Browser display of a usable bookmarks file	470
18.6	Source view of HTML for web page shown in Figure 18.5	471
18.7	Definition of the asBookmarkFileEntry method	472
18.8	Method to create file allowing selection of bookmarks in web page	472
18.9	Reading from a text file	474
18.10	The retrieveBookmarks method	475
18.11	Output format desired for Exercise 18.7.1	481
19.1	Iterative version of linear search of a Number list.	484
19.2	Recursive version of linear search	485
19.3	Recursive binary search	489
19.4	Iterative binary search	490
19.5	Iterative selection sort and helping methods.	493
19.6	Recursive selection sort.	496
19.7	Iterative insertion sort	498
19.8	Recursive insertion sort.	499
19.9	Recursive merge sort	502
19.10	MergeInto method used in merge sort	503
20.1	User interface for a simple Shell Game	512
20.2	First level of design for a cup class: properties and behaviors	513
20.3	First level of design for a Marble class: properties and behaviors	513
20.4	The first pass at a type for a marble	514
20.5	First level of design for a shellGame object: properties and behaviors	514
20.6	Second level of design for a Cup class: modeling properties with instance variables	515
20.7	Second level of design for a Marble class: modeling properties with instance variables	516
20.8	Second level of design for a ShellGame class: modeling properties with instance variables	517

20.9	Third level of design for a Cup class: modeling behaviors with methods . . .	518
20.10	The first pass at a type for a cup	519
20.11	Third level of design for a Marble class: modeling behaviors with methods .	520
20.12	Third level of design for a ShellGame class: modeling behaviors with methods	521
20.13	Filling in the details for a Cup class: constants and instance variablesr . . .	523
20.14	Filling in the details for a Cup class: methods	524
20.15	Filling in the details for a ShellGame class: initializing constants and instance variables.	526
20.16	Filling in the details for a ShellGame class: mouse event-handling methods .	527
20.17	A simple controller to test that a cup can be dragged	530
20.18	A simple graphic application to test that a cup with a marble can be dragged and its contents revealed	532
20.19	Extra methods required by the Cup class	535
20.20	High level design for a Shuffler class	535
20.21	Animation of shuffling	537
20.22	Filling in the design details for the shuffler class	538
20.23	Design of the application object for the animated shell game	539
20.24	Implementation of shuffling	541
20.25	Outline of nested rectangles class	544
20.26	A simple program to draw and move nested rectangles	545
20.27	A sample drawing produced by running RectangleDrawer	546
20.28	An array-based NestedRects class	547
20.29	Leaky faucet and glass before the faucet drips	549
20.30	A Hangman game	550
D.1	Layout of GUI components in a window	593

Chapter 1

What is Programming Anyway?

Most of the machines that have been developed to improve our lives serve a single purpose. Just try to drive to the store in your washing machine or vacuum the living room with your car and this becomes quite clear. Your computer, by contrast, serves many functions. In the office or library, you may find it invaluable as a word processor. Once you get home, connect to a media site on the internet and your computer takes on the role of a television. Start up Skype, FaceTime, or a Google Hangup and you have a videophone. Launch an mp3 player and you suddenly have a music system. This, of course, is just a short sample of the functions a typical personal computer can perform. Clearly, the computer is a very flexible device.

While the computer's ability to switch from one role to another is itself amazing, it is even more startling that these transformations occur without making major physical changes to the machine. Every computer system includes both hardware – the physical circuitry of which the machine is constructed – and software – the programs that determine how the machine will behave. Everything described above can be accomplished by changing the software being used without changing the machine's actual circuitry in any way. In fact, the changes that occur when you switch to a new program are often greater than those you achieve by changing a computer's hardware. If you install more memory or a faster network card, the computer will still do pretty much the same things it did before but a bit faster (hopefully!). In contrast, by downloading a new application program through your web browser, you can make it possible for your computer to perform completely new functions.

Software clearly plays a central role in the amazing success of computer technology. Yet few computer users have a clear understanding of what software really is. This book provides an introduction to the design and construction of computer software in the programming language Grace. By learning to program in Grace, you will acquire a skill that will enable you to construct software of your own and to learn other languages that are used in industry. More importantly, you will gain a clear understanding of what a program really is and how it is possible to change the behavior of a computer in radical ways by constructing a new program.

A program is a set of instructions that a computer follows. We can therefore learn a good bit about computer programs by examining the ways in which instructions written for humans resemble and differ from computer programs. In this chapter we will consider several examples of instructions for humans in order to provide you with a rudimentary

understanding of the nature of a computer program. We will then build on this understanding by presenting a very simple but complete example of a computer program written in Grace. Like instructions for humans, the instructions that make up a computer program must be communicated to the computer in a language that it comprehends. Grace is such a language. We will discuss the mechanics of actually communicating the text of a Grace program to a computer so that it can follow the instructions contained in the program. Finally, you have undoubtedly already discovered that programs don't always do what you expect them to do. When someone else's program misbehaves, you can complain. When this happens with a program you wrote yourself, you will have to figure out how to change the instructions to eliminate the problem. To prepare you for this task, we will conclude this chapter by identifying several categories of errors that can be made when writing a program.

1.1 Without Understanding

You have certainly had the experience of following instructions of one sort or another. Electronic devices from computers to cameras come with thick manuals of instructions. Forms, whether they be tax forms or the answer sheet for an SAT exam, come with instructions explaining how they should be completed. You can undoubtedly easily think of many other examples of instructions designed for people to follow.

If you have had to follow instructions, it is likely that you have also complained about the quality of the instructions. The most common complaint is probably that the instructions take too long to read. This, however, may have more to do with our impatience than the quality of the instructions. A more serious complaint is that instructions are often unclear and hard to understand.

It seems obvious that a set of instructions is more likely to be followed correctly if they are easy to understand. This "obvious" fact, however, does not generalize to the types of instructions that make up computer programs. A computer is just a machine. Understanding is something humans do, but not something machines do. How can a computer understand the instructions in a computer program? The simple answer is that it cannot. As a result, the instructions that make up a computer program have to satisfy a very challenging requirement. It must be possible to follow the individual instructions correctly *without actually understanding their overall purpose*.

This may seem like a preposterous idea. How can you follow instructions if you don't understand them? Fortunately, there are a few examples of instructions for humans that are deliberately designed so that they can be followed without understanding. Examining such instructions will give you a bit of insight into how a computer must follow the instructions in a computer program.

First, consider the "mathematical puzzle" described below. To appreciate this example, don't just read the instructions. Follow them as you read them.

1. Pick a number between 1 and 40.
2. Subtract 20 from the number you picked.
3. Multiply by 3.

4. Square the result.
5. Add up the individual digits of the result.
6. If the sum of the digits is even, divide by 2.
7. If the result is less than 5 add 5, otherwise subtract 4.
8. Multiply by 2.
9. Subtract 6.
10. Find the letter whose position in the alphabet is equal to the number you have obtained (a=1, b=2, c=3, etc.)
11. Think of a country whose name begins with this letter.
12. Think of a large mammal whose name begins with the second letter of the country's name.

You have probably seen puzzles like this before. The whole point of such puzzles is that you are supposed to be surprised that it is possible to predict the final result produced even though you are allowed to make random choices at some points in the process. In particular, this puzzle is designed to leave you thinking about elephants. Were you thinking about an elephant when you finished? Are you surprised we could predict this?

The underlying reason for such surprise is that the instructions are designed to be followed without being understood. The person following the instructions thinks that the choices they get to make in the process (choosing a number or choosing any country whose name begins with "D"), could lead to many different results. A person who understands the instructions realizes this is an illusion.

To understand why almost everyone who follows the instructions above will end up thinking about elephants, you have to identify a number of properties of the operations performed. The steps that tell you to multiply by 3 and square the result ensure that after these steps the number you are working with will be a multiple of nine. When you add up the digits of any number that is a multiple of nine, the sum will also be a multiple of nine. Furthermore, the fact that your initial number was relatively small (less than 40), implies that the multiple of nine you end up with is also relatively small. In fact, the only possible values you can get when you sum the digits are 0, 9 and 18. The next three steps are designed to turn any of these three values into a 4 leading you to the letter "D". The last step is the only point in these instructions where something could go wrong. The person following them actually has a choice at this point. There are four countries on Earth whose names begin with "D": Denmark, Djibouti, Dominica and the Dominican Republic. Luckily, for most readers of this text, Denmark is more likely to come to mind than any of the other three countries (even though the Dominican Republic is actually larger in both land mass and population).

This example should make it clear that it is possible to *follow* instructions without understanding how they work. It is equally clear that it is not possible to *write* instructions like those above without understanding how they work. This contrast provides an important insight into the relationship between a computer, a computer program and the author of

the program. A computer follows the instructions in a program the way you followed the instructions above. It can comprehend and complete each step individually but has no understanding of the overall purpose of the program, the relationships between the steps, or the ways in which these relationships ensure that the program will accomplish its overall purpose. The author of a program, on the other hand, must understand its overall purpose and ensure that the steps specified will accomplish this purpose.

Instructions like this are important enough to deserve a name. We call a set of instructions designed to accomplish some specific purpose even when followed by a human or computer that has no understanding of their purpose an *algorithm*.

There are situations where specifying an algorithm that accomplishes some purpose can actually be useful rather than merely amusing. To illustrate this, consider the standard procedure called long division. A sample of the application of the long division procedure to compute the quotient $13042144/32$ is shown below:

$$\begin{array}{r}
 407567 \\
 32 \overline{)13042144} \\
 \underline{128} \\
 242 \\
 \underline{224} \\
 181 \\
 \underline{160} \\
 214 \\
 \underline{192} \\
 224 \\
 \underline{224} \\
 0
 \end{array}$$

Although you may be rusty at it by now, you were taught the algorithm for long division sometime in elementary school. The person teaching you might have tried to help you understand why the procedure works, but ultimately you were probably simply taught to perform the process by rote. After doing enough practice problems, most people reach a point where they can perform long division but can't even precisely describe the rules they are following, let alone explain why they work. Again, this process was designed so that a human can perform the steps without understanding exactly why they work. Here, the motivation is not to surprise anyone. The value of the division algorithm is that it enables people to perform division without having to devote their mental energies to thinking about why the process works.

Finally, to demonstrate that algorithms don't always have to involve arithmetic, let's consider another example where the motivation for designing the instructions is to provide a pleasant surprise. Well before you learned the long division algorithm, you were probably occasionally entertained by the process of completing a connect-the-dots drawing like the one shown in Figure 1.1. Go ahead! It's your book. Connect the dots and complete the picture.

A connect-the-dots drawing is basically a set of instructions that enable you to draw a picture without understanding what it is you are actually drawing. Just as it wasn't clear that the arithmetic you were told to perform in our first example would lead you to think



Figure 1.1: Connect dots 1 through 82 (©2000 Monkeying Around)

of elephants, it is not obvious looking at Figure 1.1 that you are looking at instructions for drawing an elephant. Nevertheless, by following the instructions “Connect the dots” you will do just that (even if you never saw an elephant before).

This example illustrates a truth of which all potential programmers should be aware. It is harder to devise an algorithm to accomplish a given goal than it is to simply accomplish the goal. The goal of the connect-the-dots puzzle shown in Figure 1.1 is to draw an elephant. In order to construct this puzzle, you first have to learn to draw an elephant without the help of the dots. Only after you have figured out how to draw an elephant in the first place will you be able to figure out where to place the dots and how to number them. Worse yet, figuring out how to place and number the dots so the desired picture can be drawn without ever having to lift your pencil from the paper can be tricky. If all you really wanted in the first place was a picture of an elephant, it would be easier to draw one yourself. Similarly, if you have a division problem to solve (and you don’t have a calculator handy) it is easier to do the division yourself than to try to teach the long division algorithm to someone who doesn’t already know it so that they can solve the problem for you.

As you learn to program, you will see this pattern repeated frequently. Learning to convert your own knowledge of how to perform a task into a set of instructions so precise that they can be followed by a computer can be quite challenging. Developing this ability, however, is the key to learning how to program. Fortunately, you will find that as you acquire the ability to turn an informal understanding of how to solve a class of problems into a precise algorithm, you will be developing mental skills you will find valuable in many other areas.

1.2 The Grace Programming Language

An algorithm starts as an idea in one person’s mind. To become effective, it must be communicated to other people or to a computer. Communicating an algorithm requires the use of a language. A program is just an algorithm expressed in a language that a computer can comprehend.

The choice of the language in which an algorithm is expressed is important. The numeric calculation puzzle that led you to think of Danish elephants was expressed in English. If our instructions had been written in Danish, most readers of this text would not understand them.

The use of language in a connect-the-dots puzzle may not be quite as obvious. Note, however, that we could easily replace the numbers shown with numbers expressed using the roman numerals I, II, III, IV, ... LXXXII. Most of you probably understand Roman numerals, so you would still be able to complete the puzzle. You would probably have difficulty, however, if we switched to something more ancient like the numeric system used by the Babylonians or to something more modern like the binary system that most computers use internally, in which the first few dots would be labeled 1, 10, 11, 100, 101, and 110.

The use of language in the connect-the-dots example is interesting from our point of view because the language used is quite simple. Human languages like English and Japanese are very complex. It would be very difficult to build a computer that could understand a complete human language. Instead, computers are designed to interpret instructions written in simpler languages designed specifically for expressing algorithms intended for computers.

Computer languages are much more expressive than a system for writing numbers like the Roman numerals, but much simpler in structure than human languages.

One consequence of the relative simplicity of computer languages is that it is possible to write a program to translate instructions written in one computer language into another computer language. These programs are called *compilers*. The internal circuitry of a computer usually can only interpret commands written in a single language. The existence of compilers, however, makes it possible to write programs for a single machine using many different languages. Suppose that you have to work with a computer that can understand instructions written in language A but you want to write a program for the machine in language B. All you have to do is find (or write) a program written in language A that can translate instructions written in language B into equivalent instructions in language A. This program would be called a compiler for B. You can then write your programs in language B and use the compiler for B to translate the programs you write into language A so the computer can comprehend the instructions.

Each computer language has its own advantages and disadvantages. Some are simpler than others. This makes it easier to construct compilers that process programs written in these languages. At the same time, a language that is simple can limit the way you can express yourself, making it more difficult to describe an algorithm. Think again about the process of constructing the elephant connect-the-dot puzzle. It is easier to draw an elephant if you let yourself use curved lines than if you restrict yourself to straight lines. To describe an elephant in the language of connect-the-dot puzzles, however, you have to find a way to use only straight lines. On the other hand, a language that is too complex can be difficult to learn and use.

In this text, we will teach you how to use a language named Grace to write programs. Grace is a language that was designed to be used in teaching novices how to program in an object-oriented style, a style that we feel provides the best introduction to modern programming practices.

Other approaches to programming use programming languages like Java and Python. Java is a very popular object-oriented programming language, but it is designed for professional programmers. As a result it has many bells and whistles that are irrelevant for novice programmers, but that make the language fairly complex. These features are required even in relatively simple programs, making it a real challenge to introduce Java in a rational way to novice programmers. The original version of this book used Java, but the language complexities required us to focus more on the language idiosyncrasies and less on the essentials of programming than we would have hoped.

Python is another language that has become popular for teaching novices. It is simpler than Java, but has its own idiosyncrasies. In particular, its support for object-oriented programming is missing some of the key features (e.g., information hiding) that are key to writing good programs.

Both of these language are also showing their age. Java was first released by Sun Microsystems in 1995, nearly 20 years before this book was published. Python is even older, with the first code released to the internet in 1991, and version 1.0 of the language released in 1994.

Twenty years later, we believe that Grace is a superior language for learning to program.

Because it was designed for novices, the language designers have at each stage of development tried to keep the language as simple as possible, introducing new features only where necessary, and avoiding features that unnecessarily complicate the language.

Our approach to programming includes an emphasis on what is known as *event-driven programming*. In this approach, programs are designed to react to *events* generated by the user or system. The programs that you are used to using on computers use the event-driven approach. You do something – press on a mouse button, select an item from a menu, etc. – and the computer reacts to the “event” generated by that action. In the early days of computing, programs were provided with a collection of input data and then run to completion. Many text books still teach that approach to programming. In this text we take the more intuitive event-driven approach to programming. As a result, you will be able to create programs more like the ones you use every day.

1.3 Your First Encounter with Grace

The task of learning any new language can be broken down into at least two parts: studying the language’s rules of grammar and learning its vocabulary. This is true whether the language is a foreign language, such as French or Japanese, or a computer programming language, such as Grace. In the case of a programming language, the vocabulary you must learn consists primarily of verbs that can be used to command the computer to do things like “**show** the number 47.2 on the screen” or “**move** the image of the game piece to the center of the window.” The grammatical structures of a computer language enable you to form phrases that instruct the computer to perform several primitive commands in sequence or to choose among several primitive commands based on user input.

When learning a new human language, one undertakes the tasks of learning vocabulary and grammar simultaneously. One must know at least a little vocabulary before one can understand examples of grammatical structure, while developing an extensive vocabulary without any knowledge of the grammar used to combine words would be silly. The same applies to learning a programming language. Accordingly, we will begin your introduction to Grace by presenting a few sample programs that illustrate fundamentals of the grammatical structure of Grace programs using just enough vocabulary to enable us to produce some interesting examples.

1.3.1 Programming Tools

Writing a program isn’t enough. You also have to get the program into your computer and convince your computer to follow the instructions it contains.

A computer program is just a fragment of text. You already know ways to get other forms of textual information into a computer. You use a word processor to write papers. When entering the body of an email message you use an email application like Apple Mail or Outlook, or you use a web-based mail program like Gmail. Just as there are computer applications designed to allow you to enter these forms of text, there are applications designed to enable you to enter the text of a program.

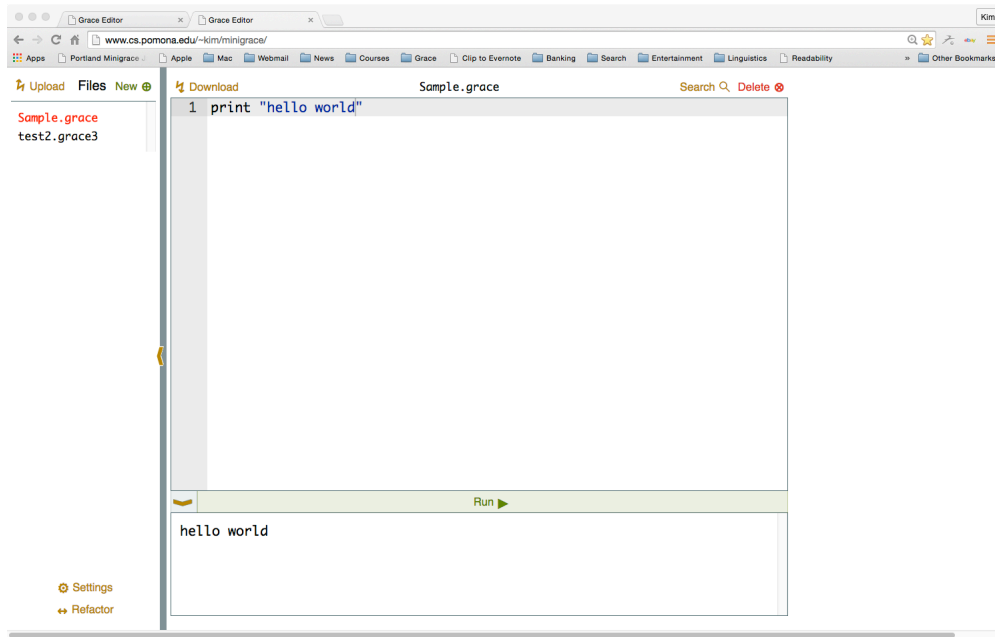


Figure 1.2: A Web Page to Run Grace Programs

Entering the text of your program is only the first step. As explained earlier, unless you write your program in the language that the machine’s circuits were designed to interpret, you need to use a compiler to translate your instructions into a language the machine can comprehend. Finally, after this translation is complete you still need to somehow tell the computer to treat the file(s) created by the translation process as instructions and to follow them.

Typically, the support needed to accomplish all three of these steps is incorporated into a single application called an *integrated development environment* or IDE. It is also possible to provide separate applications to support each step. Which approach you use will likely depend on the facilities available to you and the inclination of the instructor teaching you to program. There are too many possibilities for us to attempt to cover them all in this text. To provide you with a simple way of running Grace programs, however, we will sketch how you can write and execute Grace programs using the Firefox web browser.¹

Open Chrome (or Firefox) and go to the page with URL <http://www.cs.pomona.edu/~kim/minigrace/>. You should get a page that looks roughly like that shown in Figure 1.2

If a page like this does not appear, you may have set your browser to block Javascript, the language that the web application is written in. If you have problems, go to the preferences for your browser to see how to turn Javascript on again, if you have it turned off.

You can now write your first Grace program in the editor pane in the upper right part of the window. First click on the green “New” near the top-left corner of the screen. When it asks for the name of a new file, type any name that does not have a “.” in it. When you click

¹ Other browsers may work, but the layout on the screen may be different. We urge you to use Firefox as all of our software has been optimized for that platform.

the "OK" button, it will add a suffix of ".grace" and add it to the pane on the left side of the window.

Click in the edit area in the upper right hand side of the window and then type the following exactly as shown:

```
print "hello world!"
```

Click on the Run button directly below the editing area. It will then compile and run your program. Try it! This should result in the string "hello" (without the double quotes) being printed in the pane in the bottom right hand portion of the screen (below the run button).

Congratulations, you have executed your first Grace program! Next, let's modify the program to make it more personal. Click in the edit pane erase everything after "hello", but before the double quote. Add a comma, a space and your name. Make sure they are all inside the double quotes. For example, if your name were Jilan, the program would read

```
print "Hello, Jilan"
```

Now add a new line after the first in the edit pane:

```
print "This is your first Grace program!"
```

Again, click on the build button to compile your program. When it changes to "run", click on the run button to execute the program. Assuming that you haven't made any typos, the program should print out:

```
Hello, Jilan
```

```
This is your first Grace program!
```

Let's back up for a minute to talk about the code you've written. `print` is a command to display things on the screen (we still call it *print* even though nothing gets printed to paper any more). Words in quotes are to be taken as data consisting of the sequence of letters enclosed in quotes. These are referred to as string literals. Thus in the simple program `print "hello"`, `print` is a command, while `"hello"` is a string literal.

Of course programs that just print out a greeting are not very interesting. Soon we'll look at programs that can draw images in a window.

Before we end this section, let's see how to save your work. To save your program, hold down the control key (typically in the bottom left corner of the keyboard) and press the mouse button on the "Download" button at the top of the window. From the pop-up menu you get, select "Save Link As ...". Navigate to the folder where you want to save your program and make sure the "save" window has the right name for your file (fix it if it doesn't). When everything is the way you want it, click on the save button and your program will be retained on our system.

If you don't save it like this, then when you log out, your program will be lost, so don't forget to save!

1.3.2 Definitions & Methods

In this section we look at how to teach Grace to do new things by defining methods. However, we first start with a brief excursion into naming things.

It is often useful to be able to associate names with values in Grace. For example, the following definition associates the name `pi` with the value 3.14159265359.

```
def pi = 3.14159265359
```

Once we have this defined we can print the area of a circle with radius 7 by evaluating `pi * 7 * 7`.

In computer science we usually use the symbol `*` to represent multiplication, though the other arithmetic operators are the more familiar ones: `+`, `-`, `/`. We must also be careful to put spaces on both sides of operator symbols in Grace like `*`, `+`, etc.

As another example, we can write

```
def newFriend = "Jilan"
```

Now `print (newFriend)` would result in printing out the string "Jilan".

It is convenient to associate names to values, especially when the values themselves might be long and complicated. For example, it's much easier to write `pi` than remember the long string of digits. Moreover, we can associate descriptive names with values, making it easier to figure out why role they play in programs. Thus your program will be easier to read if you use an identifier like `candleHeight` rather than just a number like `173` in expressions.

The ability to name things is critically important in programming. (As it is in real life. Imagine if everything needed to be specified by giving a complete description!) Any value in Grace can be associated with a name via a **def** declaration.

Identifiers are used to name things in Grace. Examples are `pi`, `newFriend`, and the rather absurd `reallyLongNameWithNumber47Int!`. Identifiers must start with a letter between `a` and `z` (either lower case or upper case) and then include letters, digits (0 through 9), and the special character `'`.

A *method* is a named sequence of program instructions. Thus methods allow you to specify names for operations that you may wish to use repeatedly. A simple example is the following greeting method

```
method greet (name) {  
    print "Hello there, {name}"  
}
```

When we request Grace to `greet ("Alessandro")`, it will print out the string "Hello there, Alessandro" in the output pane.

As you might guess from the example, surrounding an item inside a string literal by curly braces causes the material inside to be evaluated, and then inserted at that point in the string. If we had not surrounded `name` by curly braces, the method execution would have printed "Hello there, name", not at all what we wanted!

If instead we wrote `greet ("Beau")`, then the string "Hello there, Beau" would be printed instead. The identifier `name` in the method header (the line starting with the word **method**) is said to be a *parameter* of the method. It represents a value that can be supplied for each

```

method square (value) {
    value * value
}

print "The results of squaring 7 is {square (7)}."

```

Figure 1.3: Program to calculate the square of 7

execution of the method. You can see that `name` is mentioned in the next line, where it is inserted after “Hello there,” and the result printed.

Notice that neither occurrence of `name` is surrounded by quotes. That is because `name` is an identifier that represents a string. When the system is executing `greet ("Alessandro")`, the value associated with `name` will be the string “Alessandro”.

Thus methods give us a way of defining operations that can be executed with different values. You should already be familiar with similar operations in mathematics. For example, we could also define a method to square numbers:

```

method square (value) {
    value * value
}

```

When this method is provided with a number, it will return the square of that number. Thus `square(7)` evaluates to 49.

We can use this method in a larger program as shown in figure 1.3. That program will print

```
The result of squaring 7 is 49.
```

The curly braces in the print statement above are doing a bit more work than we have seen previously. Numbers and strings are different and incompatible in Grace. In general, an expression surrounded by curly braces inside a string literal is first evaluated, then converted to a string, and then inserted into the containing string. Thus, in the example above, `square(7)` is evaluated to the number 49, then converted to the string 49, and then finally inserted into the string literal just before the final period.

Of course methods can be composed of more than one command

```

method betterGreeting (toName, fromName) {
    print "Hello there, {toName}!"
    print "My name is {fromName}."
}

```

When a method is requested, the code enclosed in braces is executed. Thus, if we evaluate `betterGreeting ("Luis","Zelda")`, the following will be printed:

```
hello there, Luis!
My name is Zelda.
```

You will notice that when you see `betterGreeting ("Luis","Zelda")` it is not obvious which parameter represents the greeter and which the greete. We can improve things by giving a slightly better name to the method and separating the parameters:

```

method bestGreetingTo (toName) from (fromName) {
    print "Hello there, {toName}!"
    print "My name is {fromName}."
}

```

Now when we write `bestGreetingTo ("Luis")` from `("Zelda")` it is much clearer who is being greeted ("Luis") and by whom ("Zelda"). Whenever it seems helpful for understanding, we will separate the parameters with words in this way, rather than just separating them with commas.

You may have noticed that sometimes we put parentheses around the material to be printed in a print statement, and sometimes we don't. You are always safe putting in the parentheses. However, if you are printing out a single string literal (a string surrounded by double quotes) then the parentheses may be omitted, as the computer can use the double quotes to determine where the material to print starts and ends.

1.3.3 Objects

Not surprisingly, object-oriented programming is all about objects. Objects provide a way of encapsulating (grouping together) data and operations (methods) on that data. Let's see how we can do this in Grace.

Grace provides an "object" expression to create objects. The object expression can contain definitions of methods as well as data associated with the object.

```
def firstPerson = object {
  def myName = "Shezad"
  def myAge = 21
  method greet(name) {
    print "Hello there, {name}. My name is {myName}."
  }
}
```

This object definition associates the identifier `firstPerson` with an object that contains definitions and methods representing an individual with name "Shezad", who is 21 years old. The identifiers `myName` and `myAge` are specified in `def` statements within the object expression.

We can send a *method request* to an object by writing the object followed by a period, followed by the method name and any associated arguments: `firstPerson.greet ("Jilan")`.

Executing the above will print

```
Hello there, Jilan. My name is Shezad.
```

We generally define many objects in a program. For example, in Figure 1.4, we show code that also create objects representing a second person and a dog.

The object associated with `secondPerson` has the same overall features (definitions and methods) as `firstPerson`, but the values associated with the identifiers `myName` and `myAge` are different. Because of that, if we evaluate `secondPerson.greet ("Jilan")`

The program will print out

```
Hello there, Jilan. My name is Charles.
```

Our dog also has definitions for `myName` and `myAge`, but it also has a field that indicates whether or not it has been spayed. Notice the value associated with `spayed` is not surrounded

```

def secondPerson = object {
  def myName = "Charles"
  def myAge = 24
  method greet(name) {
    print ("Hello there, {name}. My name is {myName}.")
  }
}

def dog = object {
  def myName = "Rover"
  def myAge = 7
  def spayed = true
  method speak {
    print "woof"
  }
  method spin(n) {
    print "{myName} spins {n} times on command"
  }
}

```

Figure 1.4: Defining a second person and a dog

by quotes, because it is not a string. It is a built-in value, `true`. You won't be surprised to learn there is another built-in value, `false`. These two are called Boolean values. They are the only two Boolean values. We'll see later that these values are quite important in guiding program executions.

The `dog` also has two methods, `speak` and `spin`. The first takes no parameters, while the second takes a single parameter `n` representing the number of times `dog` should spin on command.

The point here is that some objects are similar to each other in that they have the same defined fields and methods, while others can be quite different.

1.3.4 Classes

In the last section we defined two objects, `firstPerson` and `secondPerson`, that had exactly the same defined fields and method. They differed only by the values of the `myName` and `myAge` fields.

When we have a need for a number of objects with similar structures, we can define a *class* to generate those new objects. For example, look at the class `personNamed()` age below

```

class personNamed (nameVal) age (ageVal) {
  def myName = nameVal
  def myAge = ageVal
  method greet (name) {
    print ("Hello there, {name}. My name is {myName}.")
  }
}

def firstPerson = personNamed ("Shezad") age (21)

```



```
def secondPerson = personNamed("Charles") age (24)
```

As you can see, we could define both `firstPerson` and `secondPerson` directly from the class. Because of the arguments provided, `firstPerson` has name “Shezad” and age 21 (as before), while `secondPerson` has name “Charles” and age 24 (again, just as before).

It would be possible to write this instead as a method that returns an object. However, both for simplicity and because most object-oriented languages support classes like this, we will use the class construct in this text.

When we refer to a Grace class, we typically omit the information in parentheses in the declaration. We also drop the final open and closed parentheses in order to make the name a bit more compact. Thus we referred to the name of the above class as `personNamed() age`.

1.3.5 Program Layout

You may have noticed that we have been very careful in the indenting of our programs. Grace will insist that your program be laid out very consistently, as this will help you understand how pieces of a Grace program fit together.

All the items that are declared in an object should start at the same indentation, as in the examples above. Code in a method should start 4 spaces in from the method header (the line starting with the word “method”), and all subsequent lines in the method should be at the same indentation. The closing curly bracket is generally put at the same indentation as the method header.

Lines that are continuations of a previous line should always be indented by 5 more spaces so the compiler knows it is a continuation. Otherwise Grace will interpret the continuation as a new command and get confused. For example, suppose you write

```
def x = 13
  -7
```

This will result in `x` having the value of 13. While if instead you wrote

```
def x = 13
-7
```

then `x` will have the value `13-7` or 6, as it will assume that the `-7` is a continuation of the previous line. In the case where the `-7` was not indented, Grace interpreted `-7` as starting a new command, even though it doesn’t make much sense.

We recommend that you avoid using the tab key in laying out your program. Some editors can be configured so that tabs are automatically replaced by a fixed number of spaces. Any tabs in your program text will cause an error in Grace programs.

1.4 Generating Graphics with Objectdraw

Over the years, we (and many others!) have found that computer graphics is an excellent medium for learning programming. The results are more interesting than just printing out lines of text as answers. Perhaps more importantly though, when the output of your program is an image (or perhaps even an animated image), it is easy to see when your program has errors, and the broken images can lead you to understand where you have made mistakes.

Our purpose in this course is not to make you an expert graphics programmer (though some of you may go on to become that). Instead we will use graphics as a tool to help you learn to program.

In this text we use the `objectdraw` library to write Grace programs that create graphical images. It is based on a similar library (also called `objectdraw`) that was devised for teaching Java to novice programmers. You will see that it will allow us to create interesting programs relatively quickly.

We begin in the next section to understand how coordinate systems work in computer graphics, and then quickly move on to show you how to create interactive programs using the `objectdraw` graphics library.

1.4.1 The Graphics Coordinate System

Programs that display graphics on a computer screen have to deal extensively with a coordinate system similar to that you have used when plotting functions in math classes. This is not evident to users of these programs. A user of a program that displays graphics can typically specify the position or size of a graphical object using the mouse to indicate screen positions without ever thinking in terms of x and y coordinates. Writing a program to draw such graphics, however, is very different from using one. When your program runs, someone else controls the mouse. Just imagine how you would describe a position on the screen to another person if you were not allowed to point with your finger. You would have to say something like “two inches from the left edge of the screen and three inches down from the top of the screen.” Similarly, when writing programs you will specify positions on the screen using pairs of numbers to describe the coordinates of each position.

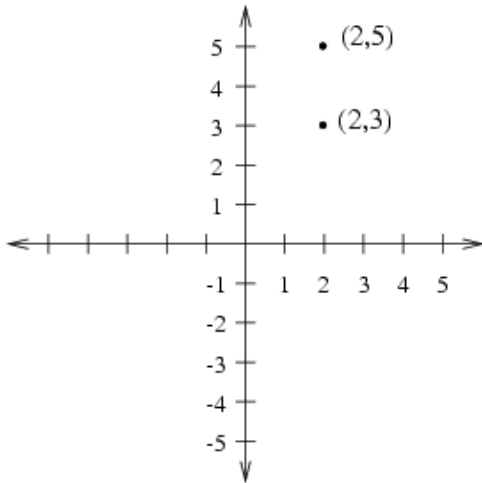
The coordinate system used for computer graphics is like the Cartesian coordinate systems studied in math classes but with one big difference: the y axis in the coordinate system used in computer graphics is upside down. Thus, while your experience in algebra class might lead you to expect the point $(2, 3)$ to appear below the point $(2, 5)$, on a computer screen just the opposite is true. This difference is illustrated by Figure 1.5, which shows where these two points fall in the normal Cartesian coordinate system and in the coordinate system used to specify positions when drawing on a computer screen.

Our Grace programs that use graphics will draw items in a computer window that contains a *canvas*. We’ll talk in more detail later about the canvas, but for now, just think of it as something like a painter’s canvas. In our beginning programs the canvas will fill the entire window, though later we will see how to put other components in the window along with the canvas.

An object representing a point on the canvas can be constructed using a term of the form $x @ y$ where x and y are the x and y coordinates of the point. Thus the location corresponding to $(2, 3)$ can be constructed using `2 @ 3`. Our classes for constructing geometric objects will use points to specify where on the canvas they will appear.

Displaying text on a canvas is more complicated than just writing it to the program output area because we must specify where the string should be written. The `objectdraw` library allows a programmer to place a string on the canvas using an object class of the form `textAt (somePoint) with (someString) on (someCanvas)`, where the italicized words are replaced with the actual information about what we want to display and where it goes.

"Normal" Coordinate System



Computer Graphics Coordinate System

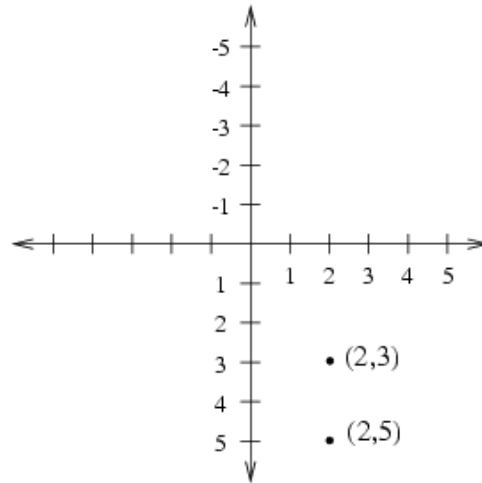


Figure 1.5: Comparison of computer and Cartesian coordinate systems

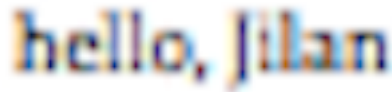


Figure 1.6: Text enlarged to make pixels visible

The graphics that appear on a computer screen are actually composed of tiny squares of different colors called *pixels*. For example, if you looked at the text displayed by our first program with a magnifying glass you would discover it is actually made up of little squares as shown in Figure 1.6. The entire screen is organized as a grid of pixels. The coordinate system used to place graphics in a window is designed to match this grid of pixels in that the basic unit of measurement in the coordinate system is the pixel. So, the coordinates (30, 50) describe the point that is 30 pixels to the right and 50 pixels down from the origin.

Another important aspect of the way in which coordinates are used to specify where graphics should appear is that there is not just a single set of coordinate axes used to describe points anywhere on the computer's screen. Instead, there is a separate set of axes associated with each window on the screen and, in some cases, even several pairs of axes for different parts of a single window.

Rather than complicating the programmer’s job, the presence of so many coordinate systems makes it simpler. Many programs may be running on a computer at once and each should only produce output in certain portions of the screen. If you are running Microsoft Word at the same time as a web browser, you would not expect text from your Word document to appear in one of the browser’s windows. To make this as simple as possible, each program’s drawing commands must specify the window or other screen area in which the drawing should take place. Then the coordinates used in these commands are interpreted using a separate coordinate system associated with that area of the screen. The origin of each of these coordinate systems is located in the upper left corner of the area in which the drawing is taking place rather than in the corner of the machine’s physical display. This makes it possible for a program to produce graphical output without being aware of the location of its window relative to the screen boundaries or the locations of other windows.

In many cases, the area in which a program can draw graphics corresponds to the entire interior of a window on the computer’s display. In other cases, however, the region used by a program may be just a subsection of a window or there may be several independent drawing areas within a given window. Accordingly, we refer to a program’s drawing area as a canvas rather than as a window.

1.4.2 A simple example

We will use the Grace program “TouchyWindow” given in Figure 1.7 to illustrate the basic concepts in our `objectdraw` graphics library. When the program is run, a window will pop up with the text “Click in this window” in the upper left. When the mouse button is pressed, the text “I’m touched” will appear in the middle of the window. When the mouse button is released, that text will disappear.

Let’s walk through the code and see what each section does.

Dialect

The first line of the program specified the “dialect” of Grace that we are using for the program. Dialects are used to restrict the constructs available in the language or to add new features. The dialect `objectdraw` is a dialect that provides commands for creating programs that draw and modify items on the canvas. The new constructs explained in this section are all from the `objectdraw` dialect of Grace. This dialect will make it much easier for us to write graphics programs in Grace.

`inherit graphicApplicationSize()`

Inheritance is used to bring in features to an object that have previously been defined in other objects. This program inherits features from `graphicApplicationSize`.

When your program creates an object inheriting from `graphicApplicationSize`, it will be capable of popping up a new window with a canvas pre-installed for drawing on. Thus virtually all of our programs using graphics will create one or more objects inheriting from `graphicApplicationSize`.

```

dialect "objectdraw"

// program that responds to a mouse press with a simple textual display
object {
  inherit graphicApplicationSize (400 @ 400)

  textAt (20 @ 20) with ("Press mouse in this window") on (canvas)

  // When the user presses mouse, write: I'm touched
  // on canvas at coordinates (180, 200)
  method onMousePress (mousePoint) {
    textAt (180 @ 200) with ("I'm touched") on (canvas)
  }

  // When mouse is released, erase the canvas
  method onMouseRelease (mousePoint) {
    canvas.clear
  }

  // create window and start graphics
  startGraphics
}

```

Figure 1.7: TouchyWindow program in Grace

Objects inheriting from `graphicApplicationSize` will also be prepared to respond to mouse events if the programmer includes the right methods in the object. For example, this program includes the methods `onMousePress` and `onMouseRelease`. Methods with these names are treated specially in objects inheriting from `graphicApplication`, as they will be requested by the system whenever the user presses or releases the mouse (or trackpad) button on the canvas.

The parameter after the word `size` indicates the size of the window. In the case of this example, the window will be 400 pixels wide by 400 pixels tall. If we instead wrote `graphicApplicationSize (800 @ 200)` then the window would be 800 pixels wide and 200 high. Because the “@” symbol builds a pair of numbers it can be used to refer to the size of a two dimensional object as well as a location.

1.4.3 Constructing Graphic Objects

The next line in the object expression constructs a text item at the coordinates (20, 20), which is written in Grace as `20 @ 20`.

```
textAt (20 @ 20) with ("Press mouse in this window") on (canvas)
```

Evaluating that object construction will display “Press mouse in this window” on the default canvas for graphics applications).

As discussed earlier, Grace will assume that the origin of the coordinate system is located at the upper left corner of the canvas in which you are drawing. Thus the lower left hand corner of the text will appear 20 pixels from the left edge of the canvas and 20 pixels down from the top.

The computer will not consider it an error if you try to draw beyond the boundaries of your program’s canvas. However, it will only display the portion of these drawings that fall within the boundaries of your canvas.

There is another statement creating text in the method named `onMousePress`:

```
textAt (180 @ 200) with ("I'm touched") on (canvas)
```

When that construction is evaluated the text “I’m touched” will be displayed starting at the point with coordinates (180, 200).

1.4.4 Methods

The two methods defined in this program are named `onMousePress` and `onMouseRelease`. They are special methods that are associated with graphics applications. As you might guess, the system requests the first method when the user presses the mouse button down, and the second one when the user releases the mouse button. The `mousePoint` parameter in parentheses after each method name represents the place on the canvas where the mouse was pointing when it was pressed or released. It is not used in this program, but must be included anyway. We will talk about how to use it in a later program.

As explained earlier, the code enclosed in curly brackets after the method name is executed when the method is requested by the system. Thus when the mouse button is pressed, a new text item stating “I’m touched” is displayed at x coordinate 180 and y coordinate 200 on the canvas. Similarly, when the mouse button is released, the canvas is cleared, as executing the command `canvas.clear` sends a request to the canvas to execute its `clear` method, which just erases everything on the canvas.

In general, the programmer is free to choose any appropriate name for a method. The method name can then be used in other parts of the program to cause the computer to perform the instructions within the method’s body. Within a class that extends `graphicApplication`, however, certain method names have special significance. In particular, if such a class contains a method which is named `onMousePress` then the instructions in that method’s body will be followed by the computer when the mouse is depressed within the program’s window. That is why this particular program reacts to a mouse press as it does.

Exercise 1.4.1 *Rewrite the line of code in `onMousePress` of program `TouchyWindow` so that it now displays the message “Hello” 60 pixels to the right and 80 pixels down from the top left corner of the window.*

1.4.5 startGraphics

The final line in the object definition is `startGraphics`. When the command `startGraphics` is executed, a window is displayed with whatever graphics have been created on the canvas to this point.

In the `TouchyWindow` example, when the program is run, the text with “Click in this window” will be displayed when `startGraphics` is executed. The code in the method `onMousePress` and `onMouseRelease` will not be executed until the system requests those methods – which will happen when the user presses or releases the mouse button.

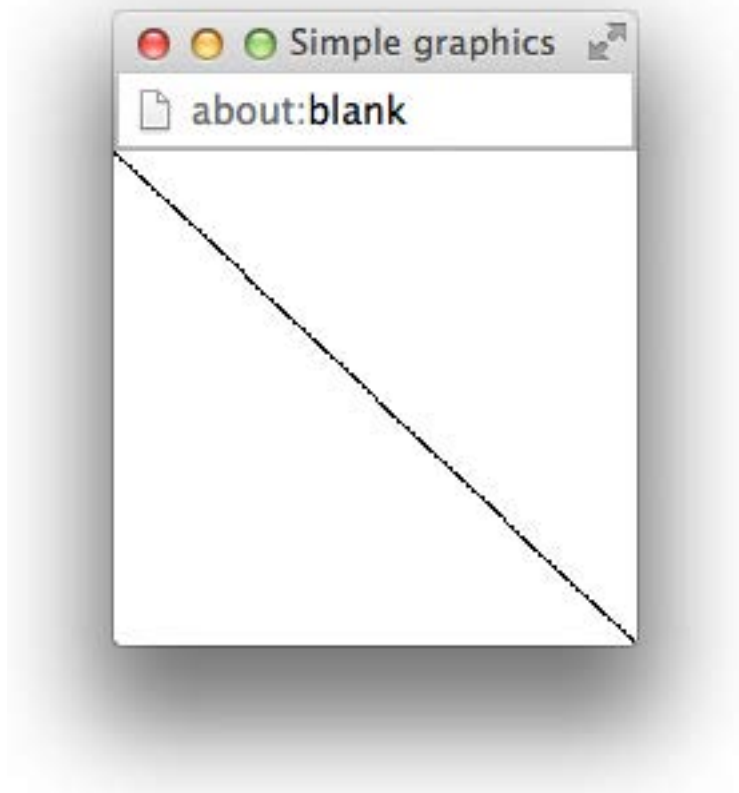


Figure 1.8: Drawing of a single line

1.5 Creating other graphics items

Now that we have seen what a simple graphics program looks like, let's see how we can construct other graphics items on a canvas.

In each case we will need to supply the point where we want the object to be drawn, information about the other attributes of the item (e.g., its width and height) and the canvas it will be drawn on. For example, for a text item, we provided the point where it is to be drawn, the text to be displayed, and the canvas it should be drawn on.

To display a line we use the class `lineFrom` (*start*) to (*end*) on (*canvas*) where *start* and *end* are the end points of the line, and *canvas* is the canvas on which it is drawn.

Thus to draw a line between the upper left and lower right corners of a canvas whose dimensions are 200 by 200, you could write:

```
lineFrom (0 @ 0) to (200 @ 200) on (canvas)
```

The line produced would look like the line shown in the window in Figure 1.8.

Similarly, to draw a line from the middle of the window, which has the coordinates (100,150), to the upper right corner, whose coordinates are (200,0), you would write:

```
lineFrom (100 @ 150) to (200 @ 0) on (canvas)
```

Such a line is shown in Figure 1.9.

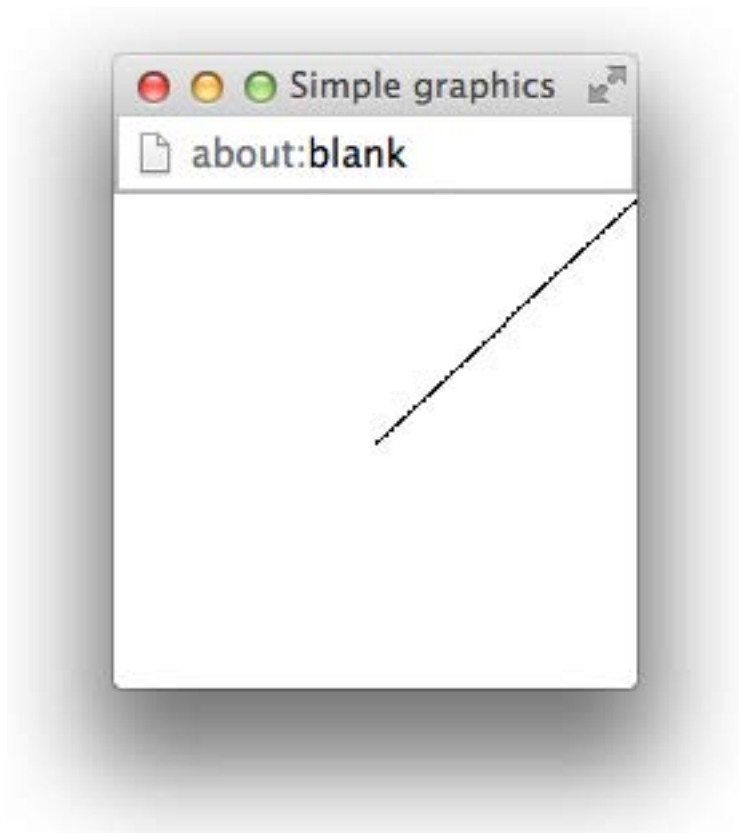


Figure 1.9: A line from (100, 150) to (200, 0)

Using combinations of these construction statements, we could replace the single instruction in the body of the `onMousePress` method shown in the `TouchyWindow` program in Figure 1.7 with one or more other instructions. Such a modified program is shown in Figure 1.10. It draws two crossed lines in the center of the window.

The only differences between this example and `TouchyWindow` are that no text is written on the screen when this object is created, and the commands in method `onMousePress` result in drawing two crossed lines when the button is pressed. The drawing produced is also shown in the figure.

There are several other forms of graphics you can display on the screen. The command:
`framedRectAt (20 @ 50) size (80 @ 40) on (canvas)`

will display the frame, or outline, of an 80 by 40 rectangular box on the canvas. The coordinates (20, 50) represent the location of the box's upper left corner. The pair (80, 40) specifies the width and height of the box. If you replace the name `FramedRect` by `FilledRect` to produce the construction

`filledRectAt (20 @ 50) size (80 @ 40) on (canvas)`

the result will instead be an 80 by 40 solid black rectangular box.

The command:

`filledOvalAt (20 @ 50) size (80 @ 40) on (canvas)`

will draw a filled oval on the screen. The parameters are interpreted just like those to the `FilledRect` construction. However, instead of drawing a rectangle, `FilledOval` draws the largest ellipse that can fit within the rectangle described by its parameters. To illustrate this, Figure 1.11 shows what the screen would contain after executing the two constructions

`framedRectAt (20 @ 50) size (80 @ 40) on (canvas)`

`filledOvalAt (20 @ 50) size (80 @ 40) on (canvas)`

The upper left corner of the rectangle shown is at the point with coordinates (20, 50). Both shapes are 80 pixels wide and 40 pixels high. You can create a framed oval by replacing `filledOval` by `framedOval` in the above command.

Other primitives allow you to draw additional shapes and to display image files on your canvas. A full listing and description of the available graphic object types and the forms of the commands used to construct them can be found in Appendix C. For now, the graphical object generators `textAt()with()on`, `lineFrom()to()on`, `framedRectAt()Size()On`, `filledRectAt()Size()On`, `framedOvalAt()Size()On`, and `filledOvalAt()Size()On` will provide enough flexibility for our purposes.

Exercise 1.5.1 *Sketch the picture that would be produced if the following constructions were executed. You should assume that the canvas associated with the program containing these instructions is 200 pixels wide and 200 pixels high.*

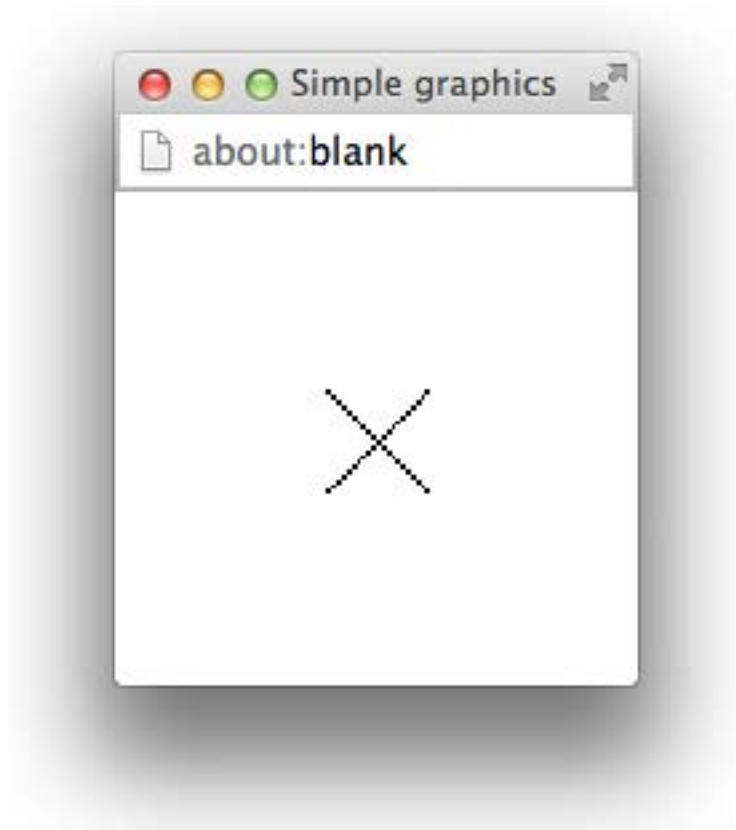
`lineFrom (0 @ 100) to (100 @ 0) on (canvas)`

`lineFrom (200 @ 100) to (100 @ 200) on (canvas)`

`lineFrom (100 @ 200) to (0 @ 100) on (canvas)`

`framedRectAt (50 @ 50) size (100 @ 100) on (canvas)`

Exercise 1.5.2 *Write a sequence of `Line` and/or `FramedRect` constructions that would produce each of the drawings shown below. In both examples, assume that the drawing will appear in a 200 by 200 pixel window. For the drawing of the three dimensional cube, there should be*



```
dialect "objectdraw"
```

```
// program that responds to a mouse press with a cross  
object {  
  inherit graphicApplicationSize (400 @ 400)  
  
  // When the user presses mouse, draw a cross on canvas  
  method onMousePress (mousePoint) {  
    lineFrom (40 @ 40) to (60 @ 60) on (canvas)  
    lineFrom (60 @ 40) to (40 @ 60) on (canvas)  
  }  
  
  // When mouse is released, erase the canvas  
  method onMouseRelease (mousePoint) {  
    canvas.clear  
  }  
  
  // create window and start graphics  
  startGraphics  
}
```

Figure 1.10: A program that draws two crossed lines.

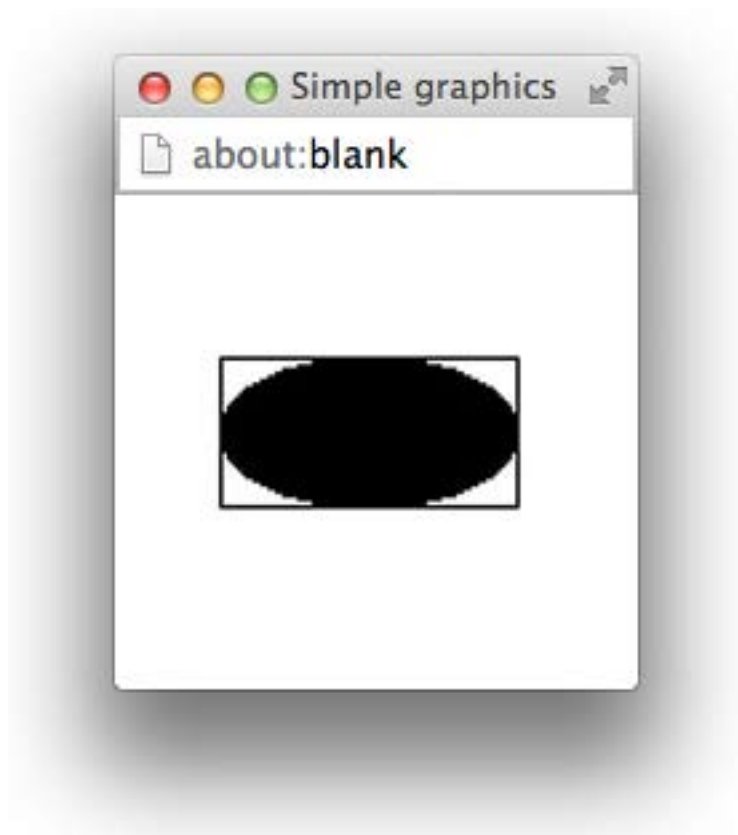
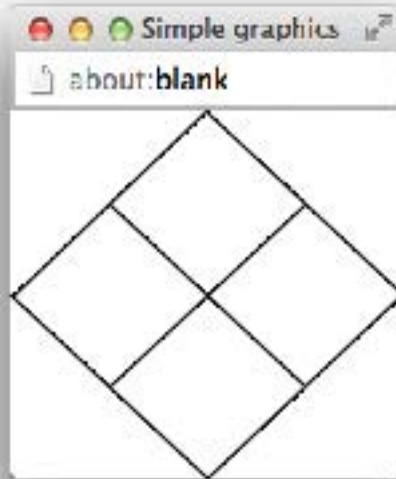


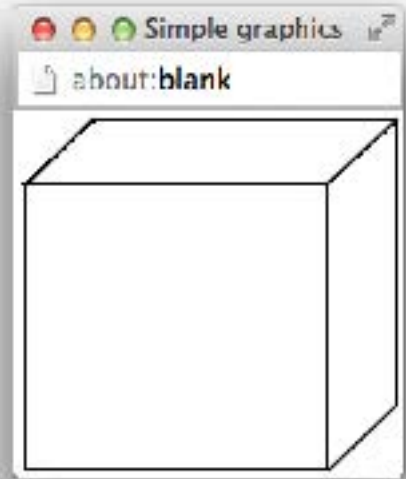
Figure 1.11: A FilledOval nested within a FramedRect

a space 5 pixels wide between the cube and the edges of the window in those areas where the cube comes closest to the edges. The rectangle drawn for the front face of the cube should be 155 pixels wide and 155 pixels high. The two visible edges of the rear of the cube should also be 155 pixels long.

a)



b)



1.6 Additional Event Handling Methods

In our examples thus far, we have used the two method names `onMousePress` and `onMouseRelease` to establish a correspondence between certain user actions and instructions we would like the computer to follow when these actions occur. In this section, we introduce several other method names that can be used to associate instructions when the user performs an action with the mouse. When the user performs such an action, the system generates a *mouse event* that can be handled by an object that extends `graphicApplication`.

1.6.1 Mouse Event Handling Methods

In addition to `onMousePress` and `onMouseRelease`, there are five other method names that have special significance for handling mouse events. If you include definitions for any of these methods within a class that inherits `graphicApplicationSize`, then the instructions within the methods you include will be executed when the associated events occur.

The definitions of all these methods have the same form. You have seen that the header for the `onMousePress` method looks like:

method `onMousePress` (point)

The headers for the other methods are identical except that `onMousePress` is replaced by the appropriate method name.

All of the mouse event handling methods are described below:

`onMousePress` specifies the actions the computer should perform when the mouse button is depressed.

`onMouseRelease` specifies the actions the computer should perform when the mouse button is released.

`onMouseClicked` specifies the actions the computer should perform if the mouse is pressed and then quickly released without significant mouse movement between the two events. The actions specified in this method will be performed in addition to (and after) any instructions in `onMousePress` and `onMouseRelease`.

`onMouseEnter` specifies the actions the computer should perform when the mouse enters the program's canvas.

`onMouseExit` specifies the actions the computer should perform when the mouse leaves the program's canvas.

`onMouseMove` specifies the actions the computer should perform periodically while the mouse is being moved about without its button depressed.

`onMouseDown` specifies the actions the computer should perform periodically while the mouse is being moved about with its button depressed.

If your computer uses a trackpad, touch screen, or other input mechanism, the above methods will be triggered by the equivalent actions with that input.

Exercise 1.6.1 *Write the complete method header for the `onMouseMove` method.*

Exercise 1.6.2 *Write a method that draws a filled square on the canvas when the mouse enters the canvas. The square should be 100×100 pixels with the upper left corner at the origin.*

Exercise 1.6.3 *Write a complete program that will display “I’m inside” when the mouse is inside the program’s window and “I’m outside” when the mouse is outside the window. The screen should be blank when the program first begins to execute and should stay blank until the mouse is moved in or out of the window.*

1.6.2 The Initialization code

Code that is inside an object expression but is not included in a method body will be executed when the object is evaluated. In graphics applications this code is generally responsible for creating the initial image presented on the canvas and doing any other initialization necessary before the user begins interacting with the object or application. Typically the last line in an object inheriting from `graphicApplication` will be `startGraphics`, which displays the window and readies it to respond to user actions.

In our `TouchyWindow` program shown in Figure 1.7, the initialization code created a text item that displays the instructions on the canvas. Finally, the command `startGraphics` created the window and made it ready to respond to mouse clicks. The code in the methods `onMousePress` and `onMouseRelease` was only executed after a mouse press or release event.

1.7 To Err is Human

We all make mistakes. Worse yet, we often make the same mistakes over and over again.

If we make a mistake while giving instructions to another person, the other person can frequently figure out what we really meant. For example, if you give someone driving directions and say “turn left” somewhere you should have said “turn right” chances are that they will realize after driving in the wrong direction for a while that they are not seeing any of the landmarks the remaining instructions mention, go back to the last turn before things stopped making sense and try turning in the other direction. Our ability to deal with such incorrect instructions, however, depends on our ability to understand the intent of the instructions we follow. Computers, unfortunately, never really understand the intent of the instructions they are given so they are far less capable of dealing with errors.

There are several distinct types of errors you can make while writing or entering a program. The computer will respond differently depending on the type of mistake you make. The first type of mistake we discuss is called a *syntax error*. The defining feature of a syntax error is that the IDE can detect that there is a problem before you try to run your program. As we have explained, a computer program must be written in a specially designed language that the computer can interpret or at least translate into a language which it can interpret. Computer languages have rules of grammar just like human languages. If you violate these rules, either because you misunderstand them or simply because you make a typing mistake, the computer will recognize that something is wrong and tell you that it needs to be corrected.

The mechanisms used to inform the programmer of syntactic errors vary from one IDE to another. The web IDE for Grace examines the text you have entered and indicates fragments of your program that it has identified as errors by displaying an error message and indicating the offending line number. For example, if we accidentally left out the closing “}” after the body of an `onMousePress` method in a program `NoClicking`, the system would print out an error message like the following:

```
Syntax error: The indentation for this line must be at least 2. This is
  often caused by a missing '}'.
in "NoClicking" (line 13, column 4)
```

where line 13 is the line that should have the closing curly brace.

The bad news is that your IDE will not always provide you with a message that pinpoints your mistake so clearly. When you make a mistake, your IDE is forced to try to guess what you meant to type. As we have emphasized earlier, your computer really cannot be expected to understand what your program is intended to do or say. Given its limited understanding, it will often mistake your intention and display error messages that reveal its confusion. For example, if you type

```
    framedRectAt (10) size (20 20) on (canvas)
instead of
    framedRectAt (10 @ 20) size (20 @ 20) on (canvas)
```

in the body of the `onMouseRelease` method of our example program, the IDE will print something unhelpful like:

```
TypeError at line 1196 of objectdraw: expected initial value of var
```

```
'location' to be of type self.outer.outer.Point
...
```

In such cases, the error message may be more likely to confuse rather than help. You will just have to examine what you typed before and after the position where the IDE identified the error and use your knowledge of the Grace language to identify your mistake.

A program that is free from syntax errors is not necessarily a correct program. Think back to our instructions for performing calculations that was designed to leave you thinking about Danish elephants. If while typing these instructions we completely omitted a line, the instructions would still be grammatically correct. Following them, however, would no longer lead you to think of Danish elephants. The same is true for the example of saying “left” when you meant to say “right” while giving driving directions. Mistakes like these are not syntactic errors; they are instead called *logic errors*. They result in an algorithm that doesn’t achieve the result that its author intended. Unfortunately, to realize such a mistake has been made, you often have to understand the intended purpose of the algorithm. This is exactly what computers don’t understand. As a result, your IDE will give you relatively little help correcting logic errors.

As a simple example of a logical error, suppose that while typing the `onMouseRelease` method for the `TouchyWindow` program you got confused and typed `onMouseExit` instead of `onMouseRelease`. The result would still be a perfectly legitimate program. It just wouldn’t be the program you meant to write. Your IDE would not identify your mistake as a syntax error. Instead, when you ran the program it just would not do what you expected. When you released the mouse, the “I’m Touched” message would not disappear as expected.

This may appear to be an unlikely error, but there is a very common error which is quite similar to it. Suppose instead of typing the name `onMouseRelease` you typed the name `onMooseRelease`. Look carefully. These names are not the same. Thus `onMooseRelease` is not the name of one of the special event handling methods discussed in the preceding sections. In more advanced programs, however, we will learn that it is sometimes useful to define additional methods that do things other than handle events. The programmer is free to choose names for such methods. `onMooseRelease` would be a legitimate (if strange) name for such a method. That is, a program containing a method with this name has to be treated as a syntactically valid program by any Grace IDE. As a result, your IDE would not recognize this as a typing mistake, but when you ran the program Grace would think you had decided not to associate any instructions with mouse release events. As before, the text “I’m Touched” would never be removed from the canvas.

There are many other examples of logical errors a programmer can make. Even in a simple program like `TouchyWindow`, mistyping screen coordinates can lead to surprises. If you mistyped a `y` coordinate as in

```
textAt (400 @ 500) with ("I'm Touched") on (canvas)
```

the text would be positioned outside the visible region of the program window. It would seem as if it never appeared. If the line

```
canvas.clear
```

had been placed in the `onMousePress` method with the line to construct the message, the message would disappear so quickly that it would never be seen.

Of course, in larger programs the possibilities for making such errors just increases. You will find that careful, patient, thoughtful examination of your code as you write it and after errors are detected is essential.

1.8 Summary

Programming a computer to say “I’m Touched.” is obviously a rather modest achievement. In the process of discussing this simple program, however, we have explored many of the principles and practices that will be developed throughout this book. We have learned that computer programs are just collections of instructions. These instructions, however, are special in that they can be followed mechanically, without understanding their actual purpose. This is the notion of an algorithm, a set of instructions that can be followed to accomplish a task without understanding the task itself. We have seen that programs are produced by devising algorithms and then expressing them in a language that a computer can interpret. We have learned the rudiments of the language we will explore further throughout this text, Grace. We have also explored how we can use a web browser to enter computer programs, translate them into a form the machine can follow, and run them.

Despite our best efforts to explain how the language and development system work, there is nothing that can take the place of actually writing, entering, and running a program. We strongly urge you to do so before proceeding to read the next chapter. We cannot stress enough that you can only learn to program through writing lots of programs. Now is a good time to start.

1.9 Chapter Review Problems

Exercise 1.9.1 Consider the point located at coordinates $(100, 100)$. What are the coordinates of the following points in the Computer Graphics Coordinate System:

- a. 40 pixels **down** and 30 pixels to the **left** of $(100, 100)$
- b. 60 pixels **up** and 10 pixels to the **right** of $(100, 100)$
- c. 35 pixels **up** and 45 pixels to the **left** of $(100, 100)$
- d. 20 pixels **down** and 50 pixels to the **left** of $(100, 100)$
- e. 80 pixels **down** and 15 pixels to the **right** of $(100, 100)$

Exercise 1.9.2 Sketch the picture that the following lines of code would produce. Assume the window is 200 pixels wide and 200 pixels high.

```
framedRectAt (20 @ 20) size (160 @ 160) on (canvas)
lineFrom (20 @ 180) to (100 @ 20) on (canvas)
lineFrom (100 @ 20) to (100 @ 180) on (canvas)
```



```
filledOvalAt (100 @ 100) size (80 @ 80) on (canvas)
lineFrom (180 @ 100) to (100 @ 100) on (canvas)
```

1.10 Programming Problems

Exercise 1.10.1 *Learn how to enter and run a program using the tools available to you by entering the TouchyWindow program discussed in this chapter. Once you are able to enter the program:*

- *Enter smaller or larger values where we had used the numbers 40 and 50 and see how the behavior of the program changes when run.*
- *Interchange the bodies of the two methods so that the construction appears in `onMouseRelease` and the `canvas.clear` is in `onMousePress`. How does the modified program behave?*

Chapter 2

What's in a name?

An important feature of a programming language is that the vocabulary used can be expanded by the programmer. Suppose you want to draw a line that ends at the current position of the mouse. The actual point will not be determined until the program you write is being used. To talk about this position in your program you must introduce a name that will function as a place holder for the information describing the mouse's position. Such names are somewhat like proper names used to refer to the character in a story. You cannot determine their meanings by simply looking them up in a standard dictionary. Instead, the information that enables you to interpret them is part of the story itself. In this chapter, we will continue your introduction to programming in Grace by discussing in more detail how to introduce and use such names in Grace programs. In addition, we will introduce additional details of the primitives used to display graphics.

2.1 Naming and Modifying Objects

Constructions like:

`lineFrom (onePoint) to (anotherPoint) on (canvas)`

provide the means to place a variety of graphic images on a computer screen. Most programs that display graphics, however, do more than just place graphics on the screen. Instead, as they run they modify the appearance of the graphics they have displayed in a variety of ways. Items are moved about the screen, buttons change color when the mouse cursor is pointed at them, text is highlighted, and often items are simply removed from the display. To learn how to produce such behavior in a Grace program, we must learn about operations that change the properties of objects after they have been constructed. These operations are called *mutator methods*, based on the non-biological meaning of the word “mutate”, to change or alter.

2.1.1 Mutator Methods

Just as each class of graphical objects has a specific name that must be used in a construction, each mutator method has a specific name. The names are chosen to suggest the change associated with the method, but there are some subtleties. For example, there are two

mutator methods that can be used to move a graphical object to a new position on the screen. They are named `moveBy` and `moveTo`. The first tells an object to move a certain distance from its current position. The second is used to move an object to a position described by a pair of coordinates regardless of its previous position.

With most mutator methods, including `moveBy` and `moveTo`, the programmer must specify additional pieces of information that determine the details of the operation applied. For example, when you send a request `moveBy` to an object, you need to tell it how far. The syntax used to provide such information is similar to that used to provide extra information in a construction. A comma-separated list of values is placed in parentheses after the method name. Thus, to move an object associated with the identifier `someObj` by 30 pixels to the right and 15 pixels down the screen one would say:

```
someObj.moveBy (30, 15)
```

While the use of a mutator method shares portions of the syntax of a construction, there are major syntactic and conceptual differences between the two. A construction produces a new object. A mutator method is used to modify an already existing object. To make this clear, let us consider a simple example.

Many programs start by displaying an entertaining animation. With our limited knowledge of Grace, we can't yet manage an entertaining animation, but we can, with a bit of help from the program's user, create a very simple animation. In particular, we can write a program that displays a circle near the bottom of the canvas and then moves the circle up a bit each time the mouse is clicked. With a bit of imagination, you can think of the circle as the sun rising at the dawn of a new day.

Without knowing anything about mutator methods, you should be able to imagine the rough outline of such a program. Basically, from the description it is clear that the object inheriting from `graphicApplication` needs to construct a filled oval. It is also clear that the program will need to define an `onMouseClicked` method that uses the `moveBy` mutator method. You don't know enough to write this method yet. The fact that `onMouseClicked` will be used, however, should tell you a bit more about the code in the object. If the user of our program needs to click the mouse to get it to function, then, just as we did in our improved version of `TouchyWindow`, we should include code to display instructions telling the user to do this. So, your first draft of an object expression might look something like:

```
object {  
  inherit graphicApplicationSize (200 @ 200)  
  
  // Circle that represents the sun  
  def sun = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)  
  
  // Place instructions on the screen  
  textAt (20 @ 20) with ("Click the mouse repeatedly") on (canvas)  
  
  startGraphics  
}
```

This code will produce an image like that shown in Figure 2.1. In an effort to make it look like the sun is rising over the horizon, the oval is positioned so that its bottom half extends off the bottom of the window leaving only the top half visible. Of course, it might

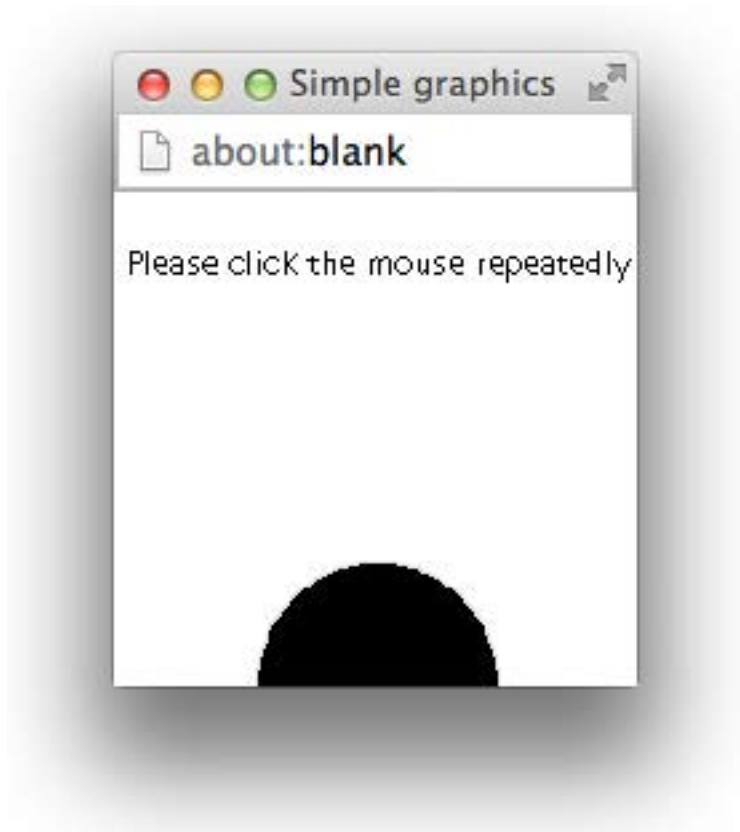


Figure 2.1: An oval rises over the horizon

help your imagination if the “sun” were yellow, but we will have to wait until we learn a bit more Grace before we can fix that.

Now, consider how we would complete the program by writing the `onClick` method. To make an object move directly upwards, we need to use the `move` method specifying 0 as the distance to move horizontally and some negative number for the amount of vertical motion desired since `y` coordinates decrease as we move up the screen. Something like:

```
moveBy (0, -5)
```

might seem appropriate. The problem is that if this is all we say, Grace will not know what to move. In the body of the **object** expression above, we construct two graphical objects, the filled circle and the text displaying the instructions. Since they are both graphical objects, we could move either of them. If all we say is `moveBy`, then Grace has no way of knowing what we want moved.

To avoid ambiguities like this, Grace won't let us simply say `moveBy`. Instead we have to tell a particular object to move. In general, Grace requires us to identify a particular object as the target whenever we wish to use a mutator method. You have already seen an example of the Grace syntax used to provide such information. In our first example program, when we wanted to remove a message from the screen we included a line of the form:

```
canvas.clear
```

where `clear` is a mutator method associated with drawing areas. The word `canvas` is the name Grace gives to the area in which we can draw. Saying `canvas.clear` tells the area in which we can draw that all previous drawings should be erased. When we tell an object to perform a method in this way we say we have *requested* the method. Alternately, we might say that we sent a `clear` message to the `canvas`.

In general, to apply a method to a particular object, Grace expects us to provide a name or some other means of identifying the object followed by a period and the name of the method to be used. So, in order to move the oval created in the body of the **object** expression, we first have to tell Grace to associate a name with the oval.

2.1.2 Definitions

First, we have to choose a name to use. Grace puts a few restrictions on the names we can pick. Names that satisfy these restrictions are called *identifiers*. An identifier must start with a letter. After the first letter, you can use any combination of letters, digits, underscores, or single quotes (`'`). So, we could name our oval something like `sunspot`, `oval2move` or `sun_ra'`. Case is significant, so `sun` is different from `Sun`. An identifier can be as long (or short) as you like, but it must be just one word (i.e. no blanks or punctuation marks are allowed in the middle of an identifier). A common convention used to make up for the inability to separate parts of a name using spaces is to start each part of a name with a capital letter. For example, we might use a name like `ovalToMove`. It is also a convention to use identifiers starting with lower case letters to name variables to help distinguish them from the names of types (which we will talk about later).

A sequence of letters, numbers, single quotes, and underscores can be used as an identifier in a Grace program even if it has no meaning in English. Grace would be perfectly happy if we named our box `e2d_iw0'`. It is much better, however, to choose a name that suggests the

```

dialect "objectdraw"

object {
  inherit graphicApplicationSize (200 @ 200)

  def sun = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)

  textAt (20 @ 20) with ("Click the mouse repeatedly") on (canvas)

  method onMouseClick(point) {
    sun.moveBy (0, -5)
  }

  startGraphics
}

```

Figure 2.2: Declaring sun in the RisingSun program

role of an object. Such names make it much easier for you and others reading your code to understand its meaning. We suggested earlier that you could think of the display produced by the program we are trying to write as an animation of the sun rising. In this case, `sun` would be an excellent name for the oval. We will use this name to complete this example.

As we saw in the last chapter, the syntax of a definition is very simple. For each name you plan to use, you enter the word `def` followed by the name you wish to introduce, an equals sign, and the value of the identifier. So, to define the name `sun`, which we intend to use to refer to a `FilledOval`, we would add the declaration:

```
def sun = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)
```

Now that we have associated a name with the oval, we can tell `sun` to move it by writing `sun.moveBy (0, -5)`

As a result, the contents of the program file for our animation of the sunrise might begin with the code shown in Figure 2.2.

The form and placement of a definition within a program determines where in the program the name can be used. This region is called the *scope* of the name. In particular, we will want to refer to the name `sun` in the body of the `object` expression and in the `onMouseClick` method of the program we are designing. The declaration of names that will be used in several methods should be placed within the braces that surround the body of our object (or class), but outside any of the method bodies. We recommend that these definitions be placed before all the method declarations in order to make them easier to find when reading the rest of the code. Only code within the object we are defining is allowed to refer to this name. Method names, by way of contrast, are generally accessible outside of the class. We will learn later how to modify the visibility of definitions and methods.

It is important to note that we can use previously defined identifiers in initializing new ones. Thus we can write

```

def pi = 3.14159
def radius = 5
def area = pi * radius * radius

```

Defining new values in terms of existing ones is often very helpful because it can help keep values in sync when code is modified. For example, if we decided to change the value of `radius` to 10, the value of `area` would change automatically.

One word of caution is that you can only use the values of identifiers that have already been initialized. As a result, the following code would result in an error message:

```
def pi = 3.14159
def area = pi * radius * radius
def radius = 5
```

One purpose of requiring an identifier to be introduced via a definition is to enable Grace to give you helpful feedback if you make a mistake. Suppose that after deciding to use the name `sun` in our program we made a typing mistake and typed `sin` in one line where we meant to type `sun`. It would be nice if when Grace tried to run this program it could notice such a mistake and provide advice on how to fix the error similar to that provided by a spelling checker. To do this, however, Grace needs the equivalent of a dictionary against which it can check the names used in the program. The definitions a programmer includes in their program constitute this dictionary. If Grace encounters a name that was not declared, it reports it as the equivalent of a spelling mistake.

2.1.3 Variable Declarations

In the previous section we saw how we could associate names to objects via definitions. As we have seen with the `sun` example, we may request mutator methods on these objects, and they may change their properties. For example, when we evaluated `sun.moveBy(0, -5)`, the object associated with `sun` shifted up 5 pixels on the screen. Once a value is associated with a name in a definition, however, that name may not refer to a different object. For example, once the system has processed the definition, we could not associate `sun` with a rectangle or even a different filled oval.

If we want a name that can be associated with different objects over time then we must declare a *variable*. We can see the differences between definitions and variables in everyday conversation. Proper names are usually used as definitions: "Barack Obama" refers to the individual who served as U.S. president from 2008 to 2016. In 2007, the phrase "the president of the U.S." referred to George Bush. In 2009, it referred to Barack Obama.

When we introduce identifiers in a language, we need to decide what kind of identifier they will be. If they will always refer to the same object then we introduce them with a definition. If they sometimes refer to one object, but at others a different one then they will be represented via a variable – that is, what they refer to may vary. A simple variable declaration uses the keyword `var` as follows:

```
var president := "Barack Obama"
```

There are two key differences between a variable declaration and a definition. First, we use the keyword `var` instead of `def`. Second, the value is associated with a variable using `:=`, while the value is associated with a definition using `=`. The `=` should suggest that the name is always identical to the value associated with it, while the variant of `:=` indicates that different values may be associated with the name over the course of the execution of the program.

A variable declared in an object is known as an instance variable. We will see later that we can also declare variables inside methods. It is also possible to declare a variable

without assigning it a value. Of course we will have to assign a value to it before we use it, as otherwise we will get an error.

We will see several examples of variables in programs later in this chapter.

2.1.4 Comments

In the complete version of the program, we introduce one additional and very important feature of Grace, the *comment*. As programs become complex, it can be difficult to understand their operation by just reading the Grace code. It is often useful to annotate this code with English text that explains more about its purpose and organization. In Grace, one can include such comments in the program text itself as long as you follow conventions designed to enable the computer to distinguish the actual instructions it is to follow from the comments. This is done by preceding such comments with a pair of slashes: `//`. Any text that appears on a line after a pair of slashes is treated as a comment by Grace.

The program we are writing seems a good example in which to introduce comments. Although the program is short and simple, it is not clear that someone reading the code would have the imagination to realize that the black circle created by the program was actually intended to reproduce the beauty of a sunrise. The Grace language isn't rich enough to allow one to express non-technical ideas in code, but we can include them in comments. We include some general guidance on using comments to make your programs easier to read and understand in Appendix A.

```
dialect "objectdraw"

// A program that produces an animation of the sun rising.
//The animation is driven by clicking the mouse button.
//The faster the mouse is clicked, the faster the sun will rise.
object {
  inherit graphicApplicationSize (200 @ 200)

  // Circle that represents the sun
  def sun = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)

  // Place instructions on the screen
  textAt (20 @ 20) with ("Click the mouse repeatedly") on (canvas)

  // Move the sun up a bit each time the mouse is clicked
  method onMouseClick (point){
    sun.moveBy (0, -5)
  }

  startGraphics
}
```

Figure 2.3: Commented code for rising sun example

The object definition in Figure 2.3 is preceded by three lines of comments. We add a line of comment for each of the major segments of the program. In this case, the definition of the

```

// pi is the mathematical constant
def pi: Number = 3.14159265359

// a daughter of the author
def daughter: String = "Jahan"

// number of times method square has been called
var count: Number := 0

// return the square of value
method square(value: Number) -> Number {
    count := count + 1
    value * value
}

// Print introduction to toName from fromName
method bestGreetingTo (toName: String) from (fromName: String) -> Done {
    print "Hello there, {toName}!"
    print "My name is {fromName}."
}

```

Figure 2.4: Adding type annotations to Grace code

circle, the display of instructions, and the declaration of the `onMouseClicked` method.

2.1.5 Labeling identifiers with types

Types in programming languages provide information about the kinds of values that can be associated with an identifier. For example, in Grace values like 3, -5, and 3.46 have type which is written `Number`, while values like "hello", "Beau", and "four" have type `String`.

In Grace, associating types with identifiers is optional. However, it is generally considered good programming practice to label identifiers with their types when they are introduced to make it clearer what sort of values they can be associated with. Thus it is a form of documentation like comments. However, the Grace compiler can also use this information to help find mistakes if you use an identifier in a way inconsistent with the type with which it is labeled. That is, we can consider associating identifiers with types as a form of machine-checkable documentation.

We can associate a type with an identifier by placing a colon and the type name immediately after the occurrence of the identifier name when it is introduced. An example of code with type declarations can be found in Figure 2.4.

The two `def` declarations indicate that `pi` has type `Number` and that `daughter` has type `String`. By convention, we write types in Grace as starting with a capital letter. Thus all of the built-in types, like `Number` and `String`, start with a capital letter. When you define your own types, we will expect you to start them with capitals as well.

The variable `count` is also declared to have type `Number`. When that value is updated in method `square`, we do *not* write the type again. It is important that the type be associated with an identifier only when it is first introduced.

The method `square` has a parameter named `value` that is also declared as having type `Number`. The arrow (written `->`) at the end of the first line of method `square` indicates what kind of value is returned by a method. Because `square` ends with `-> Number`, the programmer is telling us that it should always return a value of type `Number`. This is not a surprise because if `value` represents a number then `value * value`, which is what is returned by the method, is also a number.

The declaration of the next method works similarly, but has more pieces. The parameters `toName` and `fromName` both have type `String`. However the return type of the method is `Done`. The type `Done` is used to indicate that the method doesn't actually return a value. Instead it just terminates at the end, in this case after printing out two lines of text.

When your program is executed, Grace will make sure that your identifiers are used in ways consistent with their declarations. For example, when the `defs` are checked, it will make sure that their associated values are appropriate. Because the value associated with `pi` is a number, and the value associated with `daughter` is a string, these definitions will not result in an error.

On the other hand if we were to write

```
def daughter: String = 17
```

then when the program was executed we would receive an error message like

```
TypeError at line 1 of typeTest: expected initial value of def 'daughter'  
to be of type String
```

The types of values associated with variables are checked both in the initial declaration and whenever their value are updated by an assignment. The types of parameters to methods are checked when the methods are requested with actual arguments. Thus if we evaluated `square("hello")`, the system would report that the argument `"hello"` is not of type `Number`. (Of course if the type error was not caught here, it would be discovered when the attempt to multiply `"hello"` by itself was made.) The values returned by methods are also checked against the declared return types, with an error reported if the wrong type of value is returned.

Let's look at one more example – the code from the previous section. In Figure 2.5 we have added type annotations to the code.

This code includes two new types `Point` and `Graphic2D`. Both of these types are associated with the `objectdraw` library and are available in any program using the `objectdraw` dialect. The type `Point` represents the coordinates of a point on the screen. Thus the expression `50 @ 150` represents a value of type `Point`. All of our event-handling methods, like the `onMouseClicked` method in the figure, take a parameter of type `Point`, which represents the coordinates of the mouse at the time the event was generated.

The two dimensional geometric objects we have been using, e.g., the filled and framed ovals and rectangles, have type `Graphic2D`. They all have the same type because they all respond to the same methods. Other types from `objectdraw` that we will discuss later include `Text` for objects displaying text on the canvas, and `Line` for lines on the canvas.

Now that you know how to write types in your code, let's talk in more detail about why adding the type information to declaration is helpful.

1. Type information serves as important documentation for the reader of your program. While a reader of the code in the figure won't have any trouble figuring out that the value

```

dialect "objectdraw"

// A program that produces an animation of the sun rising.
//The animation is driven by clicking the mouse button.
//The faster the mouse is clicked, the faster the sun will rise.
object {
  inherit graphicApplicationSize (200 @ 200)

  // Circle that represents the sun
  def sun: Graphic2D = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)

  // Place instructions on the screen
  textAt (20 @ 20) with ("Click the mouse repeatedly") on (canvas)

  // Move the sun up a bit each time the mouse is clicked
  method onMouseClick (point: Point) -> Done {
    sun.moveBy (0, -5)
  }

  startGraphics
}

```

Figure 2.5: Commented code for rising sun example

associated with `pi` is a `Number` and `daughter` is a `String`, programs typically include much more complicated definitions, and the type information can be very helpful.

With methods the type annotations are even more helpful. When the reader sees a method header they will typically want to be able to understand what the method does without having to look at the details of the body of the method. Again, the methods illustrated in the examples in this section are simple enough that it is pretty easy to see what is happening. However, in general, that is not the case. We will also find it very helpful to have type annotations associated with method parameters so that we know the kind of values to provide them at run-time.

2. Type information is a form of machine-readable and machine-checkable documentation. While the discussion above focused on documentation for a human, the fact that it can be checked by the computer means that the reader can have some confidence that it is accurate (at least if the code runs without error). That is not the case with comments, which can get woefully out of date if the programmer doesn't modify them whenever the code is updated.

From now on we will make it a practice to label identifiers with their types if there is a built-in type that corresponds to the kind of values associated with them. Later we will see that we can define our own types when we start constructing new forms of objects.

2.1.6 Additional Mutator Methods

Let's go back and talk about more mutator methods available in Grace. There are several other operations that can be applied to graphical objects once we have the ability to associate names with the objects. For example, as we mentioned earlier, there is a mutator method named `moveTo` which moves an object to a specific point on the screen.

Many objects also have mutator methods that are designed to look like assignment statements. As we will see below, there is a method named `visible:=` that can be used to temporarily remove or add a graphical item from or to the screen. We can use these methods to extend the behavior of our `RisingSun` program.

First, the version of the program shown above becomes totally uninteresting after the mouse has been clicked often enough to push the filled oval off the top of the canvas. Once this happens, additional mouse clicks have no visible effect. It would be nice if there was a way to tell the program to restart by placing the sun back at the bottom of the canvas. Second, as soon as the user starts to click the mouse, the instructions asking the user to click become superfluous. Worse yet, at some point, the rising sun will bump into the instructions. It would be nice to remove the instructions from the display temporarily and then restore them when the program is reset.

All that we need to do to permit the resetting of the sun is to add the following definition of the `onMouseExit` method to our `RisingSun` class.

```
method onMouseExit (point: Point) -> Done {  
    sun.moveTo (startSun)  
}
```

With this addition, the user can reset the program by simply moving the mouse out of the program's canvas. When this happens, the body of the `onMouseExit` method will tell Grace to move the sun oval back to its initial position.

Making the instructions disappear and then reappear is a bit more work. In order to apply mutator methods to the instructions, we will have to tell Grace to associate a name with the text item created to display the instructions. As we did to define the name `sun`, we will have to both declare the name we wish to use and associate it with the creation of the text.

An obvious name for this object is "instructions." If this is our choice, then we would need to tell Grace that we planned to use this name by adding a declaration of the form:

```
def instructions: Text = textAt (startInstructions) with ("Click the mouse repeatedly")  
    on (canvas)
```

to the body of the object expression.

As in this example, it is sometimes helpful to split a long command into several lines. It can make your code much easier to read and understand. Using multiple lines for one command like this is perfectly acceptable in Grace. You can split an instruction between two lines at any point where you could type a space except within quoted text. Be careful, however, to make sure the continuation line is further indented than the beginning of the statement. You will also find that the use of indentation and blank lines can make groups of related commands stand out to the reader. These issues are discussed further in Appendix A.

The mutator method named `visible:=` provides the means to temporarily remove or add a bit of graphics from the display. To make the text disappear when the mouse is clicked, we

```

dialect "objectdraw"

// A program that produces an animation of the sun rising.
// The animation is driven by clicking the mouse button.
// The faster the mouse is clicked, the faster the sun will rise.

object {
  inherit graphicApplicationSize (200 @ 200)

  // starting position of the sun
  def startSun: Point = 50 @ 150

  // Circle that represents the sun
  def sun: Graphic2D = filledOvalAt (startSun) size (100 @ 100) on (canvas)

  // instructions for the program
  def instructions: Text = textAt (20 @ 20) with ("Click the mouse repeatedly") on (canvas)

  // Move the sun up a bit each time the mouse is clicked
  method onMouseClick (point: Point) -> Done {
    sun.moveBy (0, -5)
    instructions.visible := false
  }

  // Move the sun back to its starting position and redisplay
  method onMouseExit (point: Point) -> Done {
    sun.moveTo (startSun)
    instructions.visible := true
  }

  startGraphics
}

```

Figure 2.6: Rising sun program with reset feature

would include an instruction of the form:

```
instructions.visible := false
```

in the `onMouseClicked` method. Each time the mouse is clicked, the object associated with instructions will become invisible. Of course, once they are invisible, telling them to become invisible again has no effect.

This mutator method is intentionally designed to look like a variable assignment statement, because it changes an attribute (i.e., a quality or characteristic) of the object. The main difference is that the left side of the statement includes a “.”.

When the program is reset, we want the instructions to reappear. To do this, we need to include an instruction of the form

```
instructions.visible := true
```

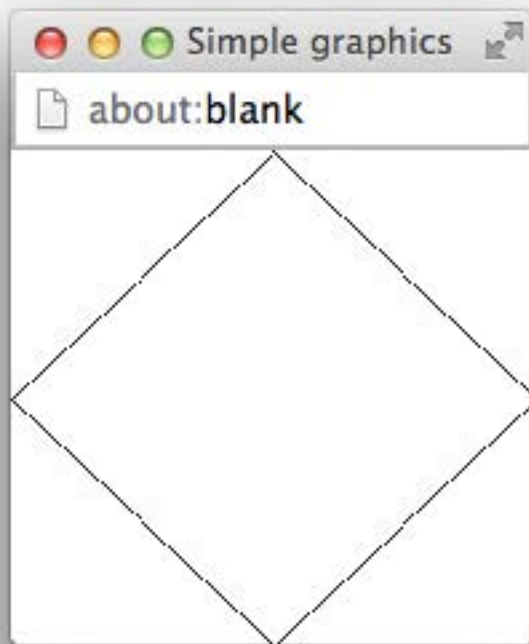
in the `onMouseExit` method. The complete text of this revised program is shown in Figure 2.6.

The `visible:=` method should only be used to hide an object when it is being removed from

the canvas *temporarily*. If an object is being removed *permanently*, that is, you know that your program will never make it visible again, there is another method named `removeFromCanvas` that is more appropriate. The `removeFromCanvas` method irreversibly removes a graphical object from the display. There is no way to put an object that has been removed in this way back on the display. When an object is made invisible, the system must save information needed to display the object in case it is made visible again. This consumes space in the computer's memory. Using `removeFromCanvas` instead allows the system to totally remove information about the object from the computer's memory.

2.1.7 Exercises

Below are several exercises in which we will ask you to consider how one could write a program that displayed a diamond that appeared to grow a bit each time the mouse was clicked. The following constructions will produce a drawing of the initial diamond shape desired if executed in a program whose canvas is 200 by 200 pixels.

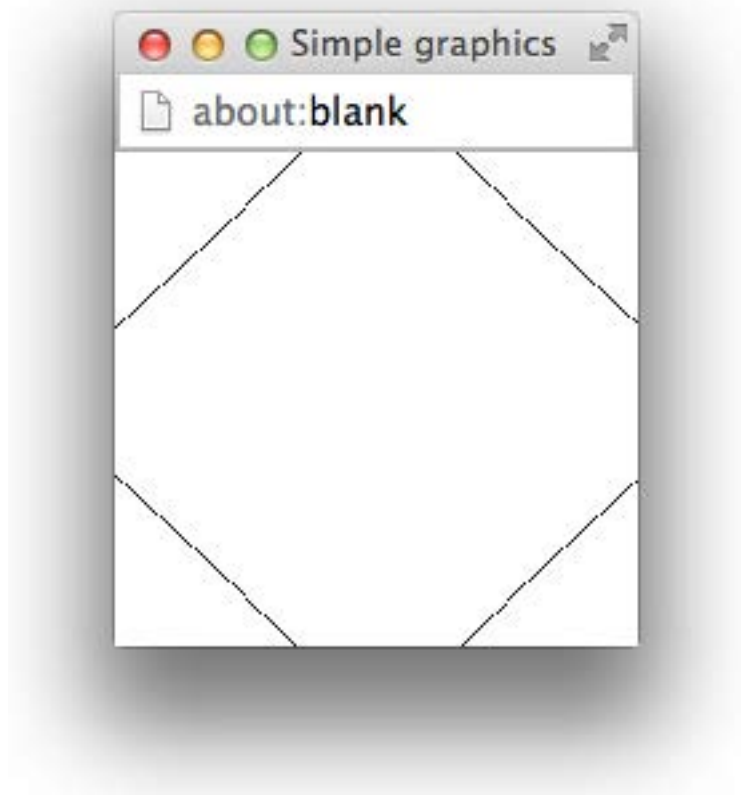


```
def left: Point = 0 @ 100
def right: Point = 200 @ 100
def top: Point = 100 @ 0
def bottom: Point = 100 @ 200
```

```
lineFrom (top) to (right) on (canvas)
lineFrom (top) to (left) on (canvas)
lineFrom (bottom) to (right) on (canvas)
```

lineFrom (bottom) to (left) on (canvas)

The complete program we have in mind won't actually make the diamond grow. Instead, all it will do is move each of the four lines drawn by the constructions a bit closer to the corner closest to the line each time the mouse is clicked. For example, each time the mouse is clicked, the line in the upper left corner of the window will be moved one pixel to the left and one pixel up so that it ends up closer to the upper left corner of the window. The line that starts in the upper right quarter of the window, on the other hand, will be moved one pixel to the right and one pixel up with each click of the mouse. If this is done, after a number of clicks, the display will look like picture shown below. The four lines won't be any longer than they were at the start, but they will appear to be part of a bigger diamond than was originally drawn, a diamond that is too big to fit in the window.



Exercise 2.1.1 Before we can move one of these lines or any other graphical object, we must first associate a name with the object. A list of possible names that might be used to refer to the line of the diamond shape drawn by the construction

lineFrom (bottom) to (right) on (canvas)

are shown below. For each name, indicate whether it would be:

- **invalid** according to Grace's rules for forming names,
- **inappropriate** as a name for this particular line because it would not help a person working with the program remember the purpose of the name or it does not conform to Grace's naming conventions, or

- **appropriate** as a name for this particular line.

In each case, briefly explain your answer.

<i>3rdLine</i>	<i>thirdLine</i>	<i>line3</i>
<i>Leftlower</i>	<i>lower right line</i>	<i>lowerLeft</i>
<i>southEast</i>	<i>S.E.</i>	<i>south-east</i>

Exercise 2.1.2 Suppose that you selected the instance variable names *leftToTop*, *topToRight*, *rightToBottom*, and *bottomToLeft* to describe the four lines that form the diamond. Show the declarations required to introduce these names in a Grace program and associate the lines with them.

Exercise 2.1.3 Assuming that the names *leftToTop*, *topToRight*, *rightToBottom*, and *bottomToLeft* had been declared appropriately and associated with the lines of the diamond using assignment statements, show the statements required to invoke the *move* method on each line to move the line closer to the nearest corner of the program's window. Each line should be moved one pixel horizontally and one pixel vertically. In which method should these commands be placed if you want the lines to move each time the mouse is clicked?

Exercise 2.1.4 Assuming that the names *leftToTop*, *topToRight*, *rightToBottom*, and *bottomToLeft* had been declared appropriately and associated with the lines of the diamond using assignment statements, describe the effect the following statements would have on the lines each time they were executed. In addition, sketch the contents of the window after these lines were executed 100 times.

```

leftToTop.moveBy (0 1)
topToRight.moveBy (0, 1)
rightToBottom.moveBy (0, -1)
bottomToLeft.moveBy (0, -1)

```

2.2 Non-graphical Classes of Objects

We have seen how to construct graphical objects of several varieties, associate names with them and apply methods to these objects. All of the objects we have worked with, however, have shared the property that they are graphical in nature. When we create any of these objects, they actually appear somewhere on the computer's screen. This is not a required property of objects that can be manipulated by a Grace program. In this section we will introduce two classes of objects that are related to producing graphics but do not correspond to particular shapes that appear on your screen.

2.2.1 Colors

So far, all the graphics we have drawn on the screen have appeared in black, as black is the default color in which graphics are drawn. To add a little variety to the display, we can change the color of any graphical object in the object draw library by a statement of the form

```
anObject.color := newColor
```

where `anObject` is the graphical object and `newColor` is the new color. Just as we saw earlier with `visible:=`, `color:=` is actually a method request with parameter `newColor`.

It would certainly be an improvement to make the sun displayed by our `RisingSun` program appear yellow or any other more “sun-like” color than black. It is quite simple to make this change. All we have to do is add a line to reset the color to our object. The revised program would look like:

```
dialect "objectdraw"
```

```
// A program that produces an animation of the sun rising.
//The animation is driven by clicking the mouse button.
//The faster the mouse is clicked, the faster the sun will rise.
object {
  inherit graphicApplicationSize (200 @ 200)

  // Circle that represents the sun
  def sun = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)
  sun.color := colorGen.yellow

  // Place instructions on the screen
  textAt (20 @ 20) with ("Click the mouse repeatedly") on (canvas)

  // Move the sun up a bit each time the mouse is clicked
  method onMouseClick (point) {
    sun.moveBy (0, -5)
  }

  startGraphics
}
```

When resetting the color, we must specify a color value. To assist with this we have a `colorGen` object that provides many useful methods to return colors. For example, the simplest way to specify a color is to ask the `colorGen` object for one of Grace’s built-in color names. We chose `colorGen.yellow` for our sun, but if we wanted a more dramatic sunrise we could have instead used `colorGen.red`. The color object provides access to all the basic colors including `colorGen.white`, `colorGen.black`, `colorGen.green`, `colorGen.red`, `colorGen.gray`, `colorGen.blue`, `colorGen.cyan`, `colorGen.magenta`, and `colorGen.neutral`. Of course, there are too many shades of each color to assign a name to every one. Accordingly, the `colorGen` object provides a method, `r()g()b()` for creating new colors.

When we want to construct a new color for use in a program, we must provide a numerical description of the color as parameters to the construction. Grace uses a system that is fairly common for specifying colors on computer systems. Each color is described as a mixture of the three primary colors: red, green and blue.¹ The mixture desired is specified by giving three numbers, each of which specifies how much of a particular color should be included in the mixture. Each of the numbers can range from 0 (“use none of this particular primary color”) to 255 (“use as much of this color as possible”). If the numbers for the red, green, and

¹If you thought the primary colors were red, yellow and blue, you aren’t confused. Those are the primary colors when mixing materials that absorb light (like paint). When mixing light itself (as in flashlight beams or the light given off by the phosphors on a computer screen), red, blue and green act as the primary colors. Mixing red and blue lights produces purple. Mixing red and green produces yellow. Mixing all three primary colors together produces white.

blue components are all 0's then the color is black. If all are 255 then the color is white. If the green component is 255 and the other two are 0 then, not surprisingly, we get green. If both the red and blue components are 255, while the green is 0 then we get a shade of purple. So if you want to make the sun purple you could construct the desired color by saying:

```
colorGen.r (255) g (0) b (255)
```

Colors are objects. Therefore, just as you can associate names with objects such as FilledOvals, you can associate names with colors. If you wanted to use the color purple in a program you might first define the identifier purple as:

```
def purple = colorGen.r (255) g (0) b (255)
```

You could then make the sun purple by replacing the statement used to make the sun yellow by:

```
sun.color := purple
```

It is worth noting that it is not actually necessary to introduce a new name to use a color created using a construction. When we update the color of an object, we have to provide the color we wish to use as to the right of :=, but we can describe the desired color in several ways. In particular, we could make the sun purple by simply writing

```
sun.color := colorGen.r (255) g (0) b (255)
```

The construction describes the color just as well as the name purple from Grace's point of view.

In general, wherever Grace allows us to identify an object or value by name, it will accept any other phrase that describes the equivalent object.

Just as we can build a color using numbers representing their red, green, and blue components, we can send method requests to colors asking for those same numeric components. The names of the methods are red, green, and blue. Thus if someColor is an identifier associated with a color, somecolorGen.red will return a number corresponding to the red component.

If we wish to print out information about a string, we can also ask it for a string representation of the color using the method asString. For example, if purple is as defined above then purple.asString returns the string "rgb(255,0,255)", which could be used in a print statement.

We will find it useful to include a method asString for most of our objects so that we can retrieve and print useful information about them. For example, if we execute the following code snippet:

```
def myRect: Graphic2D = framedRectAt (0 @ 0) size (50 @ 100) on (canvas)
print (myRect.asString)
```

then the system will print out FramedRect at (0,0) with height 50, width 100, and color rgb(0,0,0).

In fact, because asString is provided for all objects, you may leave off the asString in a print statement and the system will automatically apply the method. Thus you may simply write print(myRect) and the exact same string will be printed.

```

type Color = {
  red  -> Number    // The red component of the color.
  green -> Number   // The green component of the color.
  blue -> Number    // The blue component of the color.
}

type ColorFactory = {
  // create a new color with components (r',g',b')
  r (r': Number) g (g': Number) b (b': Number) -> Color

  // return a new random color
  random -> Color

  // return the following predefined colors
  white -> Color
  black -> Color
  green -> Color
  red   -> Color
  gray  -> Color
  blue  -> Color
  cyan  -> Color
  magenta -> Color
  yellow -> Color
  neutral -> Color
}

```

Figure 2.7: The types Color and ColorFactory

Aside from selecting a pre-named color or using the `r()g()b()` method, you can also request the `random` method on `colorGen` to get a new randomly chosen color.

Unlike our graphical objects, colors are immutable. That is, there are no methods that modify its state. While we can use the mutator method `moveBy` on a framed rectangle in order to change its position, there are no mutator methods for colors. If you want a color that is slightly different from an existing color, you must create a new color rather than modifying the original.

For example, suppose you wanted a color that was a bit lighter than the existing color associated with the identifier `someColor`. Then we could create a new color whose components are each 5 units greater than that by writing

```
colorGen.r (someColor.red + 5) g (someColor.green + 5) b (someColor.blue + 5)
```

The object `color` described in this section is different from most of the objects we have seen so far. It is an object with several methods that are used to create objects, all of the same type. Objects like `color` are sometimes known as “factory” objects. In this case `color` is a factory object that generates objects of type `Color`.

The type `Color` and the type `ColorFactory` are described in Figure 2.7. The type of `colorGen` is `ColorFactory`, while the type of the objects returned from method requests on `colorGen`, e.g. `colorGen.blue`, `colorGen.r (100) g (50) b (200)`, and `colorGen.random`, have type `Color`.

Exercise 2.2.1 Write a method that creates a red oval when the mouse is clicked. The oval

```
dialect "objectdraw"
```

```
object {  
  inherit graphicApplicationSize (400 @ 400)  
  var horizontalCorner: Point := 0 @ 0  
  var verticalCorner: Point := 0 @ 0  
  
  method onMouseClick (point: Point) -> Done {  
    filledRectAt (verticalCorner) size (5 @ 200) on (canvas )  
    filledRectAt (horizontalCorner) size ( 200 @ 5) on (canvas )  
    verticalCorner := verticalCorner + (10 @ 0)  
    horizontalCorner := horizontalCorner + (0 @ 10)  
  }  
  
  startGraphics  
}
```

Figure 2.8: An application of the + operation on points

should be 100 pixels in width and 75 pixels in height and should appear 50 pixels down from the top of the canvas and 50 pixels in from the left of the canvas.

2.2.2 Creating Points

As we have seen above, we can use a triple of numbers to construct an object representing a color. Earlier we saw that we could use pairs of numbers to create points in our drawing window by constructing objects using `x @ y`.

Like the color objects described above, points are immutable. There are no mutator methods that they respond to. The methods that are associated with points are `x`, `y`, `+`, `-`, and `distanceTo`.

Not surprisingly, the first two return the `x` and `y` coordinates of the point. Writing `startingPoint + otherPoint`, where both are points returns a new point whose `x` and `y` coordinates are calculated by summing the `x` and `y` coordinates, respectively, of `startingPoint` and `otherPoint`. Similarly the method `-` returns the point obtained by subtracting the corresponding `x` and `y` coordinates. The method request `startingPoint.distanceTo (endPoint)` returns the distance from `startingPoint` to `endPoint`.

To clarify how `+` works, consider the sample program shown in Figure 2.8. Each time the mouse is clicked, this program will draw what appear to be a pair of thick, perpendicular lines, though in reality they are very thin rectangles. The first lines drawn will intersect at the upper left corner of the window. With each click, the lines drawn will be placed to the left and below the preceding lines so that the window is eventually filled with a grid pattern.

The program includes two variables named `verticalCorner` and `horizontalCorner`. Each of these names describes the point at which one of a pair of filled rectangles will be drawn when the user clicks the mouse.

When the object expression is evaluated the computer creates two objects that both describe the point at the origin of the coordinate system, the upper left corner of the canvas. One of these objects is associated with the name `verticalCorner` and the other with the name



Figure 2.9: Display after one click

`horizontalCorner`. Although they both refer to the origin initially, two distinct variables are needed because they will be updated to describe different points using the `+` operation in other parts of the program.

The names assigned to these two points reflect the way they are used in the program's other method, `onMouseClicked`. Each time the mouse is clicked, this method creates two long, thin rectangles on the screen. The rectangle created by the first construction in `onMouseClicked` is a long vertical rectangle. The position of its upper left corner is determined by `verticalCorner`. The other rectangle is a long horizontal rectangle and its position is determined by `horizontalCorner`.

Since both of the points initially describe the upper left corner of the canvas, the first time the mouse is clicked, the rectangles created by the execution of the first two lines of `onMouseClicked` will produce a drawing like that shown in Figure 2.9.

The last two commands in `onMouseClicked` tell the `Point` objects named `verticalCorner` and `horizontalCorner` to create new points using `+` so that they describe new positions on the screen. `verticalCorner + (10 @ 0)` returns the position 10 pixels to the right of its initial position. That new point is then associated with `verticalCorner` by the assignment statement with `verticalCorner` on the left.

The expression `horizontalCorner + (0 @ 10)` provides a point 10 pixels down from `horizontalCorner`. After this value has been calculated, `horizontalCorner` is updated to this new

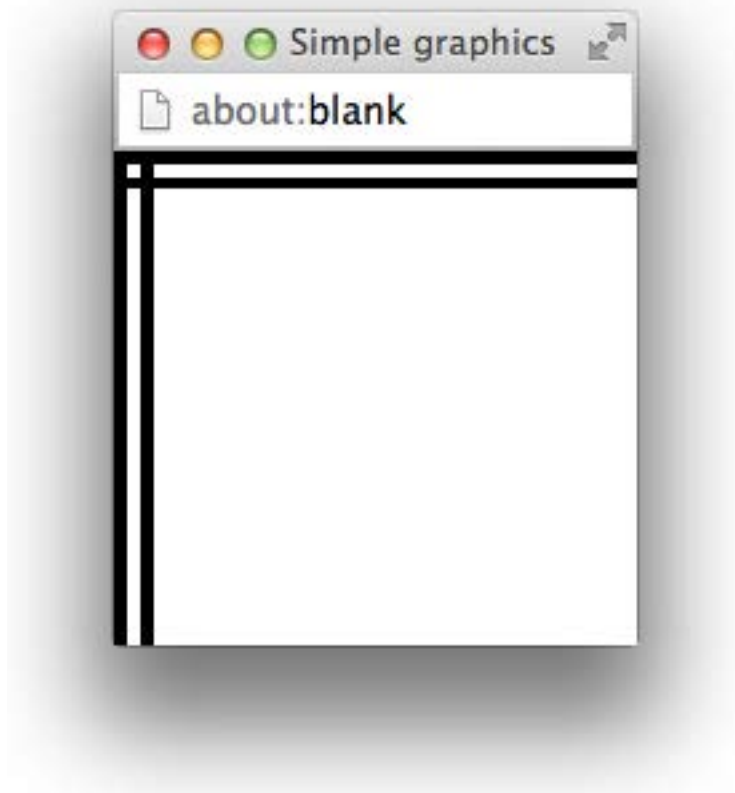


Figure 2.10: Display after second click

point.

When these assignment statements are completed, nothing changes on the screen. The program's window will still appear as shown in Figure 2.9 after they have been performed. However, the variables will refer to the new positions

The next time the mouse is clicked the two constructions at the beginning of `onMouseClicked` are performed based on the new points associated with the two variables. Accordingly, the vertical rectangle created will appear a bit farther to the right and the horizontal rectangle will appear a bit farther down the screen as shown in Figure 2.10. After these rectangles are created, the last two lines of the method will again create a new position farther to the right and down the screen so that the rectangles produced by the next click will appear at the correct points.

This process will be repeated each time the mouse is clicked. After about 15 additional clicks, the window will be nearly filled with a grid of lines as shown in Figure 2.11. Just a few more clicks will complete the process of filling the screen.

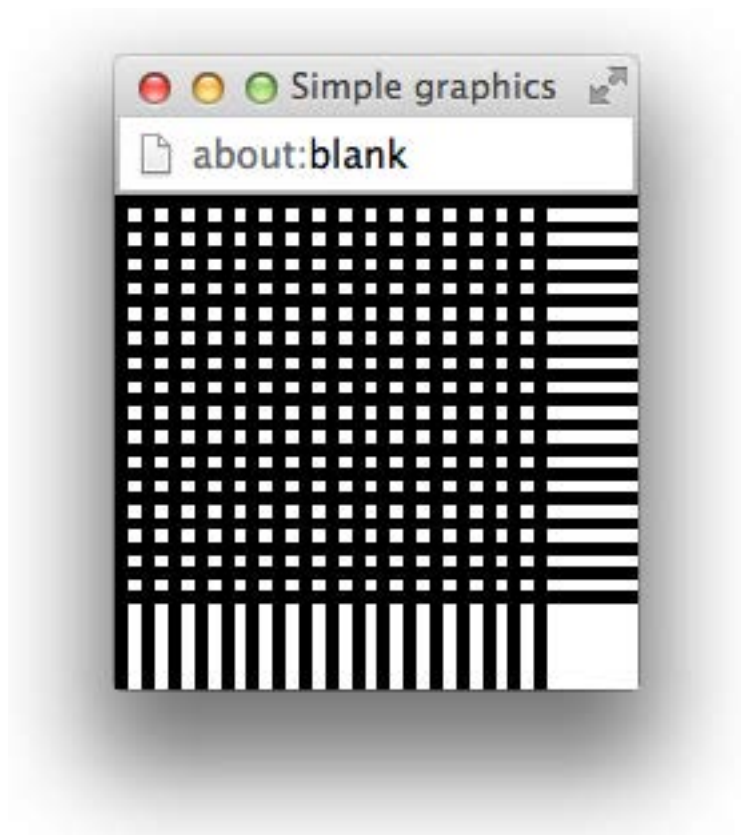


Figure 2.11: Display after many clicks

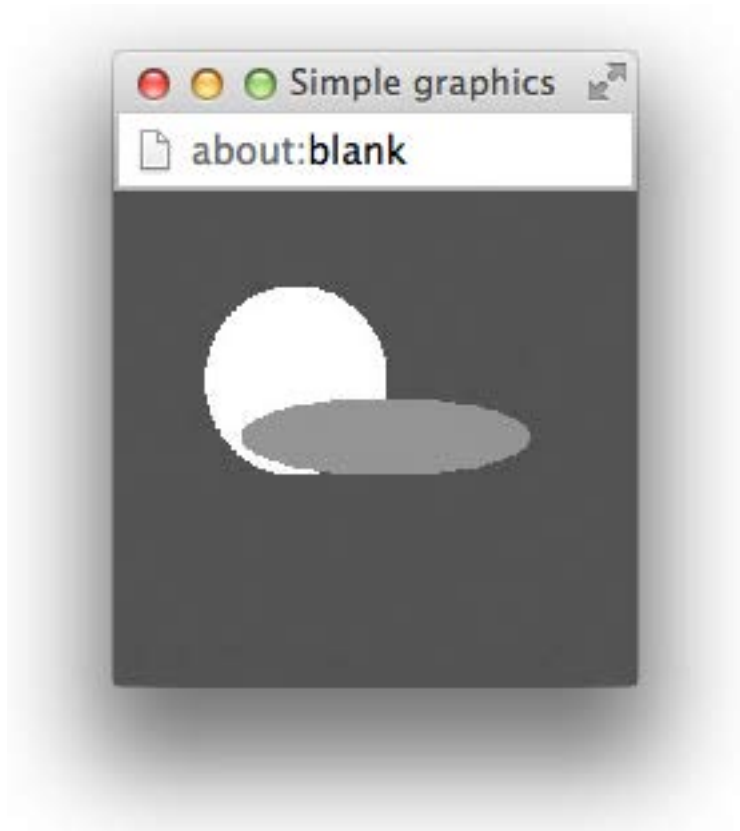


Figure 2.12: Today's forecast: Partly cloudy

2.3 Layering on the Canvas

Now that we can set the color of a graphical object, we could actually make a much prettier picture for our rising sun program. We will add a blue sky and some white clouds to go with the sun. The picture we have in mind is shown in Figure 2.12.

While we could just create the objects for the sky, sun, and cloud, we will need to associate them with names so that we can set the colors:

```
def sky: Graphic2D = filledRectAt (0 @ 0) size (300 @ 300) on (canvas)
def sun: Graphic2D = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)
def cloud: Graphic2D = filledOvalAt (70 @ 110) size (160 @ 40) on (canvas)
```

We then set their colors appropriately using their names:

```
sky.color := colorGen.blue
sun.color := colorGen.yellow
cloud.color := colorGen.white
```

Suppose we changed this code by reordering the definitions so that the sun is created after the cloud as in:

```
def sky = filledRectAt (0 @ 0) size (300 @ 300) on (canvas)
def cloud = filledOvalAt (startCloud) size (160 @ 40) on (canvas)
def sun = filledOvalAt (50 @ 150) size (100 @ 100) on (canvas)
```

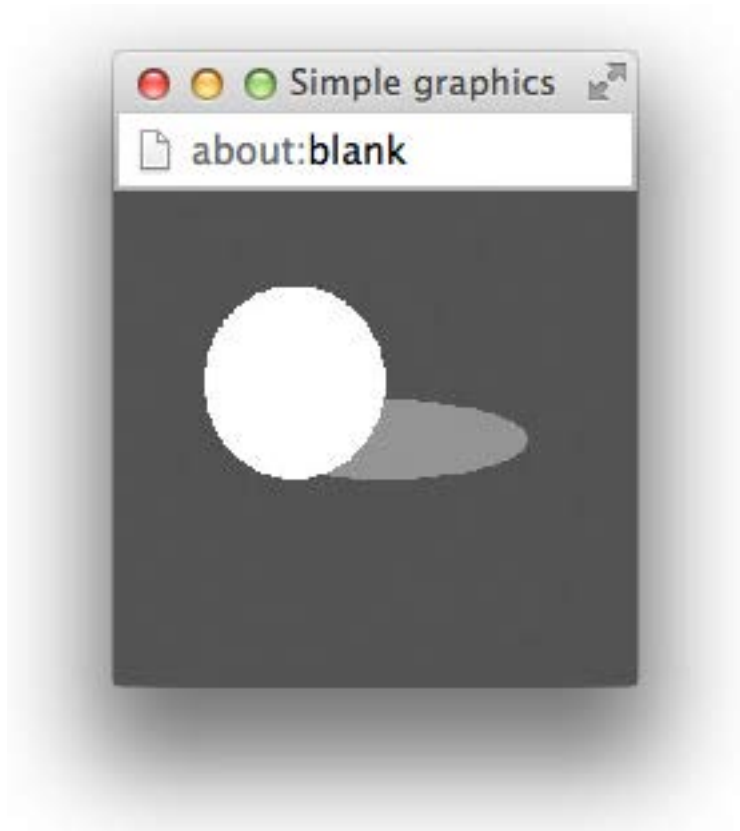


Figure 2.13: Tonight's forecast: Partly sunny?

This will change the picture drawn so that it resembles the drawing shown in Figure 2.13. The sun and the cloud still overlap, but now part of the cloud is hidden behind the sun rather than the other way around. (We know, this can never happen in real life, but let's pretend!)

The canvas views the collection of objects it has been asked to display as a series of layers. When a new object is created it is placed in a layer above all the older objects on the canvas. When two objects overlap, the object on the lower level will be partially or completely hidden by the object on the upper level. In the code to produce Figure 2.12, the sun was created before the cloud and therefore was drawn as if it was underneath the cloud. In the revised code, the moon was created last and therefore is drawn as if it is above the cloud.

None of the methods we have considered so far changes this layering. Changing an object's color or moving it will not change the way in which it is drawn when it overlaps with other objects. If we included code to move the sun as in our `RisingSun` program, the sun would move vertically on the screen, but it would still remain on a layer below the cloud. It would therefore appear to slide upward while remaining behind the cloud. Even executing the `visible:=` method preserves this ordering. If we execute the `visible:=` method of an object to make it visible, it will reappear underneath the same objects that have been above it before it was hidden. In particular, it does not appear at the top level as if it had just been created.

There are, however, several methods that are provided to enable a programmer to rearrange

```

dialect "objectdraw"

// program that displays the word "pressed" wherever
// the mouse is pressed
object {
  inherit graphicApplicationSize (400 @ 400)

  // When the user presses mouse, write "Pressed" where mouse was pressed.
  method onMousePress (point: Point) -> Done {
    textAt (point) with ("Pressed") on (canvas)
  }

  // create window and start graphics
  startGraphics
}

```

Figure 2.14: A Program that Uses the Mouse Location

the layering of objects on the screen. The methods `sendForward` and `sendBackward` move an object up or down one layer. The methods `sendToFront` and `sendToBack` move an object to the top or bottom of all the layers drawn. Thus, if we had created the picture shown in Figure 2.12 and then executed either the command

```
cloud.sendBackward
```

or the command

```
sun.sendToFront
```

the picture displayed would change to look like Figure 2.13.

2.4 Accessing the Mouse Position

In the header of every mouse event handling method we have included the identifier `point` in parentheses after the name of the method. Now that we have explained what a point is, we can explain the purpose of this phrase. It provides a means by which we can refer to the point at which the mouse cursor was located when the event handled by a method occurred. Basically, within the body of a mouse event handling method that includes the identifier `point` we can use the name `point` to refer to place in the canvas that the mouse was positioned.

As a simple example, we can write a variant of our very first example program “Touchy-Window”. This new program will display a bit of text on the screen when the mouse is pressed, just like `TouchyWindow`, but:

1. it will display the word “Pressed” instead of the phrase “I’m Touched.”,
2. it will place the word where the mouse was clicked instead of in the center of the canvas, and finally
3. it will not erase the canvas each time the mouse is released

The code for this new example is shown in Figure 2.14. Note that the name `point` is used in

```

dialect "objectdraw"

// program that displays the word "pressed" wherever
// the mouse is pressed
object {
  inherit graphicApplicationSize (400 @ 400)

  // When the user presses mouse, write ``Pressed''
  // on canvas where the mouse was pressed
  method onMousePress (mousePosition: Point) -> Done {
    textAt (mousePosition) with ("Pressed") on (canvas)
  }

  // create window and start graphics
  startGraphics
}

```

Figure 2.15: Using a different parameter name

the construction that places the text item “Pressed” on the screen. Because `point` is included in the method’s header, Grace knows that we want the computer to make this name refer to the position on the canvas where the mouse was clicked. Therefore Grace will place the word “Pressed” wherever the mouse is pressed.

An identifier that is enclosed by parentheses in a method header is known as a *formal parameter* or just a *parameter*. As in variable declarations and definitions, we are free to use any name we want when we declare a formal parameter. There is nothing special about the name `point` (except that we have used it in all our examples so far). We can choose any name we want for the mouse position as long as we place the name in parentheses in the header of the method. For example, the program shown in Figure 2.15, which uses the name `mousePosition` as its formal parameter instead of `point`, will behave exactly like the version that used the name `point`.

2.5 Sharing Parameter Information between Methods

Another important aspect of the behavior of formal parameter names like `mousePosition` and `point` is that each formal parameter name is meaningful only within the method whose header contains its declaration. To illustrate this, consider the program shown in Figure 2.16. This program displays the word “Pressed” at the current mouse position each time the mouse button is pushed, and it displays the word “Released” at the current mouse position each time the mouse is released. The `onMousePress` method in this example is identical to the corresponding method from the `Pressed` example except for the name chosen for its formal parameter. The `onMouseRelease` method is also quite similar to the earlier program’s `onMousePress`.

Exercise 2.5.1 *When the above program is executing, if the user clicks the mouse (a quick press and release without moving the mouse), then the words “Pressed” and “Released” will be written on top of each other. What happens if the user presses the mouse, drags it to*

```

dialect "objectdraw"

// program that displays the words "pressed" and
// "released" when the mouse is pressed or released
object {
  inherit graphicApplicationSize (400 @ 400)

  // When the user presses mouse, write "Pressed"
  // on canvas at current mouse position
  method onMousePress (pressPosition: Point) -> Done {
    textAt (pressPosition) with ("Pressed") on (canvas)
  }

  // When the user releases mouse, write "Released"
  // on canvas at current mouse position
  method onMouseRelease(releasePoint: Point) -> Done {
    textAt (releasePoint) with ("Released") on (canvas)
  }

  // create window and start graphics
  startGraphics
}

```

Figure 2.16: A program to record mouse button changes

somewhere else on the canvas and only then releases the mouse?

Suppose now that we wanted to write another program that would display the words “Pressed” and “Released” just as the `UpsAndDowns` example but would also draw a line connecting the point where the mouse button was pressed to the point where the mouse button was released. A snapshot of what the window of such a program would look like right after the mouse was pressed, dragged across the screen, and then released is shown in Figure 2.17.

Given that the `UpsAndDowns` program has names that refer to both the point where the mouse button was pressed and the point where the mouse button was released, it might seem quite easy to modify this program to add the desired line drawing feature. In particular, it probably seems that we could simply add a construction of the form:

```
lineFrom (pressPosition) to (releasePoint) on (canvas)
```

to the program’s `onMouseRelease` method.

THIS WILL NOT WORK!

Because `pressPosition` is declared as a formal parameter in `onMousePress`, Grace will not allow the programmer to refer to it in any other method. Grace will treat the use of this name in `onMouseRelease` as an error and refuse to run the program. In general, parameter names can not be used to share information between two different methods.

If we want to share information about the mouse position between two event handling methods, we must use formal parameters and instance variables together. We have already seen that instance variables and definitions can be used to provide information to methods. In the `RisingSun` example, evaluating the object created the oval that was then associated with

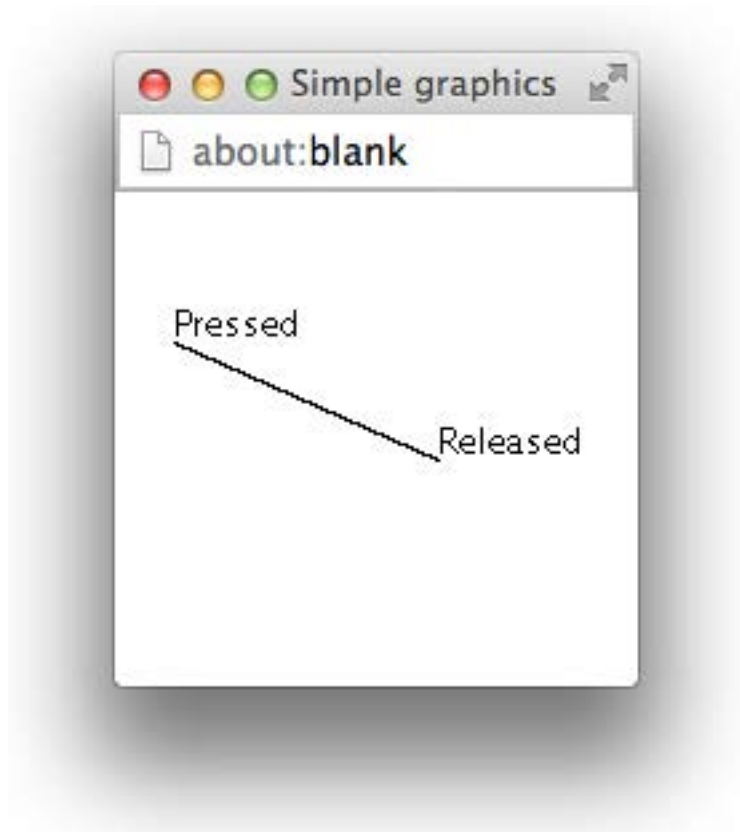


Figure 2.17: Connecting the ends of a mouse motion

the identifier `sun`. The method `onMousePress` later moved the oval by sending a method request to the object associated with `sun`.

In order to write a program to draw a line between the points where the mouse was pressed and released, we will have to associate an instance variable name with the position where the mouse was pressed. This variable will then make it possible for `onMousePress` to share the needed information with `onMouseRelease`. We will choose `firstPoint` as the name for this variable.

The completed program will look like the code shown in Figure 2.18.

```
dialect "objectdraw"

// program that displays the words "pressed" and
// "released" when the mouse is pressed or released
// It also draws a line from the pressed to the
// released position.
object {
  inherit graphicApplicationSize (400 @ 400)

  // Where mouse was pressed
  var firstPoint: Point

  // When the user presses mouse, write "Pressed"
  // on canvas at current mouse position
  method onMousePress (pressPosition: Point) -> Done {
    textAt (pressPosition) with ("Pressed") on (canvas)
    firstPoint := pressPosition
  }

  // When the user releases mouse, write "Released"
  // on canvas at current mouse position
  method onMouseRelease (releasePoint: Point) -> Done {
    textAt (releasePoint) with ("Released") on (canvas)
    lineFrom (firstPoint) to (releasePoint) on (canvas)
  }

  // create window and start graphics
  startGraphics
}
```

Figure 2.18: A Program to track mouse actions

When the mouse is pressed, the assignment

```
firstPoint := pressPosition
```

associates the name `firstPoint` with the position of the mouse. This assignment is interesting in several ways. It is the first assignment we have encountered in which the right side of the equal sign represents something other than the construction of a new object. In the general form of the assignment statement, the expression on the right side of the `:=` can be any phrase that describes the object we wish to associate with the name on the left. So, in this case, rather than creating a new object, we take the existing object that is named `pressPosition` and

give it a second name, `firstPoint`.

Immediately after this assignment, the position that describes the mouse position has two names. This may seem unusual, but it isn't. Most of us can also be identified using multiple names (e.g. your first name, a nickname, or Mr. or Ms. followed by your last name).

This assignment also illustrates the fact that a variable in a Grace program may refer to different things at different times. Suppose when the program starts you click the mouse at the point with coordinates (5,5). As soon as you do this, `onMousePress` is invoked and the name `firstPoint` is associated with a position that represents the point (5,5). If you then drag the mouse across the screen, release the mouse button and then press it again at the point (150,140), `onMousePress` is invoked again and the assignment statement in its body tells Grace to associate `firstPoint` with a position that describes the point (150,140). At this point, Grace forgets that `firstPoint` ever referred to (5,5). A variable in Grace may refer to different objects at different times, but at any given time it refers to exactly one thing (or to nothing if no value has yet been assigned to the name).

Exercise 2.5.2 *Why did we declare `firstPoint` to be a variable rather than a definition?*

In fact, the values associated with most instance variables are changed frequently. To illustrate the usefulness of such changes, we can write a simple drawing program.

Complex drawing programs provide many tools for drawing shapes, lines, and curves on the screen. We will write a program to implement the behavior of just one of these tools, the one that allows the user to scribble on the screen with the mouse as if it was a pencil. A sample of the kind of scribbling we have in mind is shown in Figure 2.19. The program should allow the user to trace a line on the screen by depressing the mouse button and then dragging the mouse around the screen with the button depressed. The program should not draw anything if the mouse is moved without depressing the button.

The trick to writing this program is to realize that what appears to be a curved line on a computer screen is really just a lot of straight lines linked together. In particular, to write this program what we want to do is notice each time the mouse is moved (with the button depressed) and draw a line from the place where the mouse started to its new position. Each time the mouse is moved with its button depressed, Grace will follow the instruction in the `onMouseDown` method. So, within this method, we want to include an instruction like:

```
lineFrom (previousPosition) to (currentPosition) on (canvas)
```

where `previousPosition` and `currentPosition` are names that refer to the previous and current positions of the mouse. The trick is to also include statements that will ensure that Grace associates these names with the correct positions.

Associating the correct point with the name `currentPosition` is easy. When `onMouseDown` is invoked, the computer will automatically associate the current mouse position with whatever name we choose to use as the method's formal parameter name. So, if the header we use when declaring `onMouseDown` looks like:

```
onMouseDown (currentPosition: Point) -> Done
```

we can assume that the name `currentPosition` will refer to the position of the mouse when the method is invoked.

Getting the correct position associated with `previousPosition` is a bit trickier. Think carefully for a moment about the beginning of the process of drawing with this program. The user

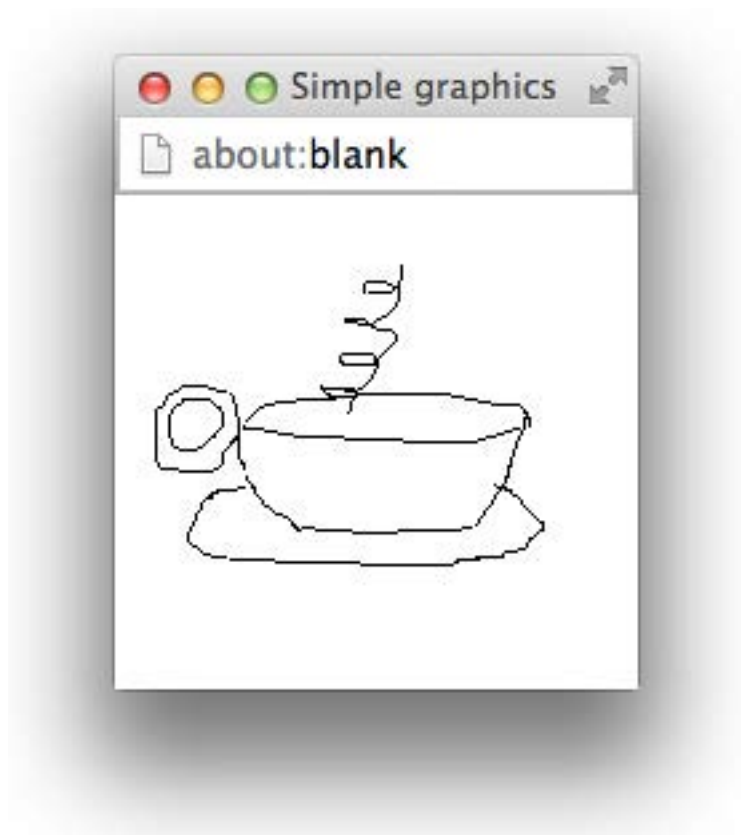


Figure 2.19: Scribbling with a computer's mouse

will position the mouse wherever the first line is to be drawn. Then the user will depress the mouse button and begin to drag the mouse. The first line drawn should start at the position where the mouse button was depressed and extend to the position to which the mouse was first dragged. This situation is similar to the problem we faced when we wanted to draw a line between the point where the mouse was depressed and the point where the mouse was released. The position at which the mouse button is first depressed will be available through the formal parameter of the `onMousePress` method but we need to access it in the `onMouseDrag` method because this is the method that will actually draw the line. We can arrange for `onMousePress` to share the needed information with `onMouseDrag` by declaring the name `previousPosition` as an instance variable and including an appropriate assignment in `onMousePress` to associate this name with the position where the mouse button is first pressed.

Given this analysis, we will include an instance variable declaration of the form:

```
var previousPosition: Point
```

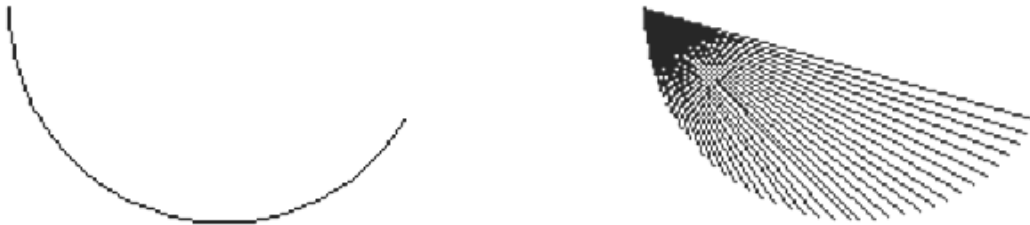
and then write the following definition for `onMousePress`:

```
method onMousePress(pressPosition: Point) -> Done {
  previousPosition := pressPosition
}
```

We also know that the `onMouseDrag` method must contain the line construction shown above and that its formal parameter should be named `currentPosition`. So, a first draft of this method would be:

```
onMouseDrag(currentPosition: Point) -> Done {
  lineFrom (previousPosition) to (currentPosition) on (canvas)
}
```

Unfortunately, if we were actually to use this code, the program would not behave as we want. For example, if we were to start near the upper left corner of the screen and then drag the mouse in an arc counterclockwise hoping to draw the picture shown below on the left, the program would actually draw the picture shown on the right.



The problem is that we are not changing the point associated with `previousPosition` as often as we should. In `onMousePress`, we tell the computer to make this name refer to the point where the mouse is first pressed and it continues to refer to this point until the mouse is released and pressed again. As a result, as we drag the mouse all the lines created start at the point where the mouse button was first pressed. Instead, after the first line has been drawn, we always want `previousPosition` to refer to the mouse's position when the last line was drawn. To do this, we must add the assignment statement:

```

dialect "objectdraw"

object {
  inherit graphicApplicationSize (400 @ 400)

  // beginning point for next line
  var previousPosition: Point

  // Choose color randomly for next line and remember start
  method onMousePress (pressPosition: Point) -> Done {
    previousPosition := pressPosition
  }

  // Draw lines to follow mouse
  method onMouseDrag (currentPosition: Point) -> Done {
    lineFrom (previousPosition) to (currentPosition) on (canvas)
    previousPosition := currentPosition
  }

  startGraphics
}

```

Figure 2.20: A simple sketching program

```

previousPosition := currentPosition

```

at the end of the `onMouseDrag` method yielding the complete program shown in Figure 2.20.

Exercise 2.5.3 *Explain why the assignment statement is still needed in `onMousePress` as shown below even though `previousPosition` is always updated in `onMouseDrag`*

```

method onMousePress (pressPosition: Point) -> Done {
  previousPosition := pressPosition
}

```

2.6 Summary

In this chapter we explored the importance of the use of names to refer to the objects our programs manipulate. Instance variable and definition names were used to share information between methods, and formal parameter names provided a means to pass information from outside the program into a method body.

Identifiers introduced in definitions need to have a value associated with them in the definition. Once associated with an object, they may not be associated with other objects, though mutator methods requested on the object may change its attributes. Variables have to be assigned a value using `:=` before they are used. They could be assigned to a new object at any time using `:=`. Formal parameters are introduced in method headers. They are associated

with values when the method is executed. For example, when `onMousePress` is executed, the formal parameter is associated with the point where the mouse is at that time.

Identifiers introduced in programs can also be declared with types that specify the kind of objects that can be associated with them. The use of these types can make your program more readable, while also providing extra computer generated checks to make sure the values match the specification of the type.

We learned more about displaying simple graphical objects on our program's canvas and learned how to modify the properties of these objects using mutator methods. In addition, we learned that the `objectdraw` library contains classes that can generate objects representing points and colors. These objects are not themselves visible on our screen, but are used in creating and modifying graphic images on the canvas.

In case you did not notice, the last example program we discussed, the `Scribble` program, was different from most of the other examples in one important regard. It actually is (at least part of) a useful program. This reflects the fact that the features we have explored are fundamental to the construction of all Grace programs. With this background, we are now well prepared to expand our knowledge of the facilities Grace provides.

2.7 Chapter Review Problems

Exercise 2.7.1 *Revise `TouchyWindow` so that the `Text` object has a name. Create the text object at the beginning, but don't show it unless the mouse is pressed. Change the `onMouseRelease` method so that it does not use the `canvas.clear` command but hides the text when the mouse is released.*

Exercise 2.7.2 *Suppose that you want to clear the contents of the canvas if the mouse exits the window. Write the appropriate method to do this.*

Exercise 2.7.3 *What is wrong with the following line of code?*

```
def message: Text = textat(60 @ 60) with ("Welcome to Hangman v1.0") on (canvas)
```

Exercise 2.7.4 *Write a program that draws a filled 20×20 square at the mouse position each time the mouse is pressed. When the mouse is released, the frame of the square should remain. The user should then be able to press again to create a new filled square at a new position that will leave a frame when the mouse is released.*

Exercise 2.7.5 *What is a variable and what are the two important steps needed to create it before being able to use it?*

Exercise 2.7.6 *Modify `Scribble` so that everything drawn in the window is red instead of black.*

Exercise 2.7.7 *Look at the two pieces of code. Do they produce the same or different outcomes? If the outcomes are different, explain the difference.*

- a. (a) **method** `onMouseClicked (point: Point) → Done {`
 `textAt (point) with ("Hello.") on (canvas)`
 `}`
- (b) **method** `onMouseClicked (clickPoint: Point) → Done {`
 `textAt (clickPoint) with ("Hello.") on (canvas)`
 `}`
- b. Assume `clickPoint` is a variable that has already been initialized correctly.
- (a) **method** `onMouseClicked (point: Point) → Done {`
 `point := clickPoint`
 `textAt (clickPoint) with ("Hi") on (canvas)`
 `}`
- (b) **method** `onMouseClicked (point: Point) → Done {`
 `clickPoint := point`
 `textAt (clickPoint) with ("Hi") on (canvas)`
 `}`

2.8 Programming Problems

Exercise 2.8.1 Write a simple program that does the following:

- When the program begins, a red square with a black frame is drawn. Each side of the square is 60 pixels, and the square is located 70 pixels from the right and 60 pixels down from the top left corner.
- When the mouse is clicked, the square turns blue.
- When the mouse exits the window the square disappears.
- When the mouse re-enters the window the square reappears and is once again red.

Exercise 2.8.2 Write a program that displays the frame of a rectangle when the mouse is pressed. The rectangle should be 100×100 pixels with the upper-left corner at the point where the mouse was clicked. When the mouse is dragged, the frame should move with the mouse, with the upper-left corner of the rectangle always following the cursor. When the mouse is released, the rectangle should be filled in so it is no longer just a frame.

Exercise 2.8.3 a. Write a simple program with these features:

- When the program begins there should be a message telling the user “Measles: Click to catch the disease!”
- When the user clicks the mouse a red dot should form at the point where the mouse was clicked. Make the diameter of the dots 5 pixels. The introductory message should no longer be visible.
- When the mouse exits the window the user should be cured. Thus all the spots should disappear and a message saying “You have been cured!” should appear.

- *When the mouse re-enters the window the original message should reappear.*

Hint: *Hidden items are also removed from the canvas when you clear it.*

- b.** *Modify your code from part (a) so that the spot is centered at the point where the mouse was clicked.*

Chapter 3

Working with Numbers

In the preceding chapter, we used numbers extensively to manipulate graphical objects. They were used to specify coordinates, dimensions, and even colors. While we used numbers to describe what we wanted to do to various graphical objects, we did not do much of interest with the numbers themselves. Numbers are of critical importance to Grace. Just as Grace provides operations to let us work with graphical objects, it provides operations to let us work with numbers. In this chapter we will explore some of these operations. We will see how to obtain numerical values describing properties of existing objects, how to perform basic arithmetic computations, how to work with numeric variables, and how to display numeric values.

3.1 Writing numbers

We've already seen several examples of using numbers in Grace, but there are some rules you should be aware of. Integral values can be written without decimal points and may optionally be preceded by a minus sign: 5, 147, -12, 0.

Non-integral values can be written using a decimal point. If a decimal point is used, there must be a digit on either side. Thus 15.3, -1.7, and 0.0005 are fine, but 12. and .7 are not legal in Grace.

3.2 Introduction to Accessor Methods

When we create a new point by evaluating a term like `50 @ 150` we use two numbers to create a single point. There are many situations where it is useful to do the opposite. That is, we have a point object and want to access the numerical values of the x and y coordinates associated with that point.

Suppose, for example, that we decided to change the `RisingSun` program so that rather than having to click the mouse to move the sun, the user could simply drag the mouse up and down and the sun would follow it. The desired behavior is similar to the way in which the scrollbars found in many programs react to mouse movement. If you grab the scroll box displayed in a vertical scrollbar you can move the scroll box up and down by moving the mouse, but you can not move the scroll box to the left or right. Similarly, in the program we

have in mind, even if the mouse is dragged about in spirals, the circle that represents the sun should only move straight up and down so that its y coordinate is always the same as the y coordinate of the mouse's current position.

In this new version of `RisingSun`, which we will name `ScrollingSun`, we need to replace the `onMouseClicked` method from the previous version with an `onMouseDown` method of the form:

```
method onMouseDrag (mousePosition: Point) -> Done {  
    sun.moveTo ( ... )  
}
```

This leaves us with the question: what argument should we give to the `moveTo` method?

We want the sun to move to the position in the canvas whose x coordinate is the same as the sun's initial x coordinate, 50, and whose y coordinate is equal to the y coordinate of the mouse. Thus, the only challenge is obtaining the y coordinate of the mouse's position.

The point object `mousePosition` certainly contains enough information to determine the y coordinate of the mouse. Grace lets us ask the point for this information through a mechanism called an *accessor method*. Like the mutator methods discussed in the preceding chapter, a small collection of accessor methods is associated with each object. Point objects understand two accessor methods, `x` and `y`.

To use an accessor method we write a name that refers to the object that is the target of the request, followed by a period, the name of the accessor method, and, if necessary, a parenthesized list of argument values. So, to position the sun appropriately in the `onMouseDown` method, we can calculate the new position using `mousePosition.y` and then use that to set the new position of `sun`

```
sun.moveTo (50 @ mousePosition.y)
```

You should observe that the notation for requesting an accessor method is identical to the notation for requesting a mutator method. In the case that no arguments are required, do not include a set of empty parentheses after the name of the method. (Some other languages insist on those empty parentheses, but not Grace.)

The complete text of the revised program is shown in Figure 3.1. The only differences between this program and the one shown in Figure 2.3 are that the instructions displayed by the object have been modified, and the new `onMouseDown` method replaces the old `onMouseClicked` method.

Accessor methods are also associated with objects of the graphics classes introduced in the last chapter. The coordinates and dimensions of a graphical object can be accessed using methods named `x`, `y`, `width` and `height`. Like the methods discussed above, these accessor methods provide numeric information about an object. In addition, there are accessor methods associated with graphical objects that provide other forms of information. There is a method named `color` that returns the color of a graphical object. Similarly, `location` returns a point object describing an object's current location.

Exercise 3.2.1 *Is the `+` method on points an accessor method? Why or why not?*

Exercise 3.2.2 *What is the difference between an accessor method and a mutator method? What can an accessor method do that a mutator method cannot?*


```

dialect "objectdraw"

object {
  inherit graphicApplicationSize (200 @ 200)

  // Circle representing the sun
  def sun: Graphic2D = filledOvalAt (50 @ 20) size (50 @ 50) on (canvas)

  // Display instructions
  textAt (20 @ 20) with ("Drag the mouse up or down") on (canvas)

  // Drag to raise or lower the sun
  method onMouseDrag (mousePosition: Point) -> Done {
    sun.moveTo (50 @ mousePosition.y)
  }

  startGraphics
}

```

Figure 3.1: Program to make the sun scroll with the mouse

3.3 Accessing the Size of the Canvas

In the last chapter, to keep things simple, we assumed that we knew the size of the windows in which our programs would run. For example, the `DrawGrid` program presented in Section 2.2.2 draws a grid like the one shown in Figure 3.2. The bars in this grid are created by constructions of the form:

```

filledRectAt (verticalCorner) size (5 @ 200) on (canvas)
filledRectAt (horizontalCorner) size (200 @ 5) on (canvas)

```

placed within the program's `onMouseClicked` method. The vertical rectangles created by these constructions are 200 pixels tall and the horizontal rectangles are 200 pixels wide. The resulting drawing looks fine if the window is exactly 200 by 200, which is the window size we showed in the figures that illustrated the drawings the program would produce, but they would not look right if the window was larger. If the window was wider, we would want the program to draw wider horizontal rectangles. If the window was taller, we would want taller vertical rectangles.

When we write a program, the size of the window is determined by the parameters in `graphicApplicationSize(..., ...)`. Thus most of the programs we have written so far specify a window that is 200 by 200 pixels. However, a programmer may change the arguments to `size` at some later time. If the rest of the program is not changed, then the graphic images in the window may no longer appear in the correct positions.

There are two ways we can avoid this problem. One is to provide a definition that specifies the width and height of the window. E.g.

```

def windowWidth: Number = 200
def windowHeight: Number = 200
object {
  inherit graphicApplicationSize (windowWidth @ windowHeight)
}

```

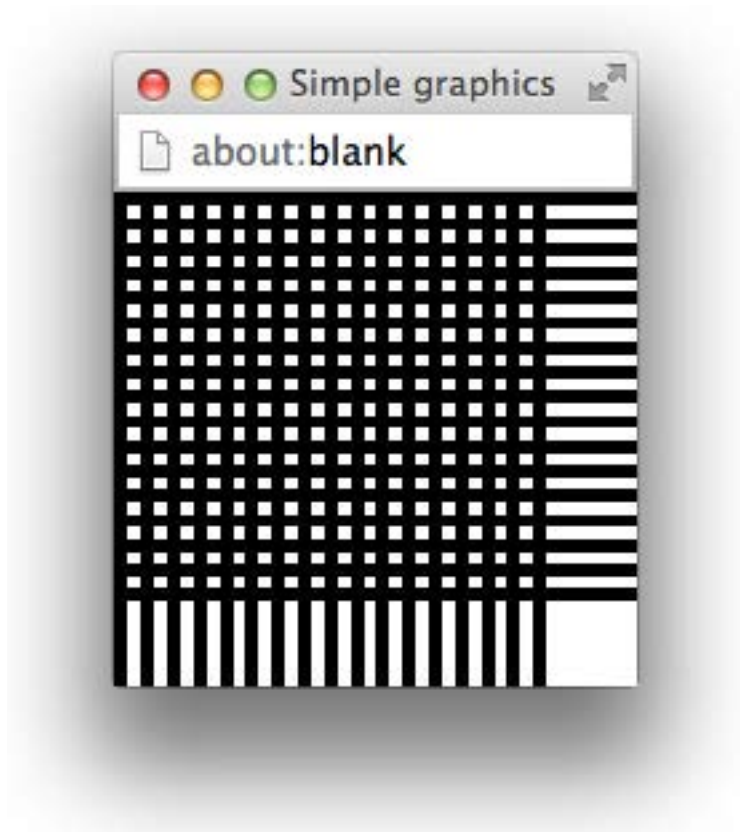


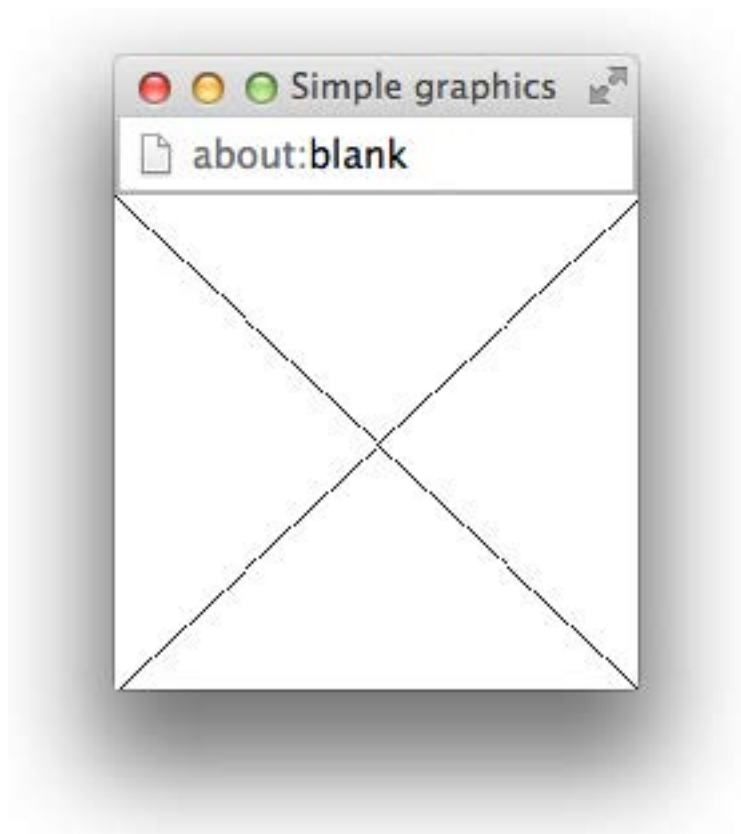
Figure 3.2: Drawing produced by the DrawGrid program

```
... windowWidth .... windowHeight  
}
```

As long as the programmer is careful to only use the identifiers `windowWidth` and `windowHeight` rather than the literal `200`, the program should continue to work correctly even if `windowWidth` and `windowHeight` were changed.

An alternative way to accomplish this is to write programs that determine the actual size of the canvas while running and adjust the objects they draw accordingly. We have seen that the canvas provides a mutator method named `clear`. The canvas also supports two accessor methods named `width` and `height` which allow a program to determine the dimensions of the drawing area. Like the `x` and `y` methods associated with points, these methods do not expect any parameter values. To produce a version of `DrawGrid` that would work correctly in any size canvas, we would simply replace the occurrences of the number `200` in the constructions shown above with appropriate uses of accessor methods to obtain the following code:

```
filledRectAt (verticalCorner) size (5 @ canvas.height) on (canvas)  
filledRectAt (horizontalCorner) size (canvas.width @ 5) on (canvas)
```



Exercise 3.3.1 Write a program that draws an *X* through the canvas using the `height` and `width` accessor methods of the canvas.

3.4 Expressions and Statements

It is important to observe that accessor methods serve a very different function than do mutator methods. A mutator method instructs an object to change in some way. An accessor method requests some information about the object's current state.

Simply asking an object for information is rarely worthwhile by itself. We also need to tell Grace what to do with the information requested. We would never write an instruction of the form:

```
mousePosition.y
```

Such an instruction would tell Grace to ask the point named `mousePosition` for its *y* coordinate, but then make no use of the information. Instead, we use accessor methods in places within a program where Grace expects the programmer to provide an object. For example, an accessor method can be used on the right-hand-side of an assignment statement to describe the object that should be associated with the name on the left-hand-side of the assignment symbol, as in the statement

```
lastY := mousePosition.y
```

as an argument to a construction, as in

```
var newLocn: Point := 10 @ mousePosition.y
```

or as the argument for a method request, like `moveBy`.

This notion of a phrase that describes an object or value is important enough to deserve a name. Such phrases are called *expressions*. We have already seen several different sorts of phrases that Grace recognizes as being expressions. Where numeric information is needed, we have often explicitly included the numbers to use by typing *numeric literals* like 50 and 150 as in the request

```
newLocn := 50 @ 150
```

In other situations, *accessor method requests* have been used to describe numeric values, as on the right-hand-side of the following assignment statement.

```
newLocn := 50 @ point.y
```

Where non-numeric information was needed, we have either used a *construction* to create the needed information as in

```
sun.color := colorGen.r (200) g (100) b (0)
```

or provided a *name* that was associated with the desired object as in

```
sun.color := purple
```

Thus, numeric literals, instance variable names, constructions, and invocations of accessor methods can all be used as expressions.

In any context where it is necessary to describe an object, Grace will accept any of the forms of expressions we have introduced. In a context where Grace expects the programmer to describe a color, we can equally well use a name associated with a color, a construction of the form `colorGen.r (...) g (...) b (...)`, or an invocation of the `color` accessor method. Grace is, however, picky about the type of value described by an expression. In a context where Grace expects us to provide a color, we can't meaningfully provide an expression that describes a number or a point instead. One of the reasons that we prefer to label parameters with their types is to make it explicit what kind of a value is needed in that context.

Not all phrases found in a Grace program are expressions. A request of a mutator method,

such as:

```
sun.moveBy (0, -5)
```

is an example of a phrase that is not an expression. This phrase contains subparts that are expressions, the numeric literals 0 and -5, but the phrase is not itself an expression, because it does not describe a value. Instead of describing a value, this phrase instructs Grace to perform an action. Such phrases are called *statements* or sometimes *instructions* or *commands*. Statements instruct Grace to perform actions that either produce output visible to the user or alter the internal state of the computer in a way that will affect the future behavior of the program. The body of each method we define in a Grace program must be a sequence of statements, possibly ending with an expression if it is to return a value.

We have seen three types of statements at this point. A request of a mutator method, such as

```
sun.moveBy (0, -5)
```

is one type of statement. The second type of statement is the assignment statement. It instructs the computer to perform the action of associating a name with an object or value. Note that the phrase on the right-hand-side of an assignment *must* be an expression.

The third type of statement we have encountered is the construction. We have used instructions like:

```
textAt (mousePosition) with ("Pressed") on (canvas)
```

to place graphics on the canvas. We have already stated, however, that a construction is an expression. Which is it? The answer is both. A construction like the one shown above describes an object. Therefore it can be used in contexts where expressions are required. At the same time, the construction of a graphical object involves the action of changing the contents of the display. Accordingly, the construction by itself can be viewed as a statement.

There are constructions that merely describe an object without having an associated action that affects any aspect of the state of the program. For example,

```
10 @ 20
```

It does not make much sense to use such a construction as a statement, because a program that contained such a statement would behave the same if the statement were removed. Grace, however, does not prevent the programmer from writing such nonsense. In fact, Grace will allow the programmer to use many kinds of expressions as if they were statements by simply placing them on new lines. In a sensible program, however, the only expressions we have introduced so far that make sense as statements are constructions of graphical objects.

Exercise 3.4.1 *Which of the following statements could actually be useful in a program and which could not? Explain.*

- a. `sun.color`
- b. `textAt (point) with ("Hello") on (canvas)`
- c. `colorGen.r (60) g (60) b (60)`
- d. `myPosition := 50 @ 50`

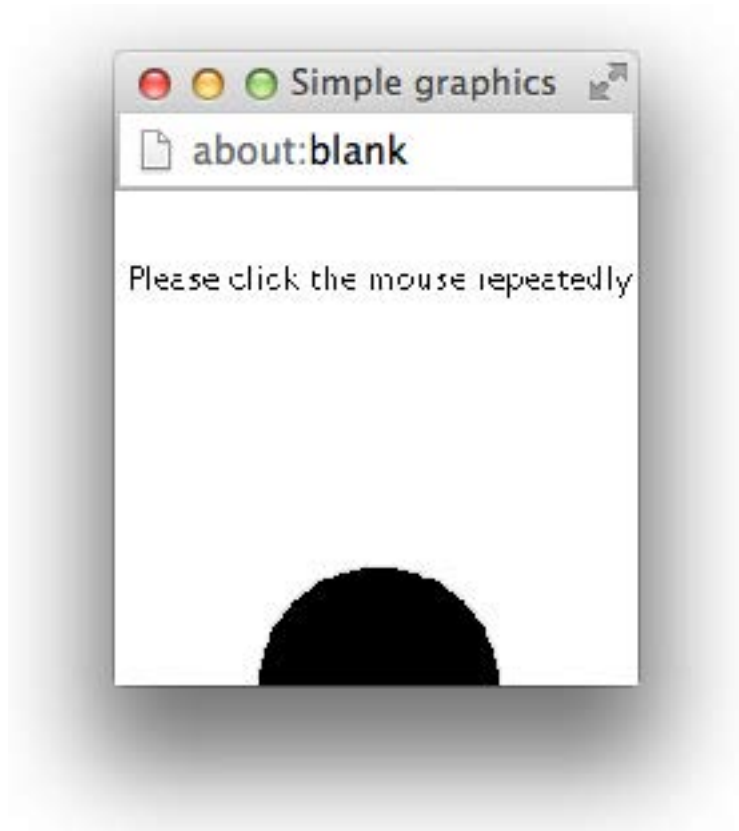


Figure 3.3: The sun rises over the horizon

3.5 Arithmetic Expressions

Sometimes it is very useful to describe a numeric value to Grace by providing a formula to compute the number. For example, to describe the x coordinate of a point slightly to the left of the current mouse position we might say something like:

```
mousePosition.x - 10
```

Grace allows the programmer to use such formulae and calls them *arithmetic expressions*. As an example of the use of arithmetic expressions, we can make some additional improvements to our `ScrollingSun` program.

Using arithmetic expressions involving the `width` and `height` methods of the canvas, we can revise the `ScrollingSun` program so that it adjusts the size and position of the circle that represents the sun based on the size of the canvas. To maintain the proportions used in the original program as shown in Figure 3.3:

- the diameter of the circle should be one quarter of the width of the canvas,
- the circle should be centered on the canvas (from left to right),
- initially, the top of the circle should be placed so that half the circle is visible above the

horizon. To do this, the top of the circle should be one half of its diameter above the bottom of the canvas.

It would also be appropriate to center the text of the instructions horizontally on the canvas. The indentation of the text from the left edge of the canvas should be equal to that on the right side. So, it should be half of the difference between the width of the text and the width of the canvas.

Each of these verbal descriptions can be turned into a formula, which can then be used in the program. The diameter of the sun, represented by identifier `sunDiameter`, is the value that should be specified as the width and height in the `filledOval` construction. It can be described as

```
canvas.width / 4
```

The x coordinate value for the left edge of the circle would be given by the formula

```
(canvas.width - sunDiameter) / 2
```

The y coordinate for the top of the circle would be described as

```
canvas.height - (sunDiameter / 2)
```

Finally, the x coordinate for the left edge of the instructions should be

```
(canvas.width - instructions.width) / 2
```

The complete `ScrollingSun` program using such formulae is shown in Figure 3.4. In most cases, we have simply replaced a number used as an expression by the appropriate formula. The only slight complication is the code to center the text. We cannot use the `width` method associated with the text object until it has been constructed because, unlike with other graphical objects, we do not specify its width when we construct it. So, when we construct the text we just use 0 as its x coordinate value. Then, once it exists we use its `width` method to figure out how big it is. Finally, we use `moveBy` to shift the text to the right to where it belongs.¹

The arithmetic expressions shown in the preceding examples use only two of the arithmetic operators, subtraction and division. It is also possible to use multiplication and addition. The symbols used to indicate addition, subtraction and division are the standard symbols from mathematics: `+`, `-`, `/`, and `÷`. As we saw earlier, multiplication is represented using an asterisk, `*`. Thus, to say “2 times the width of the canvas” one would write `2 * canvas.width`.

Grace supports two kinds of division. The `/` operator gives an answer as a decimal, e.g., $7/2 = 3.5$, while the `÷` operator gives an integer answer, with the greatest integer less than the decimal answer. Thus $7 \div 2 = 3$ and $-7 \div 2 = -4$. Other languages typically produce the integer quotient by just throwing away the fractional part, returning -3 for the second quotient. You can type the `÷` symbol on most keyboards by holding down the option key while typing `/`.

The values being operated upon are called *operands*. In the example above of multiplication, the operands are 2 and `canvas.width`.

Other fairly common arithmetic operations are exponentiation, written `^`, and modulus, written `%`. Exponentiation raises numbers to a power. For example `3 ^ 2` represents 3 times 3 and evaluates to 9. The modulus operator finds the remainder when the first number is

¹The `width` method of text objects is currently not very accurate, so you may have to make further adjustments if you really care about placement.

```

// A program that produces an animation of the sun rising and setting.
// The animation is driven by dragging the mouse.

dialect "objectdraw"

object {
  inherit graphicApplicationSize (200 @ 200)

  // diameter of the sun
  def sunDiameter: Number = canvas.width / 4

  // x coordinate of the sun
  def sunX: Number = (canvas.width - sunDiameter) / 2

  // location where the sun starts
  def startSun: Point = sunX @ (canvas.height - sunDiameter / 2)

  // Circle representing the sun
  def sun: Graphic2D = filledOvalAt (startSun) size (sunDiameter @ sunDiameter) on (canvas)
  sun.color := colorGen.yellow

  // Display instructions
  def instructions: Text = textAt (0 @ 20) with ("Drag the mouse up or down") on (canvas)
  instructions.moveBy((canvas.width - instructions.width) / 2, 0 )

  // Drag to raise or lower the sun
  method onMouseDrag (mousePosition: Point) -> Done {
    sun.moveTo (sunX @ mousePosition.y)
    instructions.visible := false
  }

  // Move sun back to original position when mouse exits window
  method onMouseExit(mousePosition: Point) -> Done {
    sun.moveTo (startSun)
  }

  startGraphics
}

```

Figure 3.4: Program to make the sun scroll with the mouse

divided by the second. For example, $14 \% 3$ is 2. If you performed the integer division of 14 by 3, the quotient would be 4, with a remainder of 2. In Grace, $m = ((m \ \text{div} \ n) * n) + (m \% n)$, even if m or n are negative.

As can be seen in this last example, Grace only stores decimals to a finite precision. When you print out the results of an expression, it will only print out 6 digits to the right of the decimal point and the last digit will be rounded to be the closest to the actual value. Thus printing the result of $5 / 3$ gives 1.666667 rather than 1.666666 because the first is closer to the actual value of the quotient.

The following table summarizes the most commonly used arithmetic operators in Grace.

+	addition
-	subtraction
*	multiplication
/	division
÷	integer division
^	exponentiation
%	modulus

Finally we note that you should always leave spaces on both sides of operators. Some Grace compilers will allow you to leave them out, but the language specification requires the spaces to be there for readability (for both the computer and humans).

Exercise 3.5.1 Write the `begin` method for a program that draws two lines that form a cross (consisting of a horizontal line and a vertical line as shown in Figure 3.5) on the canvas with the intersection of the two lines in the center of the canvas. The program should draw the cross correctly no matter what the size of the canvas.

3.5.1 Ordering of Arithmetic Operations

Two of the arithmetic expressions used in the `ScrollingSun` program shown in Figure 3.4 illustrate an issue a programmer must be aware of when writing such expressions: the rules used to determine the order in which operations are performed. The first of these is the expression

```
(canvas.width - sunDiameter) / 2
```

which is used to set the x coordinate of the sun. The second determines the initial y coordinate for the top of the sun:

```
canvas.height - sunDiameter / 2
```

Both involve a subtraction and a division. The first, however, uses parentheses to make it clear that the subtraction should be performed first and that the result of the subtraction should be divided by 2. The correct interpretation of the second expression is not as clear. In fact, Grace will first divide `sunDiameter` by 2 and then subtract the result of this division from the height of the canvas.

In the absence of parentheses that dictate otherwise, Grace always performs divisions in an expression before subtractions. Thus, the second formula is equivalent to the formula:

```
canvas.height - (canvas.width / 4)
```

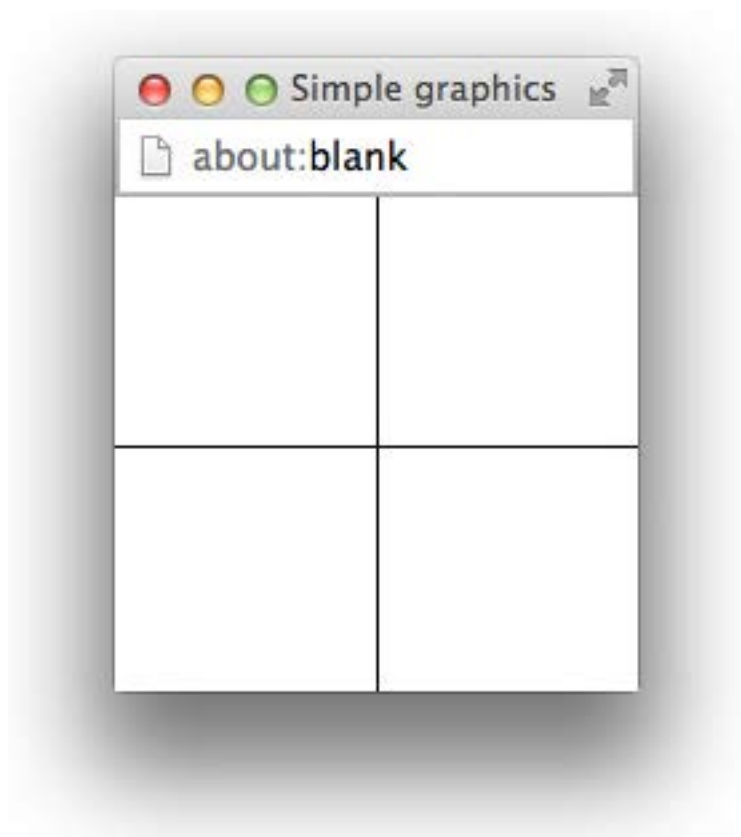


Figure 3.5: Drawing desired for Exercise 3.5.1

The rule that division is performed before subtraction is an example of a *precedence rule*. When evaluating simple arithmetic expressions, Grace follows two basic precedence rules:

- Perform divisions and multiplications before additions and subtractions. We therefore say that division has higher precedence than addition but that division and multiplication are of equal precedence.
- When performing operations of equal precedence (i.e. additions and subtractions or divisions and multiplications) perform the operations in order from left to right as written.

Parentheses can be used to override these precedence rules as seen in the first example above. Any part of a formula enclosed in parentheses will be evaluated before its result can be used to perform operations outside the parentheses.

The operators \wedge and $\%$ are not assigned a precedence in Grace, so if they are in an expression with other operators then parentheses must be included to determine the order of evaluation.

Exercise 3.5.2 *What are the values of the following expressions?*

- $4 + 3 * 8 / 2 - 3$
- $(4 + 3) * 8 / 2 - 3$
- $4 + (3 * 8) / 2 - 3$
- $4 + 3 * 8 / (2 - 3)$
- $(4 + 3) * 8 / (2 - 3)$
- $(4 + 3 * 8) / 2 - 3$

3.6 Numeric Instance Variables

In the previous chapter, we saw that it is sometimes useful to associate instance variable names with points or other objects to enable one method to communicate information to commands in another method. Unsurprisingly, it is often useful to associate names with numeric values in a similar way. We can illustrate this by adding yet another feature to our `RisingSun` program.

As the real sun rises, the sky becomes brighter and brighter. Suppose we wanted to try to simulate this in our program. For this example we will return to the original interface where the user clicks repeatedly to make the sun rise. Now, when the sun is near the bottom of the screen, we would like the background to be filled with a dark shade of gray. We can do this by constructing a `filledRect` as big as the canvas and setting its color to an appropriate shade of gray. As the user clicks, we can make the background become lighter by using `color:=` to replace the original shade of gray with lighter and lighter shades until it is eventually white.

We have seen that each color is described by a triple of numbers giving the amounts of red, green, and blue in the color. Shades of gray correspond to triples in which all three values are the same. The bigger the number used, the brighter the shade. So,

```
colorGen.r (0) g (0) b (0)
```

describes black,

```
colorGen.r (50) g (50) b (50)
```

describes a dark shade of gray,

```
colorGen.r (200) g (200) b (200)
```

would be a fairly light shade of gray and

```
colorGen.r (255) g (255) b (255)
```

is white.

To control the brightness of the background, we would like to associate an instance variable name with the number to be used to generate the shade of gray currently desired. We will use the name `brightness` for this variable. This name can then be used to construct shades of gray for the background by using the construction:

```
colorGen.r( brightness) g (brightness) b (brightness)
```

To use such a name, of course, we must first declare the name and then add assignment statements to ensure that it is associated with the correct number at each time as the program executes.

In the declaration, we will associate the name `brightness` with a number corresponding to a dark shade of gray by including an assignment statement of the form:

```
var brightness: Number := 50
```

Each time the user clicks the mouse, we want to associate a larger number with `brightness`. We can do this by including the assignment statement

```
brightness := brightness + 3
```

in `onMouseClicked`.

This statement tells the computer to take the current value associated with the name `brightness`, add three to it, and then associate the name `brightness` with the result. The first time the mouse is clicked, `brightness` will be associated with the value 50 specified in the declaration. The result of adding 3 to 50 is 53. So, after the assignment statement is executed, `brightness` will be associated with the value 53. The next time the mouse is clicked, Java will add 3 to the new value of `brightness`, 53, and set it equal to 56. Thus, each time the mouse is clicked, the value of `brightness` will become three greater and the color generated by the construction

```
colorGen.r (brightness) g (brightness) b (brightness)
```

will become a little bit brighter.

With these details we can complete the program. The code is shown in Figure 3.6.

```

// A program to simulate the brightening of the sky at sunrise

dialect "objectdraw"

object {
  inherit graphicApplicationSize(200 @ 200)

  // sky created with brightness
  def sky: Graphic2D = filledRectAt (0 @ 0) size (canvas.width @ canvas.height) on (canvas)

  // initial brightness of the sky
  var brightness: Number := 100

  // set sky color to be dark
  sky.color := colorGen.r (brightness) g (brightness) b (brightness)

  // diameter of the sun
  def sunDiameter: Number = canvas.width / 4

  // x coordinate of the sun
  def sunX: Number = (canvas.width - sunDiameter) / 2

  // location where the sun starts
  def startSun: Point = sunX @ (canvas.height - sunDiameter / 2)

  // Circle representing the sun
  def sun: Graphic2D = filledOvalAt (startSun) size (sunDiameter @ sunDiameter) on (canvas)
  sun.color := colorGen.yellow

  textAt(20 @ 20) with ("Please click the mouse repeatedly") on (canvas)

  // Each time mouse is clicked, sun rises more and sky brightens
  method onMouseClick (point: Point) -> Done {
    sun.moveBy (0, -5)
    brightness := brightness + 3
    sky.color := colorGen.r (brightness) g (brightness) b (brightness)
  }

  startGraphics
}

```

Figure 3.6: Using a numeric instance variable

Exercise 3.6.1 *What would you expect to happen in the program in Figure 3.6 if you changed the assignment of `brightness` to `brightness := 200` and then changed the first line in the `onMouseClicked` method to the following?*

`brightness := brightness - 3`

Exercise 3.6.2 *What would you expect to happen in the program in Figure 3.6 if you keep clicking the mouse for a long time? What might go wrong? Run the program and see what happens.*

Exercise 3.6.3 *Suppose instead of a gray sky in the example, we wanted the sky to start as a dark gray, but become a lighter blue each time the mouse is clicked. How would you change the code to make this happen.*

3.7 Displaying Numeric Information

We have seen how we can use the computer's ability to work with numbers to produce better drawings on the computer's screen. Sometimes, however, it is the numbers themselves rather than any drawing that we really want to see. The main purpose of many computer programs is to perform numerical calculations. Examples include programs that determine your taxes, determine your GPA, and estimate the time required to travel from one point to another. Such programs often simply display the numbers they compute rather than a drawing a graph of some sort on the screen. Even programs that are not primarily focused on numerical computations often need to display numerical information. For example, a word processor might need to display the current page number. In this section we will describe two mechanisms in Grace that can be used to display numerical information.

3.7.1 Displaying Numbers as text

As a very simple first example, let's make the computer count. You probably don't remember it, but at some point in your early childhood you most likely impressed some adult by demonstrating your remarkable ability to count to 10 or 20 or maybe even higher. To enable the computer to produce an equally impressive demonstration of its counting abilities, we will construct a program that will count. It will start at 0 and move on to the next number each time the mouse is clicked. The current value will be displayed on the computer's screen.

We have already introduced one mechanism that can be used to display numbers on the screen. We just didn't mention that it could be used with numbers at the time. In the very first program in Chapter 1, we used a construction of the form:

`textAt (clickTextPosition) with ("Press mouse in this window") on (canvas)`

and explained that the text construction requires three parameters:

- the position on the canvas where the information should be displayed, and
- the information to be displayed,
- the canvas.

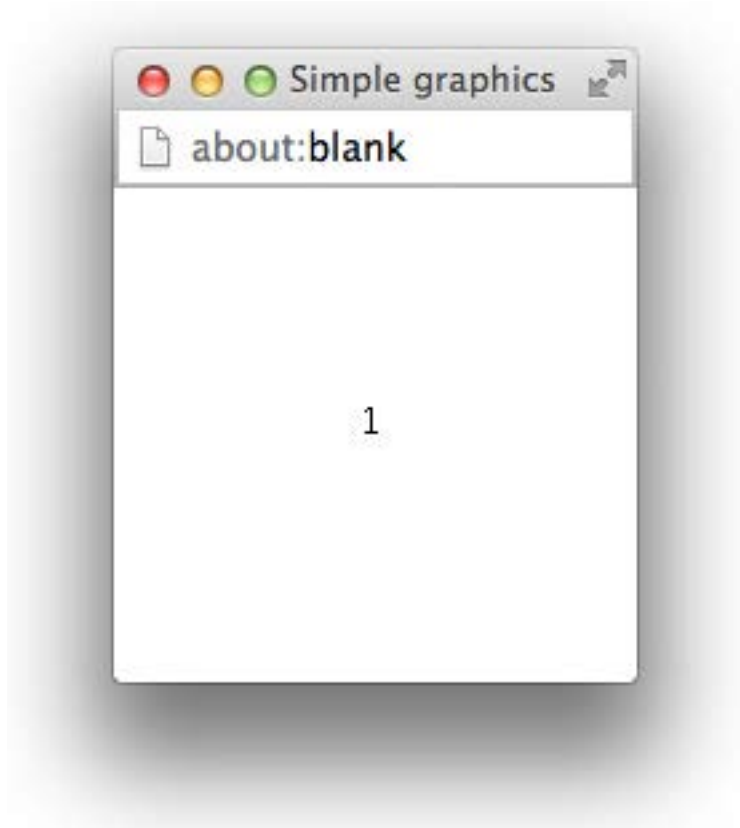


Figure 3.7: A computer counting program before the first click

In all the examples of text constructions we have seen thus far, the second parameter has been a sequence of characters surrounded by quotes. In fact, the second parameter to `textAt ()` with `()` on `()` can be any expression that returns a string.

Any object can return a string from an `asString` method request. We will ensure that each new object we define has an `asString` method that returns a string that describes the object. If we do not define `asString` ourselves, it will instead return a default string value that may not be very helpful.

In particular, if we define a variable

```
var theCount: Number := 1
```

with the intent of using it to count up from one, and then we execute a construction of the form:

```
textAt (100 @ 100) with (theCount.asString) on (canvas)
```

Grace will display the current value of the variable `theCount`, 1, at the point (100,100) in the program's window as shown in Figure 3.7.

Of course, displaying the number 1 isn't counting. The program we want to construct should start by displaying 1, but the first time the user clicks the mouse, we want to replace 1 by 2. On the next click, we want to replace 2 by 3 and so on.

In case you didn't notice, one of the examples considered in this chapter already demon-

```

// A program to count as high as you can click
dialect "objectdraw"

object {
  inherit graphicApplicationSize (200 @ 200)

  var theCount: Number := 0

  def countDisplay: Text = textAt (100 @ 100) with (theCount.asString) on (canvas)

  method onMouseClick (point: Point) -> Done {
    theCount := theCount+1
    countDisplay.contents := theCount.asString
  }

  startGraphics
}

```

Figure 3.8: A simple counting program.

strates how to teach a computer to count. In the version of the rising sun program in which the background became brighter as the sun rose, the operations we performed on the variable named `brightness` essentially told the computer to count upwards starting at 100. The instruction that we used to progress through the different values of `brightness` was

```
brightness := brightness + 3
```

A very similar assignment statement involving the variable `theCount`:

```
theCount := theCount + 1
```

is what we need to complete the counting program described above.

Such a counting program is shown in Figure 3.8. The body of the `onMouseClicked` method uses the assignment statement shown above to associate the next counting number with `theCount` each time the mouse is clicked. Note that changing the value associated with the variable `theCount` does not cause the value displayed by the text object named `countDisplay` to change. To change the text displayed, we update the contents of the item. This changes the information displayed by a text object. Like the second parameter expected in a text construction, this information can be a quoted sequence of characters or any other expression that returns a string. The statement

```
countDisplay.contents := theCount.asString
```

in the `onMouseClicked` method tells Grace to change the information displayed by the text object named `countDisplay` that was created earlier to the string corresponding to the current value of `theCount`.

An alternative to resetting the contents would be to clear the canvas and then construct a new text object displaying the new value of `theCount`. Construction of a new object, however, is a fairly time consuming process. When possible, it is better to reuse an existing object rather than create a new one. Accordingly, in an code like this is preferable to updating contents.

There are several other mutator methods that can be applied to text objects. In Chapter 1 we already showed that the method `visible:=` could change the visibility of text objects. In addition, the `moveBy` and `moveTo` methods can be used to reposition text objects, just as they can be used with rectangles and ovals. Finally, there are several special mutator methods for use with text objects. For example, the methods `fontSize` and `fontSize :=` can be used to access or change the size of displayed text. For example,

```
countDisplay.fontSize := 24
```

could be added after the definition `countDisplay` if we wanted to increase the size of the numbers displayed.

3.7.2 Using `print`

In a program that mixes graphical output with numerical or other textual information, text objects and the `contents :=` method are the most appropriate tools for displaying textual information. In programs that only display textual information, there is another tool that is often simpler and more appropriate, `print`.

All the output displayed by the Grace programs we have considered so far appears in the canvas in the pop-up window associated with the program. These programs can, however, display output in another window provided by the Grace system. There is no special name that can be used to refer to this window from within your program. It is simply known as the *output* or *console window*, and is found in the area below the program text.

The console window is more limited than the `canvas` in that it can only be used for text. On the other hand, it is more convenient for the display of text than is the `canvas`.

To display information in the console you use a method named `print`. This method takes a single parameter specifying the information to be displayed. Any expression that evaluates to a string can be used as a parameter to `print`. In particular, you can certainly use a string surrounded by double quotes or the results of sending an `asString` method request to a number (or any other object, for that matter).

In fact, `print` is a special method in that if you evaluate it with an argument, and that argument is not a string, it applies the `asString` method request before printing the result. As a result,

```
print (countDisplay.asString)
```

behaves exactly the same as

```
print (countDisplay)
```

A revised version of our counting program that uses the Grace console to display values as it counts is shown in Figure 3.9. The only text this version displays on the canvas is a message telling the user to click in order to make the program count. This will be displayed instead of the number 0 when the program first starts. The first time the user clicks, 1 is placed in the Grace console by the `print` in the `onMouseClicked` method. Each succeeding click will place another value in the console window.

When `print` is used, you do not have to provide coordinates to specify where the text should be displayed. The Grace console window displays the information you provide to

```

dialect "objectdraw"

object {
  inherit graphicApplicationSize (200 @ 200)

  // number of times the user has clicked
  var theCount: Number := 0

  // where to put instructions
  def instructionPosn: Point = 20 @ 100

  // instructions to user
  def instructions: Text = textAt (instructionPosn) with ("Click to make me count") on (canvas)

  // print the total number of times mouse has been clicked each time mouse is clicked
  method onMouseClick (point: Point) -> Done {
    theCount := theCount+1
    print (theCount)
  }

  startGraphics
}

```

Figure 3.9: Counting in the Grace console

these methods much as text might be displayed in a word processor’s window. Each time your program executes a `print`, the text specified is placed on a new line in the Grace console window, below any text placed there by previous uses of `print`.

Once the console window fills up, the older text scrolls off the top of the window leaving the newer lines visible. A scroll bar is provided so that a person running your program can look at the older items if desired. Figure 3.10 shows how both the canvas and the Grace console window might look after this program is run and its user clicks 25 times.

3.7.3 Mixing Text and Numbers

Often, a number displayed all by itself has little meaning. The difference between just displaying “3” and displaying “Strike 3” or “3 p.m.” or “Line 3” can be quite significant. Accordingly, in many programs rather than just displaying a number on the screen it is desirable to display a number combined with additional text that clarifies its meaning. Luckily, this can be done easily in Grace with both text objects and `print`.

When specifying the information to be displayed in a text object or on the Grace console, we can combine quoted text with numeric information by enclosing the numeric information in curly braces: “{...}”. Suppose, for example, that we wanted our counting programs to display a message like “You have clicked 3 times” instead of just displaying 3 on the third click. For the version that uses text objects to place the information on the screen, we could accomplish this by replacing the command

```
countDisplay.contents := theCount.asString
```

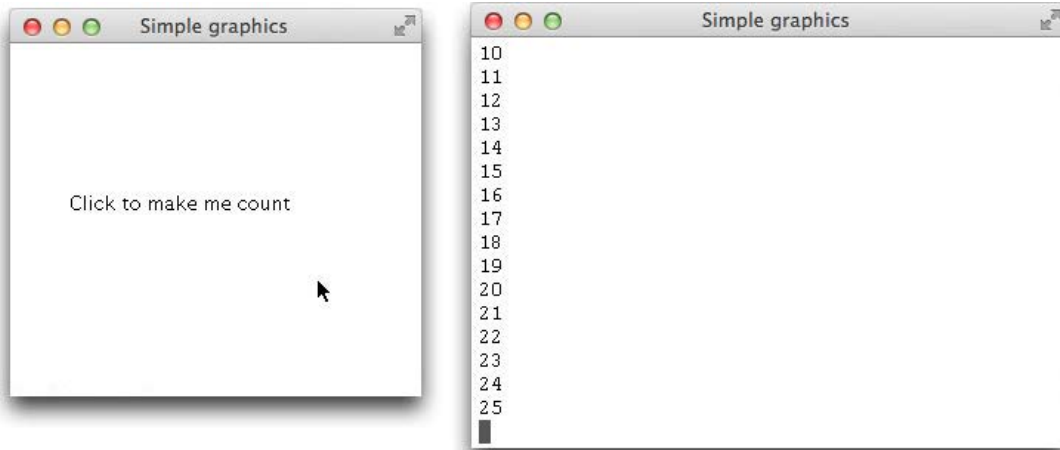


Figure 3.10: Counting using the Grace console

shown in Figure 3.8 with the command

```
countDisplay.contents := "You have clicked {theCount} times"
```

When the string is evaluated, `theCount` is evaluated, converted to a string (implicitly using `asString`) and that string is inserted into the surrounding string.

Similarly, for the print version shown in Figure 3.9 we could replace the command

```
print (theCount)
```

with the command

```
print ("You have clicked {theCount} times")
```

In fact, you can place any expression at all in the curly braces inside a string. When the string is evaluated, the expression in curly braces is evaluated, converted to a string using `asString`, and then inserted into the string.

When Grace sticks together bits of text, it doesn't think about things like words. It just sticks the letters and digits it is given together. This means you have to include all the characters you want displayed, including any spaces. If you look carefully at the `print` command shown above, you will notice that there is a space before the open brace “{” and before the close brace “}”. If these were not included, Grace would display the text

```
You have clicked3times
```

instead of displaying

```
You have clicked 3 times
```

as desired.

Exercise 3.7.1 *What output would you expect from the following statements? Assume `count` is 34.*

- a. `print ("The count is: count - 3")`

b. `print ("The count is: {count} – 3")`

c. `print ("The count is: {count – 3}")`

Exercise 3.7.2 *Suppose you are trying to write a simple program to help someone practice the multiplication tables. The program repeatedly displays a message of the form:*

What is 5 x 9?

in a Text object named question that is created in the program. The program uses two variables named factor1 and factor2 that are assigned randomly generated numbers between 0 and 9 to determine which question to display. For example, the question shown above would be displayed if factor1 equaled 5 and factor2 equaled 9.

Write the statement needed to update the message displayed by question after new values for factor1 and factor2 are chosen. (Don't worry about how the user tells the program the correct answer.)

3.8 Random numbers

“Pick a number. Any number. ...”

You might expect to hear this phrase from the hawker at a carnival game table. You might not expect it to be a useful instruction to give a computer within a Grace program, but just the opposite is true. There are many programming contexts in which it is handy to be able to ask the computer to pick a random number for you. Obvious examples are game programs. Programs that deal cards, simulate the tossing of dice, or simulate the spinning of a roulette wheel all need ways of picking items randomly. In addition to game programs, there are many programs that simulate the behavior of real systems for practical purposes that need ways to incorporate the randomness of the real world in their calculations. With this in mind, Grace and most other programming systems include what are called random number generators.

To use random numbers you must first import the library that defines them. This can be done by writing `import "random" as rand`, where `random` is the name of the library, and `rand` is an identifier used to refer to it. (You can replace `rand` by a different name if you like, but we'll stick with it for now.

The method request `rand.integerIn (m) to (n)` returns a randomly selected integer between `m` and `n`, inclusive. Thus evaluating `rand.integerIn (7) to (12)` will return a randomly chosen integer from 7 to 12, inclusive. That is, it will return one of 7, 8, 9, 10, 11, or 12, with each occurrence equally likely.

Suppose that you wanted to write a program to simulate a board game in which at each turn the player rolls two dice. Our method for generating random integers can be used to generate numbers just like a single die.² To illustrate the use of this, we will construct a simple program that simulates the rolling of a pair of dice each time the mouse is clicked.

²The English word for the little cubes you roll while playing many board games has an irregular plural form. If you have several of these cubes, you call them dice. If you have just one then it is a die.



Figure 3.11: Sample message drawn by dice simulation program

When the user clicks the mouse, we need to calculate the new values of the dice. We will provide two variables, `roll1` and `roll2`, which will each be assigned a value generated by our expression above that supplies random integers between 1 and 6.

The complete code of a simple program to simulate rolling two dice is shown in Figure 3.12. A sample of the program's output is shown in Figure 3.11.

Let's talk through the code to make sure you understand why each piece is there. As usual, we are working in the `objectdraw` dialect. In order to show all of each text item we set the window with the program output to be 300 pixels wide rather than the 200 we have used before. The height remains at 200 pixels.

We introduce the definition of `numSides` to name the number of sides on each die. If we were to change to a different size die (e.g., with 12 sides), a change in this one place in the program would automatically fix the rest of the code.

Now we create a text object to display the instructions to the user. We do not bother to give it a name as we will never need to refer to it again in the program. However, we do give the name `result` to a text object whose contents are initially an empty string, i.e., a string of length 0. We will update its contents in the `onMouseClicked` method to give us information about the new values of the dice.

We also declare the variables `roll1` and `roll2`, but don't yet assign them a value as we will

```

// A program to simulate the rolling of a pair of dice.
dialect "objectdraw"
import "random" as rand

object {
  inherit graphicApplicationSize (300 @ 200)

  // How many sides our dice have
  def numSides: Number = 6

  // A Text message giving instructions to the player
  textAt (30 @ 30) with ("Click to make me roll the dice") on (canvas)

  // The values of the two dice after a roll
  var roll1: Number
  var roll2: Number

  // Display a prompt and create the Text used to display the results
  def result: Text = textAt (30 @ 100) with ("") on (canvas)

  // Roll the dice with each click
  method onMouseClick(point: Point) -> Done {
    roll1 := rand.integerIn (1) to (numSides)
    roll2 := rand.integerIn (1) to (numSides)
    result.contents := "You rolled a {roll1} and a {roll2} for a total of { roll1 + roll2 }"
  }

  startGraphics
}

```

Figure 3.12: Simulating the rolling of a pair of dice

not roll the dice until the user clicks the mouse.

The `onMouseClicked` method evaluates our complex arithmetic expression twice to get two new random numbers, assigning them to our two variables. We then update `result` to show the values of the dice. Recall that the expressions within “{ ... }” will be evaluated and then have their string equivalents inserted into the surrounding string.

Exercise 3.8.1 *Write a program that draws a rectangle named `box` at the point (50, 50). The box should be 50 pixels wide and the height of the box should be determined by an expression involving random numbers that generates values for the height between 10 and 100 pixels. The height of `box` should change each time the mouse is clicked.*

3.9 Summary

Early applications of computers involved scientific and engineering problems that required large amounts of numerical computation. The first computers clearly earned the name “computer”. When using a word processor, reading your email, or browsing the web, it is easy to forget that computers perform arithmetic computations. Even in programs that do not appear to involve numbers, however, numerical computations continue to play a role. For example, a word processor has to do arithmetic just to determine how many words will fit on a line.

In this chapter, we have explored a few of the mechanisms Grace provides to perform numerical computations. We explored the differences between statements and expressions in Grace programs. Statements are phrases in a program that instruct the computer to perform an action that will change the visible state of the computer or change the value associated with some variable name. Expressions are phrases that describe a value or object to be used in the program. We have seen that Grace recognizes several forms of expressions: literals like “12”, variable names, constructions, invocations of accessor methods, and arithmetic formulae involving the operations of addition, subtraction, multiplication, division, exponentiation, and modulo. We also saw how to use a method returning random numbers to generate numbers in a given range.

We showed how to instruct Grace to produce textual output that included numerical information using both text objects and the `print` method. Text objects are used when such information is to be included in the display of a program that also produces graphical output. The `print` method provides a simpler mechanism for producing text output that is appropriate for programs that only produce textual output in the Grace console.

3.10 Chapter Review Problems

Exercise 3.10.1

- a. *Write the instructions for creating and positioning a 50 by 50 pixel framed rectangle so that it is centered in any window.*
- b. *Write the instructions for creating a `Text` object, that says “I am centered” and positions it so that it is centered in any window.*

Exercise 3.10.2 Write a program called `RisingMoon` which is similar to `RisingSun` except now we have a crescent moon instead of a sun and the sky is now black. Each time the user clicks, the crescent moon rises slightly. Hint: a crescent can be created by displaying an oval with the same color as the background over another oval.

Exercise 3.10.3 Assume these variables are at your disposal:

```
var myCounter: Number := 17
var yourCounter: Number := 12
```

What is the output of the following statements:

- a. `print ("The count is {myCounter} clicks")`
- b. `print ("The count is yourCounter clicks")`
- c. `print ("The count is {myCounter + yourCounter} clicks")`
- d. `print ("The count is {myCounter - yourCounter} clicks")`

Exercise 3.10.4 Why is the following code inefficient? Improve it.

```
var counter: Number := 0
method onMouseClick( point: Point ) -> Done {
  //increase the count and display it with each click
  counter := counter + 1
  canvas.clear
  displayText := textAt (50 @ 50) with ("The count is {counter}") on (canvas)
}
```

Exercise 3.10.5 What modifications would you have to make to Figure 3.12 so that you are now using a twelve sided die instead of a boring six sided one and the text is indented another 15 pixels to the right.

3.11 Programming Problems

Exercise 3.11.1 Take the `ICanCount` program one step further by writing a program called `ICanCountALot`. It will keep track of the number of mouse clicks, mouse exits and button presses. The display of the program should look similar to the one in Figure 3.13.

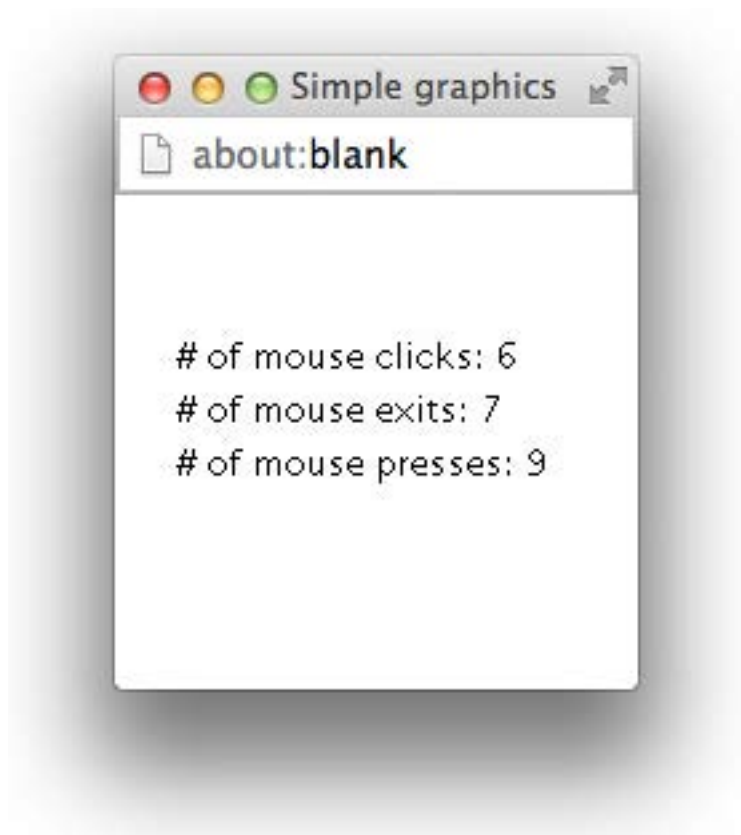


Figure 3.13: Display for ICanCountALot

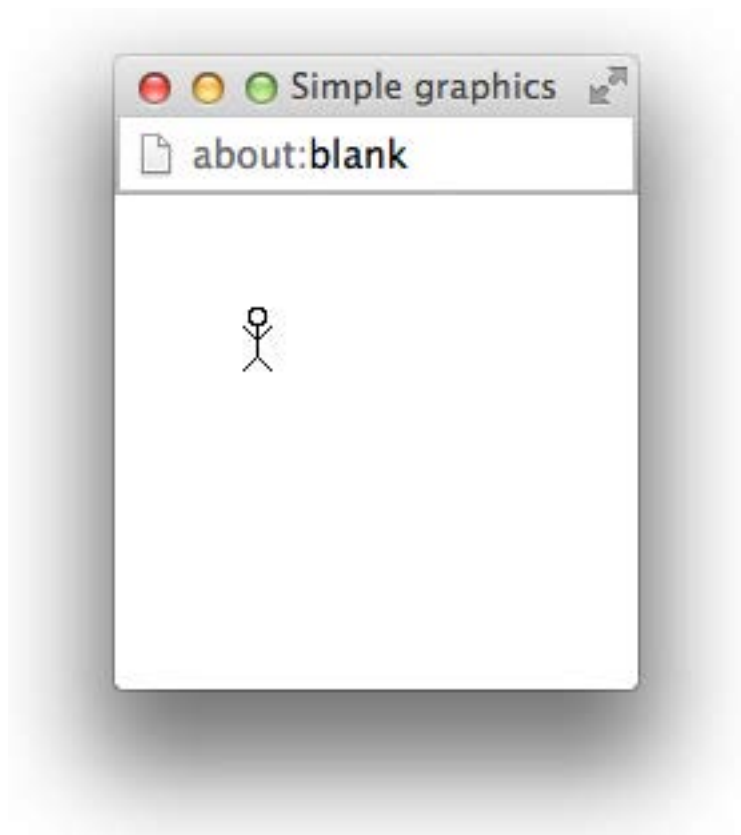


Figure 3.14: Initial display for GrowMan

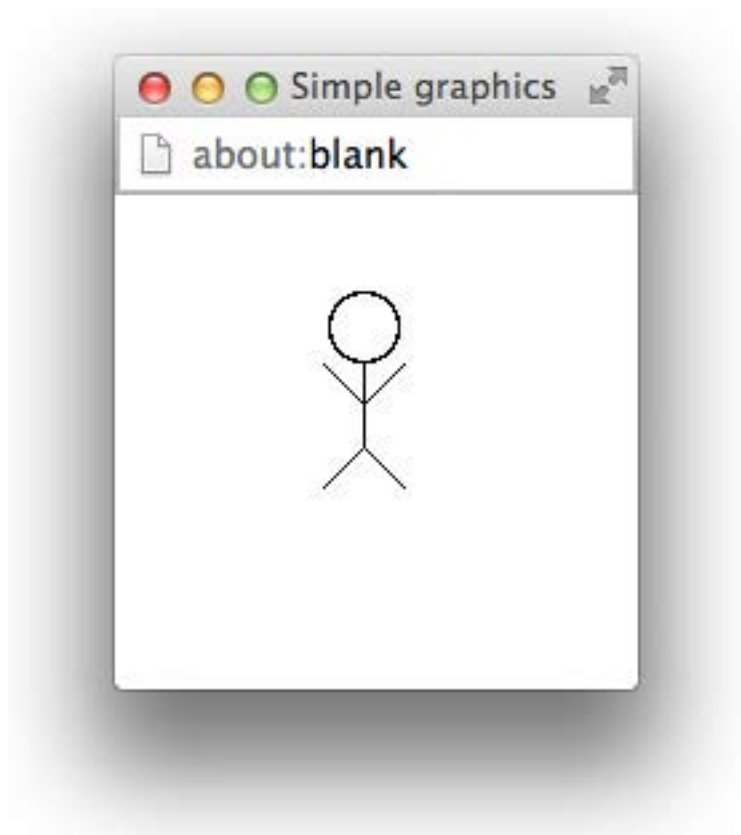


Figure 3.15: Display for GrowMan after ten clicks

Exercise 3.11.2 Write a program called *GrowMan* which initially draws a little man as shown in Figure 3.14. Each time the user clicks, the man grows by a set amount. After ten more clicks, the display should look like that in Figure 3.15. The following constants and instance variable declarations are given to you.

```
def grow: Number = 2
def headSize: Number = 6
def limbSize: Number = 5
def headStart: Number = 50

def body_X: Number = headStart + (headSize / 2)
def neck_Y: Number = headStart + headSize
def armpit_Y: Number = headStart + (2 * headSize)
def bodyEnd: Number = headStart + (3 * headSize)
def feet_Y: Number = bodyEnd + limbSize
def arms_Y: Number = armpit_Y - limbSize
def left_X: Number = body_X - limbSize
def right_X: Number = body_X + limbSize
```

Exercise 3.11.3 a. Create a program called *RandomBox* with the following features:

- When the program begins it creates a filled rectangle of random size (make your limits 20 - 120 pixels) centered on the screen.
 - When the mouse is clicked the console will report the dimensions and position of the box.
 - When the mouse exits the window the box disappears.
 - When the mouse re-enters the window, it generates a new size for the box and re-centers it on the screen.
- b. Modify the above code so that it generates a random color in addition to the random size for the box. Add code so that the red, green, and blue color components are reported in the console as well. Make sure your box has a frame so that it can easily be seen even if it is given a very light color. **Hint:** The `Color` class has accessor methods `red`, `green`, and `blue` that you may find useful.

Chapter 4

Making Choices

To write interesting programs we must have a way to make choices in Grace. For example, we might need to perform different instructions depending on whether a user has clicked inside a particular rectangle. In this chapter we show how to make choices in Grace using the *if-then* statement.

Conditional statements like the *if-then* statement are programming constructs capable of choosing between blocks of code to execute. These statements provide enormous expressive power to programmers, and yet are very easy to use and understand because they mimic the way we think. For instance, the following form a conditional statement in English:

“*If* it’s sunny outside *then* we will play frisbee, *otherwise* we will play cards.”

An example of a conditional statement that is a bit more relevant to our concerns with programming might be:

“*If* the mouse position is contained in the rectangle *then* display message “success”.
Otherwise display message “missed”.”

In Grace, the *if-then* statement is the most commonly used conditional statement. It comes in a variety of forms that we will explore, each of which is useful for certain situations. With the help of conditionals we will write programs to determine a win or loss in the game of craps and to figure out what to do on a weekend based on the weather and your finances.

After presenting a brief example illustrating the use of the *if-then* statement, we formally introduce several of its variations that will allow us to handle more complex situations. We also introduce the *Boolean* data type for expressions that can be either *true* or *false*. Finally, we provide advice on how to use conditionals clearly and effectively so that your programs can be understood correctly by the Grace compiler and, more importantly, by other programmers.

4.1 A Brief Example: Using the *if-then* Statement to Count Votes

We illustrate the use of conditionals with a simple example of a program to count votes in an election. To accomplish this we will divide the program’s canvas in half vertically so that the

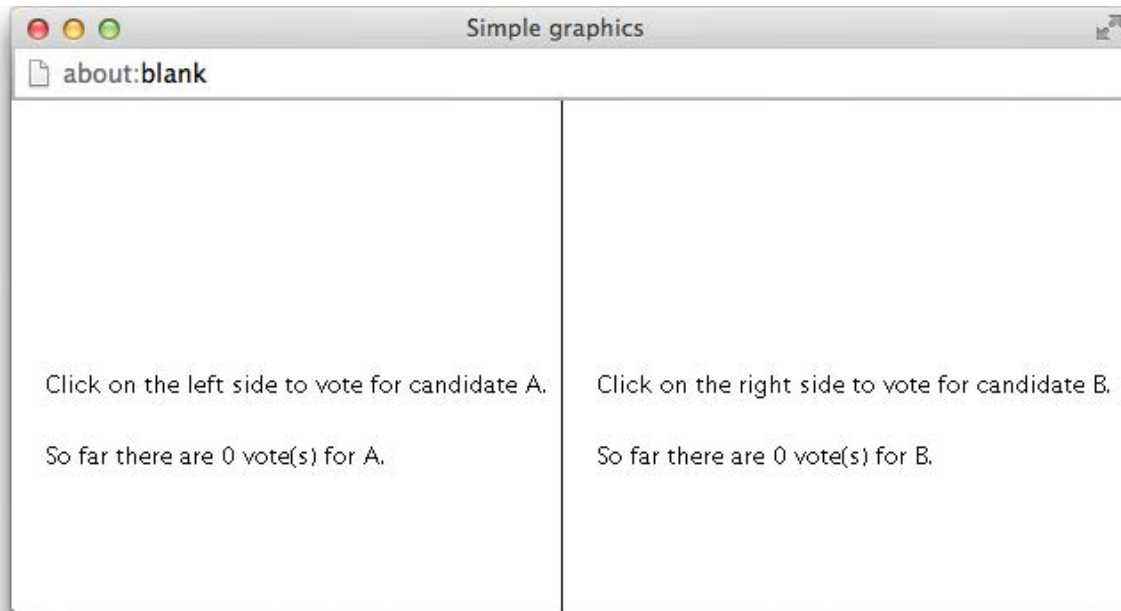


Figure 4.1: Screen shot of Voter program.

left and right sides represent candidates A and B respectively (see Figure 4.1). A mouse click on either side will be treated as a vote for that half's candidate.

Recall the program `ICanCount` in Figure 3.8, which keeps track of the number of times the user has clicked the mouse. The code for its `onMouseClicked` method is given below:

```
method onMouseClick (point: Point) -> Done {
  theCount := theCount+1
  print (theCount)
}
```

Whereas this method simply counts *all* mouse clicks on the canvas, by using the `if-then` statement our voting program's `onMouseClicked` method will be able to discriminate between votes for candidate A and B.

The code for program `Voting` is given in Figure 4.2. The constructs used should be familiar by now. The program starts by defining some `x` and `y` coordinates as well as the top and bottom of the line separating the regions. An advantage to using definitions to provide names for coordinates is that if they need adjusting, it is easy to modify them in one place while the results show up everywhere.

The program creates four new text objects, the top two of which display voting instructions while the bottom two display the current tally for each candidate. The last line before `onMouseClicked` draws the vertical line dividing the canvas.

The `onMouseClicked` method, on the other hand, contains a new programming construct. The `if-then` statement is used to determine which candidate gets each vote and then updates the appropriate Text object to display the new total. To decide who receives each vote, the program compares the `x` coordinate of the position of the mouse click to that of the middle

```

dialect "objectdraw"

object {
  inherit graphicApplicationSize (600 @ 200)

  def displayLeftX: Number = 5 // x-coordinate of location of A's votes
  def displayRightX: Number = canvas.width / 2 + displayLeftX // x-coordinate of location of B's votes

  def upperInstructsY: Number = canvas.height / 2 - 20 // y coordinates of instructions
  def upperCountY: Number = canvas.height/2 + 20 // y coordinates of vote counts

  // locations of top and bottom of separating line
  def topLine: Point = (canvas.width / 2) @ 0
  def bottomLine: Point = (canvas.width / 2) @ (canvas.height)

  // current counts of votes for A and B
  var countA: Number := 0
  var countB: Number := 0

  textAt (displayLeftX @ upperInstructsY) with
    ("Click on the left side to vote for candidate A.") on (canvas)
  textAt (displayRightX @ upperInstructsY) with
    ("Click on the right side to vote for candidate B.") on (canvas)

  // Display number of votes for A and B
  def countAText: Text = textAt (displayLeftX @ upperCountY) with
    ("So far there are {countA} votes for A.") on (canvas)
  def countBText: Text = textAt (displayRightX @ upperCountY) with
    ("So far there are {countB} votes for B.") on (canvas)

  // Line separating voting spaces
  lineFrom (topLine) to (bottomLine) on (canvas)

  // Update votes and display vote counts
  method onMouseClick(point: Point) -> Done {
    if (point.x < (canvas.width/2)) then {
      countA := countA + 1
      countAText.contents := "So far there are {countA} votes for A."
    } else {
      countB := countB + 1
      countBText.contents := "So far there are {countB} votes for B."
    }
  }
}

startGraphics
}

```

Figure 4.2: Code for Voter program.

of the canvas. Note that `canvas.width / 2` refers to the x coordinate of the middle of the canvas.

The if-then statement allows the programmer to make choices about which statements are executed in a program based on a *condition*, an expression whose value is either true or false. In the sample program, when the mouse is clicked, the program must determine whether to give a vote to candidate A or B. The condition is whether the x coordinate of the mouse click, obtained by evaluating `point.x`, is less than the x coordinate of the middle of the canvas. The condition is written in Grace as `point.x < (canvas.width / 2)`. If the condition is *not* satisfied (*i.e.*, if `point.x` is greater than or equal to `canvas.width/2`) then the code after the `else` keyword is executed.

The two lines of code following the line containing the if-then are grouped together by a pair of matching curly braces. A sequence of statements surrounded by curly braces in this manner is called a *block*. Similarly, the two statements immediately following the `else` also form a block. If the *condition* of an if-then statement is true, the block of statements immediately after the condition is executed. Otherwise, the block immediately after the `else` is executed.

Exercise 4.1.1 *What are the conditions in the following statements? For example, in the statement, “When it is cold outside, I wear a hat,” the condition is “it is cold outside”.*

- a. *When it is Tuesday, I have piano lessons.*
- b. *Since today is Halloween, it must be October.*
- c. *If Bobby says yes, we will go to the prom.*
- d. *Yesterday’s game determined the wild card, so if the Red Sox won, they made it to the playoffs.*

4.2 The if-then Statement

Now that we have seen the if-then-else statement in action, let’s carefully examine its syntax and meaning.

The code in the Voting example given in Figure 4.2 contains a form of conditional statement called the if-then-else statement. Its syntax is:

```
if (condition) then {
  if-part    // statements to be executed when condition is true
} else {
  else-part  // statements to be executed when condition is false
}
```

The *condition* in the syntax template represents an expression whose value is true or false. The phrases *if-part* and *else-part* represent sequences of Grace statements. We’ve included comments to make it clear when each of these sequences of statements is executed, even though these comments are not part of the formal syntax.

When an if-then-else statement is executed, the computer first determines whether *condition* is true. If so, it executes the statements in the block of code surrounded by the first pair of curly braces, “{” and “}”, called the *if-part*, and then skips over the rest of the statement.

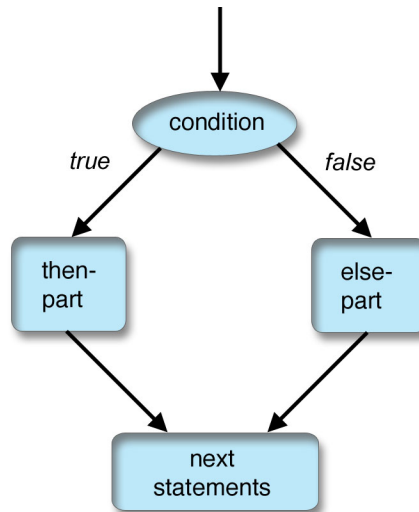


Figure 4.3: Semantics of the if-then-else statement.

```

// Update votes and display vote counts
method onMouseClicked(point: Point) -> Done {
  if (point.x < (canvas.width / 2)) then {
    countA := countA + 1
    countAText.contents := "So far there are {countA} votes for A."
  } else {
    countB := countB + 1
    countBText.contents := "So far there are {countB} votes for B."
  }
  totalText.contents := "Votes so far: {countA + countB}"
}

```

Figure 4.4: Code to display individual and total vote counts

Otherwise (*i.e.*, if *condition* is false), it skips over the *if-part* and will instead execute the block of statements after the *else* keyword, called the *else-part*. Exactly one of the two blocks of code is processed when the *if-then-else* statement is executed. When that block is completed, execution resumes immediately after the *if-then-else* statement. This execution sequence is illustrated in Figure 4.3.

The following example shows how execution resumes with the statements that follow an *if-then-else* statement. Suppose we want to modify our Voting program so that it always displays the total number of votes. Let `totalText` be defined as a text object earlier in the program. Figure 4.4 shows a revised version of `onMouseClicked` that displays the current vote total. Each time the user clicks on the canvas, the line updating the contents of `totalText` will be executed regardless of which half of the screen the mouse was clicked on.

There are many situations in which we don't need an *else-part*. Fortunately, there is a simple variant of the *if-then-else* statement, the *if-then* statement, which does not have the *else* keyword or the *else-part*.

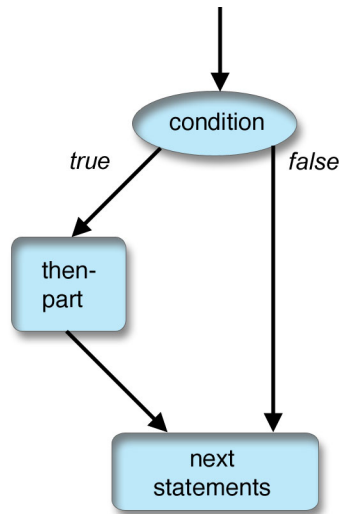


Figure 4.5: Semantics of the if-then with no else.

```

if ( condition) then {
  if-part    // statements to be executed when condition is true
}
  
```

If *condition* is true, then the *if-part* is executed as before. If it is false, however, the program simply moves on to the *next statement* after the *if-part* because there is no *else-part* to execute. The execution sequence is illustrated in Figure 4.5.

Exercise 4.2.1 Write the following statements as *if-then-else* or *if-then* statements. For example, the statement, “On Mondays I ride my bike to class. On other days, I walk.” can be rewritten as “If it is Monday, I ride my bike to class. Else, I walk.”

- a. On Sunday I eat pancakes. On other days, I eat cereal.
- b. I go to class on weekdays, but on weekends, I watch movies.
- c. In the summer I always wear sandals, but during the other seasons, I wear sneakers.
- d. If it is raining, I bring an umbrella.

4.2.1 Example: Using the if-then Statement with 2-D Objects

Suppose we want to write a program that begins by displaying a square on the canvas. If the user clicks inside the square then the computer moves the square 50 pixels to the right. If the click is not inside the square, nothing happens.

How can we determine if a point is inside a square? We could compare the coordinates of the point to the positions of the left, right, top, and bottom edges of the square. However, such tests are so common that all of the two dimensional geometric objects (*e.g.*, filled and framed rectangles and ovals) provide the accessor method `contains` that does this for us.

For example, let `square` be a variable referring to a framed rectangle, and let `point` refer to a position. Then the expression `square.contains (point)` evaluates to *true* if the object held in `square` *contains* the point, and *false* otherwise.

For the sake of brevity we will only write out the `onMouseClicked` method of this program. We assume that `square` and `xOffset` have been declared and initialized elsewhere.

```
method onMouseClick (point: Point) -> Done {
  if (square.contains (point)) then {
    square.moveBy (xOffset, 0)
  }
}
```

An English translation of the above method would read: “*If* the `square` contains the point where the mouse was clicked *then* tell the `square` to move to the right by `xOffset` pixels. Otherwise do nothing.” (Of course, the computer code doesn’t say, “Do nothing.” Instead it simply omits the *else*-part.)

We will present further variations of *if*-*then* statements later in this chapter, but first we will explore the kinds of expressions that can be used to form the *condition* part of these statements.

Exercise 4.2.2

- a. What would happen if you clicked the mouse inside `square` given the `onMouseClicked` method shown below? Assume `xOffset` is 60.

```
method onMouseClick (point: Point) -> Done {
  if (square.contains (point)) then {
    square.moveBy (xOffset, 0)
  }
  square.moveBy (-xOffset/2, 0)
}
```

- b. What would happen with the above code if you clicked the mouse outside `square`.
- c. What would happen if you now clicked the mouse outside of `square` and `onMouseClicked` was changed to the following?

```
method onMouseClick (point: Point) -> Done {
  if (square.contains (point)) then {
    square.moveBy (xOffset, 0)
  } else {
    square.moveBy (-xOffset / 2, 0)
  }
}
```

4.3 Understanding Conditions

Comparison operators like “`<`” are used in expressions that evaluate to either *true* or *false*, called boolean expressions. Grace contains several comparison operators. They include:

`<` , `>` , `==` , `<=` , `>=` , `!=`

We must use `==` to test for equality because `=` has already been used for definitions. Because `≠` isn't available on all keyboards, the symbol `!=` (read as “does not equal”) stands for inequality. Because keyboards also do not always include the symbols `≤` or `≥`, Grace uses the combinations `<=` and `>=` in their places.

As with arithmetic operators in Grace, you must surround comparison operators with spaces. Thus `3 < 4` is fine, but `3<4` will generate an error message on most Grace compilers.

Be very careful not to confuse `=` and `==`. The first is only used in definitions, while the second is used only for comparisons. Keep in mind as well that `:=` is used for assignments to variables.

Using these comparison operators, we can write `x > 4`, `y != (z+17)`, and `(x+2) <= y`. Depending on the values of the variables, each of these expressions will evaluate to either *true* or *false*. Comparison operators are not assigned a precedence, so expressions being compared must be surrounded by parentheses unless the expression is just a simple identifier or method request. Thus in the examples above, `z+17` and `x+2` are surrounded by parentheses.

Suppose the current value of `x` is 3, `y` is 6, and `z` is -10. Here are the results of evaluating the above expressions:

- `x > 4` is *false* because 3 is not greater than 4.
- `y != (z+17)` is *true* because 6 is different from -10+17.
- `(x+2) <= y` is *true* because 3+2 is less than or equal to 6.

Exercise 4.3.1 *Suppose the current value of `x` is 2, `y` is 4, and `z` is 15. Determine whether each of the following conditions evaluates to true or false.*

- `(x + 2) < y`.
- `(z - 3 * x) != (y + 5)`.
- `(x * y) == (z - 9)`.
- `z >= (3 * y)`.

4.3.1 Using Boolean Values

Grace expressions can have the values *true* and *false*. Just as one can write down values of type `Number` directly in Grace as 17, -158, or 47, we can write down values of type `Boolean` directly in Grace as **true** or **false**. We can also declare variables that can hold values of type `Boolean`.

There are a large number of expressions in Grace that return of type `Boolean`. As we have just seen, combining two integer-valued expressions with one of the comparison operators, `<`, `>`, `<=`, `>=`, `==`, and `!=`, results in a value of type `Boolean`. We have also seen the method `contains` that returns a value of type `Boolean`.

If `ok` is a variable holding values of type `Boolean` and `x` represents a number then the following are valid statements:

```
ok := true
ok := (x >= 3)
```

```

dialect "objectdraw"

object{
  inherit graphicApplicationSize (400 @ 400)
  // create box
  def box: Graphic2D = filledRectAt (200 @ 100) size (30 @ 30) on (canvas)

  // point where mouse was last seen
  var lastPoint: Number

  // whether the box has been grabbed by the mouse
  var boxGrabbed: Boolean := false

  // Save starting point and whether point was in box
  method onMousePress (point: Point) -> Done {
    lastPoint := point
    boxGrabbed := box.contains(point)
  }

  // if mouse is in box, then drag the box
  method onMouseDrag (point: Point) -> Done {
    if (boxGrabbed) then {
      box.moveBy (point.x - lastPoint.x,
                 point.y - lastPoint.y )
      lastPoint := point
    }
  }
}

startGraphics
}

```

Figure 4.6: Code for dragging a box.

In each case the expression of the right hand side evaluates to a value of type Boolean and hence can be assigned to a variable of type Boolean.

We will use both the contains method and Boolean variables in the implementation of a new program WhatADrag. The complete code listing for this program is given in Figure 4.6. The program begins by displaying a box on the screen. If the user presses the mouse down while it is pointing inside of the box, and then drags the mouse, the box will follow the mouse on the canvas. If the mouse is not pointing in the box when the mouse button is pressed, then dragging the mouse should have no effect, even if the mouse happens to cross the box at some point during the drag.

Let's look at the code in the mouse-handling methods to see how we can program this behavior. As soon as the mouse button is pressed, the `onMousePress` method is executed. The first assignment in `onMousePress`,

```
lastPoint := point
```

results in saving the current position of the mouse (as held in parameter `point`) as the value of variable `lastPoint`. The position is saved so that it can be used when `onMouseDrag` is executed

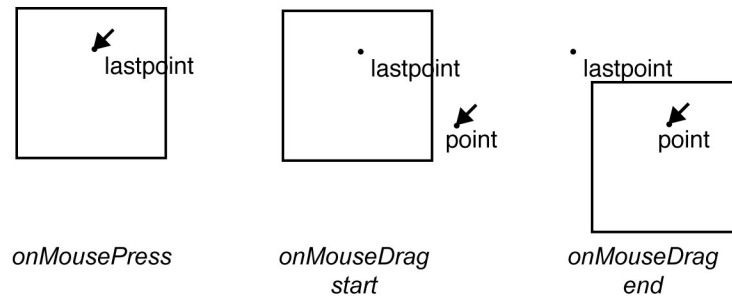


Figure 4.7: Three stages of dragging a rectangle.

later.

The second assignment in `onMousePress`,
`boxGrabbed := box.contains (point)`

determines and then remembers whether the box contained the point where the mouse was initially pressed. Because the result of evaluating `box.contains (point)` is a boolean value, `boxGrabbed` will represent either `true` or `false`. This value of `boxGrabbed`, which reflects whether the user actually pressed the mouse down inside `box`, will be used in `onMouseDown` to determine whether the `box` should be moved.

As usual, the `onMouseDown` method will be executed when the user drags the mouse. At that time the values of the variables `boxGrabbed` and `lastPoint` become relevant.

If the value of `boxGrabbed` is `true`, `box` will be moved by the distance between the last position of the mouse, saved in `lastPoint`, and the current position, held in `point`. The distance in each of the horizontal and vertical directions is needed to actually perform the move. The horizontal and vertical distances are obtained by evaluating `point.x - lastPoint.x` and `point.y - lastPoint.y`, respectively.

Figure 4.7 illustrates three stages of dragging a box in program `WhatADrag`. In the leftmost picture, the mouse button has been pressed with the mouse inside the rectangle. After the execution of `onMousePress`, the position of the mouse is stored in the instance variable `lastPoint`. In the middle picture, the mouse has been dragged down and to the right, and `onMouseDown` has just begun execution. The current position of the mouse is held in the parameter `point`, but the `if-then` statement has not yet been executed. The rightmost picture shows what has happened immediately after the `moveBy` message has been sent to `box` in the `if-then` statement during the execution of `onMouseDown`. The rectangle has been dragged to the right and down by the difference between the x coordinates and by the difference between the y coordinates of `point` and `lastPoint`. The update of the value of `lastPoint` to the position held in `point` is not shown.

Exercise 4.3.2 *Rather than using a variable to recall whether we clicked on the box, we might have written a condition that tested to see if `box` contained `point` each time the mouse*

was dragged as shown below.

```
method onMouseDrag (point: Point) -> Done {  
  if (box.contains (point)) then {  
    box.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)  
    lastPoint := point  
  }  
}
```

This would not quite work as desired. On the other hand, replacing the condition in the if-then statement with `box.contains(lastPoint)` would produce a program with exactly the same behavior as the version that used the variable `boxGrabbed`. Explain why the version using `point` as a parameter to `contains` would not work. In what circumstances and how does the other one behave differently?

Exercise 4.3.3 Suppose we replace the method `onMouseDrag` in program `WhatADrag` by the following code:

```
// if mouse is in box, then drag the box  
method onMouseDrag (point: Point) -> Done {  
  if (boxGrabbed) then {  
    box.moveTo (point)  
  }  
}
```

This is simpler than the code in Figure 4.6. For example, we no longer need to keep track of `lastPoint`. However, it does not result in as nice behavior. Explain why!

Exercise 4.3.4 Suppose the statement `lastPoint := point` inside of method `onMouseDrag` in Figure 4.6 was placed after the end of the if-then statement rather than inside the if-part. How would this change the appearance on the screen during execution?

4.4 Selecting Among Many Alternatives

The if-then-else statements discussed earlier in this section are particularly suitable when we have to choose between executing two different blocks of statements at some point in a program. However, sometimes there are more than two alternatives that have to be considered.

As a simple example, consider how to assign letter grades based on numeric scores on an examination. Scores greater than or equal to 90 are assigned an “A,” scores from 80 to 89 are assigned a “B,” scores from 70 to 79 are assigned a “C,” and those below 70 are assigned “no credit.” Suppose that a variable `score` contains the numeric score of a particular examination. We would like to display the appropriate letter grade in a text item, `gradeDisplay`. In this situation we have not just two, but *four* different possibilities to worry about, so it is clear that a simple if-then-else statement is insufficient.

Grace allows us to extend the if-then-else statement for more than two possibilities by including one or more `elseif` clauses in an if-then statement. Thus we can display the appropriate grade using the following statement:

```
if ( score >= 90 ) then {  
  gradeDisplay.contents := "The grade is A"
```

```

} elseif { score >= 80 } then {
  gradeDisplay.contents := "The grade is B"
} elseif { score >= 70 } then {
  gradeDisplay.contents := "The grade is C"
} else {
  gradeDisplay.contents := "No credit is given"
}

```

When we execute this if-then statement, the computer first evaluates the boolean expression `score >= 90`. If that is true, an “A” will be displayed and execution will continue after the last else-part of the statement. However, if it is false the program will evaluate the next boolean expression, `score >= 80`. If that is true, a grade of “B” will be displayed and execution will continue after the last else-part. If not, the expression `score >= 70` will be evaluated. If that is true then a grade of “C” will be displayed and execution will continue after the else-part. Otherwise, the statement in the else-part will be executed and a grade of “no credit” will be displayed.

When checking to see if the student should be given a “B,” why didn’t we also have to check whether `score < 90`? The reason is that to get to the condition `score >= 80`, the previous test, `score >= 90`, must have already *failed*. That is, we *only* execute the elseif clauses if `score` is less than 90. The same reasoning shows that we do not need to check if `score < 80` when we determine whether to give a “C” as the grade. We can always count on the fact that the conditions within the if-then-else statements are evaluated sequentially, and therefore that all previous tests in the if-then statement have failed before determining whether to execute the next block.

You have likely noticed that there is an odd-seeming inconsistency in the above example. The condition after the initial if is surrounded by parentheses, while the conditions after the elseifs are surrounded by curly braces. There is actually a good reason for this difference.

In Grace, an expression in curly braces is not evaluated if and until it is actually needed, and may be evaluated many times if it is necessary (we will see that this is important when we define looping constructs). When we evaluate an if statement we must always evaluate the condition following the if. However, we only evaluate the conditions after elseif if the earlier conditions were false. Thus we must surround those conditions by curly braces to make sure they are not evaluated if they aren’t needed.

This can be critically important in some cases. Consider the following example:

```

if (d == 0) then {
  ...
} elseif {(n / d) > 0} then {
  ...
}

```

Suppose the condition `(n / d) > 0` were evaluated every time the if statement was evaluated. If the value of `d` were 0, then evaluating the second condition would cause a division by 0, surely not what the programmer wanted.¹

To avoid these issues, Grace insists that the conditions following the elseif keyword be a block. That is, the boolean expression must be surrounded by curly braces.

¹As it turns out, dividing by 0 gives a result of infinity, which is actually a legal **Number**, but one that is pretty useless.

If you find it too hard to remember that an if needs a condition surrounded by parentheses and an elseif requires a condition surrounded by curly braces, then Grace will allow you to use curly braces around the if condition as well. However, we recommend that you use the conventions presented above as it will help you better understand the execution of your program.

We can summarize the execution of an if–then statement including elseif’s as follows:

- Evaluate the conditions after the if–then’s and elseif’s in order until one is found to be true.
- Execute the statements in the block following that if–then or elseif and then resume execution with the first statement after the entire if–then or elseif statement.
- If none of the conditions is true and there is an else–part, then execute the statements in the else–part. If there is no else–part, don’t execute any of the statements in the if–then statement.
- Finally, continue execution with the first statement after the if–then statement.

If there is an else clause in an if–then statement, then it must be the very last part of the if–then. That is, no further elseif’s are allowed after a plain else clause.

Exercise 4.4.1 *What is the output when the following pieces of code are executed? Why does the output of the three pieces of code differ? Assume hamburgerPrice is 7.*

- a.

```
if (hamburgerPrice < 2) then {
    print "This hamburger is super cheap."
} elseif {hamburgerPrice < 4} then {
    print "This hamburger is inexpensive."
} elseif {hamburgerPrice < 6} then {
    print "This hamburger is fairly inexpensive."
} elseif {hamburgerPrice < 8} then {
    print "This hamburger is moderately priced."
} elseif {hamburgerPrice < 10} then {
    print "This hamburger is pricey!"
} else {
    print "This hamburger is super expensive!"
}
```
- b.

```
if (hamburgerPrice >= 10) {
    print "This hamburger is super expensive."
} elseif {hamburgerPrice < 10} then {
    print "This hamburger is pricey!"
} elseif {hamburgerPrice < 8} then {
    print "This hamburger is moderately priced."
} elseif {hamburgerPrice < 6} then {
```

```

    print "This hamburger is fairly inexpensive."
} elseif {hamburgerPrice < 4} then {
    print "This hamburger is inexpensive."
} else {
    print "This hamburger is super cheap!"
}

```

c.

```

if (hamburgerPrice < 2) then {
    print ("This hamburger is super cheap.")
}
if (hamburgerPrice < 4) then {
    print ("This hamburger is inexpensive.")
}
if (hamburgerPrice < 6) then {
    print ("This hamburger is fairly inexpensive.")
}
if (hamburgerPrice < 8) then {
    print ("This hamburger is moderately priced.")
}
if (hamburgerPrice < 10) then {
    print ("This hamburger is pricey!")
}
if (hamburgerPrice >= 10) then {
    print ("This hamburger is super expensive!")
}

```

4.5 More on Boolean Expressions

The if–then–else statement in the Voting program at the beginning of this chapter was sufficient because there were only two candidates to consider when assigning a new vote. However, for more than two candidates the elseif clause introduced in the last section becomes necessary.

In the next example our program must choose between 3 candidates, A, B, and C, who each have been allotted a vertical third of the canvas. We will not rewrite the entire program here, but will instead focus on the `onMouseClicked` method.

Let `leftSeparator` and `rightSeparator` be constants representing the x coordinates of the vertical lines that divide the canvas into the three pieces. The variables `countA`, `countB`, and `countC` will keep track of the number of votes for the three candidates. Here is the code:

```

// Update votes and display vote counts for 3 candidates
method onMouseClick (point: Point) -> Done {
    if (point.x < leftSeparator) then { // clicked in left section
        countA := countA + 1
        infoA.contents := "So far there are {countA} votes for A."
    } elseif {point.x < rightSeparator} then { // clicked in center
        countB := countB + 1
        infoB.contents := "So far there are {countB} votes for B."
    }
    else { // clicked in right section
        countC := countC + 1
        infoC.contents := "So far there are {countC} votes for C."
    }
}

```

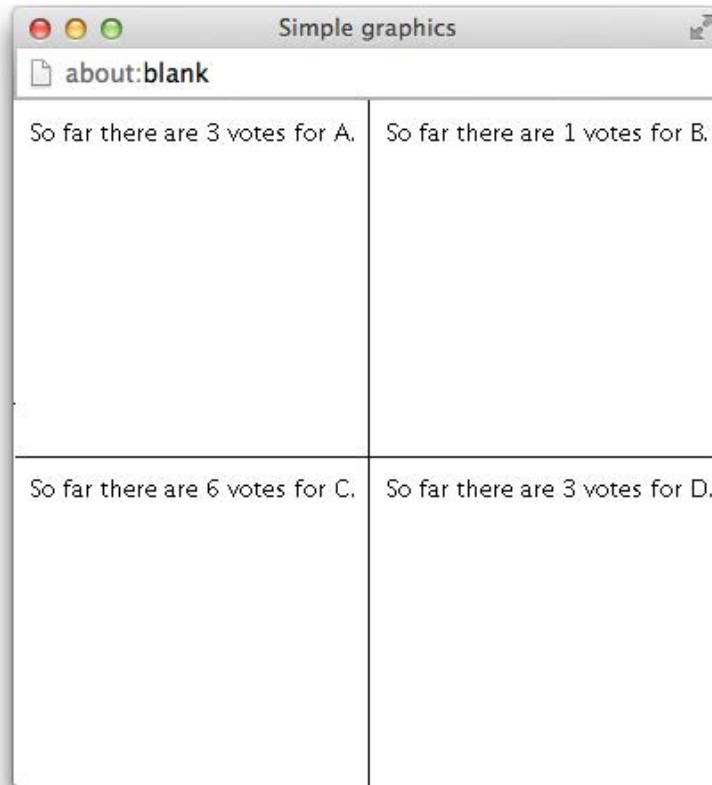


Figure 4.8: Voting for four candidates.

```
}
}
```

When determining whether the click was in the center section, why didn't we have to check that `point.x` was to the right of `leftSeparator`? As with our earlier example, the reason is that to even get to the second condition we *know* that the test `point.x < leftSeparator` must have failed (that is, it must have evaluated to *false*).

This same style solution works for determining whether clicks are in 4 or more vertical strips. However, once we get to four candidates it might make sense to divide the screen both vertically and horizontally rather than into 4 narrow, vertical strips as shown in Figure 4.8. We will count clicks in the upper-left hand corner as votes for A, in the upper-right hand corner as votes for B, lower-left for C, and lower-right for D. Thus to be a vote for A, the position where the user clicked must be *both* above the horizontal line *and* to the left of the vertical line. How can we write this as a condition?

In Grace we use the *and* operator, `&&`, between two boolean expressions to indicate that *both* must be true for the entire expression to be true. For example, we can ensure that `x` is positive and `y` is negative by writing `(x > 0) && (y < 0)`. While it would be convenient, computer programming languages do not usually allow us to combine two inequalities, as in the expression `1 <= x <= 10`. Instead we must write it out as `(1 <= x) && (x <= 10)`. *Note*

that all the parentheses are necessary as otherwise Grace wouldn't know how to group the operations.

The `&&` is an example of a logical, or boolean, operator in Grace. Just as the “+” operator on numbers takes two number values and returns a number, boolean operators take two boolean values and returns a boolean value. The `&&` (and) operator returns true exactly when both boolean values are true. Thus `(x > 0) && (y < 0)` will be true only if *both* `x` is greater than 0 *and* `y` is less than 0.

In order to ensure that point is in the upper-left corner of the canvas we would need to write:

```
if ((point.x < midX) && (point.y < midY)) then { // upper-left
    ...
}
```

The if-part is only executed if *both* `point.x < midX` *and* `point.y < midY`. If either one of those boolean expressions is false, then the entire condition evaluates to false and the if-part is not executed.

Here is the complete code to assign votes to four candidates:

```
// Update votes and display vote counts
method onMouseClick(point: Point) -> Done {
  if ((point.x < midX) && (point.y < midY)) then {
    countA := countA + 1
    countAText.contents := "So far there are {countA} votes for A."
  } elseif {(point.x >= midX) && (point.y < midY)}then {
    countB := countB + 1
    countBText.contents := "So far there are {countB} votes for B."
  } elseif {(point.x < midX) && (point.y >= midY)} then {
    countC := countC + 1
    countCText.contents := "So far there are {countC} votes for C."
  } else {
    countD := countD + 1
    countDText.contents := "So far there are {countD} votes for D."
  }
}
```

Just as Grace uses `&&` to represent the logical *and* operation, it uses `||` to represent the logical *or* operator. Thus `(x < 5) || (y > 20)` will be true if `x < 5` *or* `y > 20`. In general, if b_1 and b_2 are boolean expressions, then $b_1 || b_2$ is true if *one* or *both* of b_1 and b_2 are true.

English contains both an *inclusive* and an *exclusive* “or”. The inclusive “or” evaluates to true if either or both operands are true. The exclusive “or” evaluates to true only if exactly one of the operands is true. The `||` operator represents the inclusive “or”. You can get the effect of the exclusive “or” on boolean expressions `b1` and `b2` by writing `(b1 || b2) && !(b1 && b2)`, which states that one of `b1` and `b2` is true, but not both. (As we explain below `!` stands for negation.) Even simpler, but perhaps not as apparent, we could alternatively just write `b1 != b2`.

A good example illustrating the use of the *or* operation is determining whether someone playing the game of craps has won or lost after his or her first roll of the dice. The “shooter” in craps throws two dice. If the numbers on the faces of the dice add up to 7 or 11, then the shooter wins. A sum of 2, 3, or 12 results in an immediate loss. With any other result, play

operator	meaning
<code>&&</code>	<i>and</i>
<code> </code>	<i>or</i>
<code>!</code>	<i>not</i>
<code>==</code>	<i>equal</i>
<code>!=</code>	<i>not equal</i>
<code><</code>	<i>less than</i>
<code><=</code>	<i>less than or equal</i>
<code>></code>	<i>greater than</i>
<code>>=</code>	<i>greater than or equal</i>

Figure 4.9: A summary of the boolean and comparison operators in Grace

continues in a way that will be described later.

The if–then–else statement below has three branches that encode the relevant outcomes of the first roll of the dice. The value of `status` is simply a text object that, as usual, has been created in the object.

```

if ((roll == 7) || (roll == 11)) then {           // 7 or 11 wins on first throw
    status.contents := "You won!"
} elseif {(roll == 2) || (roll == 3) || (roll == 12)} then { // 2, 3, or 12 lose on 1st throw
    status.contents := "You lose!"
} else {                                         // play must continue
    status.contents := "The game continues"
}

```

The if–then portion determines if the player has won by checking if the roll was 7 or 11. The elseif portion determines if the player has lost by checking if roll was 2, 3, or 12. Finally the else portion is executed if further rolls of the dice will be required to determine whether the player wins or loses.

The last boolean operator to be introduced here is `!`, which stands for “not”. For example, the expression `!box.contains (point)` will be true exactly when `box.contains (point)` is false, *i.e.*, when `box.contains (point)` is *not* true.

The operator `!` has higher priority than `&&` and `||`. Thus `!a && b` means the same as `(!a) && b`. That is, first the negation of `a` is taken and that is “anded” with the value of `b`.

Although we can use `!` with equations and inequalities, it is usually clearer to rewrite the statement using a different operator. For instance, `!(x == y)` is more simply written as `x != y`, and `!(x < y)` is simplified to `x >= y`. Similarly, `!(x <= y)` is equivalent to `x > y`, which is much easier to read.

Figure 4.9 summarizes the most common operators in Grace that give a boolean result.

Exercise 4.5.1 Suppose that the current value of x is 6, y is -2, and z is 13. For each of the following conditions, determine whether they evaluate to true or false.

- $((x - 6) < y) \ \&\& \ (z == (2 * x + 1))$.
- $(!(x - 6 < y) \ \&\& \ (z == (2 * x + 1)))$.

- c. $((x - 6) < y) \parallel (z == (2 * x + 1))$.
- d. $!(((x - 6) < y) \parallel (z == (2 * x + 1)))$.

4.6 Nested Conditionals

Occasionally we run into problems that would require complex boolean conditions if they are handled using if-then or elseif statements as we have seen them used so far. Rather than constructing these complex conditions, we will introduce alternative structures for supporting the program logic. Happily, we don't need to introduce any more syntax in order to handle them; we just need to combine if-then statements in different ways.

Suppose it is a summer weekend and you are trying to figure out what to do. Your choice of recreation will depend on the weather and how much money you have to spend. The following table lists the various options and choices, where the row headings represent your possible financial situations and the column headings represent the weather possibilities.

	sunny	not sunny
rich	outdoor concert	indoor concert
not rich	ultimate frisbee	watch TV

The table entries represent the suggested recreational activity given the financial situation represented by the row and the weather as represented by the column. Thus if you are feeling rich and it is not sunny, you might want to go to an indoor concert. If you are not feeling rich and it is sunny, you might play some ultimate frisbee.

How can we represent these choices with an if-then statement? Let `rich` and `sunny` be variables of type `boolean`, and let `activityDisplay` be a variable of type `Text` that will display the selected activity. The if-then statement below uses `elseif` clauses to represent the four choices.

```

if (sunny && rich) then {
    activityDisplay.contents := "outdoor concert"
} elseif {!sunny && rich} then {
    activityDisplay.contents := "indoor concert"
} elseif {sunny && !rich} then {
    activityDisplay.contents := "ultimate frisbee"
} else { // !sunny && !rich
    activityDisplay.contents := "watch TV"
}

```

As noted earlier, `!` has *higher precedence* than `&&`. Thus the condition `!sunny && rich` is evaluated by first evaluating `!sunny` and then using the `&&` operation to determine whether both `!sunny` and `rich` are true.

This code correctly represents all four options, but is rather verbose and loses the nice structure of the table. A related problem is that by the time we arrive at the last case the program has evaluated three fairly complex boolean expressions.

We can write this so that only two evaluations of boolean variables are ever made, and furthermore that they are made without the added complication of *negation* or *and* operators. This is accomplished by *nesting* if-then statements. A nested if-then statement involves including one or more if-then statements inside another, as in the example below.

```

if (sunny) then {
  if (rich) then {
    activityDisplay.contents := "outdoor concert"
  }
  else { // not rich
    activityDisplay.contents := "ultimate frisbee"
  }
}
else { // not sunny
  if (rich) then {
    activityDisplay.contents := "indoor concert"
  }
  else { // not rich
    activityDisplay.contents := "watch TV"
  }
}

```

The advantage to using these nested if–then statements is that the organization is quite similar to that of the table. There is an outer if–then–else statement that determines whether sunny is true. This corresponds to choosing either the first or second column of the table. Inside the outer if–part there is an if–then–else statement that determines whether rich is true. This corresponds to figuring out which row of the table applies. For example, if rich is false, the outcome should correspond to the first column and second row of the table, and hence the activity should be “play ultimate”. The else–part corresponding to it not being sunny is handled in a similar fashion.

Style Note: The nested if–then–else statements are indented from the outer ones in order to make the code easier to read and understand. Grace compilers enforce this indenting because human readers generally need the cues of indenting to understand complex code like this.

Aside from the indenting, another thing that makes this code easy to understand is the inclusion of comments. In particular, notice how each else clause includes comments indicating under which conditions the else–part is executed. This has the advantage of making it absolutely clear to the reader under what circumstances this code is executed. The more complex the conditional, the more important these comments become. We strongly urge all programmers to include such comments.

While the use of nested if–then–else statements yields code that is not quite as compact and simple to understand as the table, it is much easier to see its correspondence to the table than the version involving only elseif clauses. Notice in particular that no matter what the values of sunny and rich are, only two boolean variables are ever evaluated during the execution of this code, so it is not only clearer, but it is faster as well!

Exercise 4.6.1 Use nested conditionals to rewrite the `onMouseClicked` method for tabulating votes of four candidates from Section 4.5.

Exercise 4.6.2 Using the information provided in the table and nested conditionals, write an if-statement that displays which course a student should take based on their interests in math and writing. Let `likesMath` and `likesWriting` be variables holding boolean values, and let `course` refer to a text object that will display the recommended course.

	<i>likes writing</i>	<i>doesn't like writing</i>
<i>likes math</i>	<i>Economics</i>	<i>Calculus</i>
<i>doesn't like math</i>	<i>English</i>	<i>Psychology</i>

In Figure 4.10 we provide another example of complex choices being represented by nested if–then–else statements. This program provides the code to simulate a complete game of craps. The rules of craps are as follows:

The shooter rolls a pair of dice. If the shooter rolls a 7 or 11, it is a win. If the shooter rolls a 2, 3, or 12, it is a loss. If the shooter rolls any other number, that number becomes the “point”. To win, the shooter then must roll the “point” value again before rolling a 7. Otherwise it is a loss.

The program simulates a roll of the dice by using a random number generator every time the user clicks the mouse. In order to implement the rules given above, we must organize the game logic in a way that can be represented using if–then statements. Notice that the rules for winning are quite different depending on whether this is the player’s first throw. For instance, if it is the first throw, then rolling a 7 results in a win, but if it is a second or subsequent throw, then 7 results in a loss. Therefore we will organize the first level of conditional to determine whether it is the first throw.

In order to make such a choice, the Craps program in Figure 4.10 declares a variable, `newGame`, to remember whether this is the first throw of a new game.

The outer if–then statement in the method `onMouseClicked` has the following structure:

```
if (newGame) { // starting a new game
    ...
} else { // continuing trying to make the point
    ...
}
```

The if–part of this code is itself an if–then statement with three branches, each of which encodes the relevant actions to be taken based on the first roll of the dice. Recall that we saw a simplified version of this example earlier in the chapter. The new code is reproduced below:

```
if ((roll == 7) || (roll == 11)) then { // 7 or 11 wins on first throw
    status.contents := "You won!"
} elseif {(roll == 2) || (roll == 3) || (roll == 12)} then { // 2, 3, or 12 lose on 1st throw
    status.contents := "You lose!"
} else { // Set the roll as the new point to be made and continue game
    status.contents := "Try for your point!"
    point := roll
    newGame := false // set for continuing game
}
```

Rather than having a separate branch for each possible value of the roll of the dice, there are only three. These branches correspond to winning, losing, and establishing a point to be made on subsequent rolls. The variable `newGame` remains true in the first two branches, so it need not be updated. Only the third branch requires setting `newGame` to false.

Let us now examine the else–part of the outer if–then statement. Like the if–part, this nested if–then statement also has 3 branches, though the second and third conditions are quite different from those that handle the first roll:


```

dialect "objectdraw"
import "random"

object {
  inherit graphicApplicationSize (400 @ 400)

  var newGame: Boolean := true // True if starting new game
  var point: Point // number trying to roll to win

  def maxVal: Number = 6 // max number on die

  // display instructions
  def welcomeMessage: Text = textAt (110 @ 50) with ("Let's play craps!") on (canvas)
  textAt (100 @ 90) with ("Click to roll the dice") on (canvas)

  // status of game
  def status: Text = textAt (100 @ 70) with ("") on (canvas)
  var roll: Number // total score for a roll of two dice

  // For each click, roll the dice and report the results
  method onMouseClick( pt: Point ) -> Done {
    // get values for both dice and display sum
    roll := rand.integerIn (1) to (maxVal) + rand.integerIn (1) to (maxVal)
    welcomeMessage.contents := "You rolled a {roll}!"

    if (newGame) then { // starting a new game
      if ((roll == 7) || (roll == 11)) then { // 7 or 11 wins on first throw
        status.contents := "You won!"
      } elseif {(roll == 2) || (roll == 3) || (roll == 12)} then { // 2, 3, or 12 lose on 1st throw
        status.contents := "You lose!"
      } else { // Set the roll as the new point to be made and continue game
        status.contents := "Try for your point!"
        point := roll
        newGame := false // set for continuing game
      }
    } else { // continuing trying to make the point
      if (roll == 7) then { // 7 loses when trying for point
        status.contents := "You lose!"
        newGame := true // set to start new game
      } elseif {roll == point} then { // making the point wins!
        status.contents := "You won!"
        newGame := true
      } else { // keep trying
        status.contents := "Keep trying for {point} ..."
      }
    }
  }
}

startGraphics
}

```

Figure 4.10: Craps program illustrating nested conditionals.

```

if (roll == 7) then {           // 7 loses when trying for point
    status.contents := "You lose!"
    newGame := true           // set to start new game
} elseif {roll == point} then { // making the point wins!
    status.contents := "You won!"
    newGame := true
} else { // keep trying
    status.contents := "Keep trying for {point} ..."
}

```

In this statement, both of the first two choices result in setting `newGame` back to `true` because they represent the end of a game with either a win or a loss. The third statement merely asks the player to continue rolling, so `newGame` remains `false`.

Exercise 4.6.3 *Try writing out this program using only a single if–then statement with many elseif clauses. It should become painfully clear why nested if–then statements are useful in situations with complex logic.*

4.7 Short-circuit boolean operators

Suppose we have a boolean expression of the form `a && b` or `a || b`. When these expressions are evaluated, both `a` and `b` will be evaluated in order to determine whether the result is true or false. This can occasionally lead to unfortunate surprises. For example, consider the following code

```

if ((denom != 0) && ((num / denom) > 1)) then {
    ...
}

```

If the value of `denom` is 0 then execution of this code will cause a crash because it will result in dividing `num` by 0. Instead you would need to write

```

if (denom != 0) then {
    if ((num / denom) > 1) then {
        ...
    }
}

```

to avoid performing the division if `denom` is 0.

To avoid this more awkward construction, Grace also provides “short-circuit” Boolean operators. All that is necessary is to surround the second argument in these expressions in curly braces, making it into a block:

```

if ((denom != 0) && {(num / denom) > 1}) then {
    ...
}

```

When you evaluate an expression like `b1 && {b2}`, first `b1` is evaluated. If it is false, then the expression returns false. If it is true then `b2` is evaluated and the value of the expression is the value of `b2`.

Notice that this is exactly what we want for the expression `(denom != 0) && {(num / denom) > 1}` as the second part will only be evaluated if `denom` is different from 0.

The short-circuit `||` works similarly. When `b1 || {b2}` is evaluated, first `b1` is evaluated. If it is true, then the expression returns true. If it is false then `b2` is evaluated and the value of the expression is the value of `b2`.

As a result Grace will cease evaluating an expression involving one of these expressions as soon as it can determine whether the entire expression is true or false.

Here is another example. Suppose a program includes a declaration of a variable `x` representing a number, and that it contains the expression `(x > 10) && {x <= 20}`. If the computer evaluates that expression when `x` has value 3, it will only evaluate `x > 10` without even considering `x <= 20`. Because `x > 10` is false, the expression will first evaluate to **false** `&& {x <= 20}`. As a result, Grace can determine that the final value of the `&&` expression must be false no matter what the value of `x <= 20`. However, if `x > 10` had been true, the rest of the expression would have to be evaluated to determine whether the entire `&&` expression evaluates to true or false.

Exercise 4.7.1 *Explain why the meaning of `||` given above provides an efficient evaluation of the boolean “or” operation.*

4.8 if-then-else expressions

In this chapter we have used if-then-else statements in order to make choices about what code to execute. In this section we show how we can use if-then-else as an *expression* to return values which depend on conditions.

Suppose you want to calculate the absolute value for a `Number`. The definition is as follows: the absolute value of `x` is `x` if `x >= 0` and `-x` otherwise. We can write this with an if-then-else statement as follows:

```
var absValue: Number
if (x >= 0) then {
  absValue := x
} else {
  absValue := -x
}
```

This requires us to first declare the value and then assign it a value later. However, if we use an if-then-else expression then we can declare and initialize the variable at the same time.

```
var absValue: Number := if (x >= 0) then { x } else { -x }
```

The differences in the new code are that if-then-else occurs on the right-hand side of the assignment statement, where a value would normally appear. Also in the `then` and `else` branches, we have an expression rather than a statement. The meaning of this is exactly what you would expect. If `x >= 0` then the value of the expression is `x` while otherwise the value is the value of `-x`. Whichever of those is the value of the if-then-else will be the value assigned to `absValue`.

The code included in the curly braces after `then` and `else` can include several statements as long as the last line in each of those blocks is an expression corresponding to the value of the entire if-then-else. For example, we can modify the preceding example so that it prints out information about the value to help the programmer understand their code better:

```

var absValue: Number :=
  if (x >= 0) then {
    print "{x} is non-negative, resulting in {x} as value"
    x
  } else {
    print "{x} is negative, resulting in {-x} as value"
    -x
  }

```

In this case, `absValue` would get the exact same value as before for a given value of `x`, but it would also print out a message such as: `47 is non-negative, resulting in 47 as value` if the value of `x` is 47, or `-47 is negative, resulting in 47 as value`, if `x` is -47. These `if-then-else` expressions can also include `elseif` clauses as well. The only restriction is that each of the conditional blocks ends with an expression of the same type.

While you will not need the `if-then-else` expression often, there are times when it is extremely handy. For example, if `absValue` was in a `def` statement rather than a `var` declaration then using `if-then-else` makes initializing it much simpler.

4.9 Summary

In this chapter we introduced conditional statements and the Boolean data type. The major points discussed were:

- The `if-then-else` statement is used when different code is to be executed depending on the value of a condition.
- `if-then` statements without an `else` clause are used when extra code is to be executed in one case, but nothing extra is needed in the other.
- `if-then` statements with `elseif` clauses are used if there are more than two cases to be considered in a choice. `if-then` statements with `elseif` clauses may or may not be terminated with an `else-part`, at the programmer's option.
- Conditions after `elseif` must be surrounded by curly braces, which indicate that they will be evaluated only if necessary. Conditions after `if` should normally be surrounded by parentheses (though curly braces are allowed).
- Nested `if-then` statements can be used to represent complex logic.
- Identifiers `true` and `false` represent boolean values. Comparison operators return boolean values. Boolean expressions can be combined with the boolean operators `&&`, `||`, and `!`, as well as `==` and `!=`.
- The `if-then-else` statement can be used when different values are to be returned from an expression that depend on a boolean value.

4.10 Chapter Review Problems

Exercise 4.10.1 Define the following Grace terms:

- a. *negation*
- b. *block*
- c. *!=*
- d. *condition*

Exercise 4.10.2 How would you express the following as operators in Grace?

- a. *greater than*
- b. *or*
- c. *equal*
- d. *less than or equal*
- e. *not*
- f. *and*
- g. *not less than*

Exercise 4.10.3 The database at a doctor's office stores information about each patient. For each piece of information listed below, decide whether or not it would be appropriate to store the information as a boolean.

- a. *patient's birthday*
- b. *patient's gender*
- c. *patient's address*
- d. *whether the patient has visited within the last year*
- e. *patient's insurance company*
- f. *whether the patient's last visit is fully paid*

Exercise 4.10.4 Fix the problem(s) in the following code:

```
if (x = 5) then {  
  message.contents := "You win!"  
} elseif (x < 5) then {  
  message.contents := "You lose!"  
} elseif (x > 5) then {  
  message.contents := "Try again!"  
}
```

Exercise 4.10.5 What are the values of the following expressions if $x = 8$, $y = -5$ and $z = 2$.

- a. $(x + y) != z$
- b. $!((2 * x + 3 * y) >= z)$
- c. $((x - z) * 2) < (z - 2 * y)$
- d. $(y - z + x) >= 0$
- e. $(x - 4 * z + 1) == y$

Exercise 4.10.6 Why is the following code more complex than it needs to be? Simplify it using `elseif`

```

if ((score <= 100) && (score >= 90)) then {
    gradeDisplay.contents := "You got an A"
}
if ((score < 90) && (score >= 80)) then {
    gradeDisplay.contents := "You got a B"
}
if ((score < 80) && (score >= 70)) then {
    gradeDisplay.contents := "You got a C"
}
if (score < 70) then {
    gradeDisplay.contents := "You don't get a grade"
}

```

Exercise 4.10.7 What are the values of the following expressions if $x == -2$, $y == -1$ and $z = 4$.

- a. $((x + y) < z) || (((4 * y + z) > x) \&\& (x > y))$
- b. $((x + y) < z) || ((4 * y + z) > x) \&\& (x > y)$
- c. $!((x + y) < z) || (((4 * y + z) > x) \&\& (x > y))$
- d. $((x + y) < z) || ((4 * y + z) > x) \&\& !(x > y)$
- e. $!((x + y) < z) || (((4 * y + z) > x) \&\& !(x > y))$

Exercise 4.10.8 Assume that $x = 2$, $y = -3$ and $z = 5$. What are the values of x , y and z after the following code has been executed?

- a.


```

if ( (3 * x + y) <= (z - 1) ) then {
    x := y + 2 * z
} else {
    y := z - y
    z := x - 2 * y
}

```

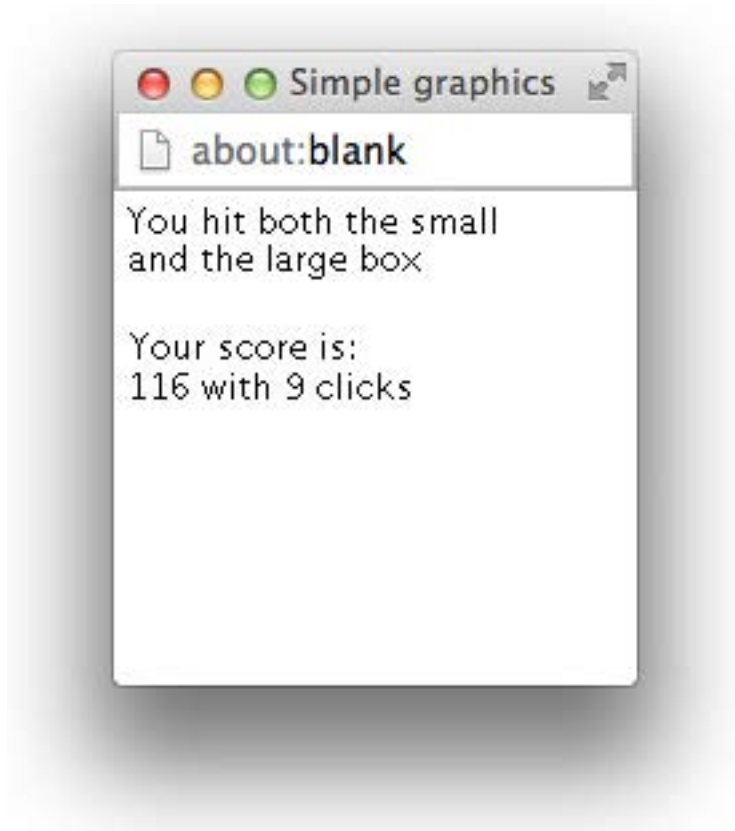


Figure 4.11: Display for InvisibleBox

```
if ( x > ( y + z ) ) then {  
  y := y - 1  
  x := x + 1  
}
```

- b. **Exercise 4.10.9** *Another variation of the game craps is to play what is called the “No Pass Line”. Now rolling a seven or eleven on the first roll loses and a two, three, or twelve wins. After the point is set, rolling a seven wins, while rolling the point again loses. Show how to modify Craps so that you now are playing the “No Pass Line” instead.*

4.11 Programming Problems

Exercise 4.11.1 *Create the game InvisibleGame to master your use of conditionals. The game will have the following features:*

- *When the game begins it will create three “invisible” boxes. The boxes will all be square but of different sizes. One should be 30 pixels wide, the second one 45 pixels wide and*

the third 80 pixels wide. These boxes should be created randomly anywhere on the screen.

- *The user will click the mouse and try to hit the boxes when doing so.*
- *When the mouse exits the window, the user will be notified of his/her success. The output should look like Figure 4.11.*
- *When the mouse re-enters the window, all variables should be reset and the boxes moved to new random positions.*
- *Points are assigned as follows*
 - *200 points: Hitting all three boxes*
 - *150 points: Hitting the small box and the medium box*
 - *125 points: Hitting the small box and the large box*
 - *110 points: Hitting the medium box and the large box*
 - *100 points: Hitting only the small box*
 - *75 points: Hitting only the medium box*
 - *50 points: Hitting only the large box*
 - *-1 point: For each mouse click*
- *Set if up so that the window is 300 pixels wide and 300 pixels high.*

Exercise 4.11.2 *Write a simple game with three dice called Dicey, which may remind you of a more popular game with five dice. When the user clicks the mouse, the three dice are “rolled” and the results are displayed on the screen. The display will also include whether the user rolled 3 of a kind, a pair or nothing of particular interest. See Figure 4.12*

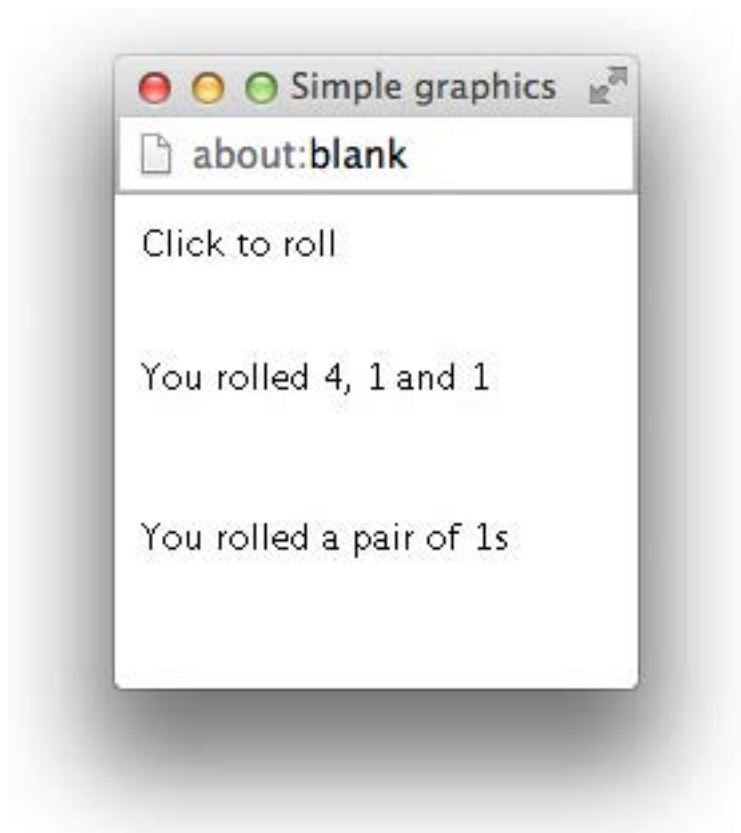


Figure 4.12: Display for Dicey

Chapter 5

Types, Numbers, and Strings

The programs that we are now writing in Grace are starting to get complex enough now that it will be useful to get help from the computer in writing correct programs. We all make mistakes when we are doing things, so it will help to anticipate that eventuality and have assistance ready.

It probably won't surprise you to know that it is easier to fix errors when they are discovered early. Many of us have put together lots of furniture pieces and children's toys. If we put something in wrong early on, but don't discover the error until we're nearly done, we generally have to take everything apart and essentially start over. However, if we recognize the error right away, it is much easier to fix the problem and complete the task.

Earlier we introduced the notion of type annotations in order to make it easier to understand and use code you or others have written. In this chapter, we will discuss in more detail the notion of type annotations. As we've seen, these type annotations not only make it easier for a reader to figure out what your program is trying to do, but they will also be machine checkable so that you can be warned if the Grace language processor detects any inconsistencies in your program. Especially helpfully, this machine check can happen before your program is executed, helping you to find errors faster and typically providing better information on where the error occurs.

In this chapter, we will discuss three kinds of types in some detail. We begin by discussing the types of the geometric objects introduced in the `objectdraw` library. After that we will provide more information about the built-in types `Number`, `String`, and `Boolean` that were introduced earlier.

5.1 What is a type?

Back in section 2.1.5 we introduced the notion of type annotations. In Grace, a type is a specification of the methods that an object can respond to. It is also useful to think of a type as the collection of all objects that can respond to the methods specified by the type.

You have already seen built-in types like `Number`, `Boolean`, and `String`. We'll come back and talk about the methods in those types, but let's first consider the types of objects from the `objectdraw` library.

```

type Point = {
  // coordinates of point
  x -> Number
  y -> Number

  // return new point shifted by other's x and y components
  + (other: Point) -> Point

  // return new point shifted backwards by other's x and y components
  - (other: Point) -> Point

  // return distance from this point to other
  distanceTo (other: Point) -> Number

  // return if other at same place as self
  == (other: Point) -> Boolean

  // return string representation of point
  asString -> String
}

```

Figure 5.1: Point type from objectdraw

5.2 The type Point

In section 2.2.2 we specified the operations available on points. These operations are specified formally in the Grace built-in type `Point`, which is shown in Figure 5.1

Each of the items in the `Point` type is the specification of a method supported by points. Thus the first two items specify that an object of type `Point` has methods `x` and `y` that each return a number. The next two items specify that there are methods `+` and `-` that add (respectively subtract) the components of a point and return an object that is another `Point`.

In general, types are defined in declarations of the following form:

```

type NewType = {
  ...
  m (p1: T1,...,pn: Tn) -> RType
  ...
}

```

In this declaration, `NewType` is the name of the type being defined. Each line of the type expression specifies a method of the type.

In the example above, `m` is the name of the method. It takes `n` arguments or parameters. Each parameter is specified with a name and its associated type. Above, parameter `p1` has type `T1`. A “:” is used to indicate the association of identifier and type. If there is more than one parameter, the parameters (and their types) are separated by commas, and the entire collection of parameters is surrounded by parentheses. (If a method has no parameters then the parentheses are omitted.) Finally the type of value returned by the method is placed after the parameters, separated from the parameters by “->” representing an arrow.

Going back to the example of the type `Point`, we can see that methods `x` and `y` take no

parameters, but always return a number. Thus if `locn` is an identifier of type `Point`, then both `locn.x` and `locn.y` are legal expressions, and when evaluated should return numbers (assuming that `locn` has been initialized properly).

The method `distanceTo` specified in `Point` takes one parameter of type `Point` and has return type `Number`. Thus if `locn1` and `locn2` are associated with points, then `locn1.distanceTo(locn2)` is a legal expression that will return a number representing the distance between the two points.

The methods `==` and `asString` are included in all types. If `loc1` and `loc2` have type `Point` then `loc1 == loc2` will be true if and only if the two points have the same x and y coordinates. If `loc` has type `Point`, then `loc.asString` will return a string of the form `"(xc,yc)"` where `xc` represents the x coordinate of `loc` and `yc` represents the y coordinate.

5.3 Types from `objectdraw`

All of the graphic items from the `objectdraw` library *satisfy* the type `Graphic` shown in Figure 5.2. That is, they have all of the methods specified in type `Graphic`. Reading through the methods specified by the type we see that there are methods to retrieve the x and y coordinates of the graphics object as well as its location.

The next two methods in `Graphic` are used to retrieve and reset whether the object is visible on the canvas. Method `isVisible` returns true if and only if the object is currently visible, while method `visible:=` can be used to reset the visibility of the object. Thus `gobj.visible := true` sets it so that `gobj` is visible on the canvas, while `gobj.visible := false` makes it invisible. The `visible:=` method does not return a value, it just resets the visibility of the object. As a result we annotate the return type to be `Done`, indicating that nothing further will be done with the result. Because most mutator methods don't return values, they generally will have a return type of `Done` as well.

Methods like `visible:=` that end with `:=` are treated specially in Grace. Unlike regular methods, a programmer may insert one or more spaces before the `:=`. Second, the parameter of the method need not be surrounded by parentheses.

`Graphic` has two methods to move the object, `moveTo`, which takes a `Point` as a parameter, and `moveBy`, which takes two numbers indicating how far it should move to the right and down. Both have return type `Done` as they don't return a value. The method `contains`, which takes a value of type `Point` as a parameter determines whether that point is contained in the geometric object. The methods `color` and `color:=` are like `visible`, giving us access to or the ability to change the color of the object.

The methods `sendForward`, `sendBackward`, `sendToFront`, and `sendToBack` are used to change the relative order of objects on the canvas. They take no parameters and return `Done`. As usual, `asString` returns a string that describes the object. The final two methods are used to either add the graphic object to a canvas or remove it.

5.4 Specializing types

The methods in `Graphic` are available to all graphic objects in the `objectdraw` library, such as framed rectangles, filled ovals, text, and lines, but these objects generally also contain other

```

type Graphic = {
  // where object is on screen
  x -> Number
  y -> Number
  location -> Point

  // Is this object visible on the screen?
  isVisible -> Boolean

  // Determine if object is shown on screen
  visible:= (isVisible: Boolean) -> Done

  // move this object to newLocn
  moveTo (newLocn: Point) -> Done

  // move this object dx to the right and dy down
  moveBy (dx: Number, dy: Number) -> Done

  // Does this object contain locn
  contains (locn: Point) -> Boolean

  // set the color of this object to c
  color:= (c: Color) -> Done

  // return the color of this object
  color -> Color

  // Send this object up one layer on the screen
  sendForward -> Done

  // send this object down one layer on the screen
  sendBackward -> Done

  // send this object to the top layer on the screen
  sendToFront -> Done

  // send this object to the bottom layer on the screen
  sendToBack -> Done

  // Return a string representation of the object
  asString -> String

  // Add this object to canvas c
  addToCanvas (c: DrawingCanvas) -> Done

  // Remove this object from its canvas
  removeFromCanvas->Done
}

```

Figure 5.2: The type Graphic from objectdraw

```

type Text = Graphic & type{
  // return the string shown in the text object
  contents -> String

  // reset the string shown in the text object to s
  contents:= (s: String) -> Done

  // return the width of the text object
  width -> Number

  // return the font size of the text object
  fontSize -> Number

  // reset the font size of the text object
  fontSize:= (size: Number) -> Done
}

```

Figure 5.3: The type Text from the objectdraw library

```

type Line = Graphic & type {
  // start and end of line
  start -> Point
  end -> Point

  // set start and end of line
  start:= (start!: Point) -> Done
  end:= (end!: Point) -> Done
  setEndpoints (start!: Point, end!: Point) -> Done
}

```

Figure 5.4: The type Line from the objectdraw library

specialized methods. For example, objects created via `text` have type `Text`, shown in Figure 5.3.

The `&` in the first line of the type definition indicates that `Text` contains all of the methods in type `Graphic` as well as those listed after the keyword `type`. In this case we can access the width of the text object as well as get and set the contents of the string and the `fontSize` of the string displayed on the canvas.

Lines and two-dimensional graphic objects will have somewhat different types. For example, lines have endpoints, while rectangles and ovals have length and width. The definition of `Line` is in Figure 5.4. Objects generated from `line` will satisfy this type.

Finally all of the two-dimensional graphics objects, including both filled and framed rectangles and ovals will satisfy the type `Graphic2D`, which is provided in Figure 5.5. Not surprisingly, they support methods obtaining and setting the width and height of the objects. The methods `size` and `size :=` allows both the width and height to be reset simultaneously. The last method, `overlaps`, returns true if and only if the receiver and the parameter overlap.¹

¹ An important proviso: `overlaps` returns true if the bounding rectangles of the objects overlap. Thus, it may return true for two ovals even if they don't overlap, but their bounding rectangles do.

```

type Graphic2D = Graphic & type {
  // dimensions of object
  width -> Number
  height->Number
  size -> Point

  // Change dimensions of object
  width:= (width: Number) -> Done
  height:= (height: Number) -> Done
  size:= (dimension: Point) -> Done

  // Does other overlap with this object
  overlaps (other: Graphic2D) -> Boolean
}

```

Figure 5.5: The type Graphic2D from the objectdraw library

```

// DrawingCanvas holding graphic objects
type DrawingCanvas = {
  // clear the canvas
  clear -> Done

  // return the current dimensions of the canvas
  width -> Number
  height -> Number

  ...
}

```

Figure 5.6: The type DrawingCanvas

Last, but not least, we have the type DrawingCanvas of the canvas that we do all of our drawing on. A partial listing of the type is given in Figure 5.6. We have left out methods that are unlikely to be used in this text. All the details may be found in the appendix.

You have already seen methods that clear the canvas and that request the current width and height of the canvas. The canvas is mainly used as a place to display graphic objects, so the classes generating them all have a parameter of type DrawingCanvas.

5.5 Type compatibility

If we have a variable of type Graphic then we can assign objects generated from both text and filledRect to it, even though objects generated from text have type Text and those generated from filledRect have type Graphic2D. The reason is that those objects support all of the methods specified in type Graphic. The fact that they support more methods than those specified in Graphic is not a problem. The same is true for **defs** and formal parameters in Grace.

We can choose to give an incomplete specification of the methods supported by an object. This is especially handy when we have a variable that may be associated with a variety of different kinds of objects. As long as we specify a type that is compatible with all the objects,

we won't have any difficulties.

We say an object type B extends object type A if B includes (at least) all of the methods in A and the parameter and return types of those methods are the same as in A. We can abbreviate this relation as $B <: A$.

In other words, B extends A if it contains all the methods in A plus possibly some extras. For example, the types `Text` and `Graphic2D` both extend `Graphic` because each has all the methods in `Graphic` and the formal parameter types and return types of the methods in `Graphic` are the same as those in `Text` and `Graphic2D`.²

We do not insist that the formal parameter names are exactly the same – only that their types are. This is because formal parameters just provide ways to talk about the run-time arguments to the methods, and the names can be changed without affecting the meaning of the method.

Clearly if we give a legal type definition

```
type B = A & type {...}
```

then B will extend A because the definition provides it explicitly with everything in A. However, it is not necessary to define B in this way in order to get a type extending A.

In general, if an expression has a type B, and B extends A, then we may safely treat the expression as if it also has type A. In particular, we can use that expression on the right side of a `def` or `var` declaration of an identifier with type A, and we may use the expression as an argument for a method's formal parameter which is declared to have type A.

More informally, if B extends A, then an element of type B can masquerade as an element of type A, as it can do anything that an element of type A can. The element of type B can be used in any definition, assignment, or as a method argument in any place that expects an element of type A.

5.6 Numbers

Most programming languages have different types of numbers. For example, many languages have different types for integers and numbers with fractional parts like 3.14159. In Grace we don't distinguish between these types and just have the single type `Number`, which represents both integers and numbers with fractional parts.

Most of the methods of type `Number`, presented in Figures 5.7 and 5.8, should be familiar to you. The operations `+`, `-`, `*`, and `/` are the usual arithmetic operations. The method written `prefix-` is the negation operator and is written before the receiver: `-n`. The specification `prefix` tells Grace to expect it before rather than after the receiver. As described in Chapter 3, `%` is the modulus operator, so `m % n` is the remainder when `m` is divided by `n`.

The comparison operators `==`, `!=`, `>`, `>=`, `<`, `<=` are also the familiar comparison operations. The method `asString` returns a string representation of the number. Method `inBase(b)` returns a string representation of the receiver in base `b`. Methods `truncated` and `rounded` respectively throw away the fractional part of a number to provide an integral value or round to the nearest integer. Thus the value of `5.74.truncated` is 5, while the value of `(-5.74).truncated` is -5.

² We can actually generalize this definition a bit by allowing specialization of return types, but we won't need this here, so we don't bother to introduce the more general — but more complicated — definition.

on the other hand, the value of `5.74.rounded` is 6, while the value of `(-5.74).rounded` is -6. The method `abs` returns the absolute value of the number.

We have explained that the common mathematical operations are methods, yet the syntax of an expression using `+`, for example, does not look like a method request. That is because we have a special syntax for methods that look like operators. With those methods, the `.` after the receiver's name must be omitted and the parentheses around the argument may also be omitted if the precedence rules make the order of operations unambiguous. However, in all other ways, these methods behave the same as other methods.

5.6.1 More numeric operations

If you review all the things you have learned to do with numbers in Grace, you should be unimpressed at best. Think about it. For just \$10 you could go to almost any store that sells office or school supplies and buy a pocket calculator that can compute trigonometric functions, take logarithms, raise any number to any power and do many other complex operations. Alternatively, if you have a smart phone, you can download an app for free or for a very small amount of money.

On the other hand, with Grace and a computer that probably costs 100 times as much as a calculator, all you have learned to do so far is add, subtract, divide, multiply, find the remainders on division, or raise numbers to powers. In this section we will improve this situation by introducing the remaining methods that provide the means to perform more advanced mathematical calculations.

The other operations supported by `Number` include methods `sqrt` to compute square roots and trigonometric functions such as `sin`, `cos`, `tan`, `asin`, `acos`, and `atan`, where the last three correspond to the arcsine, arccosine, and arctangent functions.

Thus we would write `0.sin` to obtain the value of the sine of 0 or `3.14159.cos` for the value of the cosine of a number close to π . While this may look a little odd compared to the usual `sin(0)` and `cos(3.14159)`, it is consistent with the way we make method requests on other objects.

Grace also supports methods to compute logarithms. Method `log10` finds the logarithm base 10, `lg` finds the logarithm base 2, while `ln` finds the logarithm base e . Thus `8.lg` will return 3.

5.6.2 Problems using equality between numbers

The `abs` method is sometimes used to test whether two numbers are nearly equal. If you test to see whether two `Number` values are exactly equal using `==`, the test may yield **false** even when the values being compared should be equal. This occurs because the computer only records about 15 digits of any double value.

For example, although by definition

$$a = \sqrt{a} \times \sqrt{a}$$

the Grace expression

```
3 == (3.sqrt) * (3.sqrt)
```

```

type Number = {
  // return value of sum of receiver and other
  + (other: Number) -> Number
  // return value of difference of receiver and other
  - (other: Number) -> Number
  // return value of product of receiver and other
  * (other: Number) -> Number
  // return value of quotient of receiver and other
  / (other: Number) -> Number
  // return value of remainder when receiver divides into other
  % (other: Number) -> Number
  // return value of receiver raised to power of other
  ^ (other: Number) -> Number

  // return value of negative of receiver
  prefix- -> Number

  // return whether receiver and other are the same number
  == (other: Number) -> Boolean
  // return whether receiver and other are different
  != (other: Number) -> Boolean

  // return whether receiver is less than other
  < (other: Number) -> Boolean
  // return whether receiver is less than or equal to other
  <= (other: Number) -> Boolean
  // return whether receiver is greater than other
  > (other: Number) -> Boolean
  // return whether receiver is greater than or equal to other
  >= (other: Number) -> Boolean

  // return string representation of receiver
  asString -> String

  // return the receiver as a string expressed in base b
  inBase (b: Number) -> String

  // number obtained by throwing away self's fractional part
  truncated -> Number

  // whole number closest to self
  rounded -> Number

  // the absolute value of self
  abs -> Number

  // square root of self
  sqrt -> Number
}

```

Figure 5.7: Specification of Number type

```

type Number = {
  ...

  // trigonometric sine (self in radians)
  sin -> Number

  // cosine (self in radians)
  cos -> Number

  // tangent (self in radians)
  tan -> Number

  // arcsine of self (result in radians)
  asin -> Number

  // arccosine of self (result in radians)
  acos -> Number

  // arctangent of self (result in radians)
  atan -> Number

  // log base 2 of self
  lg -> Number

  // the natural log of self
  ln -> Number

  // e raised to the power of self
  exp -> Number

  // log base 10 of self
  log10 (n: Number) -> Number
}

```

Figure 5.8: Further specification of Number type

will evaluate to **false**. Since `3.sqrt` produces a value that only approximates $3.\text{sqrt}$, the product `(3.sqrt)*(3.sqrt)` evaluates to something like 2.9999999999999996. Somewhat confusingly if you execute

```
if ((3.sqrt) * (3.sqrt) == 3) then {
  print "same"
} else {
  print "different: value is {(3.sqrt) * (3.sqrt)}"
}
```

the system will print out `different: value is 3` because the system rounds the answer before printing.

To deal with such inaccuracies, it is often better to test whether two values are approximately equal rather than exactly equal. If `a` and `b` are two numeric values, then a test of the form

```
if ((a - b).abs < epsilon) then ...
```

where `epsilon` is a constant with an appropriately small value, is a simple way to test for approximate equality.

Thus we could replace the code above by

```
if (((3.sqrt) * 3.sqrt).abs - 3) < 0.00000001) then {
  print "same "
} else {
  print "different: value is {(3.sqrt) * (3.sqrt)}"
}
```

and this time it will print out "same"

Exercise 5.6.1 *Below, you will find examples of several well-known mathematical equalities and inequalities. Using the methods discussed in this section, translate each of these mathematical statements into a Grace expression. Each of the expressions you write should produce a boolean as a result. Given that the mathematical statements are facts, all these expressions should evaluate to **true**. In fact, however, as a result of the limited precision with which computers represent double values, some of the formulas you produce may evaluate to **false**. You do not need to make any effort to address these potential “mistakes” in your answers.*

For example, if the question asked you to convert the “Triangle inequality”:

$$|x + y| \leq |x| + |y|$$

the correct answer would be the Grace expression:

```
((x + y).abs) <= (x.abs + y.abs)
```

Convert each of the following mathematical statements into Grace expressions:

a. *The reverse triangle inequality:*

$$|x - y| \geq ||x| - |y||$$

b. *the formula for the total return t received when p dollars are invested for y years at an annual rate r with interest compounded monthly:*

$$t = p(1 + r/12)^{12y}$$

c. *The triple angle identity for the cosine function:*

$$\cos 3t = 4 \cos^3 t - 3 \cos t$$

d. *A version of the half-angle identity for the sine function:*

$$\left| \sin \frac{t}{2} \right| = \sqrt{\frac{1 - \cos t}{2}}$$

e. *The exponentiation formula for the log function (use the natural log):*

$$\log x^p = p \log x$$

5.6.3 Time flies

There are many signs that computers keep track of the time. Most likely, the computer you use displays the current time of day somewhere on your screen as you work. Your computer can tell you when each of your files was created and last modified. While your web browser downloads large files, it probably displays an estimate of how much longer it will take before the process is complete.

The `sys` module in Grace provides a number of methods that can be used by a programmer. The one that we are interested measures the time elapsed during a program. To use this method, you must import module `sys` and then evaluate `sys.elapsedTime`, which will return a Number indicating the number of seconds since the program started executing.

For example, if you run a program containing the following code

```
import "sys" as system
...
print "{system.elapsedTime} seconds have elapsed since this program started"
```

Grace might produce the following output

```
0.005 seconds have elapsed since the program started
```

The `elapsedTime` method can be very helpful in measuring short intervals of time within a program. This is an ability we will need in many programs in the following chapters. To use `elapsedTime` to measure an interval, we simply ask Grace for the time at the beginning of the interval we need to measure and then again at the end. The length of the interval can then be determined by subtracting the starting time from the ending time.

As an example of how to do such timing, a program that will measure the duration of a click of the mouse button is shown in Figure 5.9. The program first uses `system.elapsedTime` in the `onMousePress` method. It assigns the time returned to the variable `startingTime` so that it can access the value after the mouse has been released. In `onMouseRelease`, the program computes the difference between the time at which the mouse was pressed and released. A sample of what the program's output might look like is shown in Figure 5.10.

```

dialect "objectdraw"
import "sys" as system

// A program to measure the duration of mouse clicks.
object {
  inherit graphicApplicationSize (400 @ 400)

  // When the mouse button was depressed
  var startingTime: Number

  // Used to display length of click
  def message: Text = textAt (30 @ 50) with ("Please depress and release the mouse")
    on (canvas)

  // Record the time that the button is pressed
  method onMousePress (point:Point) -> Done{
    startingTime := system.elapsedTime
  }

  // Display the duration of the latest press
  method onMouseRelease (point:Point) -> Done {
    message.contents :=
      "You held the button down for {system.elapsedTime - startingTime} seconds"
  }

  startGraphics
}

```

Figure 5.9: Grace program to measure mouse click duration.



Figure 5.10: Sample output of the ClickTimer program

5.7 Strings

By now it should be clear that Grace programs can manipulate many kinds of data. One of the most important kinds of data that computers process is text, which has type `String` in Grace. Text-based programs such as word processors and e-mail applications are among the most important and widely used applications on personal computers.

We have been using `String` values since our very first example programs in Chapter 1. Quoted pieces of text like the message

```
"I'm Touched"
```

which appeared in those first programs are literals describing values of the `String` type. The type `String` is immutable, just like `Number`. Thus there are no mutator methods defined on `String`.

We have seen that Grace provides useful facilities for including the results of calculations in a string. In Figure 1.3, we saw that

```
print "The results of squaring 7 is {square (7)}."
```

resulted in evaluating `square(7)`, converting the answer 49 to the string "49", and then inserting it into the string so that the program printed

```
The results of squaring 7 is 49.
```

In general, when an expression is surrounded by curly braces in a string, the expression will be evaluated, converted to a string by applying the method `asString` and then inserted into the surrounding string. This operation is known as *string interpolation*.

A useful operator provided for strings is the concatenation operator, written `++`. It is used to “glue” two strings together to form a new one. This can be helpful when a string is too long to fit on a single line. For example:

```
print ("This string is way too long to fit on a single line, so we must " ++  
      "break it into two pieces.")
```

A couple of points about this example. First, notice the blank at the end of the string on the first line of the statement. That is needed so that we don't end up with "mustbreak" in the combined string. Second, notice that we have surrounded the concatenation of the two strings by parentheses. That is necessary so Grace knows where it should stop printing. The general rule is that whatever is being printed should be surrounded by parentheses. However, if a single literal string is being printed then the parentheses may be omitted as the double quotes mark the beginning and end of the string.

The fact that Grace considers `String` a type implies that we can do several things with `Strings` that Grace allows us to do with any type. We can use `Strings` as parameters. We have done this in both text constructions and invocations of the `print` method. We can also define instance variables of `String` type and use assignment statements to associate values with these variables.

To illustrate the potential value of using `String` variables, consider the program shown in Figure 5.11. This program implements a simple interface one might use to practice encoding information in Morse code.


```

// Simple program to create Morse Code
// written by Kim Bruce, 5/16/2014

dialect "objectdraw"
import "sys" as sys

object {
  inherit graphicApplicationSize (400 @ 400)

  // Minimum time (in seconds) for a dash
  def dashTime: Number = 0.2

  // String to hold sequence entered so far
  var currentCode: String := "Code = "

  // Time when mouse was last depressed
  var pressTime: Number

  // Text used to display Morse code on canvas
  def display: Text = textAt (30 @ 30) with ("Code = ") on (canvas)

  // Record time at which mouse was depressed
  method onMousePress (point: Point) -> Done {
    pressTime := sys.elapsedTime
  }

  // Add . or - depending on how long mouse held before release
  method onMouseRelease (point: Point) -> Done {
    if ((sys.elapsedTime - pressTime) > dashTime) then {
      currentCode := currentCode ++ " -"
    } else {
      currentCode := currentCode ++ " ."
    }
    display.contents := currentCode
  }

  startGraphics
}

```

Figure 5.11: A program to display Morse code as dots and dashes

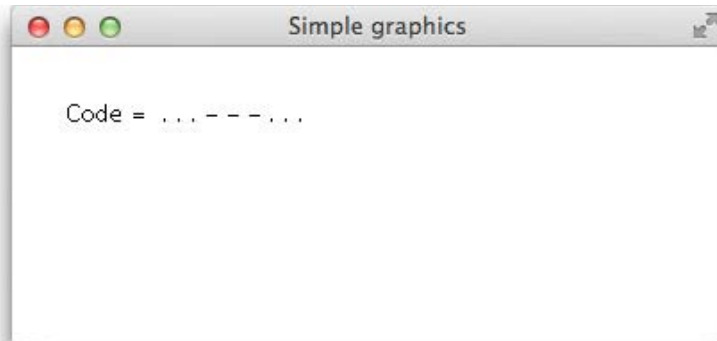


Figure 5.12: Sample of Morse code program display

In Morse code, each letter of the alphabet is encoded using a sequence of long and short signals. These signals are called dashes and dots because a standard way to represent the signals on paper is to draw a dash for a long signal and a dot for a short signal. Thus,

. . . - - - . . .

represents a message composed of three short signals followed by three long signals followed by three short signals. This happens to be the Morse code for the distress signal “SOS”.

The program in Figure 5.11 translates short and long presses of the computer’s mouse button into a sequences of dots and dashes. If someone running the program used the mouse to enter the sequence discussed above, the program would display the output shown in Figure 5.12 The idea is that a person trying to learn Morse code could use the mouse to practice signaling and then look at the sequence of dots and dashes the program displays on the screen to see if they got it right. This means that we don’t need to worry about which sequences of dots and dashes go with which letters of the alphabet to write the program. All we need to know about Morse code is that a short signal is a dot and a long signal is a dash.

The program uses a `String` variable named `currentCode` to keep track of the message it should display on the screen. Initially, the display will simply be

```
Code =
```

Then, each time the user clicks the mouse, the program will add a dot or a dash to the message. If the first thing the user does is a short click, the message will become

```
Code = .
```

If this is followed by two long clicks, the message will become

```
Code = . - -
```

and so on.

To add a dash to the value of the variable `currentCode`, the program uses an assignment statement of the form

```
currentCode := currentCode ++ " -"
```

which can be found in method `onMouseRelease`. This statement looks like and acts a great deal like the similar assignment we have used to increment number variables

```
count := count + 1
```

It takes the existing text associated with `currentCode`, adds a dash to it, and then tells Grace to make this the new value of `currentCode`.

The program uses the `sys.elapsedTime` method described in Section 5.6.3 to access the current time when the mouse is pressed and released. It compares the time that elapses between these two events to a value, `dashTime`. If the elapsed time is greater than this, it executes the assignment shown above to add a dash. Otherwise it uses a similar assignment to add a dot. Either way, after the mouse is released it uses `contents :=` to update the `Text` object named `display` to hold the extended sequences of dots and dashes.

As we have seen earlier, strings also support the “interpolation” of the results of expressions in a string literal. If `n` is an identifier representing the number 47, then `"The expression $n * 2$ evaluates to $\{n * 2\}$."` evaluates to the string `"The expression $n * 2$ evaluates to 94."` That is, the expression `n * 2` is evaluated to the number 94, then converted to the string “94” and finally inserted into the surrounding string so the “94” appears in the string where `{n * 2}` originally appeared. While we mainly use these facilities to make it easier to read output from `print` statements, they can be used in any string.

Like all other types, `String` supports `==`, `!=` (*not equals*) and `asString`.³ Grace provides many other accessor methods to operate on `Strings`. For example, `currentCode.size` will produce a `Number` value telling how many characters long a `String` is. We will consider these functions and other aspects of using `Strings` in more detail in Chapter 15.

5.8 Booleans

As we have seen, the `Boolean` data type represents the two values `true` and `false`. We have seen in Chapter 4 how to use boolean variables and how to build boolean expressions. Here we just quickly list the methods available for type `Boolean`. Like `Number` and `String`, values of type `Boolean` are immutable.

The most important methods on the type `Boolean` are given in Figure 5.13: As we discussed earlier, `&&` is “and”, `||` is “or”, while `not` or a prefix operator of `!` both represent negation. That is, if `bValue` is a boolean variable then `bValue.not` and `!bValue` both return the boolean value opposite that of the receiver.

Like all other types, `String` supports `==`, `!=`, and `asString`. However, it is relatively unusual to use `==` or `!=` with values of type `Boolean`, especially if the comparison is with one of the literals `true` or `false`. We discuss this more in Chapter 7.

³It might be hard to see why `String` would need `asString`, but it would be automatically called if a string variable is interpolated in another string. It just returns itself.

```

type Boolean = {
  // and: return true if and only if the receiver and other are true
  && (other: Boolean) -> Boolean
  // or: return false if and only if the receiver and other are false
  || (other: Boolean) -> Boolean

  // return the opposite of the receiver
  prefix! -> Boolean
  // return the opposite of the receiver
  not -> Boolean

  // return true if and only if the receiver and other are the same
  == (other: Boolean) -> Boolean
  // return true if and only if the receiver and other are the opposite
  != (other: Boolean) -> Boolean

  // "true" or "false" depending on the value of the receiver
  asString -> String
}

```

Figure 5.13: Methods of type Boolean

5.9 Chapter Review Problems

Exercise 5.9.1 Write the code necessary to define a *Number* called *num*, initialized to 5, a *Point* called *pt*, initialized to (50, 50), and a filled rectangle called *rect*, with upper left corner at (100,40) and dimensions 30 by 60. What are the differences in the definitions? Be sure to include the type information in each definition.

Exercise 5.9.2 Fill in the condition for the following code. Assume *myNumber* has been declared as a *Number*.

```

if ( // myNumber is evenly divisible by either 3 or 4 ) then {
  // do something
}

```

5.10 Programming Problems

Exercise 5.10.1 Write a program that displays the number of seconds that have elapsed between two clicks of the mouse. The program should not display a value after the first mouse click, but should display a value after every subsequent click of the mouse.

Exercise 5.10.2

- a. DNA stands for deoxyribonucleic acid and is a very long polymer of four different types of nucleotide bases. These four bases are Guanine(G), Adenine (A), Thymine (T), and Cytosine(C). It is the sequence of these bases that uniquely determine DNA. Write a simple program called `DNAGenerator` that may impress any friends of yours with

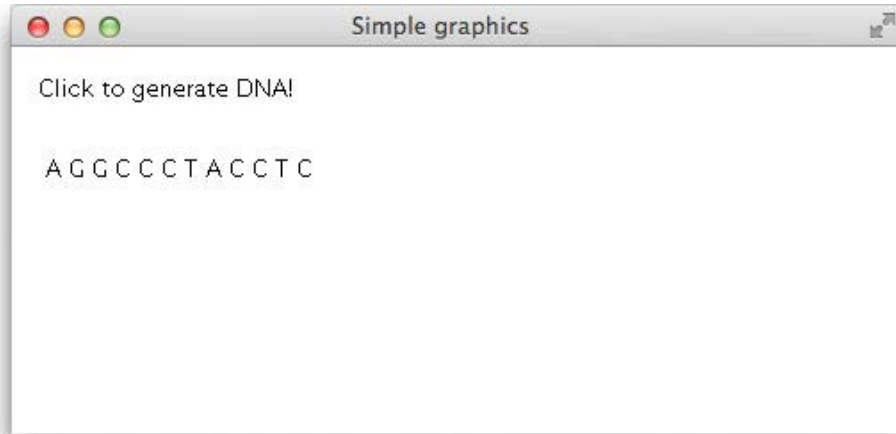


Figure 5.14: Display for DNAGenerator

interests in biology. This program will generate a strand of DNA by randomly adding one of the the four letters “G”, “A”, “T” or “C” each time the mouse is clicked. It also displays the generated DNA strand as shown in Figure 5.14.

- b.** *Modify your program so that no more than 20 bases are added to the DNA strand.*

Exercise 5.10.3 *In Morse code, a single dot is used to represent the letter “E”, a single dash represents a “T”, and the letter “D” is represented by a dash followed by two dots. This means that the sequence*

- . .

has two possible interpretations. It could be a single letter “D” or it could be the word “TEE”.

The lengths of the pauses between dots and dashes are used to resolve such ambiguities. If the pause between a pair of dots is short, then the two signals are treated as part of a single letter. If the pause is long, the signals are treated as parts of separate letters. To make it possible to represent such long and short pauses in our graphical representation of sequences of dots and dashes we might represent the sequence for “D” as

- . . .

and the sequence for “TEE” as

-

That is, we would put more spaces between pairs of symbols if the pause between them was longer. Thus, the signal for “TED” would look like

- . . . - . . .

Modify MorseCode so that it adds extra spaces for long pauses. Assume a pause is long if its duration equals (or exceeds) that of a dash.

Chapter 6

Objects & Classes

So far when we have written a program, it has been by defining a single object that inherit from `graphicApplication`. The program's behavior results from defining methods with fixed names like `onMousePress`, `onMouseClicked`, and so on. In this chapter we will be designing very different objects and classes that can generate objects that can work together in programs.

All of the geometric objects that we have been using so far have been generated from classes that have been provided for you. The classes include, for example, `filledRect`, `framedRect`, `filledOval`, `line`, etc. The first objects and classes we will be designing will provide methods similar to those used in the geometric classes. The Grace code that we will write to support these will be similar to that which we have already written in earlier programs in that we will be declaring constants and instance variables and defining methods in order to provide the desired behavior of objects.

The focus of this chapter is on how to avoid complexity by using objects and classes. Good object-oriented programmers design programs as the interaction of objects, each of which is capable of certain behaviors. While each of these behaviors may be simple, the interactions of the objects allow relatively complicated tasks to be handled simply. In particular, each object will take responsibility for a collection of actions that contribute to the program. This can make testing the objects, assembling the final program, and removing any remaining errors it contains much easier. As a result, the design of objects and the classes that generate them is one of the most important aspects of object-oriented programming.

6.1 An Example Without objects

While the focus of this chapter is on designing and writing objects and the classes to generate objects, we will begin with a simple example in which we do not create any new objects. We will later contrast this approach with one where we define an object to handle some of the more complex aspects of the program. Our goal is to convince you that the creation of new objects makes the programming process easier.

Recall the program `WhatADrag` from Figure 4.6. This program allowed the user to drag a rectangle around the screen. Only minor changes would be required to replace the rectangle by an oval. The name of the instance variable `box` would likely be replaced by a name like `circle` and the initialization would call the constructor `filledOval` rather than `filledRect`. However,

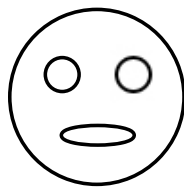


Figure 6.1: Funny face generated by `FaceDrag`

aside from these minor changes of names, the structure of the program remains essentially the same.

These minor changes work fine as long as we use a geometric object supported by our libraries, but suppose we want to drag around a triangle, which is not supported by our library. Now we have to work harder. We would have to determine how to build the triangle from three straight lines, and we have to write code to determine whether a point is contained in a triangle.

We could have had the library support a triangle, but then we would still be missing classes to support a pentagon, hexagon, or some other polygon that might be needed for some particular problem. The point here is that we will often not be lucky enough to have exactly the classes we need. In those cases we will have to figure out a way of handling this ourselves.

In this chapter we will illustrate the use and implementation of objects and classes with a fanciful example of a funny face as shown in Figure 6.1. The program in Figure 6.2 displays a funny face and allows the user to drag it around the screen, very much like the program `WhatADrag`. Because the funny face has several geometric components, the main differences with `WhatADrag` are that there are several objects that need to be moved with each drag of the mouse. The head, `leftEye`, `rightEye`, and `mouth` are declared as instance variables and are created in the `begin` method. As in `WhatADrag`, `lastPoint` will be an instance variable that keeps track of where the mouse was after the last mouse press or drag.

When we move the face in `onMouseDown`, all four pieces must be moved. However, imagine how much longer the method would be if the face had many more pieces. Worse yet, suppose we wanted to have two or more faces on the canvas at a time. It would rapidly become hard to keep track of the various pieces without forgetting one of them. This complexity is due to the fact that we do not have a class to create and manipulate funny faces in our library.

For the moment, suppose that we were provided with an object representing a funny face that does all of the initialization performed in `FaceDrag`. Furthermore, assume it has methods `contains` and `move` that determine if a `Point` is inside the funny face and that move the funny face, respectively. We could use this class exactly as we did the filled rectangle in the program `WhatADrag` in Figure 4.6.

The revised code for dragging a face is given in Figure 6.3. It is nearly identical to that of the original `WhatADrag`, and is considerably simpler than that for `FaceDrag`, because we don't have to worry about handling all of the different pieces of the funny face. We can treat the face as a single object.

The only differences from `WhatADrag` are that the variable `box` of class `Graphic2D` has been


```

dialect "objectdraw"

object{
  inherit graphicApplicationSize (400 @ 400)

  def locn: Point = 200 @ 100      // starting posn of face

  def faceHeight: Number = 60     // dimensions of the face
  def faceWidth: Number = 60
  def eyeOffset: Number = 20     // eye location and size
  def eyeRadius: Number = 8
  def leftEyeLocn: Point = (locn.x + eyeOffset - eyeRadius / 2) @ (locn.y + eyeOffset)
  def rightEyeLocn: Point = (locn.x + faceWidth - eyeOffset - eyeRadius / 2) @ (locn.y + eyeOffset)

  def mouthHeight: Number = 10   // dimensions of the mouth
  def mouthWidth: Number = faceWidth / 2
  def mouthLocn: Point = (locn.x + (faceWidth - mouthWidth) / 2) @ (locn.y + 2 * faceHeight / 3)

  // create head and facial features
  def head: Graphic2D = framedOvalAt (locn) size (faceWidth @ faceHeight) on (canvas)
  def mouth: Graphic2D = framedOvalAt (mouthLocn) size (mouthWidth @ mouthHeight) on (canvas)
  def leftEye: Graphic2D = framedOvalAt (leftEyeLocn) size (eyeRadius @ eyeRadius) on (canvas)
  def rightEye: Graphic2D = framedOvalAt (rightEyeLocn) size (eyeRadius @ eyeRadius) on (canvas)

  var lastPoint: Point // point where mouse was last seen

  // whether the box has been grabbed by the mouse
  var faceGrabbed: Boolean := false

  // Save starting point and whether point was in box
  method onMousePress (point: Point) -> Done {
    lastPoint := point
    faceGrabbed := head.contains (point)
  }

  // if mouse is in box, then drag the box
  method onMouseDrag (point: Point) -> Done {
    if (faceGrabbed) then {
      head.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)
      leftEye.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)
      rightEye.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)
      mouth.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)
      lastPoint := point
    }
  }

  startGraphics
}

```

Figure 6.2: Code for dragging a funny face.

```

dialect "objectdraw"
import "FaceClass" as fc
object{
  inherit graphicApplicationSize (400 @ 400)

  def face: fc.Draggable // object representing face

  var lastPoint: Point // point where mouse was last seen

  // whether the box has been grabbed by the mouse
  var faceGrabbed: Boolean := false

  // Save starting point and whether point was in box
  method onMousePress (point: Point) -> Done {
    lastPoint := point
    faceGrabbed := face.contains (point)
  }

  // if mouse is in box, then drag the box
  method onMouseDrag (point: Point) -> Done {
    if (faceGrabbed) then {
      face.moveBy (point.x - lastPoint.x, point.y - lastPoint.y )
      lastPoint := point
    }
  }

  startGraphics
}

```

Figure 6.3: Revised code for dragging a FunnyFace.

replaced uniformly by the variable `face`, variable `boxGrabbed` has been renamed as `faceGrabbed`, and the initialization code now creates and names a new object (whose details are omitted from the code) rather than a `Graphic2D`. Otherwise they are identical.

This “divide-and-conquer” approach makes developing programs much simpler. You can first write and test the object to create the face and then use it in other contexts that are now simpler.

In the next section we will discuss the design and implementation of objects and classes, using the funny face as a first example. Later in the chapter we will reuse the `FunnyFace` class in yet another program, illustrating the advantages of designing classes to represent objects that may be used in many different situations.

Exercise 6.1.1 *What are the advantages of creating several objects versus trying to place all of the code within one object definition?*

6.2 Designing Objects and Classes: A funny face

We saw in Chapter 1 some very simple examples of defining objects. However since then we have only designed objects that inherit from `graphicApplication`. In this section we will introduce you how to define more general objects and classes that generate objects.

6.2.1 Defining objects

Object expressions include definitions, variable declarations, method declarations, type definitions, and initialization code. One of the key characteristics of object-oriented languages is that they allow you to put all these relevant details for an object together in a single place.

As a first example of defining our own objects, let’s go back to the funny face program we wrote in the last section. In Figure 6.2 we showed a version of the program that treated the pieces of the face individually. For example, in the `onMouseDown` method we had to move each piece of the face individually. Then we imagined that there was an object representing a face, and saw in Figure 6.3 how we could simplify the code in the event-handling methods by using that object.

Of course we are not likely to find a funny face object in a general graphics library, so let’s think about how we would define such an object.

The first question we should ask when designing an object is what kind of capabilities should it have. That is, what should it be able to do and what kind of questions should it be able to answer. The answers to the first question will help determine the mutator methods of the object, while the second will give us the accessor methods.

Looking at Figure 6.3, we see that the face object is used in both the `onMousePress` and the `onMouseDown` methods. The statement containing `face` in the first is

```
faceGrabbed := face.contains(point)
```

This tells us that our face object should provide a method `contains` that takes a `Point` parameter and returns a value of type `Boolean`. We can tell it returns a value of type `Boolean` because its result is assigned to the variable `faceGrabbed`, which has type `Boolean`.

The statement mentioning `face` in `onMouseDown` is

```
face.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)
```

From this we see that the object associated with `face` needs a `moveBy` method that takes two Numbers.

Let us write a type definition that includes these two methods that we see are required for `face`. Luckily, we've already seen a type that includes these methods, type `Graphic` in Figure 5.2. We can copy those into a new type:

```
type Draggable = {  
  // move this object dx to the right and dy down  
  moveBy (dx: Number, dy :Number) -> Done  
  
  // Does this object contain locn  
  contains (locn: Point) -> Boolean  
}
```

Now that we understand exactly what methods we must support, we can focus on creating the object and providing definitions and variable declaration bodies that will allow us to support these operations.

The definition of the object expression for `face` is given in Figure 6.4. The definitions, variable declarations, and initialization code are exactly the same as before, though we left out variables like `faceGrabbed` and `lastPoint` that were specific to the application and not to the object itself.

The code for the methods is similar to that referring to the face components in the original version of the methods `onMousePress` and `onMouseDrag`. We only include the code relevant for performing the method, rather than trying to recreate everything in the mouse event-handling methods.

Thus `moveBy` includes the four lines of code that move the four pieces of the face. The formal parameters `dx` and `dy` will be associated with actual numbers when the method is requested. For example, if we evaluate `face.moveBy (7, 12)` the `dx` parameter will be associated with the value 7, while `dy` will be associated with 12. Notice that the arguments of the method request are associated with the formal parameters in exactly the order they appear. Because 7 is the first argument in the method request, it is associated with the first parameter, `x`.

In the code in Figure 6.3, the arguments to `moveBy` are complicated:

```
face.moveBy (point.x - lastPoint.x, point.y - lastPoint.y)
```

However as above, the values are associated in order. That is, the value of the parameter `dx` will be associated with the value of the argument `point.x - lastPoint.x`, while `dy` will be associated with the value of `point.y - lastPoint.y` when the method is invoked. For example suppose `lastPoint` represented the `Point` at (25, 50) and the user dragged to (28, 52), meaning the value of `point` is the `Point` at (28, 52). Then when `moveBy` is executed, the value of parameter `dx` will be $28 - 25 = 3$, while the value of `dy` will be $52 - 50 = 2$.

Because we expect only numbers for the arguments during a method request, we annotate both parameters of `moveBy` with the type `Number`. It is not expected to return a value when it finishes executing, so we provide a return type annotation of `Done`.

The `contains` method simply checks to see if the point is inside the head as the other pieces are themselves all inside the head. It has a formal parameter `pt`, which is annotated with its expected type, `Point`. The method will always return either true or false, so we annotate the method with the return type `Boolean`. When a method is to return a value, it simply returns

```

def face: Draggable = object {
  def locn: Point = 200 @ 100 // starting posn of face

  def faceHeight: Number = 60 // dimensions of the face
  def faceWidth: Number = 60

  def eyeOffset: Number = 20 // eye location and size
  def eyeRadius: Number = 8
  def leftEyeLocn: Point = (locn.x + eyeOffset - eyeRadius / 2) @ (locn.y + eyeOffset)
  def rightEyeLocn: Point = (locn.x + faceWidth - eyeOffset - eyeRadius / 2) @ (locn.y + eyeOffset)

  def mouthHeight: Number = 10 // dimensions of the mouth
  def mouthWidth: Number = faceWidth / 2
  def mouthLocn: Point = (locn.x + (faceWidth - mouthWidth) / 2) @ (locn.y + 2 * faceHeight / 3)

  // create head and facial features
  def head: Graphic2D = framedOvalAt (locn) size (faceWidth @ faceHeight) on (canvas)
  def mouth: Graphic2D = framedOvalAt (mouthLocn) size (mouthWidth @ mouthHeight) on (canvas)
  def leftEye: Graphic2D = framedOvalAt (leftEyeLocn) size (eyeRadius @ eyeRadius) on (canvas)
  def rightEye: Graphic2D = framedOvalAt (rightEyeLocn) size (eyeRadius @ eyeRadius) on (canvas)

  // move the face by dx to the right and dy down
  method moveBy (dx: Number, dy: Number) -> Done {
    head.moveBy (dx, dy)
    leftEye.moveBy (dx, dy)
    rightEye.moveBy (dx, dy)
    mouth.moveBy (dx, dy)
  }

  // determine whether the face contains pt
  method contains (pt: Point) -> Boolean {
    head.contains (pt)
  }
}

```

Figure 6.4: Definition of the face object

the value of the last expression occurring in the method body. In the case of `contains`, the method body contains only the single line `head.contains(pt)`. The value of this expression is what we need to return (a point is inside the whole face if it is anywhere inside the `head`), so it is the last expression in the method body. It returns either `true` or `false`, so this matches with the declared return type of `contains`.

The complete object expression can now be placed inside the program in Figure 6.3. You can see that inside the main program, a face is treated as an entity rather than a collection of pieces, making the code referring to it much simpler than in our original attempt in Figure 6.2.

Exercise 6.2.1 *Suppose `point` is a `Point` at $(25,50)$. When the computer executes `face.moveBy(point.x,0)`, what values are associated with the formal parameters `dx` and `dy` of the `moveBy` method of `face`?*

Exercise 6.2.2 *Suppose we defined a new feature, `nose`, of type `Graphic2D` in `face`. How would the body of the `moveBy` method change?*

6.2.2 Programming with class

Now that we have written a description of the object referred to by `face`, we should think about other uses of `face`. For example we might have programs that have the face starting at different `Points` or programs that have multiple faces. What we actually need is a way of generating objects that can be customized by any number of parameters.

The notion of class turns out to be exactly what we need here. Classes are used to generate objects and may take parameters that allow these objects to be customized based on those values. We illustrate this by writing a class that can be used to generate funny faces.

As we think about removing the definition of `face` objects from being embedded in the program that allows the dragging of the face, we need to think about two things:

1. What features used in the object definition are provided by the enclosing program?
2. What features of the object might we want to vary when it is used in other applications.

Thinking about the first of these questions, we see that the object expression providing the meaning of `face` uses the value `canvas`, which is provided by the outer object expression because it inherits from `graphicApplication`. It is not introduced in a definition or variable declaration from the object expression for `face`. Thus if we remove the object expression from this program, we will need to provide a value for `canvas`. Because of this we will provide a parameter to the class for the `canvas`.

The answer to the second question – what features might we want to vary? – is very much a judgement call. We certainly might want to vary the starting position of the face, but we might as well want to vary the width and height of the face. In fact we could vary even more features like how far the eyes are apart and so on. However, for simplicity we will be conservative here and decide the only feature we will vary is the starting `Point`. As a result we will also provide a parameter to the class that will determine the starting `Point` for the face (*which will be the upper left corner as usual*).

Thus we will write the class `funnyFace` with the two parameters determined as the results of the above questions. The first parameter, `startingPoint`, will determine where the new face

will be created while the second provides the canvas that the face will be drawn on. The complete class declaration is shown in Figure 6.5. Compare this definition with the definition of the object in Figure 6.4

Notice that, as with the constructions for the graphic items in the `objectdraw` library, class names have a “.” to separate the basic name for the constructor (in this case `funnyFace` with the identifiers (`at` and `on`) associated with the parameters. This makes use of the class constructors look very much like method requests, providing a more uniform structure to your program.

Given that this class has been defined, we can fill in the line defining `face` in Figure 6.3 by the simpler line:

```
object{
  inherit graphicApplicationSize (400 @ 400)

  def locn: Point = 200 @ 100           // starting posn of face
  def face: fc.Draggable = funnyFaceAt (locn) on (canvas) // object representing face
  ...
}
```

Notice that the arguments given to `funnyFace` are exactly the values used in the original object definition. Thus the value given in the definition of `locn` will be used in place of the formal parameter `startPoint` in the class declaration. Similarly the value associated with `canvas` (i.e., the drawing area of the window that pops up when the program is executed) will be associated with the formal parameter `theCanvas`. Thus the result will be a face at the same `Point` on the canvas when the window pops up.

If the constructor is called several times with different values for `startingPoint`, it will result in several funny faces being constructed in different `Points` on the canvas. Thus if we write

```
def funnyLeftLocn: Point = 40 @ 100
def funnyRightLocn: Point = 90 @ 100
faceLeft = funnyFaceAt (funnyLeftLocn) on (canvas)
faceRight = funnyFaceAt (funnyRightLocn) on (canvas)
```

the program will place two funny faces, `faceLeft` and `faceRight`, next to each other on canvas.

It is important to remember that each instance of a class has its own distinct collection of instance variables. Thus `faceLeft` will have geometric objects associated with `face`, `eyeLeft`, `eyeRight`, and `mouth`. The object named by `faceRight` will have its own distinct objects associated with these identifiers. For example, the oval representing the face of `faceLeft` will be 50 pixels to the left of the oval representing the face of `faceRight`.

6.2.3 Programmers create abstractions

Notice that the expression creating a new face object from the class `funnyFace` looks just like the code we used earlier to create graphic objects like framed rectangles from the `objectdraw` library. One of the important features of object-oriented languages is that you can create your own objects that look just like those provided by the system. If you ever need a new object that is not available in accessible libraries, you can define your own using an **object** or **class** and use it in exactly the same way as built-in objects.

Defining classes is clearly a big win if we need multiple objects with the same structure – differing only by values that can be passed in as parameters. On the other hand, if only one object will ever be needed then there is little reason to have a class as opposed to an

```

dialect "objectdraw"
type Draggable = {
  moveBy (dx: Number, dy: Number) -> Done
  contains (pt: Point) -> Boolean
}

// creates a funny face starting at startingPoint and on canvas theCanvas
class funnyFaceAt (startingPoint: Point) on (theCanvas: DrawingCanvas) -> Draggable {
  def faceHeight: Number = 60 // dimensions of the face
  def faceWidth: Number = 60

  def eyeOffset: Number = 20 // eye location and size
  def eyeRadius: Number = 8
  def leftEyeLocn: Point = (startingPoint.x + eyeOffset - eyeRadius / 2) @ (startingPoint.y + eyeOffset)
  def rightEyeLocn: Point = (startingPoint.x + faceWidth - eyeOffset - eyeRadius / 2) @
    (startingPoint.y + eyeOffset)

  def mouthHeight: Number = 10 // dimensions of the mouth
  def mouthWidth: Number = faceWidth / 2
  def mouthLocn: Point = (startingPoint.x + (faceWidth - mouthWidth) / 2) @
    (startingPoint.y + 2 * faceHeight / 3)

  // create pieces of the face
  def head: Graphic2D = framedOvalAt (startingPoint) size (faceWidth @ faceHeight) on (theCanvas)
  def mouth: Graphic2D = framedOvalAt (mouthLocn) size (mouthWidth @ mouthHeight) on (theCanvas)
  def leftEye: Graphic2D = framedOvalAt (leftEyeLocn) size (eyeRadius @ eyeRadius) on (theCanvas)
  def rightEye: Graphic2D = framedOvalAt (rightEyeLocn) size (eyeRadius @ eyeRadius) on (theCanvas)

  // move the face by dx to the right and dy down
  method moveBy (dx: Number, dy: Number) -> Done {
    head.moveBy (dx, dy)
    leftEye.moveBy (dx, dy)
    rightEye.moveBy (dx, dy)
    mouth.moveBy (dx, dy)
  }

  // determine whether the face contains pt
  method contains (pt: Point) -> Boolean {
    head.contains (pt)
  }
}

```

Figure 6.5: Class for generating funny faces

object expression. The most important thing is to group together all the relevant features of an object – the definitions, variable declarations, and methods, so that we can think of the object as a single entity rather than as the collection of pieces.

A guiding principle of all programming languages is to provide facilities to allow you to create your own high-level abstractions so that you can design programs based on those high-level abstractions rather than tiny details. As an example, we think of a laptop computer as having a keyboard, trackpad, and screen because those are the pieces we interact with. We tend to think of the keyboard as a whole rather than a collection of 100 or so keys with tiny springs and switches and so on. Most of the time the right level of abstraction when talking about a keyboard is to refer to what to type on the keyboard, rather than talking about the 8th key in the 3rd row from the bottom and the switch that is depressed when you click on it.

By the way, you’ll notice that when we described the computer above, we left out the internal components of the computer – the memory, the hard drive, the central processor, and the wifi card. If you are a computer user, thinking about these individual items is only rarely relevant, for example if you run out of space on your hard drive or find your computer running very slowly. For day to day use, we want to think about the computer in terms of these higher-level abstractions. Of course if we are buying or fixing a laptop then we will need to look at a different set of abstractions. In general when we are working on a problem we will need to determine what are the right set of abstractions.

We saw in this section that if we need more than one object with the same structure and methods, we can accomplish that with a class declaration. A class is a template that can be used to create new objects, which are called *instances* of the class. Classes, like object expressions, may include definitions, variable declarations, method definitions, and even type definitions. Objects and classes may also contain initialization code that is executed when the object is created. Classes, unlike objects, may also come with formal parameters. Actual values are provided for these parameters when the class is used to create an object.

Each object used in a program contains methods that can be executed in order to perform certain behaviors. The methods of an object are said to specify the *behavior* of the object. For example, a framed rectangle object has methods including `moveBy`, `moveTo`, `contains`, and `color`. These methods can be invoked or executed by making a *method request* to an object. A method request consists of the name of a method supported by the object, and parameters that provide information to be used in executing the method. You can think of making a method request of an object as being analogous to asking or telling a person to do something (perform a particular behavior) for you.

Values represented by definitions and instance variables of an object are used to maintain the *state* of an object. That is, they are used to store values that the object can use in executing the methods associated with the object. For example, a funny face object’s state consists of information that allows the methods to determine the object’s location, dimensions, height, eyes, mouth, etc.

The initialization code in a class is used to provide values for the definitions and instance variables and may also be used in the methods of the new object. When a class is invoked with appropriate arguments, a new object is constructed with new copies of all of the definitions and instance variables specified by the class and with all of the methods. The initialization

code of the class is used to provide values for the definitions and variables, and to perform any other work needed at that point.

Exercise 6.2.3 *Suppose we want to have a class that represents two framed rectangles, one inside of the other to form a picture frame. Objects generated from the class should support a `moveBy` method*

The parameters to the class should determine the upper left corner of the frame, its width and height, the separation between the two `FramedRects`, and the canvas on which it will be placed. You are free to select the exact parameters expected and their order.

Please write a complete Grace class called `pictureFrame` to create picture frame objects.

Exercise 6.2.4 *Write a class inheriting `graphicApplication` that creates two objects from class `funnyFace`. If the user presses the mouse on one of the two faces, then dragging the mouse will result in dragging that face. The code will be similar to that in Figure 6.3, but the `onMousePress` method will need to keep track of what face was selected so that the `onMouseDrag` method will move the correct face.*

6.2.4 Putting It All Together

Class definitions in Grace are written as follows:

```
className (...) nm (...) ... -> SomeType {  
  // definitions  
  // variable declarations  
  // executable code  
  // method declarations  
}
```

While Grace does not insist that the order of components be exactly that specified above, we will generally find it easier to keep the definitions and variable declarations toward the front of the class or object declaration and the method declarations at the end, with the executable statements wherever they make the most sense. (Remember that you cannot use a variable or definition until after it has been declared and initialized.)

Writing your code in a predictable order makes it easier for others to read your code. Including the type expression indicating what type of object is returned by the class lets a potential user see what capabilities (methods) the objects will have. We can then safely put the actual method declarations at the end of the class, letting a reader first learn what identifiers are used in writing that code and what they stand for before digging into the details of the implementation of the methods.

Now that we have completed `funnyFace` class, let's compare that class (shown in Figure 6.5 and the revised version of the face dragging program (in Figure 6.3) with the original program found in Figure 6.2. The first version had 5 instance variables – the four components of the face and the variable `lastPoint` to keep track of the last `Point` of the mouse. The four components of the face were moved to the `FunnyFace` class, while `lastPoint` was kept with the mouse-handling code of `RevFaceDrag` because it was more relevant to the mouse movements than to the funny face.

All of this effort to create a `FunnyFace` class may seem like a lot of work for little benefit. After all, the code of the original program is shorter than the combination of the code in

classes `FunnyFace` and `RevFaceDrag`. However, there are several advantages to creating and using the class `FunnyFace`.

One benefit is that, having created the `FunnyFace` class, we can use this class in any program now or in the future, rather than having to rewrite the functionality. In particular, this class may be used with a variety of other classes to create more complex programs. (We illustrate this later in the chapter by using `FunnyFace` in a different program.)

Another benefit is that because this program supports the same kind of methods as the geometric objects in the `ObjectDraw` library, we may use it just like these other objects. In the example above, the difference between a program allowing the user to drag around a rectangle and one allowing the user to drag around a funny face is minimal. We can thus reuse with minimal changes other code that we have already determined works with other kinds of objects.

This worked because the two methods in type `Draggable` were exactly the same as the corresponding methods in type `Graphic2D`, and they were the only two of those methods used in the program. As a result, we can associate any object of type `Graphic2D`, say a filled rectangle, with an identifier with type `Draggable`. When we say an object has a type, it means that it has at least the methods described in those types – i.e., and possibly more.

As a result, the following definition would be legal in a program where `somePt` is associated with a `Point`, `w` and `h` are numbers, and `canvas` has type `DrawingCanvas`.

```
def draggedObject: Draggable = framedRectAt (somePt) size (w @ h) on (canvas)
```

6.3 Adding Methods to our face

If we were to use objects from class `funnyFace` in other programs like those we wrote using geometric objects from the library, we would likely need other methods like those supported by the other geometric objects to the `funnyFace` class. Such methods would likely include `moveTo`, `isVisible`, `visible:=`, `color`, and `color:=`, among others. In this section we discuss how we could write such methods.

6.3.1 Some Methods Are Like Those Already Written

The code for the method `visible:=` is similar to that of `moveBy` in that it simply forwards `visible:=` method requests to each of its components.

```
// Change the visibility of face
method visible := (state: Boolean) -> Done {
  head.visible := state
  leftEye.visible := state
  rightEye.visible := state
  mouth.visible := state
}
```

The code for method `color:=` will be very similar. (We will decide to have all the components of a face to be the same color.)

Sometimes you get lucky, and the code can be even simpler. Because the components of the face objects will either all be visible or all be invisible we only need to retrieve the status

of one component to know whether the entire face is visible:

```
// return whether the face is currently visible
method isVisible -> Boolean {
  head.isVisible
}
```

Thus this method will return the visibility of the head, providing us with the visibility of the entire face. Note the similarity of this method with that of the `contains` method, where we could get away with only asking whether a location `pt` was within the head because all other pieces were within the head object.

The code for `color:=` and `color` should be easy for you to write, as they are similar to these two methods.

Exercise 6.3.1

Write the methods `color` and `color:=` for the class `funnyFace`.

Exercise 6.3.2

We didn't bother to include instance variables for representing the color of the face or whether it was visible. Consider what the code for the class would look like if we had included one or both of these. Would it make the code for the methods harder or easier? Provide reasons why we might have decided to leave them out.

6.3.2 Some Methods Are Trickier

Because `funnyFace` has a `moveBy` method, an obvious addition would be a `moveTo` method that would take a new location as a formal parameter, and result in moving the object to that location. Writing the method `moveTo` will be a bit trickier than `moveBy`. If we did the obvious thing and moved all of the components of the funny face to the same place, both of the eyes and the mouth would end up on top of each other in the upper left hand corner of the face. *Stop and make sure you understand this – draw the results of moving all the components to the same place!*

We could instead calculate where we want all of the pieces to be after the `moveTo`, and send a `moveTo` message with different parameters to each of the components of the funny face. However, it would be pretty painful to have to calculate all of those points. (Look at how horrible all the calculations were in the definitions and instance variables of the class!)

There is an alternative that is both clever and simple — an ideal combination. What we will do is figure out for the funny face *how far* to `moveBy` rather than *where* to move it. Once we calculate the horizontal and vertical distance the funny face is to move, we can pass those as actual parameters to the funny face's `moveBy` method and take advantage of the fact that we have already written that method. The key thing here is that while each of the four components of the face must be moved *to* a different point, they all need to be moved *by* the same amount so they stay in the same positions relative to the point of the head.

The position of the face is actually the position of the upper left corner of the bounding box for the head. The new position occurring as a parameter to `moveTo` will be the new position for the head. We need to figure out how far to move in each of the x and y directions to get from the current position to the new position.

Let's look at a simple example. If the `face` is currently at the point with coordinates (40,100) and we wish to move it to down and to the right to coordinates (70, 120), then we will need to move the head $70 - 40 = 30$ pixels to the right, and $120 - 100 = 20$ pixels down. Because we want the eyes and mouth to be in the same relative position with respect to the head, we also want all of them to move by the same amount: 30 to the right and 20 down.

If we take these ideas and just straightforwardly translate them to Grace code we get the following method:

```
method moveTo (newPoint: Point) -> Done {  
  def moveX: Number = newPoint.x - head.x  
  def moveY: Number = newPoint.y - head.y  
  head.moveBy (moveX, moveY)  
  leftEye.moveBy (moveX, moveY)  
  rightEye.moveBy (moveX, moveY)  
  mouth.moveBy (moveX, moveY)  
}
```

Identifiers `moveX` and `moveY` are calculated to be the amount to be moved in each of the x and y directions, and then each of the four components is moved by that amount.

That works fine, but the code is getting a bit long and messy. If we think carefully, we see that we have written other code that looks almost exactly like this – at least for the last four lines. That is almost exactly the code for the method `moveBy`.

One thing good computer scientists learn very early is never to duplicate existing code. Instead, if you know you have code that works, take advantage of it and reuse it. In this case we want to take advantage of the `moveBy` method that we have already written:

```
method moveTo (newPoint: Point) -> Done {  
  def moveX: Number = newPoint.x - head.x  
  def moveY: Number = newPoint.y - head.y  
  self.moveBy (moveX, moveY)  
}
```

So here we have replaced the four method requests for `moveBy` in the previous version by one in this one. However, the requests for `moveBy` in the previous version went to components of the face. In this version there receiver is `self`. What is that?

The identifier `self` is used to refer to the object that received the method request. So if we execute `face.moveTo (origin)`, where `face` is the object we've constructed and `origin` is a point corresponding to (0,0), then when we start executing the code for `moveTo` above, the formal parameter `newPoint` will refer to `origin`, as we have seen before. But now also `self` will refer to `face`, as that is the object that was the receiver of the `moveTo` request.

You should think of `self` as a kind of pronoun. When you need to remind yourself to do something, you may write a message to your *self*. If other people might be around, perhaps you'll put your name on it, but if, say, it is on your own desk, you may just write: "Message to self: Do ...".

On the other hand, if no one else is likely to see it, you may leave off that initial part about yourself and just write "Do ...". Grace actually let's you do the same thing. Most Grace programmers would write the above method as:

```
method moveTo (newPoint: Point) -> Done {  
  def moveX: Point = newPoint.x - head.x  
  def moveY: Point = newPoint.y - head.y
```

```

    moveBy (moveX, moveY)
  }

```

Notice that the difference here is that **self** receiver of the `moveBy` method request has been omitted. In Grace, when you omit a receiver of a method request, the system interprets that as a request on **self**.

It is also possible to attach the **self** prefix to an instance variable, but, as with methods, we may omit it (as we have to this point in all of our examples). We will generally continue to omit it in our examples.

From now on we will assume that the definition of `funnyFace` (and the type of objects it generates, `Draggable`) includes at least the method `moveTo` defined above. As suggested earlier, that will allow us to use `funnyFace` more effectively in other programs.

Exercise 6.3.3 *Normally we would make the class `funnyFace` even more flexible by also including parameters in the class to specify the width and height of the funny face being created. We left those off in this version of `funnyFace` in order to keep the code simpler. Rewrite `funnyFace` to take two extra parameters `width` and `height` of type `Number` to represent the width and height of the funny face being constructed.*

6.3.3 Using **self** as a Parameter

The identifier **self**, which refers to the object receiving the method request, has the same type as that object. As a result, we can pass **self** as a parameter of a message, so that the receiver can make requests back to the sender. This is not unlike sending someone your e-mail address so that they can correspond with you.

For example, suppose that we wish to create classes for keys and locks, and that moreover, every time we create a key, we want to associate it with the appropriate lock. We can do that by creating a `makeKey` method for the lock that will both create the key and tell the key what lock it opens.

Let us assume the class constructing key objects takes as a parameter the lock it is associated with. The class might look like:

```

class keyWithLock (myLock: Lock) -> Key {
  def theLock: Lock = myLock
  ...

  method lock -> Lock {
    theLock
  }
  ...
}

```

The class `lock` and its method `createKey` are shown below:

```

class makeLock -> Lock {
  var myKey: Key
  ...
  method createKey -> Key {
    myKey := keyWithLock (self)
    myKey
  }
}

```

```
    ...  
}
```

The use of **self** as a parameter to the `keyWithLock` class in class `makeLock` results in passing the lock itself as a parameter to the class for `keyWithLock`. For example, if `bikeLock` refers to an object of type `Lock`, then

```
def bikeKey: Key = bikeLock.createKey
```

results in `bikeKey` referring to the key that unlocks `bikeLock`. In particular, evaluating `bikeKey.lock` refers to the same value as `bikeLock`.

6.4 Another Example: Implementing a Timer Class

In this section we practice what we have learned by designing another program that uses classes. There are two main points that we would like to make with this example. The first is that we can define classes like `Point` that are useful in programs, but that do not result in anything being displayed on the canvas. The second is that we can reuse previously defined classes to solve new problems.

The application we are interested in is a very simple game in which the user tries to click on an object soon after it moves to a new position. Because success in the game depends on clicking soon after the object has moved, we will create a `Timer` class that provides facilities to calculate elapsed time. We will reuse the `funnyFace` class to create the object to be chased. Unlike the `funnyFace` class, the `Timer` does not result in anything being drawn on the screen.

To calculate elapsed time, the `Timer` class uses the method `system.elapsed` introduced in Chapter 5. Recall that this method returns the number of seconds since the program started execution. Because we are interested in the length of time intervals, we will obtain starting and ending times and then subtract them.

As we can see from the code in Figure 6.6, the `Timer` class is very simple. The class has a single instance variable `startTime` that is initialized in the constructor with the value of `system.elapsed`. It can be reinitialized in the `reset` method by reevaluating `system.elapsed`. The methods `elapsedMilliseconds` and `elapsedSeconds` calculate the elapsed time between `startTime` and the time when the methods are called. As the time values used are all in units of seconds, the number of milliseconds of elapsed time must be calculated by computing the number of seconds and then multiplying by 1000. Both methods return a value of type `Number`; neither takes any parameters.

The object `Chase` inherits `graphicApplication`. Using the class `FunnyFace`, it creates a funny face in the upper part of the canvas, and constructs a `Text` message, `infoText`, that tells the user to click to start playing the game. When the user clicks for the first time, the funny face moves to a new position on the canvas. If the user clicks on it within `timeLimit` seconds, the user wins. If the user misses the funny face with the click then a message appears indicating that the user missed, and the funny face moves to a new randomly selected position. If the user clicks on the funny face in time, a message is displayed and the user is instructed to click to start the game again. Finally, if the user clicks on the funny face, but too slowly, then the message “Too slow” appears and the funny face is moved.

The code for the chase program is given in Figures 6.7. Because this program will use the classes for timers and funny faces we will import them for use in this program. We

```

// Class allowing calculations of timing between events.
import "sys" as system

type Timer = {
  elapsedSeconds -> Number
  elapsedMilliseconds -> Number
  reset -> Done
}

class timerNew -> Timer {
  // starting time for timing interval
  var startTime: Number := system.elapsed

  // number of milliseconds since timer created or reset
  method elapsedMilliseconds -> Number {
    elapsedSeconds * 1000
  }

  // number of seconds since timer created or reset
  method elapsedSeconds -> Number {
    system.elapsed - startTime
  }

  // reset the start time of the interval
  method reset -> Done {
    startTime := system.elapsed
  }
}

```

Figure 6.6: Timer class

assume that the code for these classes are stored in files named “FaceClassTo.grace” and “Timer.grace”. The two lines

```
import "FaceClassTo" as fc
import "Timer" as tm
```

make the code in these files accessible to the program. Thus we can write `fc.funnyFace` to get access to the class `funnyFace` class and `fc.Draggable` to get access to the type `Draggable`.

The identifier `smileyFace` is associated with a new funny face, `stopWatch` with a new `Timer`, and `lstinlineinfoText` with a new `Text` object. There is a boolean instance variable, `playing`, that keeps track of whether or not the game has actually started, while `newPoint` will represent the next `Point` where `smileyFace` will move to.

The new face, which is constructed a little above the center of the screen, is associated with instance variable `funFace`. The `Text` object `infoText` is created just below the face with a startup message telling the user how to start the game. Notice that the positions of both are based on the width and height of the canvas. Thus if the programmer changes the parameters in the `inherit` expression, the positions will automatically reflect those changes.

Finally the instance variable `playing` is initialized to be false. This variable keeps track of whether the game is in progress and the system is waiting for the user to press the mouse on the funny face, or if the system is simply waiting for the user to press the mouse to start the game.

The `onMousePress` method determines how the program reacts to a mouse press. The conditional statement provides cases for when the game has not yet started (`playing` is false), when the click does not occur in `funFace`, when the user succeeds (the click does occur in `funFace` and within the appropriate time interval), and when the elapsed time is too long. The code in each of those cases is straightforward. In all the cases except when the player clicks on the face in time, the face is moved to a new randomly chosen position.

We will make several improvements to this program in the following sections as we introduce new features of Grace that are useful in writing classes.

Exercise 6.4.1 *This question has to do with the Chase program above.*

- a. *Why didn't we have to check that `playing` is true in the `elseif` clauses of the `onMousePress` method.*
- b. *Why do you think we chose to use `onMousePress` in this program rather than `onMouseClicked`?
Hint: We originally used `onMouseClicked`, but changed after playing the game for a while.*

6.5 Local Variables

So far we have seen how to give names to identifiers used in definitions, instance variables, and formal parameters. There is another kind of definition (and variable) that can be named and used in Grace classes. These are used to hold information that is only needed during the execution of a single method, but need not be remembered after the method completes execution. Definitions of this kind are called local definitions.

In order to better motivate the use of local definitions, let us look back to the `onMousePress` method in the chase program from the last section. Three of the four branches of the if-then-else statement include the following statement

```

dialect "objectdraw"
import "random" as rand
import "FaceClassTo" as fc
import "Timer" as tm

object {
  inherit graphicApplicationSize (400 @ 400)

  def timeToClick: Number = 1.5 // seconds to click on Smiley and win

  def startLocn: Point = (canvas.width / 2) @ (canvas.height / 3) // starting position of face

  def smileyFace: fc.Draggable = fc.funnyFaceAt (startLocn) on (canvas) // smiley face to be chased

  def stopWatch: tm.Timer = tm.timerNew // Timer to see if click fast enough

  var playing: Boolean := false // Is player playing or waiting to start

  def textLocn: Point = (canvas.width / 3) @ (canvas.height / 2) // where to draw infoText

  // Text item to provide instruction/feedback
  def infoText: Text = textAt (textLocn) with ("Click to start chasing the Smiley.") on (canvas)

  // Determine if user won and move smiley face if necessary
  method onMousePress (pt: Point) -> Done {
    if (!playing) then {
      playing := true
      infoText.contents := "Click quickly on the Smiley to win!"
      smileyFace.moveTo ((rand.integerIn (0) to (canvas.width)) @
                          (rand.integerIn (0) to (canvas.height)))
      stopWatch.reset
    }
    elseif {!smileyFace.contains (pt)} then {
      infoText.contents := "You missed!"
      smileyFace.moveTo ((rand.integerIn (0) to (canvas.width)) @
                          (rand.integerIn (0) to (canvas.height)))
      stopWatch.reset
    }
    elseif {stopWatch.elapsedSeconds <= timeToClick} then {
      infoText.contents := "You got the Smiley in {stopWatch.elapsedSeconds} seconds. Click to start over."
      playing := false
    }
    else { // clicked on face, but too slowly
      infoText.contents := "Too slow!"
      smileyFace.moveTo ((rand.integerIn (0) to (canvas.width)) @
                          (rand.integerIn (0) to (canvas.height)))
      stopWatch.reset
    }
  }
}
startGraphics
}

```

Figure 6.7: Chase program

```
smileyFace.moveTo ((rand.integerIn (0) to (canvas.width)) @ (rand.integerIn (0) to (canvas.height)))
```

We prefer not to have repeated code, and we especially don't like to have code that is so complex in the program. Thus it makes sense to provide an identifier with the expression calculating the new position. However, while it is recalculated each time we click, it doesn't make much sense to make it into an instance variable as there is no need to remember the value after the method completes.

To avoid taking up space when it is not needed, we associate this new position with a new identifier *inside* the method. This is called a *local definition* because it is provided inside the method body and thus goes away when the method has completed. It is only accessible *locally*, that is, within that method. The revised code can be found in Figure 6.8.

Local definitions are used to help in complex or repeated calculations for values needed inside a method, but the values need not be retained after the method has completed. Thus they help us save memory space for each item constructed, compared with instance variables, whose space is always retained while the object is in use. If the identifier is associated with different values during the same method invocation then we can define a local variable instead of a local definition. However we will not see much need for local variables until we discuss statements that provide for repeated invocation of a block of code.

As a last improvement, we can simplify the names of types and classes that are imported by giving them new local names. For example, when we access the type `Timer` we have to write `tm.Timer` and when we access the class, we must write `tm.timerNew`. While those each only occur once in this program, if they occurred multiple times these longer names might be harder to read. In that case we could begin the program with

```
import "Timer" as tm
type Timer = tm.Timer
method timerNew -> Timer {tm.timerNew}
```

As you can see, it is easy to give a simpler name to a type, just use a type definition. Objects are also simple, just write `def newName: newType = oldName`.

You must work a bit harder with classes. Classes can be treated as methods as they take parameters and return values. As you can see above this is a bit more wordy. In general the new method definition must also include the parameters in the declaration. In general, we prefer to rename types and objects, but just use the longer names for classes rather than trying to rename them.

Because we have defined `Timer` to be the same as `tm.Timer`, we can now just use `Timer` without the prefix in the rest of the program.

Exercise 6.5.1

- a. *How are local definitions different from instance definitions?*
- b. *Explain why `newPoint` in the revised chase program can be a local definition.*

6.6 What about `graphicApplication`

Let us return to the original objects that we defined in this text. These are the object expressions that inherited from `graphicApplication` and that resulted in popping up a window

```

// Game chasing face around screen

dialect "objectdraw"
import "random" as rand
import "FaceClass" as fc
import "Timer" as tm

object {
  inherit graphicApplicationSize (400 @ 400)

  def timeToClick: Number = 1.5 // seconds to click on Smiley and win

  def startLocn: Point = (canvas.width / 2) @ (canvas.height / 3)
  def smileyFace: fc.Draggable = fc.funnyFaceAt (startLocn) on (canvas) // smiley face to be chased

  def stopWatch: tm.Timer = tm.timerNew // Timer to see if click fast enough

  var playing: Boolean := false // Is player playing or waiting to start

  def textLocn: Point = (canvas.width / 3) @ (canvas.height / 2)
  def infoText: Text = textAt (textLocn) with ("Click to start chasing the Smiley.")
    on (canvas) // Text item to provide instruction/feedback

  // Determine if user won and move smiley face if necessary
  method onMousePress (pt: Point) -> Done {
    def newPoint: Point = (rand.integerIn (0) to (canvas.width)) @ (rand.integerIn (0) to (canvas.height))
    if (!playing) then {
      playing := true
      infoText.contents := "Click quickly on the Smiley to win!"
      smileyFace.moveTo (newPoint)
      stopWatch.reset
    }
    elseif {!smileyFace.contains (pt)} then {
      infoText.contents := "You missed!"
      smileyFace.moveTo (newPoint)
      stopWatch.reset
    }
    elseif {stopWatch.elapsedSeconds <= timeToClick} then {
      infoText.contents := "You got the Smiley in {stopWatch.elapsedSeconds} seconds. Click to start over."
      playing := false
    }
    else { // clicked on face, but too slowly
      infoText.contents := "Too slow!"
      smileyFace.moveTo (newPoint)
      stopWatch.reset
    }
  }
}
startGraphics
}

```

Figure 6.8: Chase game with a local definition newPoint.

```

type GraphicApplication = Application & type {
  // canvas holds graphic objects on screen
  canvas -> DrawingCanvas

  // Respond to a mouse click (press and release) in the canvas at the given
  // point.
  onMouseClick (mouse : Point) -> Done

  // Respond to a mouse press in the canvas at the given point.
  onMousePress (mouse : Point) -> Done

  // Respond to a mouse release in the canvas at the given point.
  onMouseRelease (mouse : Point) -> Done

  // Respond to a mouse move in the canvas at the given point.
  onMouseMove (mouse : Point) -> Done

  // Respond to a mouse drag (move during press) in the canvas at the given
  // point.
  onMouseDrag (mouse : Point) -> Done

  // Respond to a mouse entering the canvas at the given point.
  onMouseEnter (mouse : Point) -> Done

  // Respond to a mouse exiting the canvas at the given point.
  onMouseExit (mouse : Point) -> Done

  // must be invoked to create window and its contents as well as prepare the
  // window to handle mouse events
  startGraphics -> Done
}

```

Figure 6.9: The type `GraphicApplication` of graphic programs in `objectdraw`

that supported interaction using mouse events.

Like all the other objects we have talked about, these objects also have a type. It is the type `GraphicApplication`, whose definition is given in Figure 6.9.

The methods listed in the type are all methods that we have been using in our programs. The first method provides us with the canvas on which we place our graphical items. The next seven methods are those used to respond to mouse events: clicking, pressing, releasing, moving, dragging, as well as entering and exiting the canvas with the mouse.

The last method, `startGraphics`, is what we have been using to pop up a window with the `canvas` installed in it, as well as enabling the object to start responding to the mouse events.

One curious thing is that the objects we have been defining have not had all of the methods defined above. Typically they only have a few of the mouse-event handling methods. So, how could these objects we are defining have type `GraphicApplication`?

The secret is in the **inherit** clause. If we define an object (or class) that **inherit** `graphicApplicationSize`

(w @ h) then it will “inherit” the versions of these methods that are defined in the class `graphicApplication`. Each of the mouse-event-handling methods is defined in `graphicApplication`, but they have an empty body. In other words they do nothing at all. When you write a method `onMousePress` in your object, it will override (replace) the version of the method in `graphicApplication` with the one you wrote.

Thus in your program, you can override as many of these methods as you like. If you perform a mouse action corresponding to a method that you have not overridden, it will perform the action from `graphicApplication`. I.e., it will do nothing. However if you perform a mouse action corresponding to a method that you have overridden (by including the method in your object), then it will execute your code instead.

This is one of the reasons that you must be sure not to misspell the names of these methods or to give them the wrong number or type of parameters in your programs. If you make one of those mistakes then your code will not override the code from `graphicApplication`, and so the program will execute the “do-nothing” code from `graphicApplication` rather than the code you wrote.

Do *not* override the methods `canvas` and `startGraphics` in your object as the definitions from `graphicApplication` are necessary to provide the expected behavior in your program!

From now on we will provide names for these objects that we have been defining that control the behavior of your program. For example, we could write the chase program in the last section as

```
def chaseFace: GraphicApplication = object {  
  inherit graphicApplicationSize (400 @ 400)  
  ...  
}
```

6.7 Summary

In this section we showed how to design new objects and classes in Grace. Object definitions are used if we only need a single copy, while classes are useful if we need more than one copy of similar objects. Thus we can design classes to represent interesting collections of objects with similar behaviors. We illustrated these ideas by building classes `FunnyFace` and `Timer`. The use of these classes made it much simpler to define programs to drag around a funny face on the screen and a game in which the user is to click on the funny face within a small time interval.

Objects and classes are composed of

- Definitions providing names for objects.
- Instance variables representing the features of the object or class that can be changed at run time
- Methods representing the possible behavior of the objects of the class.

We showed how a method could call another method, `m`, of the same class by writing `self.m(...)` or just `m(...)`.

We introduced local definitions and variables in order to hold values that need only be saved temporarily during the execution of a single method. We also introduced the type `GraphicApplication` of the objects controlling the execution of our programs and explained the function of the **inherit** clause in those objects.

6.8 Chapter Review Problems

Exercise 6.8.1 *Define the following Grace terms:*

- a. instance of a class
- b. return type
- c. local definition

Exercise 6.8.2 *What is meant by formal parameters and arguments and how do they relate to each other?*

Exercise 6.8.3 *Write the following methods for `FunnyFace`*

- a. `removeFromCanvas`
- b. `x`
- c. `setEyeColor`

Exercise 6.8.4 *Why does the following `moveTo` method not do what is expected?*

```
method moveTo (x: Number, y: Number) {
  head.moveTo (x, y)
  mouth.moveTo (x, y)
  leftEye.moveTo (x, y)
  rightEye.moveTo (x, y)
}
```

Exercise 6.8.5 *What would you have to do to `Chase` if you wanted to have the console display how long the game has been running when the mouse exits the window? Write the `onMouseExit` method.*

6.9 Programming Problems

Exercise 6.9.1 *Create a class called `Man` which looks like the stick-man in Figure 6.10. A `Man` should have the following methods in addition to a constructor: `move`, `moveTo`, `color:=`, `color`, `removeFromCanvas`, `getHead` and `contains`.*

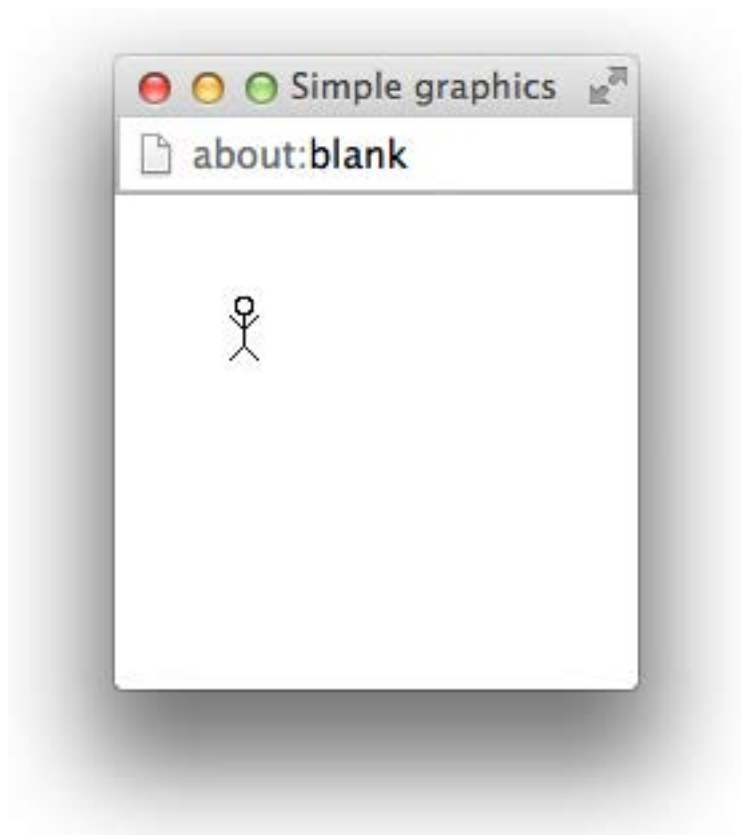


Figure 6.10: A small stick-man

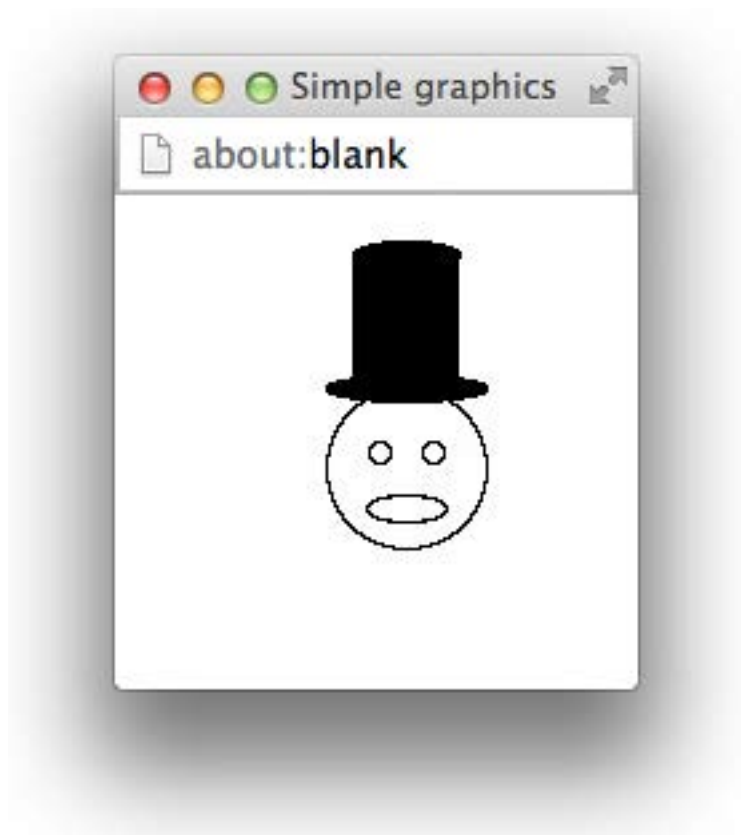


Figure 6.11: FunnyFace with a TopHat

Exercise 6.9.2 Write a class called `TopHat` which will be a hat like that in Figure 6.11 for a `FunnyFace`. Modify the `Chase` class so that there is also a `TopHat` on the `FunnyFace`. If the user clicks on the face or the hat in time, it will generate the normal correct response. However, if the user is too slow or misses the hat and face then the hat will change color. The hat should switch color either from red to black or from black to red.

Hint: You may want to add additional accessor methods to `FunnyFace`.

Chapter 7

Control Structures

This chapter discusses *control structures*. These are mechanisms that allow a programmer to control the order in which program statements are executed.

You have already learned a bit about control structures. For example, blocks of code in braces indicate that sequential execution of those statements will take place. You also learned about the conditional statement in Chapter 4. In fact, you saw a variety of different forms of the conditional statement. All of the variants of the conditional statement allow the programmer to specify a choice of statements to be executed, depending upon conditions that arise during the running of a program. This chapter expands on what you learned earlier, providing advice for writing clear conditional statements even in cases that are complex.

This chapter also introduces the *while* statement. This is one of several ways to express the idea of executing code repeatedly in Grace. The while statement will allow you to create complex drawings. Learning about the while statement will also serve as a foundation for Chapter 9, in which you will generate simple animations.

7.1 Repetition and while Loops

Many of the programs we have written have involved repetition. The Craps example from Chapter 4, for example, involved repeatedly rolling two dice. The grid shown in Figure 2.11 and repeated in Figure 7.8 involved repeatedly drawing rectangles to form a grid pattern. These examples, and many of the others we have shown you, have something in common – their repetitive behavior is controlled by a user through the mouse. But what if we want to draw a complex pattern with just a single click (or with no click at all!)? This requires a mechanism for telling Grace to perform an action repeatedly. In this section we introduce the *while* statement, which is one of several ways to express the notion of executing code repeatedly in Grace.

Let's begin our discussion of repetition with a simple example. Consider the drawing of a grassy field shown in Figure 7.1. It is easy to imagine how this scene might be created. The sun is a filled oval, the base of the grass is a filled rectangle, and each blade of grass is a line. The only problem is that, with the tools you have so far, the process of creating so many blades of grass would be horribly tedious. The code would be extremely repetitive, as illustrated in Figure 7.2. For each individual blade of grass drawn on the canvas, we must

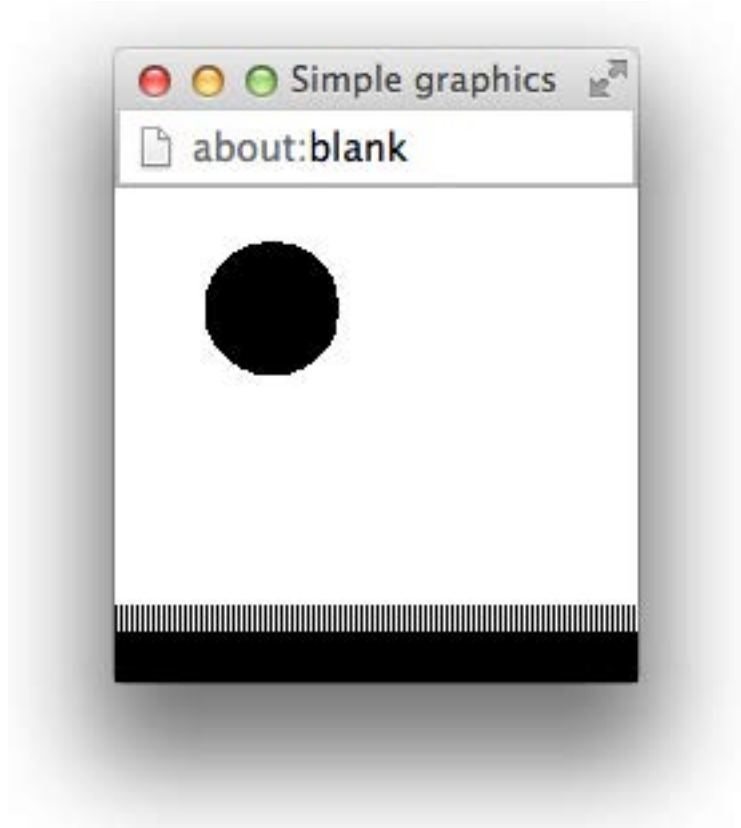


Figure 7.1: Grass

create the positions of the endpoints of the line and then create the line itself.

In the code in the figure we used the variable `nextGrassX` to keep track of the x-component of the grass blade and simply changed the value before repeating the same block of three statements to create the positions of the end points of the line and the line itself. This made it easier to cut and paste most of the code and just change the assignment to `nextGrassX` for each new blade.

Even so, this is still extremely tedious. Imagine how painful it would be if you wanted to draw hundreds of blades of grass.

Naturally, it would be nice to think about how we could make this a bit less tedious. One thing we might notice is that the only difference between the four line blocks of code to draw each blade is that the variable `nextGrassX` increases by 4 pixels. So we might think about how we can say something like, “repeatedly draw a line with an x coordinate 4 more than the previous one.”

What we can do is to initialize `nextGrassX` to 2 as before, but then replace the further assignments with

```
nextGrassX := nextGrassX + 4
```

As we saw earlier with similar examples, this assignment increases the value of `nextGrassX` by 4 each time it is executed. Thus it will go from 2 to 6 to 10 to ..., just as before. The code

dialect "objectdraw"

```
def tediousGrass: GraphicApplication = object {  
  inherit graphicApplicationSize (400 @ 400)  
  
  // height of each blade of grass  
  def grassHeight: Number = 20  
  
  // draw ground  
  def groundY: Number = 300  
  filledRectAt (0 @ groundY) size (canvas.width @ (canvas.height - groundY)) on (canvas)  
  
  // draw sun  
  filledOvalAt (80 @ 40) size (50 @ 50) on (canvas)  
  
  // add the blades of grass  
  var nextGrassX: Number := 2  
  var grassBottom: Point := nextGrassX @ groundY  
  var grassTop: Point := grassBottom + (0 @ (-grassHeight))  
  lineFrom (grassBottom) to (grassTop) on (canvas)  
  
  nextGrassX := 6  
  grassBottom := nextGrassX @ groundY  
  grassTop := grassBottom + (0 @ (-grassHeight))  
  lineFrom (grassBottom) to (grassTop) on (canvas)  
  
  nextGrassX := 10  
  grassBottom := nextGrassX @ groundY  
  grassTop := grassBottom + (0 @ (-grassHeight))  
  lineFrom (grassBottom) to (grassTop) on (canvas)  
  
  startGraphics  
}
```

Figure 7.2: Code for drawing the grass scene.

```

// add the blades of grass}
var nextGrassX: Number := 2

var grassBottom: Point := nextGrassX @ groundY
var grassTop: Point := grassBottom + (0 @ (-grassHeight))
lineFrom (grassBottom) to (grassTop) on (canvas)

nextGrassX := nextGrassX + 4

grassBottom := nextGrassX @ groundY
grassTop := grassBottom + (0 @ (-grassHeight))
lineFrom (grassBottom) to (grassTop) on (canvas)

nextGrassX := nextGrassX + 4

grassBottom := nextGrassX @ groundY
grassTop := grassBottom + (0 @ (-grassHeight))
lineFrom (grassBottom) to (grassTop) on (canvas)

nextGrassX := nextGrassX + 4
...

```

Figure 7.3: Alternate code for drawing grass. Note the pattern that has emerged in the code.

that draws the grass can now be expressed as in Figure 7.3

This has a number of advantages over the previous approach. For instance, the process of creating the multiple blades of grass involves cutting and pasting without modification of any of the individual lines. Unfortunately, this is still painfully tedious.

An alternative would be to have the user of this class draw each blade of grass with a mouse click, as shown in Figure 7.4. Again, `nextGrassX` holds the x coordinate of the blade of grass to be drawn. The `onMouseClicked` method contains all four of the repeated lines from the earlier version (though notice that to get exact repetitions we moved the original assignment of 2 to `nextGrassX` to outside the method and instead included the statement to increase the value of `nextGrassX` by 4 after the other three statements).

The only extra feature here is that we check whether the x coordinate is within the bounds of the canvas, so that a line is only drawn if it will be visible. While this is fairly straightforward to write, it would be terribly dull for the user to click enough to fill the window with grass.

Exercise 7.1.1 *What would have happened if we had included the code to initialize `nextGrassX` to 2 in the `onMouseClicked` method?*

Fortunately, Grace provides several constructs for expressing repetition concisely. One of these is the `while` statement, sometimes called the `while` loop.

Figure 7.5 defines a class that draws the entire grass scene. The important change from the earlier programs is that drawing of the grass is now expressed as a `while` loop rather than in the `onMouseClicked` method. Notice that all of the important elements of the drawing algorithm were already in place in Figure 7.4: drawing a single line, changing the value of

```

// add the blades of grass
var nextGrassX: Number := 2
var grassBottom: Point
var grassTop: Point

method onMouseClick(pt: Point) -> Done {
  if (nextGrassX < canvas.width) then {
    grassBottom := nextGrassX @ groundY
    grassTop := grassBottom + (0 @ (-grassHeight))
    lineFrom(grassBottom) to (grassTop) on (canvas)

    nextGrassX := nextGrassX + 4
  }
}

```

Figure 7.4: Code that allows a user to draw the grass in the grass scene by clicking.

`nextGrassX`, and deciding whether more should be drawn. All that was required to write a while loop was modifying the syntax a bit.

Let's look at the structure of a *while* statement in general first, so that we can better analyze how the example loop draws grass. The syntax of the *while* statement is as follows:

```

while {condition} do {
  // statements to be repeated
}

```

The *condition* specifies the circumstances under which the statements in the loop are to be executed. It must be an expression that describes a boolean value. If *condition* is false, the statement is skipped, and control moves to the statement immediately following the while loop.

If *condition* is true, the body of the loop is executed. Once the body has been executed, *condition* is tested again. If *condition* is false, execution of the loop is terminated, and control moves to the statement following the loop. Otherwise, the statements in the body of the loop are executed again, and so on. Braces mark the beginning and end of the block of statements to be repeated. These statements are sometimes called the *body* of the loop.

Now we can look at the structure of our grass drawing while statement in Figure 7.5 in detail. We use the variable `nextGrassX` of type `Number` to hold the x coordinate of the next blade of grass to be drawn, just as we did before. Drawing begins at the extreme left-hand side of the canvas. Before drawing each blade of grass, we check whether `nextGrassX` is within the bounds of the canvas. If so, we draw a line and advance the x coordinate, `nextGrassX`, and then return to the top of the while to test the condition again. If the newly updated x coordinate is still within the bounds of the canvas, we draw a line, advance `nextGrassX` and again go back to test the condition. If the condition evaluates to false, we skip the body of the loop. We can express this construct in words as follows: as long as `nextGrassX` is within the canvas, draw a new line and advance `nextGrassX` to the next drawing spot.

We can be certain that the while loop will terminate. If `nextGrassX` is initially set to two and then incremented each time through the loop, it will eventually exceed `canvas.width`, terminating the loop.

```

dialect "objectdraw"

def grass: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 400)

  def grassHeight: Number = 20

  // draw ground
  def groundY: Number = 300
  filledRectAt (0 @ groundY) size (canvas.width @ (canvas.height - groundY)) on (canvas)

  // draw sun
  filledOvalAt (80 @ 40) size (50 @ 50) on (canvas)

  // add the blades of grass
  var nextGrassX: Number := 2
  var grassBottom: Point
  var grassTop: Point

  while {nextGrassX < canvas.width} do {
    grassBottom := nextGrassX @ groundY
    grassTop := grassBottom + (0 @ (-grassHeight))
    lineFrom(grassBottom) to (grassTop) on (canvas)

    nextGrassX := nextGrassX + 4
  }

  startGraphics
}

```

Figure 7.5: Using a while loop to draw blades of grass.

Notice the similarity between this version of grass drawing and the version in Figure 7.4. In the new version, there are only two differences from the original. First, we replaced `if` with `while`. Second (and more subtly), we surrounded the condition in the `while` loop with braces rather than parentheses.

Braces in Grace are very important. They signify that an expression may be evaluated multiple times. When we look at the `while` loop in the program, it is not obvious how many times the condition and the body of the `while` loop will be evaluated. The condition will be evaluated at least once, but probably lots of times. The body of the `while` loop might not be evaluated at all (if `nextGrassX >= canvas.width` initially), but likely will also be evaluated many times.

The general rule in Grace is that if an expression may be evaluated many times (or not at all) then it should be surrounded by braces. It is interesting to compare the difference between `if` statements and `while` loops in these features. In an `if` statement like

```
if (condition) do {  
    // statements to be executed if condition is true  
}
```

the condition is always executed exactly once (and hence does not require braces), while the set of statements may not be executed at all (if condition evaluates to **false**), and thus is surrounded by braces.

In the `while` statement

```
while {condition} do {  
    // statements to be repeated  
}
```

the condition may be executed multiple times and hence is surrounded by braces. The set of statements also may be executed multiple times and hence must be surrounded by braces.

Code surrounded by braces is called a *block* in Grace. Blocks are ubiquitous in Grace. Blocks form the body of **class** and **object** expressions as well as the body of **method** declarations. In general, blocks may contain multiple statements and may include both type and regular definitions, as well as variable declarations.

If a block is to return a value, the value returned is that of the last expression in the block. For example, the condition in a `while` loop is typically only one boolean valued expression, and its value is returned every time it is executed. Technically, we could put multiple statements in the condition (*though we won't*), as long as the last statement is an expression that evaluates to a boolean.

Exercise 7.1.2 *What is the difference between the function of a `while` loop and an `if` statement with no else part?*

Exercise 7.1.3 *Assume `gameOver` is a Boolean variable that is initialized to **false**. What is the difference between the output of the following two control structures?*

```
a. if (!gameOver) then {  
    print "Roll again."  
}  
  
while {!gameOver} do {  
    print "Roll again."  
}
```

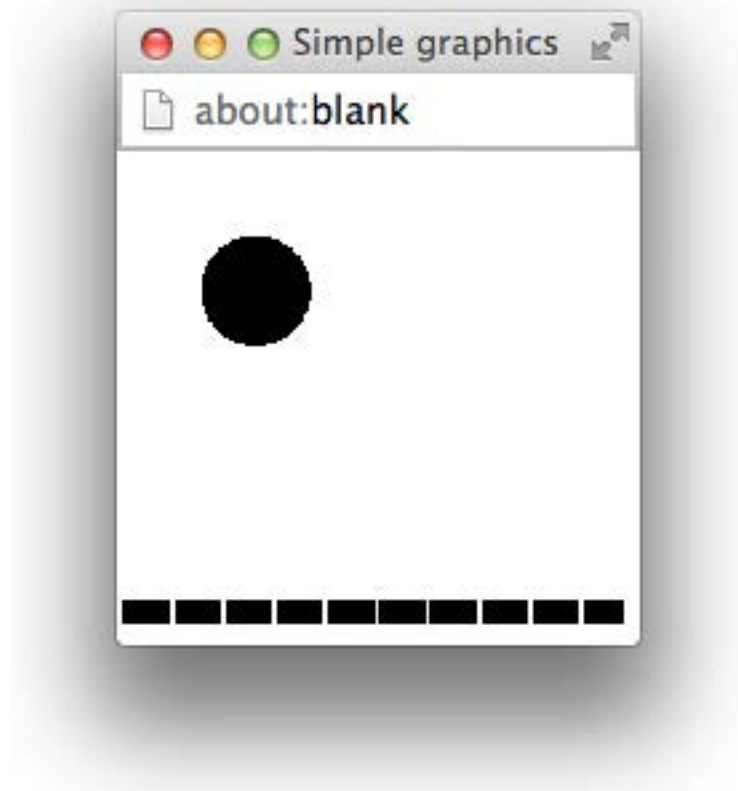


Figure 7.6: Bricks

7.2 More Examples Using while Loops

We now move from the country to the city and consider the scene in Figure 7.6. In the picture we see a moon over a city rooftop. Again, we see a pattern in the picture – the row of bricks that form the top of the roof. A loop is an appropriate construct to use to help us draw the bricks, as shown in the code in Figure 7.7.

Let's consider in some detail how we might go about writing the while loop in Figure 7.7. Each brick is a filled rectangle of a specified width and height. The only difference between any two bricks is their position on the canvas. More specifically, the difference is in the x coordinate that specifies the left edge of each brick. The y coordinates are all the same, as the tops of the bricks line up with each other. Since the top, width, and height of a brick are all fixed, we define constants for each of these. The x coordinate of the upper left corner of each brick will change, however. We use the variable `brickPosition` to hold this information.

We start with `brickPosition` set to 0, so that the first brick is drawn at the left edge of the canvas. To draw a single brick, we construct a new filled rectangle,

```
filledRectAt (brickPosition) size (brickWidth @ brickHeight) on (canvas)
```

This is what we want to do over and over in our loop, but each time a new brick is created, we need to be sure that `brickPosition` has been changed. We move `brickPosition` to the right by the width of a brick and also a small amount, `brickspacing`, to account for the spacing between

```

// Draw a row of bricks
dialect "objectdraw"

def bricks1Row: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 400)

  // brick dimensions and top
  def brickWidth: Number = 40
  def brickHeight: Number = 20
  def brickTop: Number = canvas.height - brickHeight - 10

  // spacing between bricks
  def brickSpacing: Number = 2

  def moonLocation: Point = 60 @ 60 // location of moon

  // draw moon
  filledOvalAt (moonLocation) size (50 @ 50) on (canvas)

  // add the blades of grass
  var brickPosition: Point := 0 @ brickTop

  // draw 1 row of bricks across canvas
  while {brickPosition.x < canvas.width} do {
    filledRectAt (brickPosition) size (brickWidth @ brickHeight) on (canvas)

    brickPosition := brickPosition + ((brickWidth + brickSpacing) @ 0)
  }

  startGraphics
}

```

Figure 7.7: A while loop to draw a row of bricks.

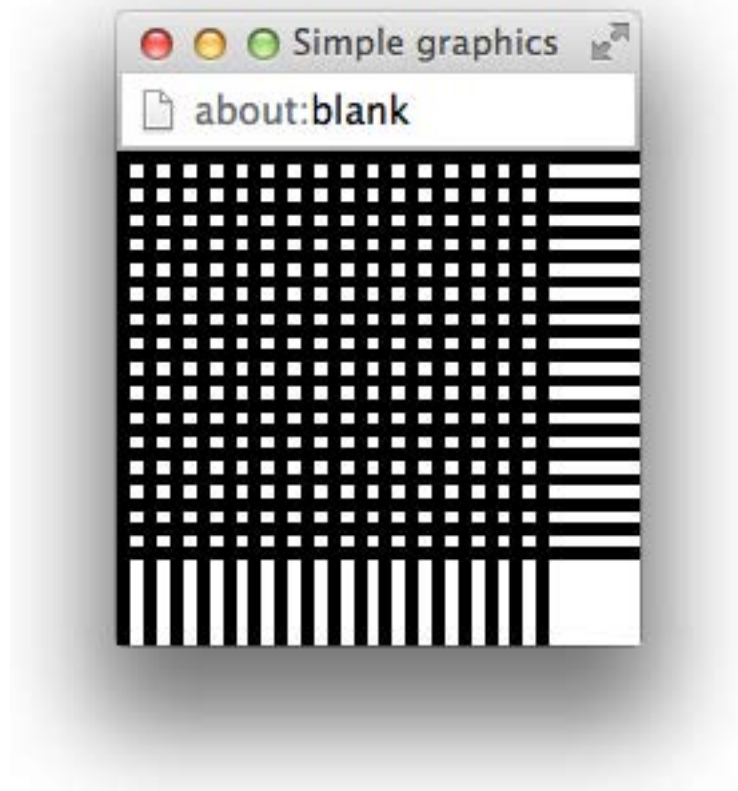


Figure 7.8: Grid

bricks. These two lines, the construction of a brick and the movement of `brickPosition` form the body of the loop.

Of course, we don't want an infinite number of bricks to be drawn. We only want to construct new bricks if they will be visible on the canvas. This helps us to formulate the expression that describes the condition under which our loop is to continue executing:

```
while {brickPosition.x < canvas.width} do {...}
```

That is, the loop should be executed as long as the x coordinate of the next brick is less than the width of the canvas.

As a slightly more complex example, consider the drawing in Figure 7.8, which should be familiar to you from Chapter 1. The grid has a clear pattern, so it is a good candidate for a while loop. The way the grid was drawn in Chapter 1 is a good starting point for thinking about how to write the while loop here. There, with each click of the mouse, two long, thin rectangles were drawn. The position of the vertical rectangle was held in a variable `verticalCorner` and the position of the horizontal rectangle was in `horizontalCorner`. With each mouse click, the two rectangles were drawn, and then the vertical and horizontal corners were shifted. These statements will form the body of our loop, as shown in Figure 7.9.

We also need to specify a condition for the while statement. Let's draw grid lines (i.e., rectangles) as long as they are at least partially visible on the canvas. Since vertical rectangles are drawn from left to right on the canvas, we need to be sure that the x coordinate of the

```

while {(verticalCorner.x < canvas.width) || (horizontalCorner.y < canvas.height)} do {
    filledRectAt (verticalCorner) size (5 @ canvas.height) on (canvas )
    filledRectAt (horizontalCorner) size (canvas.width @ 5) on (canvas )
    verticalCorner := verticalCorner + (10 @ 0)
    horizontalCorner := horizontalCorner + (0 @ 10)
}

```

Figure 7.9: A while loop to draw the grid.

vertical corner is within the width of the canvas. Analogously, since horizontal rectangles are drawn from the top of the canvas to the bottom, we need to check that the y coordinate of the horizontal corner is within the height of the canvas. As long as either of these is true, that is, as long as either rectangle is visible, we execute the body of the loop.

Exercise 7.2.1 Write a program called *AlmostATarget* that draws a variation of a target on the canvas. The biggest circle should be 100 pixels wide and 100 pixels long, while the smallest circle should be 20 pixels wide and 20 pixels long. Each circle should be 20 pixels smaller than the circle surrounding it in both length and height. The circles should all have their upper-left corner at the Point (10, 10). Your drawing should resemble the picture in Figure 7.10.

7.3 Loops that Count

Let's now go back to our example in which we drew bricks across a canvas. In Figure 7.7 we drew bricks, starting at the left edge of the canvas and continuing until we passed the right edge of the canvas. But what if we, instead, wanted to draw precisely ten bricks, starting at the same initial position.

A while loop can certainly help us accomplish this task. In fact, much of our work is already done. The main difference between the loop in Figure 7.7 and the one we want to write will be the condition on the while loop. In the earlier version, we drew bricks while the x position of the next brick was within the visible canvas. This time we will draw bricks as long as the number we have drawn so far is less than ten.

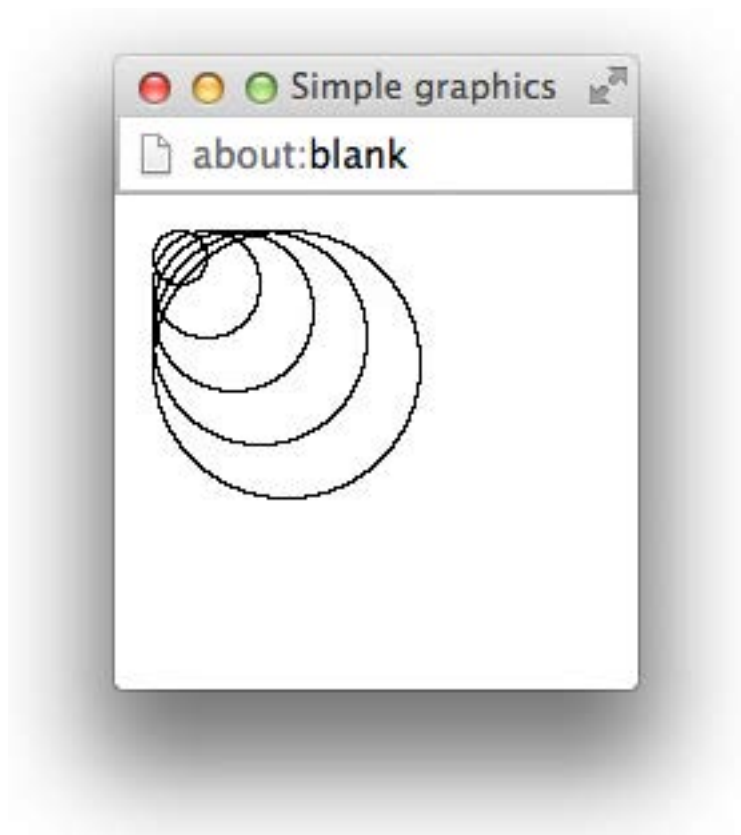
Figure 7.11 shows the code to draw precisely ten bricks of the sort pictured in Figure 7.6. As just discussed, the while loop is to be executed as long as the number of bricks drawn so far, `brickCount`, is less than the total number desired. We initialize the counter variable `brickCount` to 1 before the loop. Then each time through the loop, i.e., each time we draw a brick, we increment the counter. As you will see in Chapters 9 and 12, the loop shown in Figure 7.11 follows a general pattern of counting loops using while. The pattern is as follows:

```

var counter: Number := initialValue
while {counter <= lastValue} do {
    // do stuff
    counter := counter + 1
}

```

Exercise 7.3.1 Write a program that uses a while loop to draw a ruler on the canvas. The ruler should show 12 inches and include tick marks for each quarter of an inch. (The ruler



b.

Figure 7.10: The picture for the AlmostBullseye exercise

```

// Draw a row of 10 bricks
dialect "objectdraw"

object {
  inherit graphicApplicationSize (500 @ 400)

  def numBricksToDraw: Number = 10
  def brickWidth: Number = 40
  def brickHeight: Number = 20
  def brickTop: Number = canvas.height - brickHeight - 10
  def brickSpacing: Number = 2
  def moonInset: Number = 60

  // draw moon
  filledOvalAt (moonInset @ moonInset) size (50 @ 50) on (canvas)

  // add the bricks
  var brickPosition: Point := 0 @ brickTop
  var brickCount: Number := 1

  while {brickCount <= numBricksToDraw} do {
    filledRectAt (brickPosition) size (brickWidth @ brickHeight) on (canvas)
    brickPosition := brickPosition + ((brickWidth + brickSpacing) @ 0)
    brickCount := brickCount + 1
  }

  startGraphics
}

```

Figure 7.11: A while loop to draw a row of exactly ten bricks.

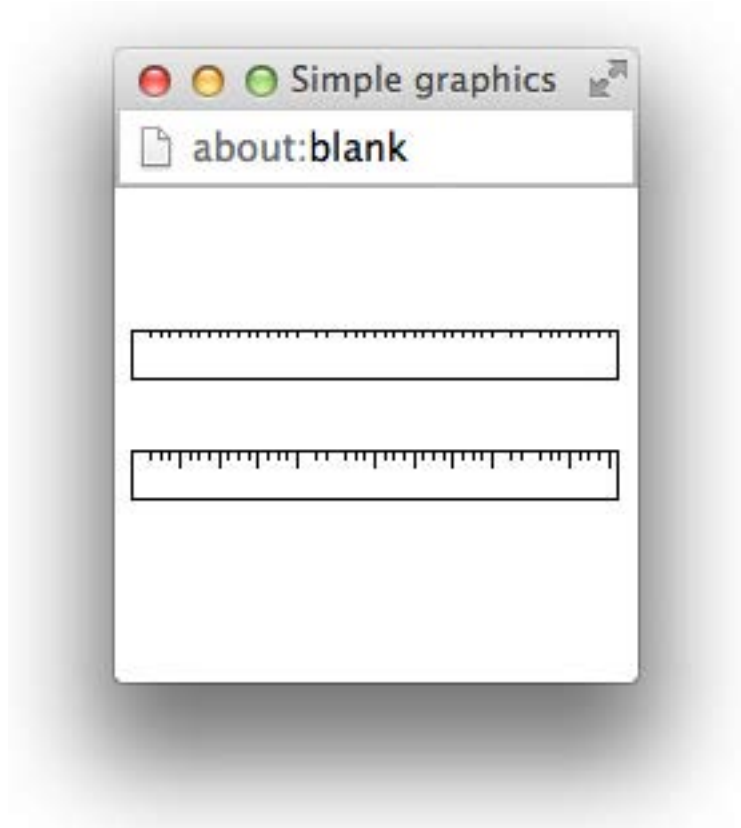


Figure 7.12: Pictures of rulers for exercises 7.3.1 and 7.3.2

does not need to be to scale!) The result should look similar to the ruler shown on top in Figure 7.12.

Exercise 7.3.2 *Modify your program from Exercise 7.3.1 so that the tick marks for each inch are longer than those for the fractions of an inch. The result should look similar to the ruler shown on the bottom in Figure 7.12.*

7.4 Nested Loops

The statements that make up the body of a while loop may be any legal Grace statements. In fact, a loop body might contain another loop. To understand this notion of *nested loops*, consider the brick wall shown in Figure 7.13. By this point, you can pretty comfortably imagine how you would draw a single row of bricks. To draw the entire wall, we simply need to draw a row of bricks repeatedly, starting at a new position each time. The complete code to draw the brick wall is given in Figure 7.14.

The variable `brickPosition` contains the position of the next brick to be drawn. The variables `brickCountInRow` and `brickLevel` are counter variables. The first counts the number of the bricks

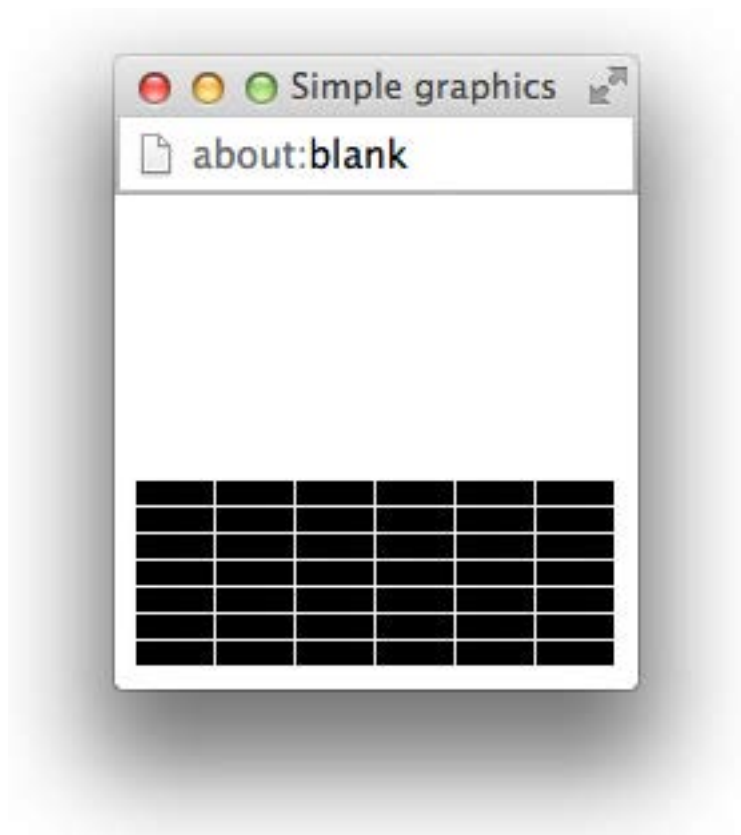


Figure 7.13: A simple brick wall

```

// Draw a brick wall
dialect "objectdraw"

def brickWall: GraphicApplication = object {
  inherit graphicApplicationSize (450 @ 400)

  // brick dimensions and spacing
  def brickWidth: Number = 40
  def brickHeight: Number = 20
  def brickSpacing: Number = 2

  // starting coords of wall
  def wallX: Number = 0
  def wallY: Number = canvas.height - brickHeight - 15

  def wallHeight: Number = 5 // number of levels of bricks to be drawn
  def wallWidth: Number = 10 // Number of bricks in a row of wall

  def moonLocation: Point = 60 @ 60 // location of moon

  // draw moon
  filledOvalAt (moonLocation) size (50 @ 50) on (canvas)

  // number of levels of bricks so far
  var brickLevel: Number := 1
  var brickY: Number := wallY

  // add the bricks
  while {brickLevel <= wallHeight} do {
    var brickCountInRow: Number := 1 // number of bricks in current row

    // location of first brick in this row
    var brickPosition: Point := wallX @ brickY

    // Draw 1 row of bricks
    while {brickCountInRow <= wallWidth} do {
      filledRectAt (brickPosition) size (brickWidth @ brickHeight) on (canvas)
      brickPosition := brickPosition + ((brickWidth + brickSpacing) @ 0)
      brickCountInRow := brickCountInRow + 1
    }

    brickY := brickY - brickHeight - brickSpacing
    brickLevel := brickLevel + 1
  }
  startGraphics
}

```

Figure 7.14: Nested loops to draw a brick wall

```

dialect "objectdraw"
def brickWallAlternating: GraphicApplication = object {
  inherit graphicApplicationSize (450 @ 400)

  // brick dimensions and spacing
  def brickWidth: Number = 40
  def brickHeight: Number = 20
  def brickSpacing: Number = 2

  def wallX: Number = 0 // starting coords of wall
  def wallY: Number = canvas.height - brickHeight - 15

  def wallHeight: Number = 7 // number of levels of bricks to be drawn
  def wallWidth: Number = 10 // Number of bricks in a row of wall

  def moonLocation: Point = 60 @ 60 // location of moon

  // draw moon
  filledOvalAt (moonLocation) size (50 @ 50) on (canvas)

  // level of bricks working on
  var brickLevel: Number := 1
  var brickY: Number := wallY

  // add the bricks
  while {brickLevel <= wallHeight} do {
    var brickCountInRow: Number := 1 // brick number in current row
    var brickPosition: Point := wallX @ brickY

    // Different rows start offset from each other
    if ((brickLevel % 2) == 0) then {
      filledRectAt (brickPosition) size ((brickWidth / 2) @ brickHeight) on (canvas)
      def lastBrickPosn: Point =
        (wallX + wallWidth * (brickWidth + brickSpacing) - brickWidth / 2) @ brickY
      filledRectAt (lastBrickPosn) size ((brickWidth/2 - brickSpacing) @ brickHeight) on (canvas)
      brickPosition := brickPosition + ((brickWidth / 2 + brickSpacing) @ 0)
      brickCountInRow := 2
    }

    // Draw a row of bricks
    while {brickCountInRow <= wallWidth} do {
      filledRectAt (brickPosition) size (brickWidth @ brickHeight) on (canvas)
      brickPosition := brickPosition + ((brickWidth + brickSpacing) @ 0)
      brickCountInRow := brickCountInRow + 1
    }
    brickY := brickY - brickHeight - brickSpacing
    brickLevel := brickLevel + 1
  }
  startGraphics
}

```

Figure 7.15: Nested loops to draw an alternating brick wall

already drawn in the current level, while the second counts the number of levels the program has completed. Both of these are initialized to 1.

The inner loop is responsible for drawing a single row of bricks. We draw each brick as a filled rectangle. After a brick is drawn, `brickPosition` is modified using the `+` method to refer to the position of the next brick on the current level. The counter `brickCountInRow` is incremented as well. The inner loop draws bricks as long as the desired number of bricks in the level has not yet been reached.

The outer loop is responsible for drawing several rows of bricks. Each time the loop executes, a new row is drawn. Once each row has been completed, `brickY` is modified to indicate the y position of the next row. (Note that we've chosen to draw the wall from the bottom up.) In addition, the counter variable `level` is incremented to indicate that another level of the wall is complete.

Before each level is drawn by the inner loop, we reset `brickCountInRow` back to 1 and `brickPosition` to the position of the first brick at the beginning of the next row.

Exercise 7.4.1 *What would happen if we moved the first two lines of the outer while loop, i.e., the lines:*

```
var brickCountInRow: Number := 1 // brick number in current row
var brickPosition: Point := wallX @ brickY
```

to just before that while loop.

A modification of the code in Figure 7.14 would allow us to draw a picture like that in Figure 7.16. The key here is to note that the odd levels (i.e., the bottom, the third from the bottom, etc.) are exactly like the rows of bricks drawn in Figure 7.13. The even levels are slightly different. Therefore, before drawing a row, we first determine whether it should be drawn as before or not. We do this by checking whether the value of `level` is evenly divisible by 2. If so, i.e., if we are on an even level, if not, odd. If we are on an even level, then we draw two half-size bricks at the left and right ends of the wall. We then increase `brickCountInRow` to 2, since the two half-size bricks make up one full brick. We also set the value of `brickPosition` so that we can begin drawing the remainder of the bricks at that level in the right position. Everything else is as before. The code for drawing the better brick wall is shown in Figure 7.15.

Exercise 7.4.2 *Modify your program from Exercise 7.3.2 so that your solution makes use of nested loops.*

7.5 Style Guidelines for Control Structures

In this section we provide some style guidelines for writing control structures. As we have seen in this chapter and in Chapter 4, while loops and if statements can be quite complex. Our aim is to benefit from the power we gain from these statements, while keeping our programs as simple and understandable as possible.

It is extremely important to write clear and concise code not only for the sake of others who might read it (including your teachers!), but also for *yourself*. Programmers sometimes waste a great deal of time trying to understand code that they wrote themselves months,

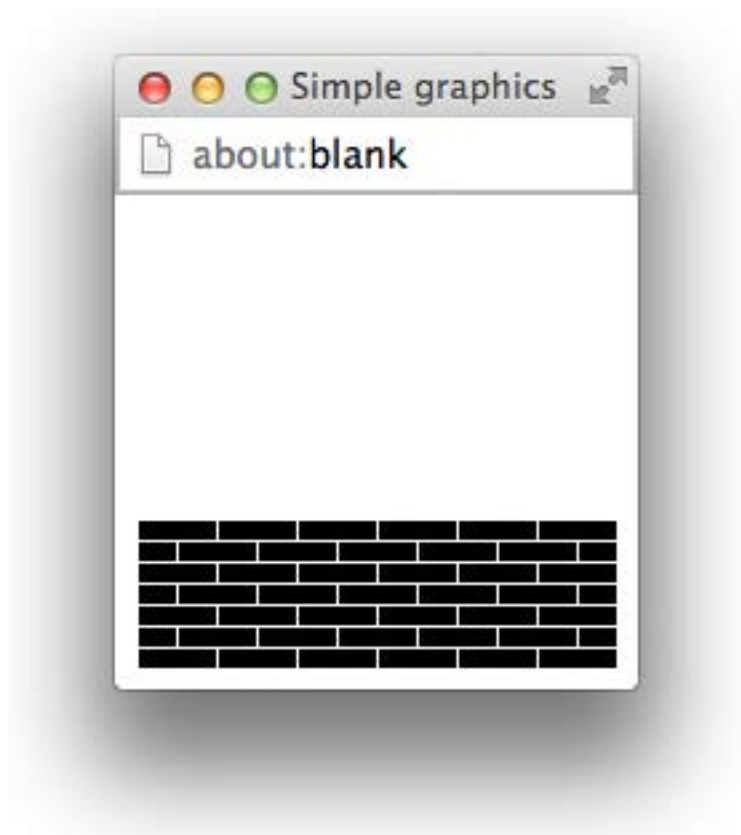


Figure 7.16: A better brick wall

weeks, or even days earlier because they did not pay sufficient attention to clarity and style. As a result we consider it essential to learn and use good coding style.

The “bad” examples we illustrate here are legal Grace code. However, they can be replaced by code that is much simpler and easier to understand.

Avoid empty if–parts. We mentioned in Chapter 4 that an if statement can omit the else–part if it is not needed. However, what if we need the else–part, but not the if–part of an if statement? For example, suppose we want to increment a variable named `counter` exactly when `point` is not contained in a rectangle called `box`, but do nothing otherwise. We could write:

```
if (box.contains (point)) then {  
  // do nothing here  
} else {  
  counter := counter + 1  
}
```

However, writing it that way is considered very bad style. We should instead rewrite it as:

```
if (!box.contains (point)) then {  
  counter := counter + 1  
}
```

Here we have used `!` to negate the value of the condition so that the statement we wish to execute now belongs in the if–part. Notice that this version of the statement is much shorter and simpler. In general, we will be looking for short, clear ways of writing code in order to make it more understandable to a reader.

Don’t be afraid to use boolean expressions in assignments. Recall the class `WhatADrag` contained in Figure 4.6. In that program, `box` was a filled rectangle, `point` was a `Point`, and `boxGrabbed` was a Boolean variable.

Many beginning programmers are more comfortable writing

```
if (box.contains (point)) then {  
  boxGrabbed := true  
} else {  
  boxGrabbed := false  
}
```

rather than the code we actually wrote in that program

```
boxGrabbed := box.contains (point)
```

However, you should become comfortable enough with boolean variables that this kind of assignment makes sense to you. The assignment is simpler and easier to understand than the if–then–else statement for an experienced Grace programmer.

Don’t use true or false in conditions. Again, let `boxGrabbed` be a Boolean variable. Suppose you want a program to do something exactly when `boxGrabbed` is true. Look at the following code excerpt:

```
if (boxGrabbed == true) then {  
  ...  
}
```

Can you determine why this code is considered bad style? The code can, and should, be written more simply as:

```
if (boxGrabbed) then {  
    ...  
}
```

Both code segments do exactly the same thing. However the first is unnecessarily verbose. It is like the difference between saying “if it is true that it is raining then I will stay home” and “if it is raining then I will stay home.” The second version is clearly preferable as it is less complex and easier to understand.

```
Similarly, rather than writing  
while {done == false} do {  
    ...  
}
```

or

```
while {done != true} do {  
    ...  
}
```

we prefer

```
while {!done} do {  
    ...  
}
```

A good general rule is to avoid using either **true** or **false** in a condition, as their presence represents verbose and redundant code. The literals **true** and **false** are most commonly used in assignment statements to initialize boolean variables, and never in conditional expressions.

Exercise 7.5.1 *Simplify the following statements so they use good style. Assume hungry is a Boolean and eat and sleep are methods.*

a.

```
if (hungry == true) then {  
    eat  
}
```

b.

```
if (hungry == true) then {  
  
} else {  
    sleep  
}
```

7.6 DeMorgan’s Laws and Complex Boolean Expressions*

As you write more complex programs, the conditions in your if and while statements will likely become fairly complicated. In order to ensure that the conditions in your statements mean what you want, it is worth taking a small excursion into propositional logic to study DeMorgan’s laws. DeMorgan’s laws are rules for understanding complicated boolean expressions by identifying logically equivalent expressions. Of particular interest will be DeMorgan’s laws involving negations, **&&**, and **||**.

Recall that if A and B are boolean expressions, then the expression $A \ \&\& \ B$ will be true exactly when both A and B are true. We can illustrate this with a diagram called a truth table:

A	B	$A \ \&\& \ B$
true	true	true
true	false	false
false	true	false
false	false	false

A truth table is similar to an addition or multiplication table in that it indicates the results of an operation for given values of the operands.

The first column of the table shows possible values of A, the second column shows possible values of B, while the third column shows the corresponding value of $A \ \&\& \ B$ for those values of A and B. Thus the first row shows that if A and B are true then $A \ \&\& \ B$ is also true. The second row shows that if A is true and B is false then $A \ \&\& \ B$ is false.

We can make a similar truth table for the boolean operator $\ ||$:

A	B	$A \ \ B$
true	true	true
true	false	true
false	true	true
false	false	false

This table shows that $A \ || \ B$ is only false if both A and B are false.

The truth table for the boolean operator $!$ is shorter:

A	$!A$
true	false
false	true

It simply shows that the negation of an expression is always the opposite of the original value.

The table for $!A$ only has two rows because $!$ is a unary operator. That is, it only takes a single operand. Because the single operand has only two possible values, **true** and **false**, only two rows are necessary. The tables for binary operations $\ \&\&$ and $\ ||$ have four rows because each operation has two operands and there are a total of four distinct combinations of boolean values for those two operands.

We can build more complicated truth tables by adding extra columns to the tables. The following table is formed by starting with the table for $A \ \&\& \ B$ and adding an extra column for the negation, $!(A \ \&\& \ B)$

A	B	$A \ \&\& \ B$	$!(A \ \&\& \ B)$
true	true	true	false
true	false	false	true
false	true	false	true
false	false	false	true

The values for the last column are obtained by negating the values in the previous column – the values of $A \ \&\& \ B$.

Next we build the table for $(!A) \ || \ (!B)$. To build this table, we will need to compute the values of $!A$ and $!B$ before we compute the *or*.

A	B	!A	!B	(!A) (!B)
true	true	false	false	false
true	false	false	true	true
false	true	true	false	true
false	false	true	true	true

The columns for $!A$ and $!B$ are obtained by negating the values in the corresponding positions in columns A and B . The values for column $(!A) \ || \ (!B)$ are obtained from the values in the columns for $!A$ and $!B$. Because an *or* only fails when both operands are false, only the first row results in a value of false for $(!A) \ || \ (!B)$.

It is interesting to observe that the column for $(!A) \ || \ (!B)$ is exactly the same as for $!(A \ \&\& \ B)$. This shows that for all possible combinations of values of A and B , those two boolean expressions have exactly the same resulting value. In other words those two boolean expressions are equivalent. This equivalence is one of DeMorgan's laws of logic.

This should make sense intuitively. Here is a simple example in English that is roughly equivalent. Suppose I say that I had wanted to learn to both sing and dance, but I failed. Then I either didn't learn to sing or I didn't learn to dance.

Below we write the truth tables for the expressions $!(A \ || \ B)$ and $(!A) \ \&\& \ (!B)$.

A	B	A B	!(A B)
true	true	true	false
true	false	true	false
false	true	true	false
false	false	false	true

A	B	!A	!B	(!A) && (!B)
true	true	false	false	false
true	false	false	true	false
false	true	true	false	false
false	false	true	true	true

Because the final columns for each of these tables are the same, we know that the boolean expressions $!(A \ || \ B)$ and $!(A \ \&\& \ B)$ are equivalent. This is another of DeMorgan's laws.

Whenever possible we will use DeMorgan's laws and the rules for simplifying the negations of comparison operators to simplify complex boolean expressions.

Exercise 7.6.1 Use the truth table for negation to show that $!!A$ is equivalent to A . That is, for each boolean value for A , the value of $!!A$ is exactly the same.

Exercise 7.6.2 Give an intuitive argument for the equivalence of $!(A \ || \ B)$ and $!(A \ \&\& \ B)$.

Exercise 7.6.3 Convince yourself of the equivalence of $!((x > 0) \&\& (x \leq 10))$ and $(x \leq 0) \parallel (x > 10)$. Draw a number line and shade in the regions represented by $(x > 0) \&\& (x \leq 10)$ and $(x \leq 0) \parallel (x > 10)$. The two regions do not overlap and contain the entire number line between them. Hence $!((x > 0) \&\& (x \leq 10))$ is equivalent to $(x \leq 0) \parallel (x > 10)$.

Exercise 7.6.4 Use DeMorgan's laws and the rules for simplifying the negations of comparison operators to simplify the following boolean expression:

$!((x == 0) \parallel (x >= 100))$

Exercise 7.6.5 Write a nested if statement that displays the snack of choice based on the table below using `print`. Use the Boolean values `sweet` and `warm` to control the statement.

	<i>sweet</i>	<i>not sweet</i>
<i>warm</i>	<i>brownie</i>	<i>hot pretzel</i>
<i>not warm</i>	<i>ice cream</i>	<i>potato chips</i>

Exercise 7.6.6 Translate the following English statements into a nested if statement that prints out the weather. “If it is winter, then in New England there is snow, but in Florida there is sun. If it is summer, there is sun in New England and Florida.” Use the boolean identifiers `winter`, `summer`, `florida` and `newEngland` to control the statement.

Exercise 7.6.7 Assume that $x = 6$, $y = 8$ and $z = -5$. What are the values of x , y and z after the following code has been executed?

a.

```

if ( x - (3 + y) <= z - 1) then {
    x := y + 2 * z
} elseif { (x - 2 * y) > (2 * z) } then {
    y := z + y
    z := z + y
}

```

```

if ((1 - z) >= (2 * x - y) ) then {
    y := y + 1
    x := x + y + z
}

```

7.7 The match Statement

The match statement is useful in situations where different actions are to be taken based on the value of an expression. Figure 7.17 shows a simple example that uses a random number generator to select a color from several options. You should assume that `colorChoice` is a variable of type `Color`.

When the statement in Figure 7.17 is executed, a random number between 1 and 6 is obtained. If its value is one of the values listed after the `case` keyword, then the statements

```

match (rand.integerIn (1) to (6))
  case {1 -> colorChoice := colorGen.red }
  case {2 -> colorChoice := colorGen.white }
  case {3 -> colorChoice := colorGen.yellow }
  case {4 -> colorChoice := colorGen.green }
  case {5 -> colorChoice := colorGen.blue }
  case {6 -> colorChoice := colorGen.cyan}

```

Figure 7.17: Using a match statement to randomly select a color.

after the corresponding case are executed. When the case clause is done, execution jumps to immediately after the last case clause. In our example, since the random number generator will only generate values in the range 1 to 6, a color will always be selected for `colorChoice`.

The general syntax of a match statement is as follows:

```

match (anExpression)
  case {literal1 -> statements1 }
  case {literal2 -> statements2 }

  case {literalk -> statementsk }

```

Notice that there are no curly brackets surrounding the collection of cases, but there are curly brackets surrounding the code following each occurrence of the keyword `case`.

The value of `anExpression` is used to determine which case clause is executed. Each case statement starts with a literal value, such as `73` or `"Paula"` or `true`. This must be a primitive value and may not be a general expression or even an identifier. The `->` is intended to remind you of an arrow. If the value of `anExpression` corresponds to one of the literals given in a case statement, then all the statements after the `->` are executed for the first case that matches, and then control is transferred to the next statement after the `match` statement. If the value of `anExpression` does not match any of the literals then none of the statements in the case statements are executed and control passes to the following statement.

A match statement is often used in a context where an `if-then-elseif` compound statement could also be used. If the decision on which block of statements to execute is based on equality comparisons with literals, then a match statement may be the best choice, both from the point of view of readability and efficiency.

If you wish to do the same thing with several distinct values of the match expression, you may use the symbol `|` to separate the choices. For example, we included the following code in our initial discussion of the craps example in Section 4.5

```

if ((roll == 7) || (roll == 11)) then { // 7 or 11 wins on first throw
  status.contents := "You won!"
} elseif {(roll == 2) || (roll == 3) || (roll == 12)} then { // 2, 3, or 12 lose on 1st throw
  status.contents := "You lose!"
} else { // Set the roll as the new point to be made and continue game
  status.contents := "Try for your point!"
  point := roll
  newGame := false // set for continuing game
}

```

We can rewrite this with a match statement as follows:

```

match (roll)

```

```

match (rand.integerIn(1)to(6))
  case {1 -> colorChoice := colorGen.red }
  case {2 -> colorChoice := colorGen.white }
  case {3 -> colorChoice := colorGen.yellow }
  case {4 -> colorChoice := colorGen.green }
  case {5 -> colorChoice := colorGen.blue }
  case {6 -> colorChoice := colorGen.cyan}
  case {_ -> print "Color choice is out of range."}

```

Figure 7.18: Generating an error message for unexpected cases.

```

case {7 | 11 ->
  status.contents := "You won!"
}
case {2 | 3 | 12 ->
  status.contents := "You lose!"
}
case {_ -> // play must continue}
  status.contents := "Try for your point!"
  point := roll
}

```

We can see if `roll` has value 7 or 11, it will update `status` to reflect a win, while if `roll` is one of 2, 3, or 12, it will indicate a loss.

The last case of this example illustrates the use of a “wildcard” symbol, written `_`, that matches everything. Thus if the value of `roll` is anything aside from 2, 3, 7, 11, or 12, then `status` will be updated to show “Try for your point!” and the value of `point` will be set to the value of `roll`. Any `match` statement may include a wildcard case. However, it should always be the last case as it will always succeed, causing any later case statements to be inaccessible.

It is often useful to include a wildcard case to catch potential errors. For example, we could modify our `match` statement for color selection as shown in Figure 7.18. With the new version, if a programmer decided to modify `colorSelector` to generate numbers in the range 0 to 5, rather than 1 to 6, without making the appropriate modifications to the `match`, they would receive a warning whenever a problematic number was generated.

The `match` statement in Grace is much more powerful and flexible than we have described here. That also makes it easier to accidentally type in an incorrect case, but rather than the computer finding the error, it may do something unexpected (and wrong!). Be careful when you are using `match`–`case` statements never to use identifiers after the `case` keyword and you should stay out of trouble.

- b. Exercise 7.7.1** *Using the table below, write a match statement controlled by an channel number that sets the network to the correct station based on the channel. If the channel is not found, the program should display “Channel is out of range.”*

<i>channel</i>	<i>network</i>
<i>2</i>	<i>CBS</i>
<i>4</i>	<i>NBC</i>
<i>5</i>	<i>FOX</i>
<i>7</i>	<i>ABC</i>
<i>9</i>	<i>UPN</i>
<i>10</i>	<i>TBS</i>
<i>11</i>	<i>WB</i>

7.8 Summary

In this chapter we discussed control structures. We expanded on conditional statements, which were introduced in Chapter 4. We also discussed the notion of repetition. Grace provides a number of constructs for specifying repetition. In this chapter we focused on the while statement, particularly in the context of complex drawings with repetitive features.

We also introduced the match statement. The match statement can be useful in certain contexts where an if–then–elseif compound statement could also be used. If the decision on which block of statements to execute is based on equality comparisons with constants, then a match statement may be the best choice, both from the point of view of readability and efficiency. However, we also cautioned the reader about potential pitfalls of the match syntax if identifiers occur after case.

It is important to remember that, although these constructs provide immense power and flexibility to programmers, unless used intelligently they can produce confusing, unreadable code. If you follow the suggestions we have made, however, you can avoid this.

Here are some of our tips summarized for your convenience:

- Use parentheses to ensure that expressions are evaluated in the order desired.
- Don't negate the result of evaluating a comparison operator – change the operator instead! For example, replace `!(a== b)` by `a != b`.
- Use DeMorgan's laws to simplify complex boolean expressions.
- Never use the boolean literals `true` or `false` in a condition.
- Never omit the if–part of a conditional. Negate the condition so what used to be in the else–part now belongs in the if–part.

7.9 Chapter Review Problems

Exercise 7.9.1 Rewrite the body of the following method in a single line of code.

```
method onMouseClick (point: Point) {
    if (box.contains(point) == false) {
        outside := true
    } else {
        outside := false
    }
}
```

```
}  
}
```

Exercise 7.9.2 *Can the following conditions be written more concisely? If so, rewrite them. Assume that the variables `a`, `b`, `c`, and `d` are booleans and that the variables `x`, `y`, and `z` are ints.*

- a. `(!a && !b)`
- b. `((x > 7) == true)`
- c. `(a && !b) || (a && c) || (a && !d)`
- d. `!(x < y)`
- e. `(b != c)`
- f. `(x != y) && (x > z)`
- g. `!(a == b)`

7.10 Programming Problems

Exercise 7.10.1 *Write a program that draws a knit scarf by drawing tiny overlapping circles. The circles (or stitches) should be circles that are 12 pixels by 12 pixels and should overlap each other by 8 pixels. The scarf should contain 40 rows, each 12 stitches across. The upper left corner of the scarf should be at coordinates (50,10). Your scarf should resemble the scarf in Figure 7.19.*

Exercise 7.10.2 *Write a program that simulates the two-player game Rock, Paper, Scissors. Use a random number generator to determine whether each player picks rock, paper, or scissors. The program should play five rounds of the game. For each round of the game, the program should first determine if the players tied (the players tie when they pick the same item). If the players tied, the round should not count and the program should do that round over again. If the players did not tie, the program should display (using `print`) each player's choice and the winner of the round. After all five rounds have been played, the program should display the number of times each player has won. Base the winner of each round on the table below, where *W* = win, *L* = loss, *T* = tie.*

	<i>Rock</i>	<i>Paper</i>	<i>Scissors</i>
<i>Rock</i>	<i>T</i>	<i>L</i>	<i>W</i>
<i>Paper</i>	<i>W</i>	<i>T</i>	<i>L</i>
<i>Scissors</i>	<i>L</i>	<i>W</i>	<i>T</i>

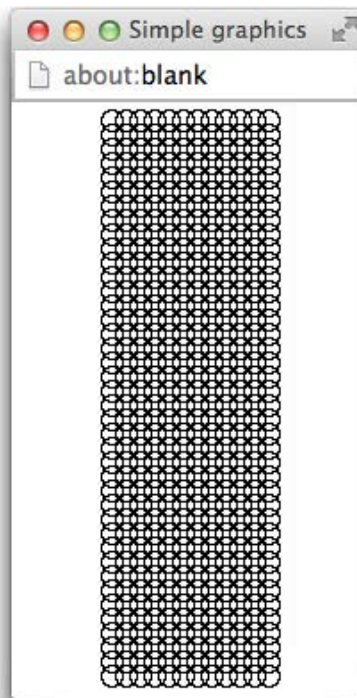


Figure 7.19: Example of scarf for Exercise 7.10.1

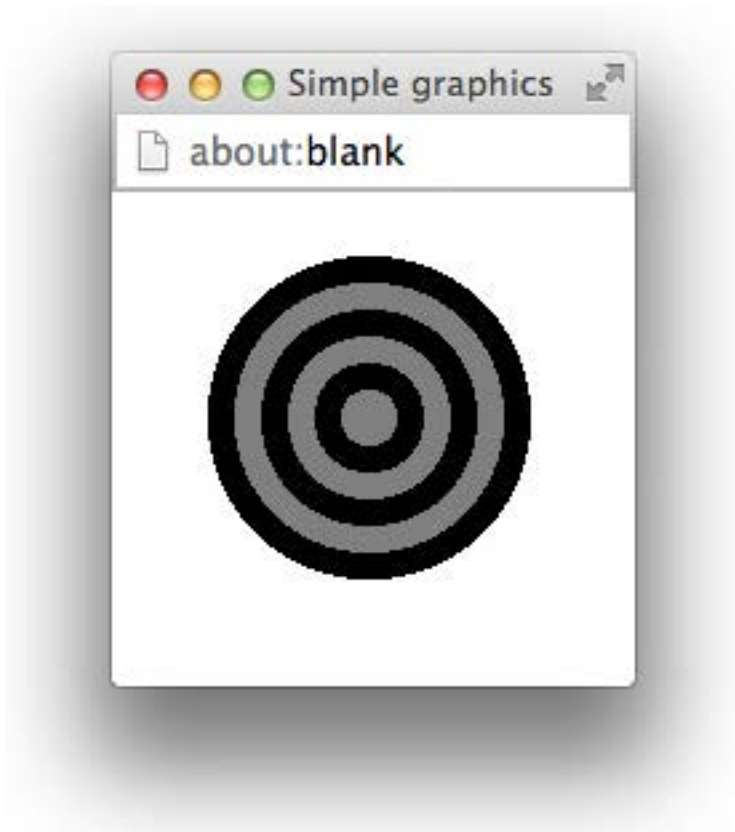


Figure 7.20: Bullseye for Exercise 7.10.3

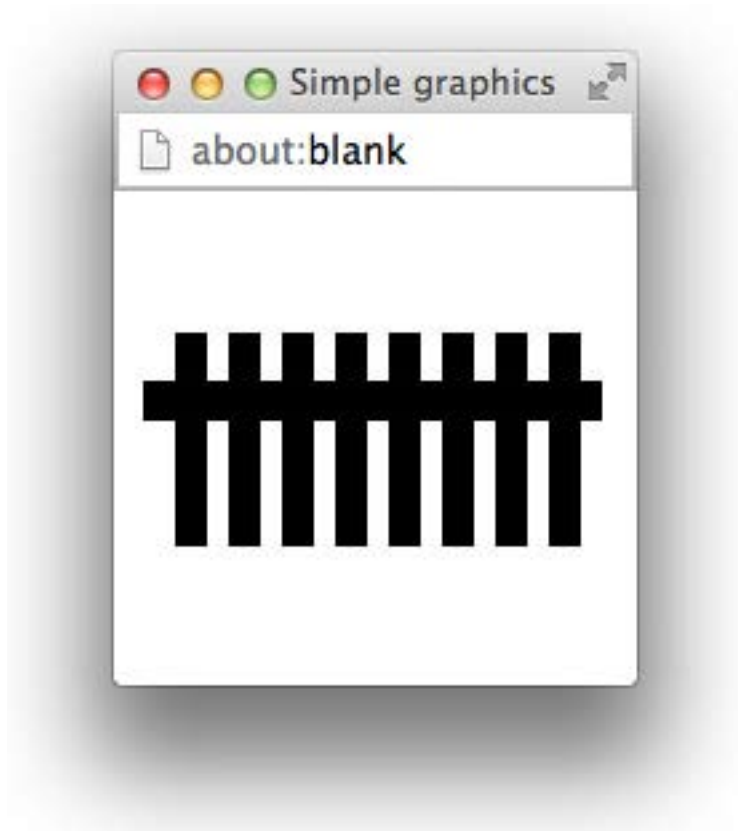


Figure 7.21: Fence for Exercise 7.10.4

Exercise 7.10.3 *Using your program called `AlmostBullseye` from Exercise 7.2.1, write a program called `Bullseye` that draws a real bullseye like the one in Figure 7.20. The program should draw six concentric circles, with the color alternating between gray and black. The biggest circle should be 120 pixels in diameter, with each subsequent circle decreasing 20 pixels in diameter.*

Exercise 7.10.4 *Write a program called `PicketFence` that draws a picket fence on the canvas. The fence should look like the picture in Figure 7.21.*

Exercise 7.10.5 *Write a program called `Railroad` that draws railroad tracks. The railroad tracks should lead to the horizon and get smaller as they get farther away from the bottom of the canvas as in Figure 7.22.*

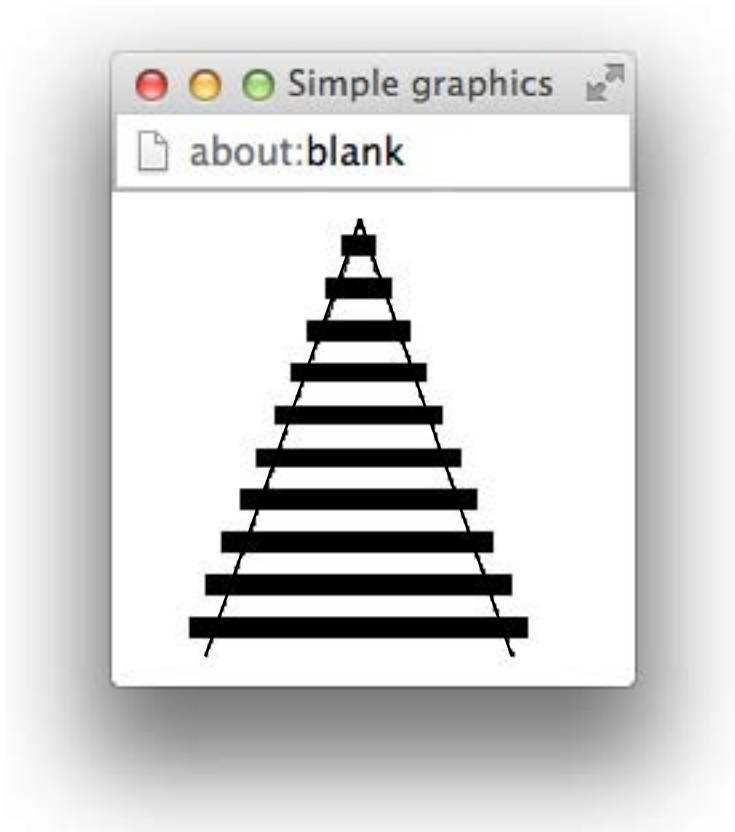


Figure 7.22: Railroad tracks for Exercise 7.10.5

Chapter 8

Declarations and Scope

In this chapter we discuss Grace's rules that govern the visibility of variables and methods declared in classes. We begin by giving a description of what the access modifiers `public` and `confidential` mean. We next review the notions of instance variables, parameters, and local variables, reviewing when each of these is the most appropriate to use.

Another important aspect of visibility has to do with the notion of *scope*. As we shall see, the notion of scope determines where an identifier is visible in a program. An important difference between the different kinds of variables and parameters has to do with scope.

The point of this chapter is not to introduce new programming language features, but instead to help you better understand the features that have already been introduced. This better understanding is the key to becoming a good programmer.

8.1 Access Control: `public` versus `confidential`

In Grace, features may be declared to have different levels of accessibility. For example, a feature in an object that is `public` will be accessible anywhere that object is accessible. A feature that is `confidential` will only be accessible inside the object where it is defined or in any other object or class that From from it.

In Grace, methods (and type definitions) have `public` accessibility by default, which explains why you could always make a method request to an object as long as you had access to the object. On the other hand, definitions, and variables have `confidential` accessibility by default, so we have only been able to access them inside the objects or classes they were defined in.

While the default access for methods is `public`, there are situations where declaring a method to be `confidential` is useful. Let us now reexamine the `onMouseClicked` method of the class `Chase` given in Figure 6.8 to see how a `confidential` method may be of help. We repeat the method in Figure 8.1 for convenience. Recall that this class was used in a game in which the user attempted to click on the funny face within a certain time limit.

Notice that there are two lines in the `onMouseClicked` method that are repeated in three different cases of the conditional statement. They are

```
smileyFace.moveTo (newPoint)
stopWatch.reset
```

```

object {
  From graphicApplicationSize (400 @ 400)
  ...

  // Determine if user won and move smiley face if necessary
  method onMousePress (pt: Point) -> Done {
    def newPoint: Point = (rand.integerIn (0) to (canvas.width)) @
      (rand.integerIn (0) to (canvas.height))
    if (!playing) then {
      playing := true
      infoText.contents := "Click quickly on the Smiley to win!"
      smileyFace.moveTo (newPoint)
      stopWatch.reset
    }
    elseif {!smileyFace.contains (pt)} then {
      infoText.contents := "You missed!"
      smileyFace.moveTo (newPoint)
      stopWatch.reset
    }
    elseif {stopWatch.elapsedSeconds <= timeToClick} then {
      infoText.contents := "You got the Smiley in {stopWatch.elapsedSeconds} seconds. Click to start over."
      playing := false
    }
    else {
      infoText.contents := "Too slow!"
      smileyFace.moveTo (newPoint)
      stopWatch.reset
    }
  }
}

```

Figure 8.1: Chase class: onMouseClick method.

```

// Move smileyFace to new randomly selected coordinates and reset timer}
method resetGame -> Done is confidential {
  def newPoint: Point = (rand.integerIn (0) to (canvas.width)) @
                        (rand.integerIn (0) to (canvas.height))
  smileyFace.moveTo(newPoint)
  stopWatch.reset
}

// Determine if user won and move smiley face if necessary
method onMousePress(pt: Point) -> Done {
  if (!playing) then { // set up to start playing
    playing := true
    infoText.contents := "Click quickly on the Smiley to win!"
    resetGame
  } elseif {!smileyFace.contains(pt)} then { // playing, but missed!
    infoText.contents := "You missed!"
    resetGame
  } elseif {stopWatch.elapsedSeconds <= timeToClick} then { // clicked in time!
    infoText.contents := "You got the Smiley in {stopWatch.elapsedSeconds} seconds. Click to start over."
    playing := false
  } else { // too slow with the click
    infoText.contents := "Too slow!"
    resetGame
  }
}

```

Figure 8.2: Chase class: revised onMouseClick method using confidential method resetGame.

If they had been repeated in all four cases in the conditional, we could have just moved them to occur immediately after the conditional. However, this code need not be executed in the case that the user clicked on the funny face in time, so moving it to the end is not a good strategy.

When we have repeated code like this it is worth trying to figure out what these statements represent, and building a method that represents that abstraction. This particular repeated set of statements resets the game so that it can be played again. Because it is conceptually a single task, we can name this block of code and make it into a method, `resetGame`. Grace code including this method and the revised code for `onMouseClick` that uses the new method are given in Figure 8.2. We have also moved the local definition of `newPoint` to the new method as it is only used in that method. Using this new method makes `onMouseClick` somewhat easier to understand, because the reader need not be concerned about the details of resetting the game.

Because this method is designed solely to help in writing other methods (in this case `onMousePress`), it should not be public. We can signal this by writing “`is confidential`” after the return type. (If the return type is not provided by the programmer, then the new phrase can be added after the closing parentheses for the parameters.) Because methods like this are designed to help in writing other methods, they are often referred to as *helper* methods.

Because we have declared `resetGame` to be a confidential method, no other object may invoke it. Confidential methods, like confidential instance variables, are only available inside the

```

class r (r') g (g') b (b') -> Color{
  // error checking code is omitted
  def red: Number is public = r'
  def green: Number is public = g'
  def blue: Number is public = b'

  method asString -> String {
    "rgb({red}, {green}, {blue})"
  }
}

```

Figure 8.3: The class `color` from the `objectdraw` library

classes in which they are declared. By way of contrast, methods like `onMousePress` must be public (which they are by default) so that the system can request them when the user presses the mouse.

What about definitions and variables? Are there ever circumstances where a programmer might want those to be accessible outside of the object? Yes, there are, but this should be used only sparingly. A good example are the identifiers `red` and `green` and `blue` defined in the `objectdraw` class `r()g()b()` generating elements of type `Color`. This class is actually defined in the `colorGen` object, so the class is accessed by writing `colorGen.r()g()b()`.

To allow access to these values, we declared them to be `public`. We display the class definition in Figure 8.3.

When we declare the definitions of `red`, `green`, and `blue` to be `public`, it results in the system creating methods with the same names as the variables. The effect is as though we rewrote the class as in Figure 8.4. In that version, the hidden definitions are renamed as `red'`, `green'` and `blue'`, while there are three new methods, named `red`, `green`, and `blue`. In fact if you look at the definition of type `Color`.

```

type Color = {
  red -> Number
  green -> Number
  blue -> Number

  asString -> String
}

```

you see that there are methods named `red`, `green`, and `blue` that return numbers. There is no indication in the type that these methods arose from public definitions.

Having public definitions really represent implicitly defined methods is useful for two reasons. First, it means that all computation really takes place via method requests and that there is no syntactic difference between method requests and definition and variable accesses. Second, as we will see later when we discuss inheritance, it means that we will be able to modify what happens during these accesses when we inherit from a class.

What about public variables? It is possible to have public variables, though it is relatively rare to allow complete access to instance variables outside of the class they are defined in. What is more common is to allow only read access to these variables. Let's look at a couple of examples.

```

// class generating immutable Points on screen
class r (r!) g (g!) b (b!) -> Color{
  // error checking code is omitted
  def red! is public = r!
  method red -> Number {red!}

  def green! is public = g!
  method green -> Number {green!}

  def blue is public = b!
  method red -> Number {blue!}

  method asString {
    "rgb({red}, {green}, {blue})"
  }
}

```

Figure 8.4: The class in object `color` as it might have been written in the `objectdraw` library

Our `Point` type represents immutable points, but we could design a different type for mutable points. Let's call it `MutablePoint`. Its interface might look like the following:

```

type MutablePoint = {
  // coordinates of mutable point
  x -> Number
  y -> Number

  // update coordinates of mutable point
  x:= (x!: Number) -> Done
  y:= (y!: Number) -> Done

  // return new point shifted by other's x and y components
  + (other: Point) -> Point

  // return new point shifted by backwards by other's x and y components
  - (other: Point) -> Point

  // return distance from this point to other
  distanceTo( other: Point) -> Number

  // return if other at same place as self
  == (other: Point) -> Boolean

  // return string representation of point
  asString -> String }

```

The only difference between this type and `Point` is the addition of the two methods `x:=()` and `y:=()`. These could be generated automatically by a class `mutablePoint` whose description starts out like the following:

```

class mutablePoint (x!: Number,y!: Number) -> MutablePoint {
  // coordinates of mutable point
  var x: Number is public := x!

```

```

    var y: Number is public := y'
    ...
}

```

When a variable is declared to be `public`, Grace generates *two* new methods for it: one accessor (with the same name as the variable) and a mutator method whose name is the variable name with `:=` attached at the end. The second of these takes a parameter as the same type as the variable and returns `Done`. In the above example, the `public` declaration of `x` results in the generation of methods `x` and `x:=`, whose types are as given in the definition of `MutablePoint`.

You will likely not be using or creating public variables unless you are creating very primitive objects that just store a group of values. However, it is fairly common to declare variables to be `readable`, which results in the implicit creation of a method (with the same name as the variable) that returns the value of the variable.

For example, suppose that we want to be able to ask an object of type `DrawingCanvas` what its height and width are, but don't want the user to be able to reset it with a command. Then we could declare variables as follows:

```

var width: Number is readable := width'
var height: Number is readable := height'

```

This will implicitly generate methods with the following types:

```

width -> Number
height -> Number

```

Of course, if `width` and `height` never change, then they could each have been defined as `public defs`, but there may be other actions (dragging the right lower corner of the window) that might internally change the value. *To be clear, our library does not recognize these changes!*

At this point, you might remember that you have seen lots of other methods in the library that look like they could have been generated by public variables, as they consist of pairs of accessors and mutators. For example, there are methods `color` and `color:=()` in type `Graphic`. However, their implementation is a bit more subtle:

```

// the color of this object
var theColor: Color is readable := colorGen.black

method color -> Color {theColor}

method color:= (newColor: Color) -> Done {
    color := newColor
    setStateChanged
}

```

Thus the methods to return and set the color are defined explicitly. This is particularly important for the mutator method as the system must be notified (via the request `setStateChanged`) that it must redraw the window every time the color of an object is changed. The handy thing is that you, as a user of these object, cannot tell whether these are written as instance variables or as methods. Thus, you can write the code to use these the same way, no matter which way the implementer has designed the class.

Other methods you have seen that appear to be generated automatically include the following from various classes that generated graphic objects: `width`, `height`, `start`, `end`, `contents`, and `fontSize`.

Not surprisingly, we can also declare variables to be `writable`. These are less common, but end up generating methods with the variable name followed by `:=`, such as `log:=()`

We end this section with an exhortation to try to avoid public definitions and instance variables as much as possible. If you find yourself needing to make too many of these public you are probably not constructing the appropriate high-level methods for your objects. We will discuss this in more detail in Chapter 20.

Exercise 8.1.1 *Why is `resetGame` in the revised `Chase` class declared to be confidential rather than public?*

8.2 Using Instance Variables, Parameters, and Local Variables

So far in this text we have introduced three different kinds of identifiers, i.e., names, that represent values. They are instance variables, local variables, and formal parameters. In the rest of this section we compare and contrast their use, providing advice to help you decide when each is the most appropriate.

Instance variables and definitions are visible inside all of the code that follows them inside the object or class expression in which they are declared. By default they are confidential, and hence access to them is limited to other code in the same object or class. Instance variables and definitions may be declared to have wider accessibility by declaring them to be `public`, `readable`, or `writable`. Local variables and parameters are defined and visible only in the classes and methods in which they are declared. They may *not* have an access annotation of `public` or `confidential`, as a compile-time error would result. Their use is actually more restricted than that of confidential instance variables.

Instance variables are typically initialized when the object is created, as was the case in class `funnyFace` in Figure 6.5. Their initial values may depend on the values of the formal parameters of the class. If an instance variable `iVar` has not been initialized before its first use, the program will generate an error something like the following:

```
RuntimeError: Requested method on uninitialised value iVar
```

Thus if a variable is not initialized in its declaration, the programmer must ensure that it is given a value before its first use in the program.

Instance variables retain their values between method invocations and can be updated inside of methods of the class. Thus they are used to hold information that must be retained to be used in later invocations of methods.

Formal parameters become associated with the values used as arguments during a method request. Parameters are typically used to transmit information needed into a method body or class. A formal parameter is not available outside of the method body and normally goes out of existence when the execution of the method or class construction in which it is contained finishes. If that information is needed elsewhere after the method terminates, then it should normally be associated with an instance variable by a statement in the method body.

Sending a message to an object referred to by a formal parameter may result in changes to the state of the corresponding argument. Look at the following example:

```
method makeYellow (rect: Graphic) -> Done {
  rect.color := colorGen.yellow
}
```

The result of executing the method invocation `someObj.makeYellow(myRect)` will be to change the color of `myRect` to yellow. Notice that we have not replaced the argument `myRect` with a different object; we have instructed the filled rectangle associated with `myRect` to change its state by making a method request for `color :=`.

Grace treats formal parameter declarations like definitions in that it is not possible to assign to them. For example, suppose we (foolishly) write the following method:

```
method tryToMakeYellowRect (rect: Graphic) -> Done {
  rect := filledRectAt (someLocn) size (20 @ 40) on (canvas) // ERROR!!
  rect.color := colorGen.yellow
}
```

The assignment to formal parameter `rect` will trigger an error message informing the programmer that no assignments are allowed to `rect` because it is the name of a formal parameter.

Local variables may be initialized in their declarations or by assignment statements inside the method or constructor in which they are declared. They are *never* provided with default values. Local definitions must be initialized in the actual definition. Local variables and definitions go out of existence when the method or constructor in which they are declared finishes executing. Hence a local variable must be reinitialized each time the method or class containing it is requested. Local variables are thus used to store temporary values needed only in a local computation.

Here are general guidelines to determine when each of these three kinds of identifiers should be used.

1. If a value is to be retained after the execution of one of the methods of an object, then it should be stored in an instance variable.
2. If a value must be obtained from the context where a class construction or method is requested, then it should be passed as a parameter.
3. If a value is only needed temporarily during the execution of a method, can be initialized locally, and need not be retained for later method calls, then it should be declared as a local variable.

Novices often use instance variables where it makes more sense to be using local variables. Why does it matter? A first reason that local variables are to be preferred to instance variables is because they make it easier to read and understand your classes. When a programmer is looking at a class definition, the instance variables provide information on what state information is retained between invocations of methods. If superfluous variables are declared as instance variables it makes it harder for the reader to understand the state of objects from the class.

Programmers also find it easiest to understand variables if they are declared close to where they are used. This makes it easier for a reader to find the definitions and hence read the comments explaining what the identifier is standing for. Because instance variables are

generally clustered at the head of a class, local variables are likely to be much closer to their uses.

Another reason to prefer local variables to instance variables is that objects with excess instance variables also take up more space in memory than they need to, a problem that may be important for large programs.

For each instance variable declaration in an object or class, consider why it is there and whether it really needs to be retained for use by later methods. Parameters provide the primary mechanism for passing data between different objects. If a method or constructor needs information that is not held in its own instance variables, it typically needs to obtain that data via a parameter.

If a value is obtained from elsewhere, but needs to be retained for use in later method executions, then it can be passed in as a parameter and saved to an instance variable. However, there is no reason to save parameter values as instance variables if they are not needed later. Just use the parameters directly in the body of the method into which they are passed.

Exercise 8.2.1 *Why is playing an instance variable of the class Chase rather than a local variable?*

8.3 Scope of Identifiers

In the last section we compared the uses of instance variables, local variables, and parameters. One important difference between these has to do with where each is visible in a class. It is important to understand the differences in these in general, but it is especially important when the same name is declared in different ways within the same class. In that circumstance it is very easy to get confused as to which entity is being referred to by a name. We can make sense of this by considering the notion of scope.

The *scope* of an identifier declaration is that section of code in which the feature named in the declaration can be referred to by that simple name. If all of the identifier names in a class are unique then the scope is easy to explain.

For example, the scope of a method or instance variable is the entire object or class in which it is declared, while the scope of a parameter declaration is the entire body of the method in which it is declared. However, the definition of the scope of a local variable is a bit more complex. The scope is confined to the block in which the declaration occurs.

Recall that a block of code is a series of statements enclosed by curly braces. For example, the entire body of a method is a block because it is surrounded by curly braces. Similarly blocks occur in if–else statements to indicate the statements to be executed depending on whether the condition evaluates to true or false.

The scope of a local variable declaration includes all of the statements from its point of declaration to the end of the most tightly enclosing block.

As an example, consider the code in Figure 8.5. The declaration of variable *v* has as its scope the entire program. The parameter *w* of method *m* has as its scope the entire body of *m*. That is, it can be accessed anywhere in *m*'s method body. The local definition of *x* is given inside of the block composing the entire method body of *m*. Thus its scope is the entire method body after the definition.

```

dialect "objectdraw"

var v: Number := -3

method m (w: Number) -> Done {
  def x: Number = 0
  if (w > 0) then {
    var y: Number := 17
    print "v+x+y is {v + x + y}"
  } else {
    def z: Number = -5
    print "v + x + z is {v + x + z}"
  }
  print (v+x)
}

method n (y: Number) -> Done{
  def w: Number = 12 + y + v
  def v: Number = 1.72 // <- error!!
  print("w is {w}")
}

type Test = {m -> Number}

method p (x: Number) -> Test {
  def y: Number = 12
  object {
    method m -> Number {
      x + y + v
    }
  }
}

def ob: Test = p (10)
v := v + 1
print "result of m is {ob.m}"

```

Figure 8.5: Scope example.

The local variable `y` is declared inside of the block corresponding to the `if`-part of the conditional. As a result, its scope includes only that block. It may not be referred to in the `else`-part or anywhere else in the method body. Similarly, the scope of the local definition of `z` is the `else`-part of the conditional. Neither `y` nor `z` can be accessed outside of the blocks corresponding to the `if`-part or `else`-part, respectively.

Method `n` reuses the identifier names `y` and `w` in the parameter declaration and a local definition. Because the scopes of the previous declarations of `y` and `w` are restricted to the method body of `m`, the duplication of names causes no difficulties. These are completely unrelated to the previous declarations of these identifiers. and each declaration has scope consisting of the method body of `n`.

However, things get more complicated when two declarations introducing the same identifier have overlapping scopes. For example the scope of the declaration of `v` at the beginning of the program includes the method body `n`, while the local definition of `v` inside `n` also includes the rest of the method body.

Grace considers it an error when a new declaration of a name is inside the scope of another declaration of the same name. Thus the definition of `v` inside method `n` results in a compilation error and a warning something like the following:

```
Syntax error: 'v' cannot be redeclared because it is already declared
                in scope as a def on line 3
```

Grace also does not allow declaring two local variables or definitions in the same method with the same name, nor does it allow declaring a local variable or definition with the same name as a parameter of the method.

The method `p` in the figure illustrates some more subtle points having to do with scope. Method `p` returns an object of type `Test`. The object it returns has a method `m` that returns the sum of `x` (the parameter of method `p`), `y` (a local definition in `p`), and `v` (which is declared at the beginning of the program).

In the last lines after the definition of `p`, we first compute `ob` as the result of evaluating `p(10)`. That is, `ob` is an object with the method `m` as described above, where the value of the parameter `x` is 10. The next line updates the value of `v` from -3 to -2. Then, when we make the method request `ob.m`, it will use the current values associated with the identifiers to return $10 + 12 + (-2) = 20$. The key idea here is that the identifiers referred to in the body of method `m` are determined when we evaluate `p(10)`. That is, the `x` refers to the formal parameter 10, the `y` refers to the definition inside `p`, and the `v` refers to the variable at the beginning of the program. However, we do not fetch the value of the variable `v` until we actually evaluate the call of method `m`. By the time we do that in the example, `v` is associated with the value -2 rather than -3, so that is used in the computation. Of course the values of definitions and parameters cannot be changed once they are evaluated.

So far we have confined our attention to the scope of a declaration from within the object or class containing the declaration. However, as you know, features declared to be public are also accessible outside of the object. They are accessible anywhere the object is accessible.

A common error made by beginning programmers is to insert a new local variable declaration when an instance variable is needed. For example, suppose we write:

```
object {
```

```

var value: Number

method initialValue (initialValue: Number) -> Done {
  var value: Number := initialValue
}
...
}

```

Here the programmer seems to be worried that he or she won't be able to access the instance variable unless it is declared within the method. Fortunately, the programmer will get an error message informing them that a new declaration of `value` in the scope of the first declaration is illegal. All the programmer will have to do then is to replace the method by

```

method initialValue (initialValue: Number) -> Done {
  value := initialValue;
}

```

Then `value` is updated as desired without accidentally declaring a new and different variable.

8.4 Summary

In this section we took a closer look at declarations and scope. We began by discussing access control for features of Grace by using the keywords `public` and `confidential`. The `confidential` features will only be available inside the object or class definition in which they are defined or any class or object that From from it, while `public` features are available outside of the class. By default, instance variables and definitions are `confidential`, but may be annotated to be `public`. If an instance definition of identifier `id` is annotated to be `public`, then a parameterless method with the same name `id` is implicitly created that returns the value of `id`. If an instance variable `vble` is annotated as `public` then both accessor and mutator methods (named `vble` and `vble:=()`) are implicitly created.

Methods in a class or object definition are `public` by default, but may be annotated to be `confidential`. Finer access control is possible over variables by declaring them to be `readable` or `writable`, implicitly creating either the accessor or setter method to be `public`.

We also compared the use of instance variables, local variables, and parameters. Instance variables are to be used when information must be retained between calls to constructors and methods. Parameters provide values to be used within the bodies of methods. They are similar to definitions in that they may not be assigned to. Formal parameters become associated with the values of corresponding actual parameters during each method invocation. These associations are not retained after the execution of the method body. Local variables are used for values that need only be saved temporarily during the execution of a constructor or method.

8.5 Chapter Review Problems

Exercise 8.5.1

What do each of the following Grace terms mean?

- a. `confidential`

b. public

c. local variable

Exercise 8.5.2 *A beginning programmer has written the program below without fully understanding declarations and scope. What problems does the following code have?*

```
class c.with (number: Number) {  
  var number: Number := number  
  
  method m -> Done{  
    var number: Number  
    if (18 > number) { ... }  
  }  
  ...  
}
```

Exercise 8.5.3 *Why should a programmer try to use local definitions and variables rather than simply declaring all the variables in a class as instance variables?*

Chapter 9

Animating Objects

In Chapter 7 you learned about Grace control constructs to perform repetition. So far the examples that you have seen have largely involved repeated patterns in drawings – blades of grass, bricks, and grid lines. We now consider another type of application in which repetition plays a major role: animation.

In this chapter, we discuss how to create animations using objects of type `Animator`. Using animators we can execute operations repeatedly with scheduled pauses between actions.

9.1 Animation

If you ever owned a “flip book” as a child, you certainly know how animation works. A series of pictures is drawn, with each picture slightly different from the one before it. The illusion of movement is created by quickly flipping through the series of pictures. As an example, consider the snapshots of a ball in Figure 9.1. These are just a few pictures from a series of drawings that illustrate the action of a ball being dropped and falling to the ground. Each image shows the ball at a slightly different (i.e., lower) position than the one before it. If we were to flip through these images quickly, it would appear to us that the ball was actually falling.

You might already be imagining how to write a program to display the falling ball animation. To display the ball, all you need to do is construct a ball using `filledOval`. Once you have constructed the ball, you can create the illusion of movement down the screen by repeatedly moving the ball in the downward direction. If `ballGraphic` is the name of a variable associated with the oval, then movement can be achieved by repeatedly doing the following:

```
ballGraphic.moveBy (0, yDisplacement)
```

where `yDisplacement` is a small value. A `while` loop can be used to achieve the repeated movement. While these ideas are important components of creating moving images, you need to know one more thing before you can begin to create animations of your own.

9.2 Animator

Ultimately we will want to create fairly complex animations. For example, we might want to create a rain animation, where each raindrop is like a ball falling from the the top of the

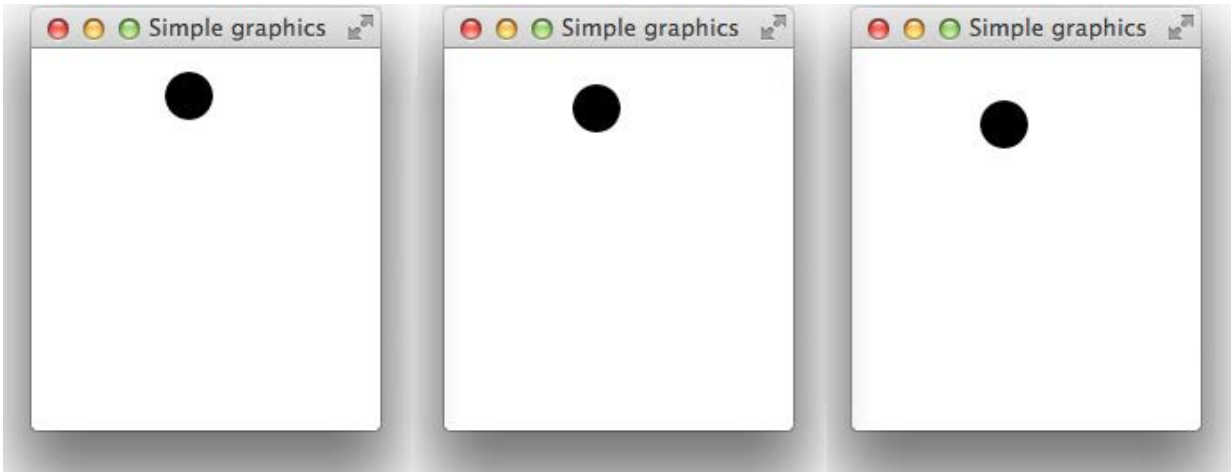


Figure 9.1: A falling ball in motion.

canvas to the bottom. How can we construct many balls that all have to fall at the same time? We will accomplish this by using a library whose job is to animate other objects.

The module that provides the methods for animation will have type `Animator`. Rather than give the complete definition of `Animator`, we will slowly introduce the most important methods that can be used to create animations. The first looks like a regular `while` loop, but with an extra argument indicating how long a pause there should be between iterations.

```
// Repeatedly execute statements while condition is true
while {condition} pausing (pauseTime) do {
  statements
}
```

The semantics or meaning of this method is very similar to that of the regular `while` loop, but it differs by introducing a pause before each execution of the body of the `while` loop. The parameter `pauseTime` is the time (in milliseconds) that execution should pause before each execution of the body of the `while` loop. It is evaluated when the `while () pausing () do ()` method is invoked, and that same value is used between all iterations.

If we now write

```
animator.while {condition} pausing (pauseTime) do { stats }
```

then `myAnimator` will evaluate `condition`. It will pause for `pauseTime` milliseconds before checking the condition. If it is true then it will execute `stats` before pausing again and repeating. Like a regular `while` loop, if `condition` is ever false then the `while` loop will terminate.

Figure 9.2 contains a simple example program using an animator. In this program, the body of the `while` loop will be executed 5 times, each time pausing 30 milliseconds before starting the loop again.

Because the `while () do () pausing ()` method is defined in the top level of the module “`animation.grace`” it can be accessed directly via `animator.while...` where `animator` is the name the module was imported as.

```

import "animation" as animator
import "sys" as sys

var x: Number := 1
animator.while {x <= 5} pausing (30) do {
  print "x: {x} at {sys.elapsedTime}"
  x := x + 1
}

print "Ended at {sys.elapsedTime}"

```

Figure 9.2: First example of using an animator with a while loop

The output of this program is as follows:

```

Ended at 0
x:1 at 0.032
x:2 at 0.062
x:3 at 0.094
x:4 at 0.125
x:5 at 0.156

```

Examining the output, you can see that there is a gap of a bit over .030 seconds (= 30 milliseconds) between successive lines of the form `x:n at n.nnn`. There are two reasons while the gap would typically be longer than the pause time for the animator. The first is that it takes time to evaluate the condition and to execute the body of the while loop. That time must be added to the pause time to calculate the time between successive invocations. Second, the system may be busy with other things it needs to do, and may not get be able to get back to execute the loop in exactly 30 seconds. Thus the system can only guarantee that the pause will be *at least* the pause time specified in the while.

Looking again at the output from this program, you can see that the print statement that follows the end of the while loop printed before the first iteration of the while loop. That illustrates a very important property of animators. The code in the while loop is executed asynchronously from the code that follows the loop. In this example the while loop paused for 30 seconds before printing out the first line. During those 30 milliseconds the computer went ahead and started executing the code following the loop. Thus the first line was printed during that first pause.

Suppose you wanted to execute some specific code immediately after the while loop terminated? As we see from the above example, you can't just place it after the request of the while method. However, there is another method to accomplish that in the animation module:

```

// Repeatedly execute block while condition is true
// when condition fails, execute endStatements.
while {condition} pausing (pauseTime) do{
  bodyStatements
} finally {
  endStatements
}

```

```

//Falling ball
dialect "objectdraw"
import "FallingBall" as fb

object {
  inherit graphicApplicationSize (400 @ 400)
  textAt (20 @ 200) with "click to drop a ball" on (canvas)

  // start ball dropping at point where mouse is pressed
  method onMousePress (pt: Point) -> Done {
    def fallingBall: fb.Startable = fb.fallingBallAt (pt) on (canvas)
    fallingBall.start
  }

  startGraphics
}

```

Figure 9.3: Program that creates a falling ball where mouse is pressed.

As suggested above, this operates like the earlier `while () pausing () do ()` method. However, after the check of condition returns false, the code in `endStatements` is executed.

A simple example of how this would work is illustrated in Figures 9.3 and 9.4.

An object created from `fallingBall` class should look like a round ball and should know how to move itself from the top of the canvas to the bottom. We will design the class so that evaluating `fallingBallAt(locn)on(canvas)` creates an object that is ready to fall down the canvas, but requires making a `start` method request for the ball to be drawn and start falling. This will allow us more control over when it starts falling.

Suppose we have such a class. We illustrate how to use it in the program in Figure 9.3. In it we display some simple instructions to the user. In addition, in `onMousePress` we construct a new falling ball, immediately followed by sending it a `start` message, which will start it falling at the location of the mouse click. Notice that because `fallingBall` was in a separate file named `FallingBall.grace`, we needed to import it and both the type `Starter` and the class `fallingBall` needed to be prefixed by the internal name of the module, `fb`.

Let's now look at the class `fallingBall`, shown in Figure 9.4, to see how we can set this up.

When the `start` method request is made we create a new filled oval at the point where the mouse was pressed and then begin animating its motion by requesting the `while()pausing () do () finally ()` method on animator. As long as the top of the ball is still on the canvas, the ball will repeatedly be moved by `yStep` pixels and then it will pause by 30 seconds. Once the top of the ball is off the canvas, the code in the `finally` block is executed, removing the ball from the canvas.

Our falling ball class implements the type `Startable`, which specifies that it has a `start` method. We will use this type for objects that support an animation that may be started.

A straightforward way to create an animated object like our falling ball is to do the following:

- define a class or object that implements `Startable`

```

//Falling ball
dialect "objectdraw"
import "animation" as animator

// type for objects with an animation that can be started
type Startable = {start -> Done}

class fallingBallAt (initialPoint: Point) on (canvas: DrawingCanvas) -> Startable {
  // object to animate the ball
  def yStep: Number = 4
  def ballSize: Number = 20
  def pauseTime: Number = 30

  // create ball at initialPoint and let it fall through bottom of canvas and then remove it
  method start -> Done {
    def ball: Graphic = filledOvalAt (initialPoint) size (ballSize @ ballSize) on (canvas)
    animator.while {ball.y < canvas.height} pausing (pauseTime) do {
      ball.moveBy (0, yStep)
    } finally {
      ball.removeFromCanvas
    }
  }
}

```

Figure 9.4: Code for a FallingBall class.

- Import the library from `animation.graceas` animator.
- from inside the start method, use the `while () pausing () do()` or `while () pausing () do() finally ()` methods of the Animator object to make the movements in your animation.

Using this Startable object is straightforward. Construct the object and then send it the start message when you want the animation to begin, as in Figure 9.3.

9.3 Using images in programs

The animation library makes it simple to write programs that produce animations. Alas, if you have Disney or Pixar in mind when we say “animations”, moving filled ovals and rectangles around the screen may not quite live up to your expectations. We can make things a bit better by using features of Grace and objectdraw that make it easy to display graphics from image files on your program’s canvas. This will allow us to animate images instead of simple geometric objects.

Support for displaying the contents of an image file on the canvas is provided by the `drawableImage` class. The description of the image comes from an image available on the web that is stored in the `png` format. To access such a file from your program you will use the `drawableImage` class. Suppose that we had a drawing of a raindrop like the one shown in Figure 9.5 stored in a file named “rainDropPic.png.”



Figure 9.5: A drawing of a raindrop

To create an image on the canvas, we need to provide the location where we want it to appear, the dimensions that we wish the image to have on the canvas, and the location of the `png` file holding the image. For example, if the raindrop image is on the web at

```
http://www.cs.pomona.edu/classes/cs051G/Images/rainDropPic.png
```

then to place a copy of our raindrop at the center of the top of the canvas we might write

```
drawableImageAt (0 @ ((canvas.width / 2) - 75) size (150 @ 150)
  url ("http://www.cs.pomona.edu/classes/cs051G/Images/rainDropPic.png") on (canvas)
```

Most of these parameters will be familiar to you. The only unusual one is labeled `url`. On the world-wide-web, `URL` is an abbreviation for “Uniform Resource Locator”. As its name suggests, it provides the address where one can find a particular object. Typically these are the usual web addresses that one can type into the top of a web browser to get to a particular location.

The biggest issue with using `drawImage` is getting the `url` parameter correct. We suggest that you first go to the web site with the image and then copy the `URL` and paste it into your code.

As an example, in Figures 9.6 and 9.7 we show how to revise the code we provided to animate the motion of a ball down the screen so that it instead draws our raindrop picture. Note how similar the code for the `FallingRainDrop` is to the code of the `FallingBall` class shown in Figure 9.4.

9.4 Including audio in programs

The `objectdraw` dialect provides support for audio as well as graphic objects. Like images, audio for your program can be stored on the world-wide web. For example, suppose we have a sound (in `wav` format) stored at

```
http://www.cs.pomona.edu/~kim/Sounds/WaterDrop.wav
```

We can make this available in our programs using the `audio` class to construct a `Grace` object that can play the song:

```
// load dripping noise from web
def dripSound: Audio = audioUrl ("http://www.cs.pomona.edu/~kim/Sounds/WaterDrop.wav")
```

Once the sound object has been created, it can be played by sending it the `play` method request. For example, here is some code using an animated while loop to play a drip sound 5 times, with successive drips separated by a pause of 8 seconds.

```
def delay: Number = 8000
var dripCount: Number := 1
```

```

// Application to show falling ball
dialect "objectdraw"
import "FallingRain" as fr

def fallingRainController: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 400)
  textAt (20 @ 200) with "click to make falling rain" on (canvas)

  // start ball dropping at point where mouse is pressed
  method onMousePress (pt: Point) -> Done {
    def fallingRain: fr.Startable = fr.fallingRainAt (pt) on (canvas)
    fallingRain.start
  }

  startGraphics
}

```

Figure 9.6: Class that creates a falling raindrop picture with each mouse click.

```

//Falling ball
dialect "objectdraw"
import "animation" as animator

// type for objects with an animation that can be started
type Startable = {start -> Done}

class fallingRainAt (initialPoint: Point) on (canvas: DrawingCanvas) -> Startable {
  // object to animate the ball
  def yStep: Number = 4
  def rainSize: Number = 50
  def pauseTime: Number = 30

  // create ball at initialPoint and let it fall through bottom of canvas and then remove it
  method start -> Done {
    def drop: Graphic = drawableImageAt (initialPoint) size (rainSize @ rainSize) url ("http://pngimg.com/
upload/pomegranate_PNG8651.png") on (canvas)

    animator.while {drop.y < canvas.height} pausing (pauseTime) do {
      drop.moveBy (0, yStep)
    } finally {
      drop.removeFromCanvas
    }
  }
}

```

Figure 9.7: Code for a FallingRainDrop class.

```

animator.while {dripCount <= 5} pausing (delay) do {
  dripSound.play
  print "drip...{dripCount}"
  dripCount := dripCount + 1
}

```

As with images, the biggest issue with using `audioUrl` is getting the parameter correct. As before, we suggest that you first go to the web site with the sound and then copy the URL and paste it into your code.

We also warn you that if you try to replay a sound before the previous one has finished, no new sound will be produced. For example, the water drop sound loaded from the above web address cannot be repeated with less than an 8 second delay (there is dead “air” in the recording after the actual sound). For each sound, you will have to experiment to see how close the sounds can be to each other.

The type of the object generated by the `audio` class is

```

type Audio = {
  source -> String           // The source URL of the audio file.
  source:= (value: String) -> Done
  play -> Done              // Play the audio.
  pause -> Done            // Pause playing the audio.
  isLooping -> Boolean     // does the audio loop back to the start?
  looping:= (value: Boolean) -> Done // determine whether the audio will loop
  isEnded -> Boolean       // whether the audio has finished
}

```

Notice that you can set the audio to loop forever. If you do this, make sure your program eventually pauses playing the audio or you will have to reload the browser to stop it playing.

9.5 Interacting with Animated Objects

As we saw above, it is convenient for classes that represent animations to include a `start` method. However, it can also be convenient for them to have other methods as well. As we will see in this section, that can help us interact with our animated objects.

There is a popular game available on a variety of mobile devices called “Fruit Ninja”. It involves falling fruits that you are to slice before they hit the ground. Our next example will not be as elaborate as that, but it will have the same flavor. It will be based on a slight elaboration of our earlier `fallingBall` class that will react to slicing motions.

```

method trySliceIt(pt: Point) -> Done {
  if (ballFalling && ball.contains(pt)) then {
    ball.color := red
  }
}

```

We can now modify our falling ball program so that every time the mouse moves, it checks to see if it has hit the falling ball. If it does, it will turn the ball red. The new program is shown in Figure 9.5. It has one additional method to handle mouse motion. We have also added a variable `started` of type `Boolean`. This variable is initialized to be false, but is reset to true when the first ball falls.


```

dialect "objectdraw"
import "FallingColorBall" as fb

object {
  inherit graphicApplicationSize (400 @ 400)
  windowTitle := "Ninja ball???"
  textAt (20 @ 200) with ("click to drop a ball") on (canvas)

  var fallingBall: fb.Slashable
  var started: Boolean := false

  // start ball dropping at point where mouse is pressed
  method onMousePress (pt: Point) -> Done {
    started := true
    fallingBall := fb.fallingColorBallAt (pt) on (canvas)
    fallingBall.start
  }

  method onMouseMove (pt: Point) -> Done {
    if (started) then {
      fallingBall.trySlicelt(pt)
    }
  }

  startGraphics
}

```

Figure 9.8: Defining a class that allows a user to drop a ball that changes color when sliced by the mouse.

The reason for having this extra variable is that we can't send a message to `fallingBall` if the variable has not been assigned a value. The variable `fallingBall` is declared as an instance variable so that it can be referred to in the `onMouseMove` method after it has been initialized in `onMousePress`. Suppose the user moves the mouse before pressing the mouse button. Then `onMouseMove` would be requested before `fallingBall` has been initialized. If `fallingBall.trySlicelt` is requested before `fallingBall` has been initialized then it will cause a system crash, printing a message informing the user that a message has been sent to an uninitialized variable.

Once the first ball has been dropped, `fallingBall` will have a value, even if that ball has been removed from the canvas.

What changes do we have to make to the falling ball class? Clearly we must add a method `trySlicelt` that determines if a point is contained in the ball. But we will also need to add an instance variable that will record whether the ball is still falling. If it is not falling, then there is no reason to check to see if the ball contains the point. The class `fallingColorBall` can be found in Figure 9.9

In the previous version of this class, the ball was defined in method `start`. However, now we must also access it in method `trySlicelt`, so it must be moved up to an instance variable or definition. However, we don't want it to appear until the start method is requested. We have two options on how to accomplish this. We can create it with a definition before the start

```

dialect "objectdraw"
import "animation" as animator

// type for objects with an animation that can be started & slashed
type Slashable = {
  start -> Done
  trySlicelt(pt: Point) -> Done
}

// falling ball that changes color when sliced
class fallingColorBallAt (initialPoint: Point) on (canvas: DrawingCanvas) -> Slashable {
  def pauseTime: Number = 30
  def yStep: Number = 4
  def ballSize: Number = 30
  var ballFalling: Boolean := false
  var ball: Graphic2D

  // start the animation dropping the ball
  method start -> Done {
    ball := filledOvalAt (initialPoint) size (ballSize @ ballSize) on (canvas)
    ballFalling := true
    animator.while {(ball.y < canvas.height) && ballFalling} pausing (pauseTime) do {
      ball.moveBy (0, yStep)
    } finally {
      ball.removeFromCanvas
    }
  }

  // If pt is in the falling ball then turn red
  method trySlicelt (pt: Point) -> Done {
    if (ballFalling && ball.contains (pt)) then {
      ball.color := red
      ballFalling := false
    }
  }
}

```

Figure 9.9: A falling ball class that responds to slices

method, but then immediately set it invisible or we can wait to create it until the `start` method is requested. In this case we decided to wait until `start` was requested. Doing this required us to define the instance variable `started`. Depending on the particular system, creating the ball and then immediately hiding it may cause a flicker on the screen as it appears and then disappears. Hence our choice of not creating it until we are ready for it to start moving.

In the method `trySliceIt` we check to make sure the ball is still falling and that it contains the point before setting it red. You can see that the variable `ballFalling` is set to be true in method `start` and then reset to false when the ball is removed from the canvas. Technically, the `contains` method and the method resetting the color would continue to work even if the ball has been removed from the screen, but we add the extra test as there is no sense performing the `contains` test and changing the color when the ball is no longer on the screen.

Exercise 9.5.1 *What happens if the user clicks twice, creating two falling balls, and then slices the first ball.*

Changing the ball to red lets us see that the ball has been hit, but maybe we want to go further and remove it from the canvas once it has been sliced. How can an action in the `trySliceIt` method affect the ball falling as determined by the `start` method. The key to this communication between methods is almost always to use instance variables. One method can change the value of the instance variable while the other one can notice the change and modify its behavior.

In this case we will add a new Boolean instance variable `ballFalling`. We will start with it set to true and then change its value to false in the `trySliceIt` method.

In the `start` method, we modify the condition of the `while` statement. As long as the ball is visible on the canvas *and is moving*, i.e., has not been stopped, it falls.

```
    animator.while {(ball.y < canvas.height) && ballFalling} pausing (pauseTime) do {
      ball.moveBy(0,yStep)
    } finally {
      ball.removeFromCanvas
    }
  }
```

The method `trySliceIt` affects the state of the ball. If this message is sent to a falling ball, the boolean variable `ballFalling` is set to false. This means that the next time the condition of the `while` is reached, the condition will evaluate to false. The loop body will not be executed and the `finally` clause will be executed to remove the ball from the canvas.

9.6 Making Animated Objects Affect Other Objects

So far we have seen how to define a class of animated objects, and we have explored ways to affect them through mutator methods. To this point, however, our animated objects have been fairly isolated, in that they don't affect other objects. In this section we explore a number of different ways in which animated objects can affect others.

9.6.1 Interacting with a Non-animated Object

Let's imagine that our falling balls don't simply disappear when they reach the bottom of the canvas. Instead, they are collected in a box that becomes ever more full with each ball

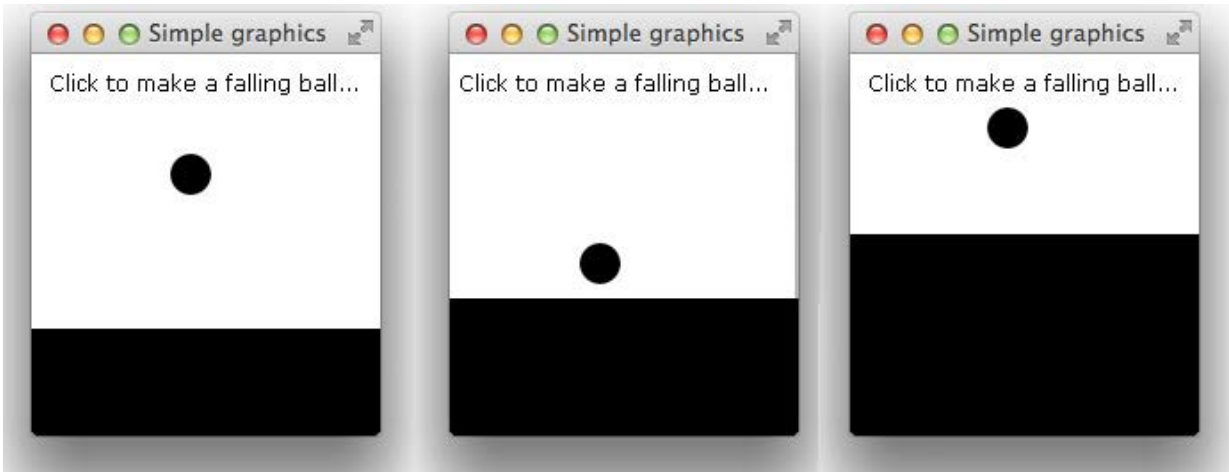


Figure 9.10: A pool filling with rain.

that falls. You might imagine that the falling balls are raindrops that collect in a pool of water. Figure 9.10 shows how such a pool might look before any drops have fallen and how it might look after many drops have fallen.

Creating the first image in Figure 9.10 is easy. The controller will simply construct a collector, i.e., a `FilledRect`, at the bottom of the canvas. We then need to make several changes to the `FallingBall` class. We'll call this new class `Droplet`, and the first modification we will make is to change the instance variable name `ball` to `droplet`.

We want every falling droplet to know about the collector. The reason for this is that each droplet will check whether it has reached the collector's surface, and then send a message to the collector telling it to raise its level a bit. Therefore, the next change we need to make is to add an instance variable, `collector`, as in Figure 9.12. As before, the constructor creates the actual droplet. We will need the collector to grow when the droplet reaches the bottom of the screen. In order for the droplet to be able to change the size of the collector, we will need to pass the collector along as a parameter. This is just another example of the general principle that if one object needs to know about some other, then it needs to be sent it as a parameter.

The `start` method of the class `droplet` is similar to the previous example with the falling ball. We remember from before that the droplet should only continue to move down the screen as long as it is visible. Therefore we set the condition in the `while` statement so that the droplet will stop moving as soon as it falls below the top of the collector.

Once the loop terminates, we need to remove the droplet as we did before, but we need to do additional work as well. If the collector is not yet full, we need to fill it a bit. In this case we have decided to increase its height by a quarter of the droplet size. To make the collector more full, we change the height of the rectangle and move it up on the canvas so that we can see the new size. These statements as well as the statement to remove the droplet from the canvas are included in the `finally` clause, which is executed after the loop terminates.

Figure 9.11 shows a program that creates a drop every time the user clicks. An instance variable, `collector`, a filled rectangle, represents the pool. (Note that the definition of `collectorRatio` is a fraction that specifies the initial size of the collector relative to the entire canvas.) In

```

dialect "objectdraw"

import "Droplet" as drop

object {
  inherit graphicApplicationSize (400 @ 400)
  textAt (20 @ 50) with ("click to make a falling raindrop ...") on (canvas)

  // fractional increase in size of collector for each drop
  def collectorRatio: Number = 0.1

  // create the collector
  def collectorPosn: Point = 0 @ (canvas.height - canvas.height * collectorRatio)
  def collector: Graphic2D = filledRectAt (collectorPosn)
    size (canvas.width @ (canvas.height * collectorRatio)) on (canvas)

  // start drop falling at point where mouse is pressed
  method onMousePress (pt: Point) -> Done {
    def dropper: drop.Startable =
      drop.dropletAt (pt) on (canvas) with (collector)
    dropper.start
  }

  startGraphics
}

```

Figure 9.11: Adding a collector to collect falling raindrops.

```

dialect "objectdraw"
import "animation" as animator

// type for objects with an animation that can be started
type Startable = {start -> Done}

class dropletAt (startPt: Point) on (canvas: DrawingCanvas) with (collector:Graphic) -> Startable {
  def pauseTime: Number = 30
  def yStep: Number = 4 // how far droplet falls in each step
  def ballSize: Number = 20 // diameter of droplet

  method start -> Done {
    def aDroplet: Graphic = filledOvalAt (startPt) size (ballSize @ ballSize) on (canvas)
    animator.while {aDroplet.y < canvas.height} pausing (pauseTime) do {
      aDroplet.moveBy (0, yStep)
    } finally {
      aDroplet.removeFromCanvas
      if (collector.y > 0) then {
        collector.moveBy (0, -ballSize/4)
        collector.height := collector.height + ballSize / 4
      }
    }
  }
}

```

Figure 9.12: Making a droplet fill a collector.

order to allow our falling droplets to affect the pool, we need to pass `collector` as a parameter to the `Droplet` constructor. In this way, each falling drop will know about the pool, so that it can effectively change it.

In general, if we want an object to have the ability to affect other objects, we can do so by making sure the first object knows about the others. One way to do so is to pass that information in as a parameter to the class definition as we did above. As long as the object remembers that information for later, it can communicate with the other object as much as it needs to.

In the example above, you might have wondered why the droplets were responsible for modifying the collector. Clearly there has to be some communication between the droplets and the collector. The way we have shown you is just one option. Another possibility is to have the main program manage the collector. After completing a fall, a droplet could send a message to the controller, telling it to increase the collector size. Note that it would not be a good idea for the collector to know about the droplets, as it would then be responsible for keeping track of potentially many at once.

9.6.2 Animated Objects that Construct Other Animated Objects

We just saw that animated objects can affect non-animated objects. They can also interact with other animated objects. In particular, they can create animated objects.

We have begun to think about our falling balls as raindrops that can fall into a collector. Naturally, raindrops should fall from a cloud. Rather than having each drop fall with the click of the mouse, let's construct a cloud program that drops a large group of drops every time the user presses the mouse.

The rain cloud program is shown in Figure 9.13. We will treat objects of the class `Droplet`, as in Figure 9.12, as our raindrops. Because we don't want them all starting at the same time, we create another loop by using the `while () pausing () do ()` method to generate a new drop every 400 milliseconds.

An important question is where should we start each raindrop. In order to simulate real rain, we should probably have each drop fall from a different `Point` in the sky (in our case, the top of the canvas). We can do this by using the method `rand.integerln (0) to (canvas.width)` to provide x coordinates from the left side of the window to the right. The program uses those x coordinates to start the raindrops at random positions across the top of the canvas.

In order to limit the amount of rain to fall, we define a constant `maxDrops` that gives us the maximum number of droplets to be dropped. In order to count droplets, we declare a local variable, `dropCount`, in the `run` method that will keep track of the number dropped so far. This is initialized to 1. The condition of the `while` statement specifies that the body of the loop should execute as long as the count of drops is less than or equal to the maximum allowed. The statements in the body construct a new `Droplet`, passing a point, the canvas, and collector as parameters. After a raindrop is created, started, and `dropCount` incremented, the program pauses briefly before executing the commands in the `while` loop again, creating a new drop.

Note that we did not have to limit the number of raindrops. We could have had the program generate raindrops indefinitely by replacing the `while` loop as follows:

```
animator.while{true} pausing (delayTime) do {  
  def dropPosn: Point = (rand.integerln (0) to (canvas.width)) @ 0
```

```

dialect "objectdraw"
import "random" as rand
import "animation" as animator
import "Droplet" as drop

object {
  inherit graphicApplicationSize (400 @ 400)
  textAt (20 @ 50) with ("click to start a rainstorm ...") on (canvas)

  // fractional increase in size of collector for each drop
  def collectorRatio: Number = 0.1

  def maxDrops: Number = 40 // number of drops to be generated
  def delayTime: Number = 400 // delay between generation of successive drops

  // create the collector
  def collectorPosn: Point = 0 @ (canvas.height - canvas.height * collectorRatio)
  def collector: Graphic2D = filledRectAt (collectorPosn)
    size (canvas.width @ (canvas.height * collectorRatio)) on (canvas)

  // start drop falling at point where mouse is pressed
  method onMousePress (pt: Point) -> Done {
    var dropCount: Number := 1
    animator.while {dropCount <= maxDrops} pausing (delayTime) do {
      def dropPosn: Point = (rand.integerIn (0) to (canvas.width)) @ 0
      def dropper: drop.Startable =
        drop.dropletAt (dropPosn) on (canvas) with (collector)
      dropper.start
      dropCount := dropCount + 1
    }
  }
}

startGraphics
}

```

Figure 9.13: A rain cloud class.


```

def dropper: drop.Startable =
  drop.dropletAt (dropPosn) on (canvas) with (collector)
  dropper.start
}

```

Here, as long as the condition evaluates to true, which it does for each iteration, raindrops will be generated.

Exercise 9.6.1

- a. *What would happen if we omitted the statement*
`dropCount := dropCount + 1`
from the while loop of program RainCloud shown in Figure 9.13?
- b. *What if we wrote the same while loop, but without the animator?*

9.7 Animated Objects without Loops

So far in this chapter, each time we wanted to animate an object (or even execute general code with a delay in between, we created an Animator object and requested a while method to control the repeated behavior. At this point you might be wondering whether you *must* use a while loop. The answer is no. There is no requirement that there be a while loop when you want to animate some behavior.

The while method in type Animator will solve almost all of our problems, but there may be times where we can use something more primitive – a timer. There is a built-in timer module in Grace that can be used to delay the execution of code. The syntax is:

```
timer.after (pauseTime) do {block}
```

When this is executed, Grace will set a timer for `pauseTime`. After that time has elapsed it will execute the code in `block`. It is important to note that Grace will continue executing other code while waiting for the timer to file. Thus if we execute:

```
timer.after (1000) do {print "after one second"}
print "after timer"
```

then “after timer” will be printed *before* “after one second” Thus if you wish to delay a series of actions so they follow each other with delays then you will have to nest the `after` methods.

As an example, consider the way that movie credits scroll on the screen at the end of a film. Typically they rise from the bottom of the screen, float to the top, and then disappear. We can think of each line of the credits as being controlled by an Animator object. It’s behavior is similar to our falling ball or droplet, except that it moves up, rather than down. Let’s examine it briefly before we get to the more interesting main program to create the credits.

The `credit` constructor takes three parameters: the String containing the information to be displayed, the font size, and the canvas. When the `credit` constructor is executed, it creates an Animator object whose while loop pauses every 30 millisecond, and sets how far the credit rises on the screen each time through the loop.

When the `start` method is invoked, it constructs a Text object with the appropriate String and sets its font size to the value passed in. We then start its motion up the canvas, similarly to the way we moved a falling ball down the canvas. We move the line until its top is 15

```

dialect "objectdraw"
import "animation" as animator

// type for objects with an animation that can be started
type Startable = {start -> Done}

class credit.with (scrollingLine: String) size (fontSize: Number) on (canvas:DrawingCanvas) -> Startable {
  def pauseTime: Number = 30
  def yStep: Number = 1 // how far credit rises in each step

  method start -> Done {
    def roleAndName: Text = textAt (10 @ canvas.height) with (scrollingLine) on (canvas)
    roleAndName.fontSize := fontSize
    animator.while {roleAndName.y > -15} pausing (pauseTime) do {
      roleAndName.moveBy (0, -yStep)
    } finally {
      roleAndName.removeFromCanvas
    }
  }
}

```

Figure 9.14: A class of individual movie credit lines.

pixels off the screen. We anticipate that will be sufficient to make sure all of it is off the canvas.

Once it has moved completely off the canvas, we remove it. This class should be comfortably familiar to you now given its similarity to moving balls.

Our main program is in charge of rolling the credits when the user presses the mouse. It could generate a line, wait a bit, then generate another line, and so on. But there is a big difference between raindrops and film credits. Each line in the credits is different from any other. The textual information it conveys is unique. With the tools you have learned so far, the best way to generate a series of movie credits would be with the `CreditScroller` program shown in Figure 9.15, where the `Credit` class is as shown in Figure 9.14.

This program imports two modules. The first is “timer” which provides access to the `after` method of the timer, while the second imports the module containing the class `credit` for constructing credits. We do not import `animation` because we will be using the more primitive timer methods.

When the user presses on the mouse, the program first removes the instructions and then creates credits for producer, director, writer, script, and star. Notice that we are creating all of the objects before setting them in motion. A good reason for this is that construction of objects may take quite a while, so if they are controlled by a timer or `Animator` object, the delays may be larger than expected.

The next portion of code requests the start methods for each of these objects, but controls them by nested `after` methods. We start the `producer` without any delays, but then pause for nearly a second before starting the `director` credit moving. When that is done (and still in the scope of the `after` method), we reset the timer to wait again before starting the `writer`. We do the same for `costumes` and `star`. (Notice that the `star` negotiated a larger pause before his

```

dialect "objectdraw"

import "timer" as timer
import "Credit" as cr

object {
  inherit graphicApplicationSize (400 @ 400)
  // instructions for user
  def instructions:Text = textAt (20 @ 50) with ("click to start the credits ...")
    on (canvas)

  def pauseTime: Number = 800 // pause time between creating credits

  // start credits rolling when mouse is pressed
  method onMousePress (pt: Point) -> Done {
    instructions.removeFromCanvas

    def producer:cr.Startable =
      cr.credit.with ("Producer ... Martha Washington") size (16) on (canvas)
    def director:cr.Startable =
      cr.credit.with ("Director ... George Washington") size (16) on (canvas)
    def writer:cr.Startable =
      cr.credit.with ("Script ... Thomas Jefferson") size (16) on (canvas)
    def costumes:cr.Startable =
      cr.credit.with ("Costumes ... Betsy Ross") size (16) on (canvas)
    def star:cr.Startable =
      cr.credit.with ("... and starring ... Paul Revere") size (24) on (canvas)

    producer.start
    timer.after (pauseTime) do {
      director.start
      timer.after (pauseTime) do {
        writer.start
        timer.after (pauseTime) do {
          costumes.start
          timer.after (3 * pauseTime) do {
            star.start
          }
        }
      }
    }
  }
}

startGraphics
}

```

Figure 9.15: A class to generate movie credits.

credit rolls and his name is also is displayed in a larger font – vanity!)

This structure ensures that each new timer begins only after the previous credit has started rolling up the screen.

Exercise 9.7.1 *What would have happened if we replaced the code above for starting the objects by the similar unnested code:*

```
producer.start
timer.after (pauseTime) do {
  director.start
}
timer.after (pauseTime) do {
  writer.start
}
timer.after (pauseTime) do {
  costumes.start
}
timer.after (3 * pauseTime) do {
  star.start
}
}
```

Be careful as the answer is not completely obvious.

9.8 Summary

The focus of this chapter has been on how to perform animations. The main points that you should take from this chapter are:

- Animation is a context in which while statements are used often.
- To define a class of animated objects we import animation so that we can access the methods in animator, and then use the while () pausing () do () or while () pausing () do () finally () methods to perform the animation
 - A more primitive way of performing animations is to import timer and use its after method to delay a block from executing until the given time has expired.
 - Executing while loops on animator, and invoking after methods on timers does not delay any of the code that follows the parameters of the delaying methods.

9.9 Chapter Review Problems

Exercise 9.9.1 *Consider the definition of a class, slidingBox. A SlidingBox is a rectangle that moves across the canvas from left to right. It should be removed when it goes across the right edge of the canvas. Unfortunately, the class definition below has an error that will result in no box being visible on the screen. Please fix it.*

```
import "animation" as animator
type SlidingBox = { ... }
class slidingBox (boxLocation: Point) on (canvas:DrawingCanvas) -> SlidingBox {
```

```

// constants omitted

// the box
def box: Graphic2D = filledRectAt (boxLocation) size (boxSize @ boxSize) on (canvas)

method start -> Done{
  animator.while (box.x < canvas.width) pausing (delayTime) then {
    box.moveBy (xSpeed, 0)
  }
  box.removeFromCanvas
}
}

```

Exercise 9.9.2 Suppose that class `droplet` had the line

```
aDropletGraphic.visible := false
```

instead of

```
aDropletGraphic.removeFromCanvas
```

What this difference does this make? Why is it better to use `removeFromCanvas`?

9.10 Programming Problems

Exercise 9.10.1

- a. Revisit the program `LightenUp` from Figure 3.6 and modify it so that now the sun rises slowly and the sky brightens as it rises. The sun should begin rising as soon as the program starts. The sun should stop rising before it begins to exit the window.
- b. Add methods so that the sun also stops rising if the mouse exits the window.

Exercise 9.10.2 Write a program that creates a `ResizableBall` centered around the point where the mouse is clicked. A `ResizableBall` is an animated ball that grows up to twice its original size and then shrinks down to half its size before beginning to grow again. It continues to grow and shrink until the program is stopped. The center of the `ResizableBall` should never move.

Exercise 9.10.3 Write a program called `HitTheTarget` with the following attributes: Refer to Figure 9.16

- When the program begins there should be a target moving back and forth horizontally between the left and right edges of the canvas. This is depicted as the large, light-colored oval in Figure 9.16.

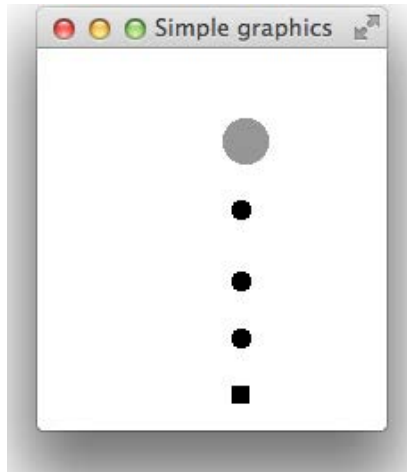


Figure 9.16: Run of HitTheTarget

- *When the user clicks the mouse, a ball shooter is created at the bottom of the window, and aligned with the mouse. The ball shooter will shoot 3 balls that move up the screen, one after another.*
- *If the user hits the target with a ball, the console should indicate this.*

Chapter 10

Graphical User Interfaces in Grace

Users typically interact with computers via *graphical user interfaces*. Because that is a mouthful, they are typically referred to as *GUI*'s, where GUI is pronounced as "gooey". These user interfaces provide the user with multiple windows on the screen and support the use of a mouse to click on buttons, drag items around, pull down and select items from menus, select text fields in which to type responses, scroll through windows, and perform many other operations. GUI components (sometimes called "widgets") are items like buttons and text fields that can be added to the user interface to provide a way for users to interact with a program.

So far we have not been able to program using these components. We can draw geometric objects on a canvas and interact with programs using mouse actions, but we haven't yet seen how to construct and interact with GUI components in the way that one does with most programs running on personal computers. In this chapter we will introduce you to techniques for programming a simple graphical user interface with your Grace programs.

The `objectdraw` library includes facilities to assist in the programming of GUI interfaces. Grace's primitive facilities are based on the GTK library for C, but the objects and methods provided by Grace are higher-level and similar to those provide by the Java Swing library. We will only use the facilities of the `objectdraw` library.

10.1 Text fields

As our first example of using GUI components we introduce the class `textField`. An object from the class `textField` displays a single line of user-updatable text.

A `textField` object differs from objects of `objectdraw`'s `Text` class in two important ways. First, while only the programmer can change the contents of a `Text` object, a user may change the contents of a `TextField` object by clicking in the field and typing new text. Thus `TextFields` are very useful for obtaining user input.

Another important difference between the two is the way objects are placed in a window. A `Text` object can be placed at any point desired on a `canvas`. A `TextField` can be added to a window, but we cannot specify its exact position in the window. Grace will determine the exact position it will be placed.

Figure 10.1 contains the code for an object that displays a window with a text field at

```

dialect "objectdraw"

def textController: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 400)

  // the field to hold user input
  def input: TextField = textFieldLabeled ("Enter text here")

  // add the actual text field to top of window
  prepend(input)

  // respond to mouse presses on the canvas
  method onMousePress (point: Point) -> Done {
    textAt (point) with (input.text) on (canvas)
  }

  startGraphics
}

```

Figure 10.1: textController using a text field

the top of the window and a canvas filling the rest of the window. Each time the user clicks on the canvas, the program generates a `Text` object at that spot with the contents of the text field. Figure 10.2 displays a window that shows the operation of this program.

10.1.1 Constructing a text field

Text fields are constructed using the `textField` class that is part of `objectdraw`. The type of the object constructed is `TextField`. You can use the `textField` class like any other class. For example, the `textController` program in Figure 10.1 contains the instance variable declaration:

```
def input: TextField = textFieldLabeled( "Enter text here" )
```

The class `textField` takes a `String` parameter specifying the initial contents appearing in the text field. If you do not want the text field to contain any text initially, you can just use the empty string, `""`, as the argument.

10.1.2 Adding a TextField to a Window

While the construction of a `Text` object makes it appear immediately on the `canvas` used as a parameter, creating a `TextField` object does not make it appear in the window. To make a component appear in a window, we must add it to the window.

The window constructed by `Grace` comes with a *layout manager*. The layout manager takes care of arranging components in the window according to the instructions of the programmer. Objects inheriting `graphicApplication` start with the `canvas` as the only item in the window. You can use `prepend` to add new items before (above) the `canvas` or you can use `append` for adding after (below) the `canvas`. We'll see later how we can get finer control over this layout using containers.

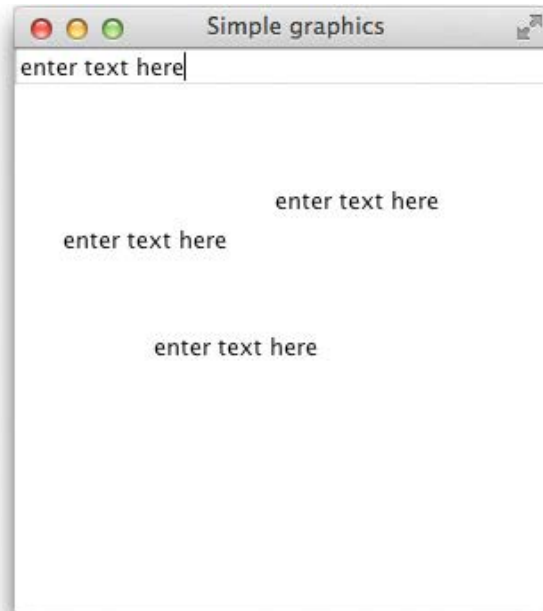


Figure 10.2: TextField in window.

For example, the input text field is added to the top edge of the window when the following statement is executed:

```
prepend (input)
```

10.1.3 Getting Information from a Text Field

The program in Figure 10.2 constructs a new `Text` object on the canvas every time the user clicks the mouse on the canvas. The string displayed in the new `Text` object is obtained from `input`, an object of type `TextField`. The contents of a text field are obtained by sending it a `text` message, as is shown in the `onMouseClicked` method. It is also possible to send a text field a `text:=()` message to reset the contents of the field.

In summary, a text field can be a useful way of getting text input from a user into a program. To use a text field you must construct it and then add it to the desired part of the window. The `text` method can be used to retrieve the contents of a text field.

Exercise 10.1.1 Write a program `textMimic` that places two text fields on the top of the window. When the mouse is clicked on the top half of the canvas, the program should display the contents of the first text field at the top of the window as a `Text` object at the point the mouse was clicked. Similarly, if the user presses the mouse button on the bottom half of the screen, the program should display the contents of the second text field as a `Text` object at the position the mouse was clicked. For example, suppose the canvas was 400 pixels tall, the first text field at the top showed “Hello world”, while the second showed “Goodbye world”. If the mouse is pressed at the position (100, 100), then the program would display a `Text` object at

the position (100, 100) that read "Hello world." If it is pressed at (150,300) then "Goodbye world" would be displayed at that point on the canvas.

10.1.4 Text Boxes

A `TextBox` is like a text field, but the user cannot change the contents. It has methods `contents` and `contents:=` rather than `text` and `text:=`. Thus the programmer can change the text, but the user cannot.

Text boxes are typically used as labels or to give the user instructions. A text box may be created by evaluating an expression of the form.

```
textBoxWith("initial contents")
```

10.1.5 Number Fields

A `NumberField` is like a `TextField` except that the field can only hold numbers. Its type differs from `TextField` only in that the methods `text` and `text:=` have been replaced by `number` and `number:=`. If the user attempts to enter a non-numeric value to the field in the program window, then it will simply be ignored.

A number field is generated by evaluating an expression of the form

```
numberFieldLabeled("initial contents")
```

10.2 Buttons and Events in Grace

In the previous section we showed how to construct, add, and get the contents of a `TextField`. In this section, we show how to use buttons in Grace. Buttons require slightly more programming effort than text fields because we want a click on a button to trigger a program action. We explain how to accomplish this using Grace events and event-handling methods.

As a simple example of the use of a button in a Grace program, we will modify the program from the previous section so that the canvas is erased whenever a button is pressed. The new program is shown in Figure 10.3.

10.2.1 Creating and Adding Buttons

Buttons are represented in the `objectdraw` library by the type `Button`. Buttons may be constructed and added in a way similar to text fields. The button `clearButton` is declared in the program as follows

```
def clearButton: Button = buttonLabeled ("Clear screen")
```

The class `button` creates the button with the label provided as a parameter. In this case it is labelled "Clear screen". As with a text field, the button must also be added to the window. We again add it to the top part of the window:

```
// add the button to the top of the canvas  
prepend (clearButton)
```

Because `prepend (input)` comes later in the program than `prepend (clearButton)`, `input` will appear first. I.e., if you `prepend` an item, it will occur before all others already added.

```

dialect "objectdraw"

def textButtonController: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 400)

  // define the button
  def clearButton: Button = buttonLabeled ("Clear screen")

  // add the listener
  clearButton.onMousePressDo {mevt: MouseEvent ->
    canvas.clear
  }

  // add the button to the top of the canvas
  prepend (clearButton)

  def input:TextField = textFieldLabeled ("Enter text here")
  prepend (input)

  // respond to mouse presses on the canvas
  method onMousePress (point:Point) -> Done {
    textAt (point) with (input.text) on (canvas)
  }

  startGraphics
}

```

Figure 10.3: TextButtonController program using TextField

10.2.2 Handling Events

We already know that when the user clicks on the canvas, the code in the `onMouseClicked` method is executed. When a user clicks on a button constructed from the class `Button`, however, the `onMouseClicked` method is not executed. Instead, the operating system will look to see if the programmer has used an `onMousePressDo` or `onMouseClickedDo` (or even `onMouseReleaseDo`) on the button to associate an action that should be taken when `buttonPressed`, that is designed to handle button clicks.

To understand how this works, let's back up for a minute to get a better understanding of how `objectdraw` programs respond to user actions. `Objectdraw` supports a style of programming known as *event-driven programming*. We have been programming in this style with our programs that react to mouse actions. For example, whenever the user clicks the mouse button on the canvas, it generates an event that results in the `onMouseClicked` method being executed by the computer.

A button generates an event of type `MouseEvent` whenever the user interacts with it with the mouse (e.g., by clicking or pressing on it).

Suppose the user presses the mouse on a button. The operating system the looks to see if the programmer has associated an action with the button to be executed when the mouse

was pressed. If it finds one, then it performs the action, supplying the actual `MouseEvent` so that the user can see where the mouse was, if desired.

The user associates an action with a mouse press on a button by writing a statement of the form

```
myButton.onMousePressDo {mouseEvt: MouseEvent ->
    // commands to be executed
}
```

In our example above this statement was:

```
// add an action to be executed when the user presses on the mouse over the button
clearButton.onMousePressDo {mouseEvt: MouseEvent ->
    canvas.clear
}
```

The identifier `mouseEvt` supports method `at`, which returns the `Point` where the mouse was pressed, where the coordinates are measured relative to the upper left corner of the button. It is rare for you to actually use the location of the mouse in these actions. Thus, while you must write the mouse event, it is generally ignored in the actions. In our case we simply cleared the canvas when the mouse was pressed on the button.

Just as the `onMousePress` method on the canvas is provided with a parameter representing the point the mouse was clicked on, the action associated with `onMousePressDo` is provided with a parameter of type `MouseEvent` that contains information about the object that triggered the event and where the mouse was when pressed. For example, `evt.source` returns the button that triggered the event, `evt`, and `evt.sourceLabel` returns the label on that button. While we don't need that information for this example, we will present examples later that do use that information.

Pressing on a button triggers the execution of all actions that were associated with the button using the `onMousePressDo` method.

We will generally handle interaction with GUI components similarly to the way in which we have been handling mouse actions. That is, we will have the main program object contain the method requests that determine the actions to be taken when the user interacts with a GUI object. Thus we wrote

```
// add an action to be executed when the user presses on the mouse over the button
clearButton.onMousePressDo {mouseEvt: MouseEvent ->
    canvas.clear
}
```

in the code of the object after the definition of `clearButton`. As described earlier, the action associated with `clearButton` by this method request erases the canvas.

We could have handled mouse actions on the canvas in exactly this way from the beginning of this text. That is, rather than write

```
method onMousePress (pt: Point) -> Done {
    //commands using pt
}
```

we could have instead written:

```
canvas.onMousePressDo {mouseEvt: MouseEvent ->
```

```
def pt: Point = mouseEvt.at
  // commands using pt
}
```

We did not do this for two reasons. First we wanted to avoid the extra overhead of extracting the Point from the mouse event. Second, we wanted the mouse handling methods to look as similar as possible to other methods.

10.3 Checklist for Using GUI Components in a Program

We have now seen examples using both text fields and buttons. With that background, we can now summarize the actions necessary to construct and use a GUI component in an object inheriting `graphicApplication`, where that class also serves as the listener for the component.

1. **Construct the GUI component:**

```
def input: TextField = textFieldLabeled ("enter text here")
def clearButton: Button = buttonLabeled ("Clear Canvas")
```

2. **Add the component to the window:**

```
prepend (input)
append (clearButton)
```

3. **If the program is to respond to events generated by the component, associate actions to be taken in response to mouse actions on the component.** For example:

```
clearButton.onMousePressDo {mouseEvt: MouseEvent ->
  canvas.clear
}
```

While the `onMousePressDo` method is appropriate for a button, the `onChangeDo` method is the correct method to use for text fields. It will be triggered when the user hits the enter key after typing in the field.

```
input.onChangeDo {evt: Event ->
  print (input.text)
}
```

Do not forget that the action must take an appropriate event as a parameter.

While different kinds of GUI components generate different kinds of events requiring different event-handling methods and listener types, the checklist above summarizes the steps a programmer must take in order to add and use any kind of GUI component. Always make sure that you have taken care of each of these requirements when using GUI components.

Exercise 10.3.1 Write a program `ClickMe` that places a button on the top of the window. The button should be labeled “Click Me.” When the button is clicked, a `Text` object should display the number of times the user has clicked. For example, after the user has clicked twice, the canvas should read, “You have clicked 2 times.” Be sure to follow steps 1 through 3 to add the button.

Exercise 10.3.2 Write a program `RandomCircles` that places a button on the top of the window. The button should be labeled “Draw a circle.” When the button is clicked, the program should draw a circle of a random size (between 10 and 100 pixels in diameter) at a random position on the canvas. Be sure to follow steps 1 through 3 to add the button.

10.4 Choice menus

A popular GUI component used in Grace programs is a pop-up menu, which is generated by class `menuWithOptions` that generates object of type `Choice`. In Figures 10.4 and 10.5 we illustrate how a menu can be used to choose what kind of figures to put on the screen in a simple drawing program.

This program has a few new features we haven’t seen before. First we define an object `empty` that responds in the simplest possible way to any method request. Its type is `Movable`, which has methods `moveBy`, `contains`, and `color :=`. It is intended to be a placeholder at the beginning of the program when nothing has yet been drawn. Because it will be associated with the variable `newShape` that will also be associated with geometric objects, it will need to be able to respond to the same method requests that will be made to these geometric objects in this program. It turns out that the program only uses those methods listed in `Movable`. By defining `Movable`, and defining `newShape` to that type (rather than `Graphic2D`), we only have to implement those three methods in object `empty`.

The object `empty` essentially represents a non-existent object on the canvas. Thus when we tell it in `onMousePress` to change its color, it doesn’t need to do anything. (We included the methods `moveBy` and `contains` in anticipation of a later example.) Notice that because all of our geometric objects have methods `moveBy`, `contains`, and `color :=`, we can also associate the variable `newShape` with framed rectangles, framed ovals, and filled rectangles.

Now let’s look at the main drawingProgram. It defines an identifier `shapes`, which will be associated with our pop-up menu. We construct a menu by writing

```
def shapes = menuWithOptions ("FramedSquare", "FramedCircle", "FilledSquare")
```

The class definition `menuWithOptions` is unusual, in that we may supply as many parameters as we like in the form of a *line-up*, which is a list of elements separated by commas, and surrounded by square brackets. In this case we are supplying three options, but we could just as easily had 6 or more.¹ With the declaration above, the menu will have three items, showing in the same order as written in the class construction when the user presses the mouse on the menu.

This menu is added to the window using a `prepend` or `append` method as with buttons and text fields. In this case, the menu is added to the top of the window, just above the canvas.

¹We say that this parameter has type `Iterable[[String]]`, meaning that it supports a line-up of arguments, all of the type `String`.

While we could associate an action with changing the selection in the menu, we will hold off on that until later. Instead in this program, if the user clicks on the drawing area then a geometric object will be drawn on the canvas at that point. The kind of object to be drawn will depend on what was the last item selected from the pop-up menu: a framed square, framed circle, or filled square.

If the user instead presses the mouse on one of the colored rectangles on the right edge of the screen then the last object drawn on the screen will be changed to that color.

A danger in many programs is sending a method request to a variable that has no value associated with it. These are referred to as uninitialized variables. If you send a method request to an uninitialized variable then your program will crash with a run-time error. To avoid that we initialized the variable `newShape` to refer to the empty object. Then if we sent a method request of `color:=` it would do nothing, exactly as desired!

Inside the `onMousePress` method we see the use of a `match-case` statement. This is a handy way to handle multiple options. In this case, the choice of case depends on the value showing on the pop-up menu. We can obtain this string by writing `shapes.selected`. The case clauses correspond to the three possible strings that could have been selected on the `shapes` menu. In each case we create the appropriate type of object where the user pressed the mouse.

Let's now take a look at another example using menus. In Figure 10.6 we introduce a slight variant of the `FallingBall` class from Figure 9.4 in Chapter 9. In this new version we have added a new method `setSpeed` that allows us to change how far the ball falls between pauses. Using a large value for the integer parameter makes the ball appear to move faster, while a small value makes it appear to move slower.

We have also included an object `notABall`, that has the same type as `fallingBall`. It is designed to serve as a place-holder for variables of type `AnimatedSpeed` before they have an actual ball assigned to them. It accepts requests of `start` and `changeSpeed`, but does nothing in response.

We will write an object that inherits `graphicApplication` to handle user interactions with the ball. As with the previous case, when the user clicks on the canvas, the program will construct a ball that will begin falling. However, we will also introduce a menu to allow the user to change the speed at which the ball will fall. This will be accomplished in the object `speedBallController` in Figure 10.8. The menu will include three alternatives, "Slow", "Medium", and "Fast." A picture showing the window with the menu at the bottom is given in Figure 10.7.

Let's follow through the checklist from the previous section to make sure that we do everything necessary to properly add and use a menu GUI component generated from the class `menuWithOptions`. As we do this, we will also highlight those places where menus are handled differently from buttons or text fields.

1. **Construct the menu:** Menus are constructed by providing a list of the options to be shown. The relevant code in the class is:

```
def speedMenu: Choice = menuWithOptions ["slow", "medium", "fast"]
```

The first line constructs the menu and associates it with variable `speedMenu`. The items "Slow", "Medium", and "Fast" are added to the menu when it is constructed.

Items appear on the pop-up menu in the order that they are added, with the first

item added showing when the menu is first displayed. Thus “Slow” would normally be displayed when the menu first appears.

2. **Add the menu to the window:**

```
append(speedMenu)
```

This places the menu after the canvas – on the bottom of the window.

3. **Associate actions to the menu:** Apply method `onChangeDo` to an action that responds appropriately to the item selected. As is common with menus, the action associated with changes to the menu will use a match statement to select the correct response.

```
speedMenu.onChangeDo{ evt: Event ->
  match (speedMenu.selected)
  case {"slow"    -> speed := slowSpeed}
  case {"medium" -> speed := mediumSpeed}
  case {"fast"   -> speed := fastSpeed}
  ball.changeSpeed(speed)
}
```

There is little difference between adding and using a menu and a button. The action associated with the menu resets the speed of the last ball dropped as well as balls to be dropped in the future. To make these changes to the speed, it needs a way to determine the current setting of a menu. The class `menuWithOptions` provides the method `selected`, which returns the string that the user chose from the menu.

Exercise 10.4.1 *If you click the mouse in the window several times in rapid succession, multiple balls will be constructed and will fall toward the bottom of the window. If you use the menu to change the speed, only the last ball constructed will change speed. Explain why.*

Exercise 10.4.2 *This exercise builds on the previous one. Suppose you wish to use the menu to change the speed of all of the balls still falling. How would you change the `speedBallController` and `fallingBall` classes so that this will happen? Hint: The ball must be able to know the speed setting at all times.*

Exercise 10.4.3 *Write a class called `BoxColor` that initially draws a black `FilledRect` on the canvas. Add a menu to the bottom of the window. The menu should include the items “red,” “yellow,” “green,” “blue,” and “black.” When a color is chosen from the menu, the color of the `FilledRect` should change appropriately.*

10.5 Using Containers

We now know how to add a text field, a button, or a menu to a window. However, the buttons and menus may not look as expected because they are just inserted as they fit either before or after the canvas. Often we want more control over how elements are laid out in the window to make it easier for the user to find the appropriate action.

In this section we will learn how to solve these problems and make more attractive user interfaces by using containers.

We begin by solving the following simple problem. Rather than using a menu to set the speed of the falling ball, as in the `MenuBallController` class, we would like to have three buttons to set the speed to three different values. We will also include a menu for selecting the color of the falling ball.

We could add each of the buttons and the menu all to the bottom of the window. However if we just append each of them, we will end up with a layout like that shown on the left side in Figure 10.9, which looks relatively unorganized and difficult to use. Instead, we'd like to put the menu above the three buttons, as shown on the right of the figure.

To help solve problems like this, `objectdraw` provides a component called a `Container`. A container is a kind of component that can hold GUI components just like a window does, but can also serve as a component in the same way as a button or menu. Containers are typically used to organize the window into collections of subcomponents. In particular, if we wish to arrange items more logically in one area of the window, we can construct a panel, add the GUI components to the panel, and then add the panel to the window.

This program will reuse out animated falling ball from Figure 10.6.

We have placed the code for handling user interactions in Figures 10.10 and 10.11. In part 1, we create a `colorMenu` as well as three buttons to regulate the speed of the falling ball. The action associated to `colorMenu` assures that the variable `ballColor` will be updated to reflect the new color menu selection. We also associate actions to each of the three buttons, namely to change the speed of the ball. The standard `onMousePress` method creates a new ball when the mouse is pressed and starts it on its way down the screen.

Part 2 of the program, provided in Figure 10.11 is primarily concerned with the layout of the GUI components. We will lay out these components in two rows below the canvas, where the first row will hold just the color menu, while the second row will hold the three buttons.

The program will use three containers to arrange the items. The `Container row1` will hold the color menu, `row2` will hold the three buttons, while `bottomBox` will contain the two rows.

We can create a new `Container` with `emptyContainer`, and then add items (from left to right) by appending the components. Once `row1` and `row2` are created and filled, we will place them both in `bottomBox`. However, if we just add them without doing anything else, they will be added from left to right. Instead we want them listed vertically – with the first at the top, and the second below it. We can do this by executing

```
bottomBox.arrangeVertical
```

You won't be surprised to know that `Containers` also have a method `arrangeHorizontal`, but since that is the default, you likely will never use it.

Once the two rows have been appended to the `bottomBox`, all that remains is to add it to the window, which can be done by executing `append(bottomBox)`.

In general, it takes some experimentation with containers in order to get the screen layout to look good. All you really need to remember is that any component can be added to a container, and containers are themselves components, so containers can be added to containers.

Exercise 10.5.1 *Revise `adjustableBallController` to add a menu to change the speed of the ball. Make sure the menus are on top of each other, and put them both at the top of the window.*

10.6 Handling Keystroke Events

We can write Grace programs so that every keystroke made by the user generates an event that can be handled by the program. This can be useful in processing input one character at a time or in using keyboard input to control actions in games. For example, in a game that controls a rocket ship, one could use the arrow keys on the keyboard to change the direction of movement of the ship or use the space bar to trigger an action such as shooting a missile or putting up defense shields.

Keyboard events are generally associated with the canvas, but can also be associated with text and number fields. Because the canvas already exists, we can skip the usual steps of creating and adding a component. Instead all we need to do is associate a listener with the keyboard events, and make sure the listener has the appropriate methods to handle key events.

We need only include a statement of the form `begin method:`

```
onKeyPressDo{keyEvt: KeyEvent -> ... action to be taken ...}
```

When this statement is executed in a program, it will begin executing the action whenever a key is pressed. There are also other methods `onKeyReleaseDo` and `onKeyTypedDo` that may be used to associate actions with releasing a key or with pressing and releasing a key (much like the difference between a mouse click and separate presses and releases). For simple applications typically only one of the three may be needed.

The only methods available on a `KeyEvent` are `source` and `code`. The method `code` is the one you will find most useful. It returns a code of type `Number` representing the ASCII/Unicode code of the character. This is more useful than returning the character typed in that not all keys (e.g., the arrow and shift keys) return distinct characters.

In Figure 10.12 is a very simple program that prints the codes for the keys pressed while the program is running. The codes for different keys can be found by experimentation with this program or by looking them up in a table of ASCII or Unicode codes. However, here are the codes for many of the important keys:

- the letters "a" to "z" have codes 65 to 90,
- the digits from "0" to "9" have codes 48 to 57,
- the left arrow, up arrow, right arrow, and down arrow have codes 37, 38, 39, and 40, respectively,
- the shift key has code 16, and
- the space bar has code 32.

Sample code that uses arrow keys to change the speed of a falling ball is shown in Figure 10.13. When the up arrow key is pressed the ball's speed increases, while the down arrow key reduces the speed (and can even make it negative).

10.7 Summary

In this chapter we introduced the standard Grace event model and a number of GUI components. While there is a lot more that can be learned about the design of graphic user interfaces, the information provided here should be enough to get you started in writing programs using simple GUI components.

The tasks to be performed in creating and using GUI components include the following:

1. Construct the GUI component.
2. Add the GUI component to the window.
3. If the program should respond to an interaction with the item, use the `onMousePressDo` or other appropriate method on the GUI component to associate an action with the component.

More details on the types and methods of the GUI components are available in Appendix D

10.8 Chapter Review Problems

Exercise 10.8.1 Write a program called `sunriseSunset` that draws a sun near the bottom of the canvas. Create a button on the bottom of the screen that is labeled "Rise." When the button is clicked, the sun should rise by moving up 5 pixels. Once the top of the sun reaches the top of the canvas, the label of the button should change to "Set" and clicking the button should cause the sun to move down 5 pixels. When the sun reaches the bottom of the canvas, the button should be relabeled "Rise."

Exercise 10.8.2 Modify your `sunriseSunset` program so that the rising and setting is controlled by the arrow keys on the keyboard instead of buttons. When the user presses the up arrow, the sun should move up 5 pixels (unless it is already at the top of the screen). When the user presses the down arrow, the sun should move down 5 pixels (unless it is already at the bottom of the screen).

Exercise 10.8.3 Write a program called `bubbles` that places a Choice menu on the bottom of the screen and a button at the top of the screen. The menu should include as items the numbers 0 to 10. When a number is selected, the program should draw that number of framed ovals on the screen. The size and location of each framed oval should be determined randomly. Be sure to use the `selected` method of the Choice to access the selected item. You can convert that to a number by making a method request of `asNumber` to the resulting string. The button should be labeled "Erase." When the button is pressed, all the bubbles on the screen should disappear.

Exercise 10.8.4 Write a program called `sizableColorfulBox` that initially draws a black filled rectangle on the canvas. Add two buttons and a menu to the top of the window. The buttons should be labeled "bigger" and "smaller." Pressing the "bigger" button should increase the size of the `FilledRect` by 10 pixels in each dimension and the "smaller" button should

decrease the size of the `FilledRect` by 10 pixels in each dimension. The menu should include items "black," "red," "yellow," "green," and "blue" and should change the color of the box appropriately.

10.9 Programming Problems

Exercise 10.9.1 Write a program called `votingBooth` that simulates voting. Use three different buttons labeled "Candidate 1," "Candidate 2", and "Candidate 3" to collect the votes. Above each button, the program should display the number of votes the candidate has received.

Exercise 10.9.2 Write a program called `iceCreamStand` that allows the user to "purchase" ice cream. On the top of the window there should be three buttons and a Choice menu. The buttons should be labeled "Small," "Medium," and "Large." The menu should include 5 different ice cream flavors. Below each button there should be a label that displays the price of that particular size of ice cream. When the user clicks a button, the bill should be displayed on the canvas as a series of objects of type `Text` lined up one above the next. The bill should include the size, flavor, and price of the ice cream. On the bottom of the window, there should be a "New Order" button that resets the bill so that it is blank.

dialect "objectdraw"

```
// type of an object that can be dragged and have its color changed
type Movable = {
  moveBy (dx: Number, dy: Number) -> Done
  contains (pt: Point) -> Boolean
  color:= (c: Color) -> Done
}

// an empty object that responds to the same messages as the
// graphic objects, but doesn't do anything.
// It is used to initialize the newShape variable in case
// someone tries to change properties before a geometric
// object has been created
def empty: Movable = object {
  method moveBy (dx: Number,dy: Number) -> Done { }
  method contains (pt: Point) -> Boolean {false}
  method color:= (c:Color) -> Done { }
}

def drawingProgram: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 400)
  windowTitle:= "Drawing Program"

  // size of the drawing area
  def drawingAreaSize: Number = 300

  // size of the color buttons
  def buttonHeight: Number = 40
  def buttonWidth: Number = 40

  // x coordinate for the color buttons
  def colorsX: Number = buttonWidth + drawingAreaSize + 2

  // default size for geometric objects
  def defaultSize: Number = 50

  // boundary around the drawing area
  def drawingArea: Graphic2D =
    framedRectAt (buttonWidth @ 0) size (drawingAreaSize @ drawingAreaSize) on (canvas)

  // the four color rectangles allowing us to change the color of the object
  def redRect: Graphic2D = filledRectAt (colorsX @ 0) size (buttonWidth @ buttonHeight) on (canvas)
  redRect.color := colorGen.red

  def blueRect: Graphic2D = filledRectAt (colorsX @ buttonHeight) size (buttonWidth @ buttonHeight)
    on (canvas)
  blueRect.color := colorGen.blue

  // to be continued
```

Figure 10.4: Drawing program, part 1

```

def greenRect: Graphic2D = filledRectAt (colorsX @ (2 * buttonHeight)) size (buttonWidth @ buttonHeight)
  on (canvas)
greenRect.color := colorGen.green

def yellowRect: Graphic2D = filledRectAt (colorsX @ (3 * buttonHeight)) size (buttonWidth @ buttonHeight)
  on (canvas)
yellowRect.color := colorGen.yellow

// pop up menu allowing choice of kind of object to be drawn
def shapes: Choice = menuWithOptions (["FramedSquare", "FramedCircle", "FilledSquare"])
prepend (shapes)

// most recent geometric object drawn on the screen.
var newShape: Movable := empty

// When mouse is clicked create a new object (if clicked in boundaries)
// or change color (if clicked on a color button)
method onMouseClick (pt:Point) -> Done {
  if (drawingArea.contains (pt)) then {
    match(shapes.selected)
      case {"FramedSquare" ->
        newShape := framedRectAt (pt) size (defaultSize @ defaultSize) on (canvas)
      }

      case {"FramedCircle" ->
        newShape := framedOvalAt (pt) size (defaultSize @ defaultSize) on (canvas)
      }

      case {"FilledSquare" ->
        newShape := filledRectAt (pt) size (defaultSize @ defaultSize) on (canvas)
      }

    } elseif {redRect.contains (pt)} then {
      newShape.color := colorGen.red
    } elseif {greenRect.contains (pt)} then {
      newShape.color := colorGen.green
    } elseif {blueRect.contains (pt)} then {
      newShape.color := colorGen.blue
    } elseif {yellowRect.contains (pt)} then {
      newShape.color := colorGen.yellow
    }
  }
}

startGraphics
}

```

Figure 10.5: Program for drawing, part 2

```

dialect "objectdraw"
// \emph{Animate a falling ball}
import "animation" as animator

type AnimatedSpeed = {
  start -> Done
  changeSpeed (newYDiff: Number) -> Done
}

// a ball that falls starting at pt and moves each time by yDiff
class fallingBallAt (pt: Point) speed (yDiff: Number)
  color (ballColor: Color) on (canvas: DrawingCanvas) -> AnimatedSpeed {
  // diameter of the ball
  def diameter: Number = 20

  // delay between moves
  def delay: Number = 30

  // how far to move each time
  var yChange: Number := yDiff

  // start the animation and drop the ball
  method start -> Done {
    def ball: Graphic2D = filledOvalAt (pt) size (diameter @ diameter) on (canvas)
    ball.color := ballColor
    animator.while {pt.y < canvas.height} pausing (delay) do {
      ball.moveBy (0, yChange)
    } finally {
      ball.removeFromCanvas
    }
  }

  // change the speed of the ball
  method changeSpeed (newYDiff: Number) -> Done {
    yChange := newYDiff
  }
}

// An object that can stand in for a ball when none has
// yet been created
def notABall: AnimatedSpeed = object {
  method start -> Done {}
  method changeSpeed (newYDiff: Number) -> Done {}
}

```

Figure 10.6: FallingBall class

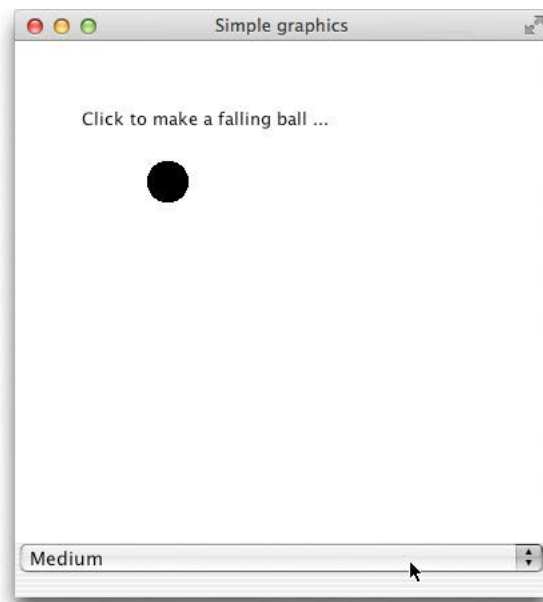


Figure 10.7: Menu constructed by MenuBallController.


```

def speedBallController: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 500)

  var ball: AnimatedSpeed := notABall

  def speedMenu: Choice = menuWithOptions ["slow", "medium", "fast"]

  def fastSpeed: Number = 8
  def mediumSpeed: Number = 4
  def slowSpeed: Number = 2
  var speed: Number := slowSpeed

  speedMenu.onChangeDo{ evt: Event ->
    match (speedMenu.selected)
      case {"slow"   -> speed := slowSpeed}
      case {"medium" -> speed := mediumSpeed}
      case {"fast"   -> speed := fastSpeed}
    ball.changeSpeed(speed)
  }

  // start the ball falling
  method onMousePress(pt: Point) {

    ball := fallingBallAt (pt) speed (speed) color (colorGen.black) on (canvas)
    ball.start
  }

  append(speedMenu)

  startGraphics
}

```

Figure 10.8: MenuBallController class

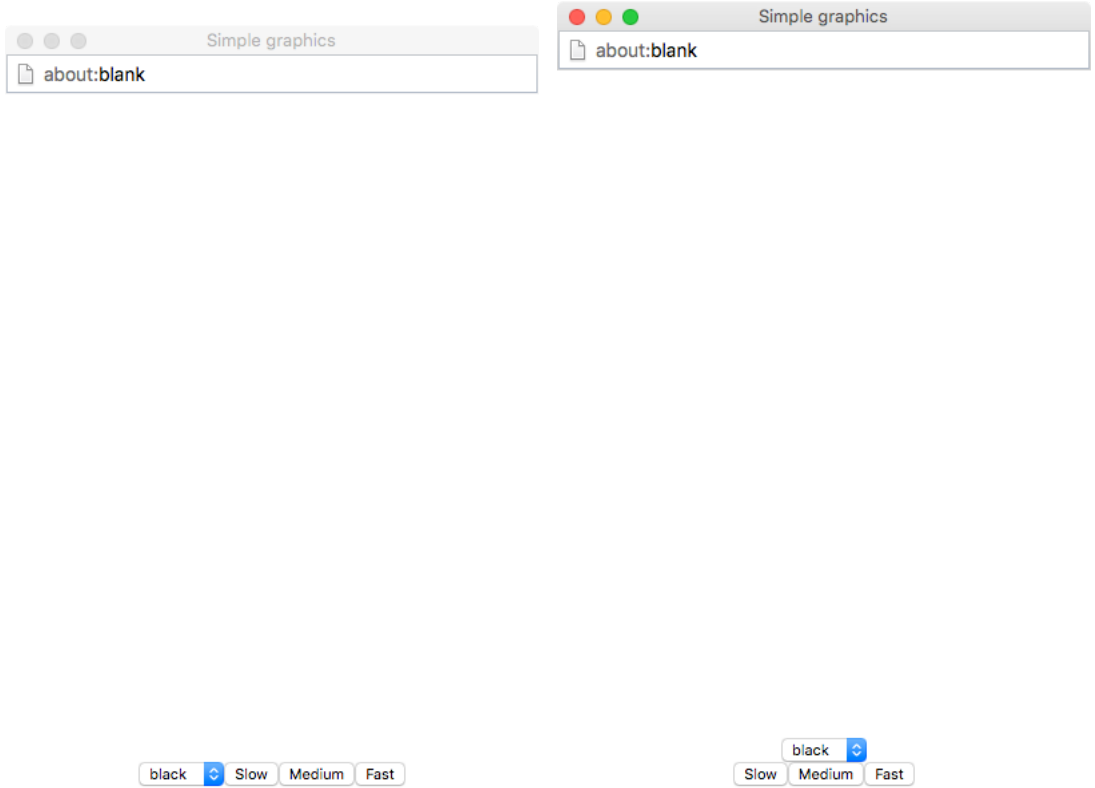


Figure 10.9: Windows before and after use of containers.

```

def adjustableBallController: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 500)

  var ball: AnimatedSpeed := notABall

  def colorMenu: Choice = menuWithOptions ["black", "red", "blue", "green", "yellow"]

  var ballColor: Color := colorGen.black

  colorMenu.onChangeDo{ evt: Event ->
    match (colorMenu.selected)
    case {"black" -> ballColor := colorGen.black}
    case {"red" -> ballColor := colorGen.red}
    case {"blue" -> ballColor := colorGen.blue}
    case {"green" -> ballColor := colorGen.green}
    case {"yellow" -> ballColor := colorGen.yellow}
  }

  def fastSpeed: Number = 8
  def mediumSpeed: Number = 4
  def slowSpeed: Number = 2
  def startingSpeed: Number = mediumSpeed

  // define the speed buttons
  def fastButton: Button = buttonLabeled ("Fast")
  def mediumButton: Button = buttonLabeled ("Medium")
  def slowButton: Button = buttonLabeled ("Slow")

  // Pressing buttons changes speed of ball
  fastButton.onMousePressDo {mevt: MouseEvent ->
    ball.changeSpeed (fastSpeed)
  }

  mediumButton.onMousePressDo {mevt:MouseEvent ->
    ball.changeSpeed (mediumSpeed)
  }

  slowButton.onMousePressDo {mevt:MouseEvent ->
    ball.changeSpeed (slowSpeed)
  }

  // start the ball falling
  method onMousePress(pt: Point) {
    ball := fallingBallAt (pt) speed (startingSpeed) color (ballColor) on (canvas)
    ball.start
  }
}

```

Figure 10.10: adjustableBallController with multiple buttons and a menu. Part 1.

```

// row1 will hold just the colorMenu
def row1: Container = emptyContainer
row1.append (colorMenu)

// row 2 will hold all three buttons controlling the speed of the ball
def row2: Container = emptyContainer
row2.append (slowButton)
row2.append (mediumButton)
row2.append (fastButton)

// bottomBox will hold the two rows
def bottomBox: Container = emptyContainer
// rows are laid out from top to bottom
bottomBox.arrangeVertical

// Add row1 and row2 to the box and then put bottomBox below the canvas
bottomBox.append(row1)
bottomBox.append(row2)
append(bottomBox)

startGraphics
}

```

Figure 10.11: adjustableBallController with multiple buttons in a panel. Part 2.

```

dialect "objectdraw"

def keyDemo: Application = object {
  inherit applicationTitle("Test Keys") size (300 @ 300)

  onKeyPressDo{keyEvt: KeyEvent ->
    print (keyEvt.code)
  }

  startApplication
}

```

Figure 10.12: Program to print key codes

```

def keyBallController: GraphicApplication = object {
  inherit graphicApplicationSize (400 @ 500)

  var ball: AnimatedSpeed := notABall

  var speed: Number := 4

  onKeyPressDo{ evt: KeyEvent ->
    match (evt.code)
      case {38 -> speed := speed + 1}
      case {40 -> speed := speed - 1}
      case {_ -> } // ignore presses of other keys
    ball.changeSpeed(speed)
  }

  // start the ball falling
  method onMousePress(pt: Point) {

    ball := fallingBallAt (pt) speed (speed) color (colorGen.black) on (canvas)
    ball.start
  }

  startGraphics
}

```

Figure 10.13: Controlling ball speed with the keyboard

Chapter 11

Recursion

A key idea in designing a program to solve a complex problem is to break it into simpler problems, solve the simpler problems, and then assemble the final answer from these simpler pieces. Sometimes it is possible to fashion a solution to a problem by solving one or more simpler versions of the *same* problem, and then using those to complete the solution to the original version of the problem. This technique is called *recursion*.

In this chapter we explain how to write programs using recursion in Grace. Programming with recursion does not require the introduction of new features. Instead it represents an important new way of approaching problems.

For a simple example, let's look at Russian nesting dolls. You have probably encountered these in gift shops. They consist of a collection of hollow wooden dolls of decreasing sizes that nest inside each other. Each can be opened by twisting the upper body, revealing a hollow cavity inside. Typically the smallest doll is solid wood, though it can be hollow so that small items can be stored inside. The sizes of the dolls are chosen so that each doll can hold the nested collection of smaller dolls inside of it.

In particular, the second smallest doll can hold the smallest inside. The third can hold the second (which has the smallest inside). In fact each doll can hold the next largest doll with all smaller dolls nested inside.

Let's look at the problem of providing directions on how to make a collection of nested dolls. To make the problem more interesting, suppose that we wish to provide instructions that, given the size of the largest doll, will always result in the largest number of nested dolls in a collection.

Let `minSize` represent the smallest doll that can fit other dolls inside. Clearly if the size of the largest doll is smaller than `minSize` our collection will only have a single doll.

On the other hand, if we start with a doll with size `startSize` \geq `minSize`, then we can build the outer doll with space inside for the other nested dolls. Suppose that to fit inside another doll, the gap between the sizes of the dolls must be at least `gapSize`.

A procedure for building a collection of nested dolls where the outermost doll has size `startSize` is the following:

- If `startSize` $<$ `minSize`, build a doll of that size and you are done.
- If `startSize` \geq `minSize`, then do the following:

1. Build the hollow outer doll with size `startSize`.
2. Build a collection of smaller nested dolls whose outer doll has size `startSize - gapSize`. When finished you are done.

What instructions should be used in the second case to build the collection of smaller dolls? Exactly the same set of instructions can be used, because they are general instructions to build collections of nested dolls of any size.

For example, suppose there is a production line with several workers, where the first worker is given a slip of paper with the value of `startingSize` for the outer doll. Each of the workers is told to follow the instructions given above. If the starting size is less than `minSize` then the first worker builds a doll of that size and the order is complete. If the starting size is greater than or equal to `minSize` then the first worker builds a hollow doll of that size and hands a slip of paper to the next worker with the value `startingSize - gapSize`. That worker will build a doll of that size, handing a slip of paper with an even smaller value to the next worker if the number they received was greater than or equal to `minSize`. This continues until the last doll is built.

This collection of nested dolls is an example of a recursively-defined structure. Recursive structures consist of a base structure (the smallest doll in this example) as well as a way of describing more complex structures in terms of simpler ones of the same sort (a set of nested dolls consists of the largest hollow doll plus a smaller collection of nested dolls that fit inside of it). Methods can be defined based on this recursive structure.

We will begin by first examining recursive structures, those structures that contain one or more subcomponents from the same class as the entire structure. Later we investigate recursive methods, whose execution involves further invocations of the methods themselves to solve slightly simpler problems.

11.1 Recursive Structures

A *recursive structure* is one in which a piece of the structure is similar to the entire structure. Usually this is represented by defining a class that includes one or more instance variables that may refer to objects of the same type as those being defined by the class. In this section we will create different recursive structures that create and manipulate pictures of nested rectangles, create and search collections of strings, and that create and manipulate pictures of broccoli.

11.1.1 Nested Rectangles

Our first example of a recursive structure will be quite simple. We will design a class that will draw nested rectangles of the sort shown in Figure 11.1. Eventually we would like to make these have as many capabilities as the geometric objects from the `objectdraw` library. For example we would like them to have `move` and `moveTo` methods. For now, however, we'll just be content with being able to draw them.

In order to motivate our final solution, we will present three versions of classes to build and draw nested rectangles. The first will use *iteration* or loops of the sort that you have already seen, while the last two will involve recursion.

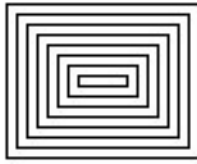


Figure 11.1: Nested rectangles

An Iterative Solution

Because you are used to thinking in terms of loops, you can probably figure out a nice iterative way of drawing a non-empty collection of nested rectangles using a `while` loop. Algorithms based on repeating a process with some sort of loop are typically described as *iterative*.

Here is an example of a constructor for the class `NestedRects` that uses a `while` loop.

```
// Draw nested rectangles at pt' with dimensions width' x height' (both of which must be non-negative)
```

```
dialect "objectdraw"
```

```
class iterativeNestedRectsAt (pt': Point) size (dimension': Point) on (canvas: DrawingCanvas) -> Movable {
  framedRectAt (pt') size(dimension') on (canvas)
  var pt: Point := pt'
  var dimension: Point := dimension'
  while {(dimension.x >= 8) && (dimension.y >= 8)} do {
    dimension := dimension - (8 @ 8)
    pt := pt + (4 @ 4)
    framedRectAt (pt) size (dimension) on (canvas)
  }
}
```

The idea of the object generated by the class is quite simple. It first draws a framed rectangle at the point `pt'` with the given dimensions. Then, while the width and height of the rectangle it has just drawn are both greater than or equal to eight, it adjusts the width, height, `x`, and `y` coordinates in order to draw a new smaller framed rectangle centered inside the one just drawn.

That is relatively simple, but suppose, however, that we want to write a `moveTo` method that moves the nested rectangle to a given location on the canvas. In order to do this we will need to associate names with the individual framed rectangles drawn as part of the nested rectangles. Unfortunately, when we write the program we do not know how many framed rectangles will be needed, as the number will vary depending on what `width'` and `height'` parameters are sent to the constructor. At this point we don't yet know any way in Grace to introduce a variable number of names to be used in a program if we do not know the

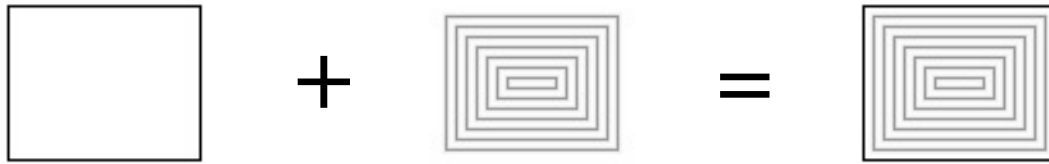


Figure 11.2: A view of nested rectangles as a recursive structure.

maximum number that might be required. As we will see, the recursive solution given in the next section does not have that problem.

A Recursive Solution

The `iterativeNestedRects` class above did not represent a recursive structure. To see a non-empty collection of nested rectangles as a recursive structure, we need to change the way we understand nested rectangles. Instead of thinking of them as a series of framed rectangles, we instead think of them as having an outer framed rectangle and then if there is enough space, a smaller collection of nested rectangles inside. In Figure 11.2 we demonstrate how a collection of nested rectangles can be formed from an outer framed rectangle and a smaller collection of nested rectangles. In general, when we look at pictures as being recursive, we try to find a smaller copy in the whole picture that we are looking at.

Figure 11.3 contains a recursive version of the class `nestedRects` that corresponds to this description. It contains an instance variable `outerRect` with type `Graphic` corresponding to the outer framed rectangle and `rest` with type `NestedRects` corresponding to the smaller collection of nested rectangles. The class `NestedRects` represents a recursive structure because the instance variable `rest` is the same kind of object as the entire structure. In particular `rest` has the same type as the object constructed from the class.

Before going through the details of this class, however, let's back up and look at the earlier definitions in the Figure. First we have a definition of the type `Movable`. It represents objects that can be moved around and removed from the canvas.

The definition of `emptyRects` should look a bit odd to you. Let's first start with the motivation. The object `emptyRects` is supposed to represent an empty collection of nested rectangles. Let's put off for a moment why we might want such a thing and instead think about what it would mean to define methods on it.

Let's go back to our example of nested Russian dolls. Suppose there are a bunch of kids sitting around playing with nested dolls. Some have sets with a large number of dolls, while some have just one or two in the set, and some don't have any and just watch the other kids. If the teacher tells everyone to put away their dolls, then the kids with one or more dolls have to put them away, but the kids with none don't have to do anything.

```

dialect "objectdraw"

type Movable = {
  moveTo (pt: Point) -> Done
  moveBy (dx: Number,dy: Number) -> Number
  removeFromCanvas -> Done
}

// An empty collection of nested rectangles
// All methods do nothing!
def emptyRechts: Movable = object {
  method moveBy (dx: Number, dy: Number) -> Done {}
  method moveTo (pt: Point) -> Done {}
  method removeFromCanvas -> Done {}
}

// Create set of nested rectangles w/upper left corner at pt &
// dimensions width x height
class nestedRechtsAt (pt: Point) size (dimension: Point)
  on (canvas: DrawingCanvas) -> Movable {
  def shift: Number = 4 // gap between adjacent rectangles

  // create single outer rectangle
  def outerRect: Graphic2D = framedRectAt (pt) size (dimension) on (canvas)

  // create the rest of the nested rectangles (which may be empty)
  def rest: Movable = if ((dimension.x < (2*shift)) || (dimension.y < (2*shift))) then {
    emptyRechts
  } else {
    nestedRechtsAt (pt + (shift @ shift)) size (dimension - ((2 * shift) @ (2 * shift))) on (canvas)
  }

  // move full set of nested rectangles by (dx,dy)
  method moveBy (dx: Number, dy: Number) -> Done{
    outerRect.moveBy (dx, dy)
    rest.moveBy (dx, dy)
  }

  // move full set of nested rectangles to newPt
  method moveTo (newPt: Point) -> Done {
    def diff:Point = newPt - outerRect.location
    moveBy (diff.x, diff.y)
  }

  // remove all nested rectangles from canvas
  method removeFromCanvas -> Done {
    outerRect.removeFromCanvas
    rest.removeFromCanvas
  }
}

```

Figure 11.3: Recursive nested rectangles

Exactly the same applies here. If you don't have nested rectangles you don't have to do anything. As a result if we look at the definitions of `moveBy`, `moveTo`, and `removeFromCanvas`, there is nothing to do, because they don't have any rectangles at all.

On the other hand if we look at those methods in the class `nestedRects`, there is stuff to be done. Each object constructed from the class has at least one rectangle `outerRect` and the `rest` (which might be `emptyRects`). Thus if we look at the method `moveBy`, we see that we first must move `outerRect` by `dx` and `dy`, and then move the `rest`. Similarly with `removeFromCanvas`. First remove `outerRect` and then request `removeFromCanvas` from the `rest`. (Going back to the dolls analogy, if a child is asked to put them away, they might first put away the biggest and then finish up by putting away the `rest`.)

As for the method `moveTo`, we are going to do exactly the same thing we did in Section 6.3.2 with class `funnyFace`. We will write `moveTo` by figuring out how far the object must move and then using the previously defined `moveBy` method to move all the pieces the same amount. The code here looks a bit different than the code we wrote in the earlier example as we are using `Point` subtraction to subtract the `x` and `y` coordinates simultaneously and then extracting the difference in each of them from the object `diff`.

We've now discussed how to implement the methods in the object `emptyRects` and in objects generated by the class `nestedRects`. Now we need only explain how we construct the nested rectangles initially.

Not surprisingly, it is very easy to draw an empty set of nested rectangles on the screen. Don't do anything! Thus object `emptyRects` has no instance definitions or variables and there is no initialization code at all (matching nicely the fact that our methods don't do anything).

Creating a non-empty collection of nested rectangles is a bit more work as we can see by looking at the code in `nestedRects`. We begin by creating `outerRect`, whose upper-left hand corner and dimensions are given by the parameters of the class.

We now need to create the `rest` of the nested rectangles and associate those with the identifier `rest`. Let's remind ourselves of the rules for constructing the `rest`. If both `width` and `height` are greater than or equal to 8, then we have more rectangles to go, while if either is less than 8, we get to stop.

Because it sounds a bit simpler, let's do the case where one or the other is less than 8. We will call this the *base* case. Now it would be nice to just not do anything. However, we saw earlier that our methods `moveBy` and `removeFromCanvas` (and indirectly `moveTo`) all make method requests to `rest`. If in this case we didn't provide a value for `rest`, then our program would crash! Sending a method to an identifier that hasn't been initialized always results in an error. It's a bit like sending an e-mail message without first filling out the `To:` field. Nothing good comes of it!

This is the place where our `emptyRects` object comes in. If we set `rest` to `emptyRects`, then when we send a `moveBy` or `removeFromCanvas` method request to it, it will do the right thing – nothing!!

Think about this for a moment. Suppose you are creating an object using `nestedRects` where the `width` is 6. Then when it is evaluated, it will set `outerRect` to a framed rectangle with `width` 6 (and `height` whatever was specified). Then, because `width` is less than 8, it will set `rest` to `emptyRects`.

Now suppose you send a method request `moveBy(30, 20)` to this new object. Looking at

the body of `moveBy` we see that it will first move the `outerRect` by 30 and 20, as desired, and then send a method request `moveBy (30, 20)` to `rest`, which is `emptyRects`. It responds to that message by doing nothing, which is exactly right, and we are done.

OK, that was the base case (though at first you may find the base case more confusing!). Let's do the general *recursive* case where both the width and the height are greater than 8. We've already constructed `outerRect`. Again we need to figure out how to create `rest`. As we saw earlier, `rest` needs to also be a set of nested rectangles, but it should be 4 pixels (`shift`) to the right and down from `outerRect`, and its width and height should be 8 pixels ($2 * \text{shift}$) smaller than the original. We can do this by setting `rest` to

```
nestedRectsAt (pt + (shift @ shift)) size (dimension - ((2 * shift) @ (2 * shift))) on (canvas)
```

That's right, we just used the class we are defining to create the smaller set of nested rectangles. After all, it should be easier to do than the original, and we know eventually we are going to get down to the case where we put in `emptyRect` as the value of `rest`.

We will next illustrate exactly what happens when we actually use the class to construct an object. The trace of its execution will be relatively long and detailed, but should prove helpful in understanding better what actually happens during recursion.

Suppose we evaluate

```
nestedRectsAt (50 @ 50) size (19, @ 21) on (canvas)
```

This will result in the execution of

```
def outerRect: Graphic2D = framedRectAt (50 @ 50) size (19 @ 21) on (canvas)
```

Thus the value of `outerRect` for this collection of nested rectangles will be a 19 by 21 rectangle with upper left corner at (50, 50).

Next the value of `rest` is initialized by evaluating the if-then-else expression. Because width is 19 and height is 21, the condition is false. Thus we evaluate:

```
nestedRectsAt (54 @ 54) size (11 @ 13) on (canvas)
```

where the new parameters are calculated according to the expressions given in the constructor call. The result of evaluating this gives us a collection of nested rectangles that is then associated with `rest`.

How do we construct this set of nested rectangles that is assigned to `rest`? We execute the same constructor code as last time, but this time we have different parameters. We begin by evaluating

```
def outerRect: Graphic2D = framedRectAt (54 @ 54) size (11 @ 13) on (canvas)
```

Thus the value of `outerRect` for this new smaller collection of nested rectangles will be an 11 by 13 rectangle with upper left corner at (54,54). The if-then-else expression is again evaluated. Because width is 11 and height is 13, the condition is again false. Thus we evaluate:

```
nestedRectsAt (58, 58) size (3 @ 5) on (canvas)
```

where the new parameters are calculated according to the expressions given in the constructor call. This value is associated with `rest` for this object. Thus the value of `rest` for the set of nested rectangles at (54,54) is yet another set of nested rectangles, this time located at (58,58).

To construct these rectangles we once more execute the constructor code, but with yet another set of parameters. Thus we execute

```
def outerRect: Graphic2D = at (58 @ 58) size (3 @ 5) on (canvas)
```



Figure 11.4: Nested rectangles drawn by evaluating `nestedRectsAt (50 @ 50) size (19 @ 21) on (canvas)`.

But now when we evaluate the condition for the `if–the–else` expression, it will be true, because the new width, three, is less than eight. As a result, the `rest` instance variable of this object will be set to `emptyRects` and the constructor will terminate. This completes execution of the object resulting from evaluating `nestedRectsAt (54 @ 54) size (11 @ 13) on (canvas)`. The value just constructed (the simple 3 by 5 pixel framed rectangle) will get assigned to the instance variable `rest` of this object, completing the execution of that constructor. The resulting object consists of an outer 11 by 13 pixel framed rectangle and an inner object of type `NestedRects` that consists of a single 3 by 5 pixel framed rectangle.

Control now returns to the original constructor call `nestedRectsAt(50 @ 50) size (19 @ 21) on (canvas)`. The object just constructed (the two nested rectangles) is assigned to this object's `rest` instance variable, and the execution of that constructor now terminates. This final object has an outer rectangle that is a 19 by 21 pixel framed rectangle and `rest` consisting of two nested rectangles.

A picture illustrating the results of evaluating `nestedRectsAt (54 @ 54) size (11 @ 13) on (canvas)` is given in Figure 11.4.

In summary, the net result of the original evaluation of the expression is that the class is evaluated three times, which results in the construction of three framed rectangles. Because the class construction ended up being executed three times, there are actually three objects of the class `NestedRects` that were constructed, each of which has different values for their `outerRect` and `rest` instance variables. For example the value of `outerRect` for the initial `NestedRects` object is a framed rectangle at (50, 50), while that for the last `NestedRects` object is a framed rectangle at (58, 58).

See Figure 11.5 for a diagram showing the objects that will be constructed.

Recall that we were not able to move the nested rectangles created by the program using a `while` loop because we did not have names for all of the framed rectangles created by the constructor. However, we have written that method for this recursive version. Let's trace through the message request `moveBy (15, 20)` to the nested rectangles we created earlier by the evaluation of `nestedRectsAt (50 @ 50) size (19 @ 21) on (canvas)`, then we can see that all of the framed rectangles will be moved appropriately.

Because the value of `outerRect` for this object is the framed rectangle at (50, 50), that will be moved to (65, 70) by the first line of the `moveBy` method's code. Next the message `moveBy (15, 20)` will be sent to the `rest` instance variable, which holds the nested rectangles at (54, 54) that we saw constructed in the example above.

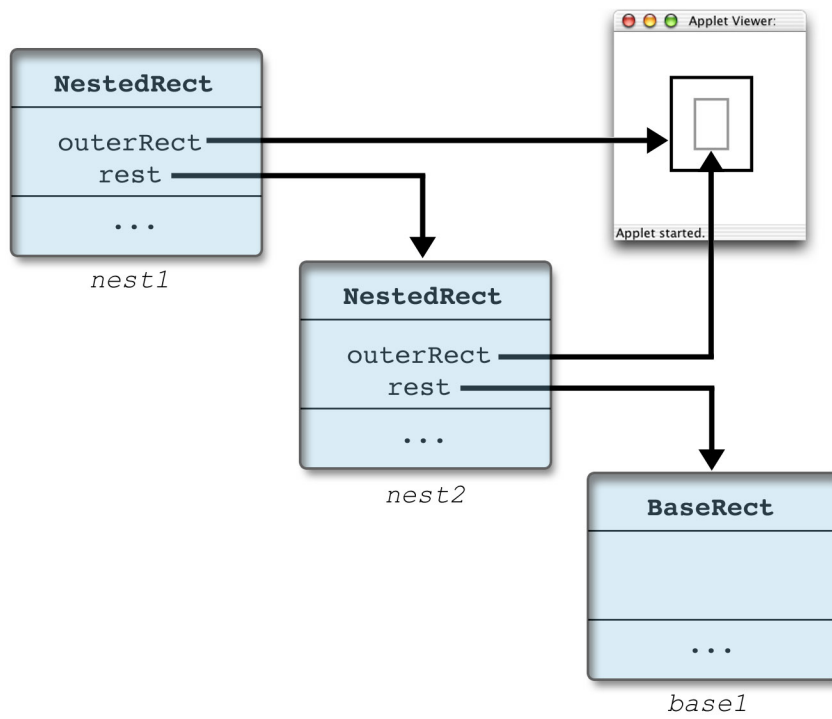


Figure 11.5: Objects constructed by evaluating `nestedRectsAt (54 @ 54) size (11 @ 13)` on (canvas).

If you trace the execution of the `moveBy (15, 20)` message on the second collection of nested rectangles you will see that its framed rectangle at (54, 54), `outerRect`, gets moved to (69, 74). Because its `rest` instance variable is a collection of nested rectangles at (58, 58), the message `moveBy (15, 20)` will be sent to `rest`.

As we saw when tracing the execution of the constructor, the collection of nested rectangles at (58, 58) has width three, which is less than eight. As a result, the value of its `rest` instance variable is `emptyRects`. Hence the evaluation of the message `moveBy (15, 20)` on that object will move the framed rectangle in its `outerRect` instance variable to (73, 78) and then will send `moveBy (15, 20)` to `rest`, which is just `emptyRects`. The result of sending `moveBy (15, 20)` to `emptyRects` is to do nothing. As a result, the method finishes execution, having moved all three framed rectangles.

From this example we see that the methods of `NestedRects` can have a structure which is based on the recursive structure of objects from the class. That is, they do what is necessary to `outerRect` and then, if there is more to do, they call the method recursively on `rest` (though possibly with slightly different parameters). Eventually this results in a method request to `emptyRects`, which just terminates execution of the method without doing anything more.

The method `removeFromCanvas` included in `NestedRects` is another example of this structure. It operates by removing `outerRect` from the canvas, and then requests `removeFromCanvas` on `rest`. If `rest` is `emptyRects` then it terminates without doing more. Otherwise it continues working, removing the `outerRects`.

Not all methods need to have this recursive structure. For example, the method `moveTo` ultimately just makes a request of its own `moveBy` method.

If we were to add a `contains` method for `nestedRects`, we would simply check to see if the `Point` parameter were contained in `outerRect`, as any point contained in the nested rectangles is contained in `outerRect`.

```
method contains (pt: Point) -> Boolean {
  outerRect.contains (pt) // pt is in nested rects if it is in the outermost one
}
```

Adding the method to `emptyRects` would be even simpler as the answer should always be false:

```
method contains (pt: Point) -> Boolean {
  false // there is nothing to be contained in
}
```

However, most methods will follow the recursive structure of the objects as the methods `moveBy` and `removeFromCanvas` from above.

Exercise 11.1.1

- a. Add methods `color:=` and `color` to both `emptyRects` and `nestedRects`. You may assume that `emptyRects` is always black.
- b. Write a program that draws a `nestedRects` object when the mouse button is pressed, drags it around the screen when the mouse is dragged, and removes it from the screen when the mouse button is released. Make the size of the outer rectangle be 75 by 47.

Exercise 11.1.2 *Trace the execution due to sending a method request of `removeFromCanvas` to the object `nest1` to make sure you understand how it moves all of the rectangles and then terminates.*

11.1.2 Building and Searching Data Collections with Recursive Structures

Many modern web browsers make it easy for you to type in the URL (universal resource locator) for a web page by offering automatic address completion. For example, if I start typing “http://www.c” my browser first suggests “http://www.cs.pomona.edu” as a completion. If that is the address that I want, then I press the return key to go to that address. On the other hand, if that is not the address I have in mind then I keep typing. For example, if I next type a “c”, my browser suggests “http://www.cccblog.org” as the completion. In order to do this the browser must keep a collection of URLs typed in by users, and then must be able to provide a sublist that includes all of the strings that match a given prefix.

We can model this collection using recursive structures. We will start with an empty list of strings. Then every time the user enters a new string (say in a `TextField`), we will construct a new list that consists of the existing list of strings plus the new one. We won’t bother to add another copy of a string that is already in the list. To avoid entering duplicates we will need a method, `contains`, that will allow us to check if a string is already in the list.

When the user starts typing a new URL, we will display (say in a `TextArea`) the existing URLs that match the portion of the URL typed so far. Thus we will need a method `getMatches` that returns a list of URLs that match. In order to convert a list of URLs into a `String` that can be displayed, we will also need a method `asString`.

The class will take care of building new lists out of old ones. Therefore the only methods that we will need are `contains`, `getMatches`, and `asString`. The type `URLList`, as well as the definition of object `emptyURLList` and class `nonEmptyURLList` are given in Figure 11.6.

Implementing an object representing an empty URL list will be straightforward, as nothing needs to be remembered for an empty list. The method `getMatches` should always return an empty URL list. Our code just returns `self`, as `self` is an empty URL list. We could also have it return `emptyURLList`, but those are just two different names of the same object. The method `contains` is always false, and `asString` always returns an empty string.

The class `nonEmptyURLList` in Figure 11.6 is more interesting. The class takes a new URL and an existing `URLList` and builds a new list by associating the new URL with the identifier `firstURL`, and the old list with identifier `rest`. This structure is very similar to the one we used with `NestedRects`. A trivial difference is that the first instance variable is called `firstURL` rather than `outerRect` and the types of these variables are different. However, these definitions play essentially the same roles in `NonemptyURLList` as the corresponding ones did in `NestedRects`.

A more significant difference between the constructors in these two classes is that the constructor in `nonEmptyURLList` is not recursive. That is, there is no evaluation of `nonEmptyURLList.adding () toList ()` inside the body of the class.¹ Instead the class takes a parameter of type `URLList` and builds a new more complicated `nonEmptyURLList` that includes the list passed in as well as the new URL passed in as the other parameter. This is fairly common when building new recursive structures out of existing structures, rather than building the entire structure

¹There is one inside the method `getMatches`, however, as we discuss later.

```

type URLList = {
  getMatches (prefix: String) -> URLList
  contains (url: String) -> Boolean
  asString -> String
}

// An empty collection of URLs
def emptyURLList: URLList = object {
  method getMatches (prefix: String) -> URLList {self}
  method contains (url:String) -> Boolean {false}
  method asString -> String {""}
}

// Create set of URLs by adding newURL to existing list
class nonEmptyURLList.adding (newURL: String) toList (existing: URLList) -> URLList {
  def firstURL: String = newURL
  def rest: URLList = existing

  // return list of URL's that begin with prefix
  method getMatches (prefix: String) -> URLList {
    if (firstURL.startsWith (prefix)) then {
      nonEmptyURLList.adding (firstURL) toList (rest.getMatches (prefix))
    } else {
      rest.getMatches (prefix)
    }
  }

  //returns true if list contains url
  method contains (url: String) -> Boolean {
    (firstURL == url) || rest.contains (url)
  }

  // returns a string representation of the list
  method asString -> String {
    firstURL ++ "\n" ++ rest.asString
  }
}

```

Figure 11.6: Recursive URL lists

at once (as we did with nested rectangles). If anything, it is easier to see that this kind of structure correctly builds a new list object with the new URL and the old list.

The methods of `nonEmptyURLList` are written in the same style as those of `nestedRects`. Let's start by looking at the `contains` method as it is one of the simplest. The `URLList` contains `url` if it is either the first element of the list or it is in the rest of the list. Thus the method returns true if `firstUrl == url` or `rest.contains(url)`.

While we could trace the execution of `contains` on a list with several strings, when trying to understand a recursive method, we are allowed to assume that the recursive calls of the method work correctly when trying to understand a call on a more complicated object. In this case, that means we can assume that the result of evaluating `rest.contains(url)` is **true** exactly when `url` is contained in the rest of the list. (We already saw that `contains` returns the correct answer – **false** – in the case of the empty list in class `emptyURLList`.) Hence the method will return true exactly when `url` equals `firstURL` or is contained in the rest of the list.

The method `asString` is very similar. The method concatenates the string in `firstUrl` with “\n” and then the result of converting `rest` into a string. If “\n” appears in a string, then when the string is printed, it is transformed into a “new line” character. That is, it forces the rest of the string to be printed on a new line.

If we assume `asString` works correctly on the smaller list in `rest`, then the result of the concatenation gives us a string representing all of the elements in the list, where consecutive elements will be printed on separate lines.

The `getMatches` method uses a helper method, `startsWith`, from `String`, that we have not seen before. The request `someString.startsWith (possiblePrefix)` returns true exactly when `possiblePrefix` is a prefix of `someString`. That is, it returns true when a sequence of letters at the start of `someString` corresponds exactly to the sequence of letters in `possiblePrefix`. We will discuss this and other `String` methods in more details in Chapter 15.

The `getMatches` method should return a `URLList` of all strings in the list that start with the value of the parameter **prefix**. Suppose `rest.getMatches (prefix)` returns a list of all the strings in `rest` that start with **prefix**. If the first element of the list, `firstUrl`, starts with **prefix**, then it should be added to the list to be returned, so we build a new list out of `firstUrl` and the list of matches from the rest of the list. Otherwise we simply return the set of matches from the rest of the list.

Exercise 11.1.3 *Check your understanding of what the methods of these classes do by determining the results of sending the messages `contains ("abcd")`, `asString`, and `getMatches ("abc")` to each of `w`, `x`, `y`, and `z` if they have been initialized as follows:*

```
w = emptyURLList
x = nonEmptyURLList.adding ("abcd") toList (w)
y = nonEmptyURLList.adding ("bca") toList (x)
z = nonEmptyURLList.adding ("abce") toList (y)
```

11.1.3 Designing Recursive Structures

We have now presented the design of two recursive structures, one representing a collection of nested rectangles and the other a list of URLs. The final recursive implementation of nested

rectangles and the implementation of the list of URLs were very similar. In each case we wrote two classes and a type, where the object and class expressions generated objects of that same type.

In each of these examples, one class was designed to represent an empty collection. In each case, the class for the empty collection of objects was not recursive. That is, it included no definitions or instance variable declarations with the same type. Similarly the methods and initialization code for empty objects were non-recursive. It was easy to verify that they did what they were supposed to – i.e., that they were correct.

In each example, a second class was designed to represent a non-empty collection. Each class included two instance variables. One represented one of the objects of the collection, with the other representing the rest. Thus these classes were recursive. The methods in these cases were also recursive. The initialization code for `nestedRects` was recursive, calling the class recursively to build a simpler collection, while the initialization code for `nonEmptyURLList` was not recursive, instead taking a simpler collection as a parameter.

Let's step back and see if we can extract from these two examples some principles that we can use to design other recursive structures:

- Recursive structures are built by defining objects (and sometimes classes) for base cases and classes for recursive cases. The objects constructed from these have the same type.
- An object represents a *base* case if it has no definitions or instance variable declarations whose type is the same as the object being defined. As we have seen, the base case is generally easy to write and verify.
- A class represents a *recursive* case if it has one or more instance variables whose type is the same as the object being defined. Designing the constructors and methods of a class representing a recursive structure requires a bit more care. The idea is that we must make sure that the constructors and methods always terminate (don't run forever), and convince ourselves that the initialization code and methods that we write work properly.

In our final recursive solution for nested rectangles the object `emptyRects` represents the base case, `nestedRects` represents the recursive case, and `Movable` is the interface that both implement. In the URL list example, the object `emptyURLList` represents the base case, `nonEmptyURLList` represents the recursive case, and `URLList` is the interface that both implement.

We write the classes and interface representing recursive structures as follows:

1. Define a type with all of the methods that both base case objects and recursive case classes must implement.
2. Define one or more objects (or classes) representing base cases. Make sure that all methods in the type are implemented. Convince yourself that the constructors and methods work correctly.
3. Define the initialization code for recursive classes. Recursive calls of the class should only create objects that are simpler than the one being built by the initialization code. Similarly, definitions and instance variables whose types are the same as the object being defined should only refer to objects simpler than the one being initialized. In

particular, the initialization of simpler objects in the class should eventually end up at a base case. Convince yourself that the initialization will be correct under the assumption that the initialization of simpler objects is correct.

4. Write each method under the assumption that it works correctly on all simpler objects. Convince yourself that the methods will be correct under the assumption that instance variables hold simpler objects.

Following the first two points in defining the type and base objects is generally very straightforward. The third point ensures that eventually the initialization will finish and that instance variables only hold simpler objects than the one being defined. If it does not hold, then the constructor will never terminate, and the computer will run out of memory.² This is used in the fourth point when reasoning about the correctness of methods.

We brushed over one key item in the instructions above: What does it mean to be simpler? This depends on the recursive structure. With a collection of nested rectangles, for example, the complexity of the collection could either be defined by the number of nested rectangles in the collection or by the width and height of the outer rectangle. In our case it will be most straightforward to say that one collection of nested rectangles is simpler than another if the width and height of the first are both smaller than the width and height of the second (though we could also have used the number of rectangles). We will say that an object from a base class like `emptyRects` is always simpler than an object from a recursive class like `NestedRects`.

For the list of strings, the easiest measure of complexity is the number of strings in the list. Thus an object from `emptyURLList` is always simpler than an object from `nonEmptyURLList`.

Let's examine the four steps as applied to the collection of nested rectangles. As expected, the definitions of the type `Movable` and `emptyRects` are straightforward, so points one and two are easily accomplished.

The third point has to do with the initialization in class `nestedRects`. If the width and height are at least eight when the initialization is performed, then the construction in the `if` statement results in the creation of a new object from class `nestedRects` that has smaller width and height, and hence is simpler. If either the width or height is less than eight then only the `emptyRects` object is created, and that is, by definition, simpler than the object of `nestedRects` being constructed. Thus it doesn't matter which branch of the `if-else` statement is executed; the object constructed and associated with `rest` is simpler than the full collection of nested rectangles.

Now let us worry about the correctness of the initialization code. First we create an outer framed rectangle with the same dimensions and location as the entire collection of nested rectangles. Next, if the width and height are both at least eight, the initialization code creates a new object using `nestedRects` that is four units to the right and below, and with width and height eight units smaller than the original. That object is simpler than the original one we are trying to create. As a result, we are allowed to assume that it really does create a collection of nested rectangles with the given dimensions and location. Under that assumption, the framed rectangle associated with `outerRect` will be drawn in just the right place to symmetrically surround the collection of nested rectangles.

²The error reported will be something like `StackOverflow error`, indicating that there is no more room on the "run-time stack" to create new objects.

On the other hand if either the width or height is less than eight, the initialization code creates the `emptyRects` object, which does not result in anything being drawn aside from the outer rectangle. Hence only a single framed rectangle is drawn at the given position, which is correct.

Thus, in either case, if we assume the initialization code in the simpler recursive calls of the class does what it is supposed to do, then the general case will work properly.

The fourth point has to do with writing and convincing yourself of the correctness of methods. According to that point, when attempting to argue that initialization code or a method is correct for an object, we are allowed to assume that it works correctly on all simpler structures.

Let's use that to investigate the design of the `moveBy` method. We've already determined that the `moveBy` method does the right thing (i.e., nothing) when we send a `moveBy` message to an object from the class `emptyRects`. Now we need to show that an object from the class `nestedRects` is moved correctly when a `moveBy` message is sent to it.

When a message of the form `moveBy (du,dv)` is sent to an object from the class `nestedRects`, first the framed rectangle associated with `outerRect` is moved by `(du,dv)`. Then the object associated with `rest` is sent the message `moveBy (du,dv)`. Because the value of `rest` is simpler than the whole object, we can assume that the method request to `rest` works correctly. In particular it moves the simpler collection of nested rectangles to a point four pixels to the right and below where the framed rectangle was sent, because it was moved by exactly the same amount as the outer framed rectangle. This results in the rectangles represented by `rest` being centered in the newly moved framed rectangle represented by `outerRect`. Hence the entire collection is moved correctly to the new location.

Similar arguments can be used to show that `removeFromCanvas` correctly removes all of the rectangles from the canvas.

11.1.4 Why Does this Work?

Reasoning according to the third and fourth points in the previous section is sufficient to ensure that the constructors and methods of a recursive class will work correctly. A proof of this could be given using mathematical induction. But since not all readers may have encountered mathematical induction yet, we will instead give a more intuitive argument for why this suffices.

Rather than giving an argument in general, let's give the justification for the specific case of the constructor for `nestedRects`. A similar justification works for other cases of recursive constructors and methods.

We will start from the smallest collections of `nestedRects` and work our way up. Let's consider evaluating `nestedRectsAt(pt)` size (width @ height) on (canvas) where the smallest of width and height is less than eight pixels, but both are greater than or equal to zero. Then the constructor will construct exactly one framed rectangle of the desired size at `pt`, and will construct `emptyRects`, which does not result in any drawing on the canvas. The recursive call of the class is not executed because the condition in the `if` statement is true. Clearly drawing the one framed rectangle is what should happen, so everything works fine when the smallest side is less than eight pixels.

Now suppose the smallest of the dimension components is less than sixteen pixels, but greater than or equal to eight pixels. Then when `nestedRectsAt (pt) size (dimension) on (canvas)` is called, a framed rectangle is drawn with the specified dimensions at `pt`. The condition on the `if` statement is now false, so the recursive construction of `nestedRectsAt (pt + (shift @ shift)) size (dimension - ((2 * shift) @ (2 * shift))) on (canvas)` is executed. Because the original smallest side was less than sixteen pixels, and the recursive call involves parameters for each dimension that are reduced by eight each, the smallest side for the recursive call is less than eight pixels. It is also drawn four pixels to the right and below the original call.

We know the call of the constructor is correct when the smallest side is less than eight pixels, so we know that the recursive call does the right thing – that is, it draws a single framed rectangle at `pt+(shift@shift)`. Because the original call resulted in drawing a framed rectangle which is eight pixels larger on each side, we know that the combination of drawing the framed rectangle and the recursive call result in drawing two nested rectangles, as desired.

Now suppose the smallest of the dimension components is less than twenty-four pixels, but greater than or equal to sixteen pixels. Examining the evaluation of `nestedRectsAt (pt) size (dimension) on (canvas)` we see that it draws a framed rectangle with the specified dimensions at `pt`, and then calls the constructor recursively with `nestedRectsAt (pt + (shift @ shift)) size (dimension - ((2 * shift) @ (2 * shift))) on (canvas)`. The recursive call now has smallest side less than sixteen pixels, but greater than or equal to eight pixels. However, we already know that a call of the constructor with smallest side of this size draws a collection of two nested rectangles correctly. Thus it is easy to convince ourselves by looking at where each of these is drawn that a call with smallest side less than twenty-four pixels, but greater than or equal to sixteen pixels, works correctly and draws three nested framed rectangles.

We could continue in this way as far as we like (*proof by exhaustion!*), but you should be able to recognize the pattern now. Each time we draw a new framed rectangle and rely on the fact that we already know that the recursive call of the constructor does what it is supposed to. This is exactly what our instructions told us to do in order to write and verify a recursive constructor or method.

Very similar arguments can be used to show that the `moveBy` and `removeFromCanvas` methods for `nestedRects` are correct. The rules for checking constructors and methods tell us exactly what must be verified in order to have confidence that our recursive constructors and methods will be correct.

Exercise 11.1.4 *What would go wrong if the initialization code for `nestedRects` resulted in a recursive call of `nestedRects` creating an object of type `Movable` with larger width and height as the value of the `rest` instance variable?*

Exercise 11.1.5 *Apply the four points above to the design of the classes and the interface for the list of URLs.*

Exercise 11.1.6 *Suppose we made a mistake in the definition of the `moveBy` method in `nestedRects` by leaving out the statement `outerRect.moveBy (dx, dy)`. What would go wrong? Show where the correctness argument for that method would break down.*

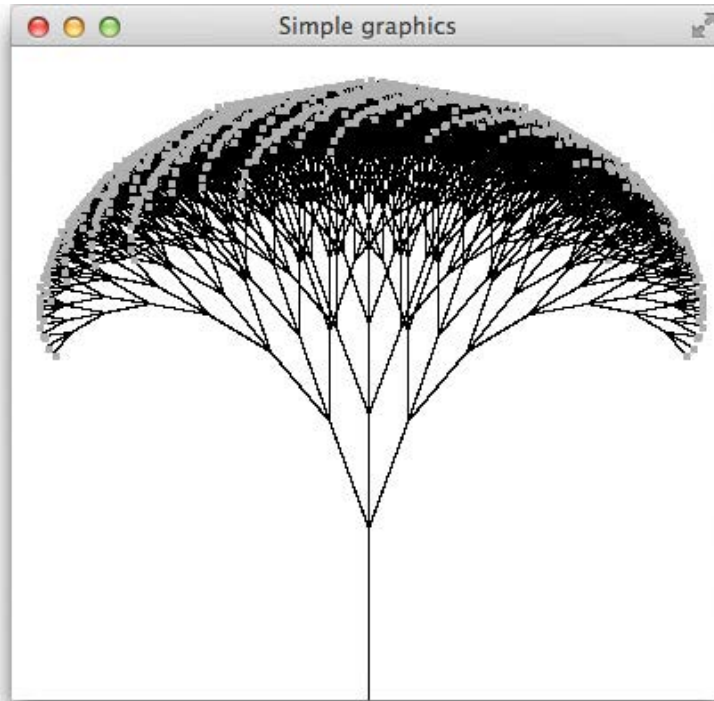


Figure 11.7: A broccoli plant

Exercise 11.1.7 *Suppose we made a mistake in the definition of the method `moveBy` in `nestedRects` by writing `rest.moveBy(x+4,y+4)` rather than `rest.moveTo(x+4,y+4)`. What would go wrong? Show where the correctness argument for that method would break down.*

11.1.5 Broccoli

Recursion can be used to create other interesting pictures. A picture of broccoli is shown in Figure 11.7. If you examine the picture carefully, you will notice the recursive structure of the plant. At the base is a stem that is about one-fourth of the height of the plant. At the upper end, the stem branches off three ways – one to the left, one straight up, and one to the right. If we look at what grows out of the upper end of the stem, we see that those structures themselves look exactly like smaller broccoli plants. We will take advantage of this structure by drawing broccoli as a stem with three smaller broccoli plants attached to the ends of the stem. Thus it can be represented as a recursive structure.

We draw broccoli by following the approach in the previous section. We will define a class `flower` to handle the base case of drawing the flowering end of broccoli, while `broccoli` will handle the recursive case. As in our previous recursive designs, we will define an interface, `Movable`, that both `flower` and `broccoli` must implement. We will keep track of the size of the broccoli by keeping track of the size of the stem. This will help us decide whether we are in the base case where we construct a flower or in the recursive case where we construct a full broccoli plant.


```

type Movable = {
    moveTo (pt: Point) -> Done
    moveBy (dx: Number, dy: Number) -> Done
    contains (pt: Point) -> Boolean
}

```

Figure 11.8: Movable type

If the stem is more than `minSize` pixels long then we will attach three smaller broccoli plants to the end of the stem. Each of these will have a stem which is 80% of the size of the one just drawn. Otherwise, if the stem is at most `minSize` pixels long then we will attach three flowers to the end of the stem. Thus the base case occurs when the stem length is at most `minSize` pixels and the recursive case when it is larger than `minSize` pixels.

In order to draw broccoli, we will need to draw lines at a variety of angles. While the first stem will be drawn vertically, successive stems will be drawn at other angles (in particular, the left-most and right-most branches will be drawn at angles of $\pi/9$ radians (or 20°) from that of the base stem. While we have only used one so far, the `Line` class in the `objectdraw` library has two ways of constructing objects of type `Line`. The one we have already been using, `lineFrom(to)on()` takes end points of a line and the canvas it will be drawn on. However, there is a second one, `lineFrom(length () direction () on ()` that takes as parameters the starting point of the line, its length, angle from the positive x-axis (measuring counterclockwise). This constructor is what we use here.

All we will require of our broccoli is that we can drag it around the screen. Thus, we need it to support `moveBy`, `moveTo`, and `contains` methods. The rather trivial definition of the type `Movable` is given in Figure 11.8. This completes step 1 in the definition of a recursive structure.

The base case, the `flower` class, is shown in Figure 11.9. The constructor for `flower` takes as parameters the starting location `startPt`, the length of the stem, size, the angle that the stem makes with the x-axis, direction, and the canvas that it will be drawn on. An instance of `Flower` is composed of an `Line`, `stem`, which is colored green, connected to a `FilledOval`, `bud`, which is colored yellow. The method `end of line` makes it easy to find the end of the stem in order to draw the bud there (with a bit of an offset to get the bud centered on the end of the stem). The `moveBy` method for `Flower` is straightforward, as it simply moves the `stem` and `bud` the appropriate distance. The `contains` method simply returns whether `stem` or `bud` contains point. The method `moveTo`, as always, depends on using `moveBy`. This completes step 2 of our guidelines.

Figure 11.10 contains the code for the recursive `broccoli` class. The constructor takes the same parameters as `flower`. It draws the stem and then, depending on `stemSize`, adds either three `Broccolis` or three `Flowers` to the end of the stem. The instance variables, `left`, `center`, and `right`, all have type `Movable`, so they can hold values from either the `broccoli` or `flower` classes. The objects associated with `center` (whether `flower` or `broccoli`) will face the same direction as the stem, while the the objects associated with `left` and `right` are tipped at angles of $\pi/9$ or 20° to the left and right of the stem.

If we start with a stem length of greater than `minSize` pixels in the constructor, then after creating new objects of class `broccoli` with the stem length reduced to 80% of its value over

```

// color of the broccoli stems
def broccoliColor: Color = colorGen.r (0) g (135) b (0)

// construct a yellow flower at the end of a stem starting at startPt whose length is stemSize
// and is at angle radians
class flowerAt (startPt: Point) size (stemSize: Number) direction (radians: Number)
  on (canvas: DrawingCanvas) -> Movable {
  def budSize: Number = 5

  // create stem of the flower at startPt of length stemSize at angle radians.
  def stem: Line = lineFrom (startPt) length (stemSize) direction (radians) on (canvas)
  stem.color := broccoliColor
  def endPt: Point = stem.end

  // create a yellow bud on the flower
  def bud: Graphic2D = filledOvalAt (endPt + ((-budSize / 2) @ (-budSize / 2)))
    size (budSize @ budSize) on (canvas)
  bud.color := colorGen.yellow

  // move the flower dx to the right and dy down
  method moveBy (dx: Number, dy: Number) -> Done {
    stem.moveBy (dx, dy)
    bud.moveBy (dx, dy)
  }

  // move bottom of stem to pt
  method moveTo (pt: Point) -> Done {
    def diff: Point = pt - stem.start
    moveBy(diff.x, diff.y)
  }

  // is pt in the stem or bud of the flower
  method contains (pt: Point) -> Boolean {
    stem.contains (pt) || bud.contains (pt)
  }
}

```

Figure 11.9: flower class implementing Movable

```

class broccoliAt (startPt: Point) size (stemSize: Number) direction (radians: Number)
    on (canvas: DrawingCanvas) -> Movable {
    def minSize: Number = 40      // below this size draw flower
    def shrinkage: Number = 0.8   // fraction to shrink when drawing successive items

    // draw stem of broccoli
    def stem: Line = lineFrom (startPt) length (stemSize) direction (radians) on (canvas)
    stem.color := broccoliColor

    // end of stem: where new pieces get added
    def endPt: Point = stem.end

    // new pieces of broccoli to go at end of the stem
    var left: Movable
    var right: Movable
    var center: Movable

    if (stemSize > minSize) then {
        left := broccoliAt (endPt) size (stemSize * shrinkage) direction (radians + math.pi / 9) on (canvas)
        center := broccoliAt (endPt) size (stemSize * shrinkage) direction (radians) on (canvas)
        right := broccoliAt (endPt) size (stemSize * shrinkage) direction (radians - math.pi / 9) on (canvas)
    } else {
        left := flowerAt (endPt) size (stemSize * shrinkage) direction (radians + math.pi / 9) on (canvas)
        center := flowerAt (endPt) size (stemSize * shrinkage) direction (radians) on (canvas)
        right := flowerAt (endPt) size (stemSize * shrinkage) direction (radians - math.pi / 9) on (canvas)
    }

    method moveBy (dx: Number, dy: Number) -> Done {
        stem.moveBy (dx, dy)
        left.moveBy (dx, dy)
        center.moveBy (dx, dy)
        right.moveBy (dx, dy)
    }

    method moveTo (pt: Point) -> Done {
        moveBy ((pt - startPt).x, (pt - startPt).y)
    }

    method contains (point: Point) -> Boolean {
        stem.contains (point) || left.contains (point) ||
        center.contains (point) || right.contains (point)
    }
}

```

Figure 11.10: broccoli class

and over, we will eventually get a stem length which is `minSize` pixels or smaller. Thus each invocation of the constructor creates a simpler structure (*i.e.*, with shorter stem length), and these simpler structures will eventually terminate with the construction of a flower.

Now we must verify that the constructor does the right thing if we assume that all simpler calls of the constructor do the right thing. In this case, it is easy to see that the constructor does the right thing, as it is easy to see from the picture that a non-trivial broccoli consists of a stem with three broccoli objects or flowers attached to the end of the stem. Thus we have satisfied the third point for the definition of recursive structures.

Now let us take a look at the `moveBy` method for broccoli. To move broccoli we just move the stem and the left, center, and right parts. Because the left, center, and right pieces are simpler than the entire broccoli, we are allowed to assume the recursive invocations of `moveBy` to the instance variables work correctly. Under this assumption, it is easy to see that the complete method works correctly: the stem is moved the appropriate amount and each of the three broccoli parts are moved the same amount. Verifying the `contains` method is similar, while `moveTo` is defined in terms of `moveBy` and may be verified under the assumption that `moveBy` is correct.

We have now satisfied all four conditions for the definition of a recursive structure, so we can have confidence that our classes and interface support the creation and manipulation of broccoli.

Figure 11.11 contains the code for a program `BroccoliDrag` that extends `WindowController`. It creates a broccoli and allows the user to drag it around the screen. . It assumes that the definitions of type `Movable` and classes `flower` and `broccoli` are in file “`broccoli.grace`”.

Exercise 11.1.8 *Add new method `removeFromCanvas` to the `broccoli` and `flower` classes.*

Exercise 11.1.9 *If you were to run the broccoli test program, you would discover that it takes a while to respond to a mouse drag, and the motion is somewhat jerky. To understand why this is so, calculate the number of lines that are included in the broccoli created in `BroccoliDraw`. You may find it easiest to do this by first counting lines for a broccoli whose stem length is at most $\text{minSize} = 25$, then, if shrinkage is 80%, moving up to at most 31.25 (25 is 80% of 31.25), 39, 48.8, 61, 76, and finally 95.*

Exercise 11.1.10 *To make the Broccoli drawing more interesting, animate its growth by drawing it slowly using the `animation` module. It should draw the stem and then pause for 500 milliseconds before creating the values of the three instance variables that correspond to the three branches.*

Exercise 11.1.11 *The first class we designed for drawing nested rectangles used a loop in the constructor to create the framed rectangles. One might imagine that one could do something similar for drawing broccoli. What is it about broccoli that makes it hard to draw it using a while loop and no recursive invocations of the constructor.*

11.2 Recursive Methods

In this section we develop recursive methods that are not part of recursive structures. Instead we write recursive methods where the complexity is controlled by an integer parameter. As

```

dialect "objectdraw"
import "broccoli" as br
import "math" as math

object {
  inherit graphicApplicationSize (600 @ 600)

  def broc: br.Movable = br.broccoliAt (300 @ 600) size (100) direction (math.pi / 2) on (canvas)

  var dragging: Boolean := false
  var lastPoint: Point
  method onMousePress(pt: Point) -> Done {
    dragging := broc.contains (pt)
    lastPoint := pt
  }

  method onMouseDrag (pt: Point) -> Done {
    if (dragging) then {
      broc.moveBy ((pt - lastPoint).x, (pt - lastPoint).y)
      lastPoint := pt
    }
  }

  startGraphics
}

```

Figure 11.11: Program to drag broccoli

before, we will ensure that recursive invocations are simpler, though in this case that will be done by making recursive invocations with smaller, but non-negative, integer parameters.

Recursive methods differ from the methods in classes supporting recursive structures in that the methods must include at least one base case in which there are no recursive invocations of the method. These base *cases* replace the base *objects* and *classes* that we needed to support recursive structures. Recursive methods typically contain a conditional statement in which at least one of the cases is a base case and at least one involves recursive invocations.

After we present the first example we will provide a slightly different set of points to write and verify recursive methods where the recursion is based on parameter values rather than recursive structures.

We present two examples of problems in the rest of this section. The first example involves a fast algorithm for raising a number to a non-negative integer power, while the second involves the solution to an interesting puzzle.

11.2.1 Fast Exponentiation

Fast algorithms for raising large numbers to large integer powers are important to the RSA algorithm for public key cryptography that is behind a large number of the secure sites on the web. We won't discuss cryptography here, but we will investigate a recursive algorithm that is substantially faster than the usual way of raising numbers to powers.

As a warm-up, we describe a simple recursive method to raise an integer to a non-negative power that is no more (or less) efficient than the obvious iterative algorithm. As usual, the basic idea is to describe how you would complete the solution to the problem if there are solutions to simpler problems. In this case, if we want to raise a number to the *n*th power, we presume the algorithm for raising the number to the *n*-1st power works correctly.

// returns base raised to exponent power as long as exponent is non-negative integer

```
method simpleRecPower (base: Number, exponent: Number) -> Number {
  if (exponent == 0) then {
    1
  } else {
    base * simpleRecPower (base, exponent - 1)
  }
}
```

Notice that the method includes a conditional statement which has both a base case and a recursive case. A simpler case is one with a smaller value of *exponent*.

The following are the rules for writing recursive methods:

1. Write the “base case”. This is the case where there is no recursive call. Convince yourself that this works correctly.
2. Write the “recursive case”.
 - Make sure all recursive calls go to simpler cases than the one you are writing. Make sure that the simpler cases will eventually get to a base case.
 - Make sure that the general case will work properly if all of the recursive calls work properly.

Let's apply these rules to `simpleRecPower`:

1. The base case is where `exponent == 0`. It returns 1, which is the correct answer for raising `base` to the 0th power. It is the base case because there is no recursive invocation of `simpleRecPower`.
2. The recursive case uses the `else` clause.
 - The recursive call is to `simpleRecPower(base, exponent - 1)`. It involves a smaller value for the second argument. Because the value of `exponent` is greater than 0 (*why?*), and the `exponent` always goes down by one, the recursive calls will eventually get down to the base case of 0.
 - Assume that evaluating `simpleRecPower(base, exponent - 1)` results in $\text{base}^{(\text{exponent} - 1)}$, the `else` clause returns

$$\begin{aligned} \text{base} * \text{simpleRecPower}(\text{base}, \text{exponent} - 1) &= \text{base} * \text{base}^{(\text{exponent} - 1)} \\ &= \text{base}^{\text{exponent}}. \end{aligned}$$

Thus we can be confident that the above algorithm calculates $\text{base}^{\text{exponent}}$. It is also easy to see that the evaluation of `simpleRecPower(base,n)` results in exactly `n` multiplications, because there is exactly one multiplication associated with each recursive call.

Using a simple modification of the above recursive program, we can get a much more efficient algorithm for calculating very large powers of integers. In particular, if we use the above program (or the equivalent simple iterative program) it will take 1024 multiplications to calculate b^{1024} , for any integer `b`, while the program we are about to present cuts this down to only 11 multiplications!

The algorithm above takes advantage of the following simple rules of exponents:

- $\text{base}^0 = 1$
- $\text{base}^{\text{exp}+1} = \text{base} * \text{base}^{\text{exp}}$

The new algorithm that we present below takes advantage of one more rule of exponents:

- $\text{base}^{m*n} = (\text{base}^m)^n$

In the program we use this rule where `m` is 2 and `n` is `exponent / 2`.

The key is that by using this rule as often as possible, we can cut down the amount of work considerably, by reducing the size of the exponents in recursive calls faster. See the code below:

// returns base raised to exponent power as long as exponent is non-negative integer

```
method fastRecPower (base: Number, exponent: Number) -> Number {
  if (exponent == 0) then {
    1
  } elseif {(exponent % 2) == 1} then {
    base * fastRecPower (base, exponent - 1)
  } else {
    fastRecPower (base * base, exponent / 2)
  }
}
```

The `fastRecPower` method performs exactly the same computation as the previous one when the exponent is 0 or an odd integer (i.e., when `exponent % 2` is equal to 1). However, it works very differently when the exponent is even. In that case, it squares the base and divides the exponent in half.

Before analyzing exactly why this method works, let's first look at an example of the use of this algorithm and count the number of multiplications.

```

fastRecPower (3, 16) = fastRecPower (9,8)           // mult 3 * 3 -> 9
                    = fastRecPower (81, 4)         // mult 9 * 9 -> 81
                    = fastRecPower (6561, 2)       // mult 81 * 81 -> 6561
                    = fastRecPower (43046721, 1)   // mult 6561 * 6561 -> 43046721
                    = 43046721 * fastRecPower (43046721, 0)
                    = 43046721 * 1                 // mult by 1
                    = 43046721

```

The computation only took five multiplications using `fastRecPower`, whereas it would have taken 16 multiplications the other way. Division by two can be done as easily in binary as divisions by 10 can be done with numbers in decimal notation. Thus we will not bother to count division by 2 or using the “%” operation with 2 as worth worrying about in terms of the time complexity of the algorithm.

In general it takes somewhere between $\log_2(\text{exponent}) + 1$ (for exponents that are exact powers of 2) and $2 * \log_2(\text{exponent})$ multiplications to compute a power this way. While the difference between $2 * \log_2(\text{exponent})$ and `exponent` is not great for small values of `exponent`, it does make quite a difference when `exponent` is large. For example, as noted above, computing `fastRecPower(b, 1024)` would only take 11 multiplications, while computing it the other way would take 1024 multiplications.

Let's once again use the standard rules for understanding recursion to see why this algorithm is correct.

1. The base case is again where `exponent == 0`. It returns 1, which is the correct answer for raising `base` to the 0th power.
2. The recursive case is in the `else` clause, but this time it divides into two cases, depending on whether `exponent` is odd or even.
 - When `exponent` is odd, the recursive call is with `exponent - 1`, while, when `exponent` is even, the recursive call is with `exponent / 2`. In either case, each of these calls is for a smaller integer power because `exponent` is greater than zero. Thus no matter which case is selected at each call, eventually the power will decrease to the base case, 0.
 - We have already given the correctness argument for the code in the odd case in our earlier analysis of the `simpleRecPower` method. Let's convince ourselves that it also is correct for the even case.

Let `exponent` be an even positive integer. Then the method returns the value of
`fastRecPower (base * base, exponent / 2)`

That is simpler than the original call because `exponent/2` is less than `exponent`.

Thus we can assume that it returns the correct answer. However,

$$\begin{aligned}(\text{base} * \text{base})^{\text{exponent}/2} &= (\text{base}^2)^{\text{exponent}/2} \\ &= \text{base}^{2 * (\text{exponent}/2)} \\ &= \text{base}^{\text{exponent}}\end{aligned}$$

Thus the result of `fastRecPower (base * base, exponent / 2)` is the correct answer.

*Note that we are only proving this algorithm for integer values of exponent. If we tried to calculate `fastRecPower (base * base, exponent / 2)` when exponent is odd, the value of the second argument, `exponent / 2` would be a fraction rather than an integer and the call would not give the right answer. In fact, it wouldn't even terminate. (Look at what would happen if you evaluated `fastRecPower (3, 0.5)`.)*

Thus we see that the `fastRecPower` algorithm is correct when the second argument is a non-negative integer.

While this algorithm can be rewritten in an iterative style, the recursive algorithm makes it clear where the rules of exponents are coming into play in the algorithm.

Exercise 11.2.1 *Both of the above algorithms can be rewritten iteratively.*

- a. *Rewrite the `simpleRecPower` algorithm with a loop.*
- b. *Write the iterative equivalent of the `fastRecPower` algorithm. Do you find the iterative or recursive version easier to understand?*

11.2.2 Towers of Hanoi

An amusing use of recursion comes up in the solution of the Towers of Hanoi puzzle. The puzzle is based on a story about a group of Buddhist monks in the Tower of Brahma. The monastery contains a large table in which are embedded 3 diamond-tipped needles. There are also 64 golden disks, each with a different radius. Each disk has a hole in the center so that it can be placed on one of the needles. At the start, the 64 golden disks are placed on the first diamond-tipped needle, arranged in order of size, with the largest on the bottom.

The monks are supposed to move the 64 golden disks from the first to the third golden-tipped needle. This sounds easy, but there are restrictions on how disks can be moved. First, only one disk at a time can be moved from one needle to another. Second, it is strictly forbidden to put a large disk on top of a smaller one. As we shall see, there is a rather simple recursive strategy for solving this puzzle, though we shall also see that completing this task will take quite a bit of work.

One can buy a children's puzzle based on this story, though these puzzles are typically made of plastic or wood, and only come with 8 or fewer disks (for reasons that will become apparent later). See the picture in Figure 11.12.

The key to solving the puzzle is to consider how you could move the biggest disk from the bottom of the first needle to the bottom of the third needle, it's final destination. Then think recursively!

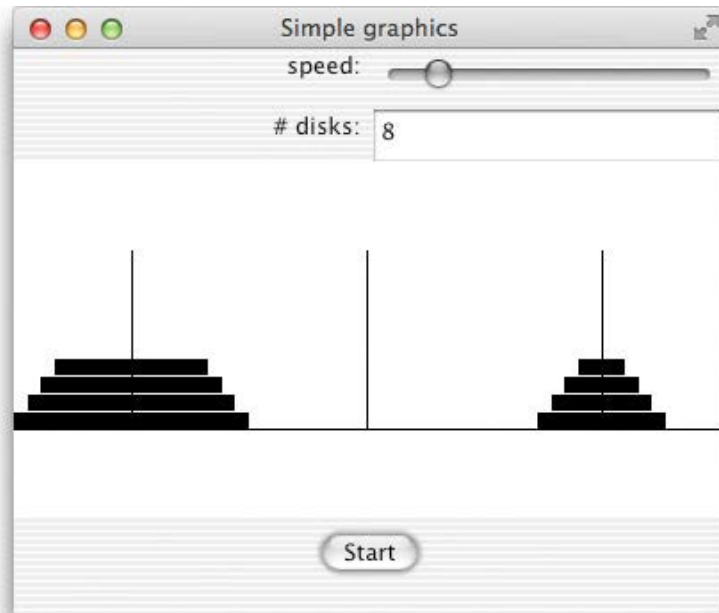


Figure 11.12: Towers of Hanoi puzzle with 8 disks.

A little thought should convince you that to move the biggest disk from the first to last needle, you must first move all of the smaller disks to the middle needle in order to get the smaller disks out of the way. If any of the smaller disks are left on the first needle then the biggest disk cannot be moved. Similarly, if any of the smaller disks are on the third needle, then we could not possibly move the biggest disk onto that needle, as that would be an illegal move.

Hence, all but the largest disk must be moved to the middle needle. Then the biggest disk must be moved to the third needle. Finally the remaining $n-1$ smaller disks must be moved from the middle needle to the last needle. We can write down this procedure more carefully as follows:

To move n disks from the start needle to the target needle using a helper needle:

1. If there is only one disk to move, just move it from the start to target needle, and you are done.
2. Otherwise move the top $n-1$ disks from the start needle to the helper needle (following the rules, of course).
3. Then move the bottom disk from the start to the target needle.
4. Then move the top $n-1$ disks from the helper to the target needle (following the rules, of course).

A method to do this in Grace can be found in Figure 11.13.

dialect "objectdraw"

```
method solveHanoi (numDisks: Number) from (first: String) to (last: String) using (helper: String) {
  if (numDisks == 1) then {
    moveDisk (1) from (first) to (last)
  } else {
    solveHanoi (numDisks - 1) from (first) to (helper) using (last)
    moveDisk (numDisks) from (first) to (last)
    solveHanoi(numDisks - 1) from (helper) to (last) using (first)
  }
}

method moveDisk (n: Number) from (current: String) to (destination: String) {
  print "Moved disk {n} from {current} to {destination}"
}
```

Figure 11.13: Solving tower of Hanoi puzzle

We can understand this method a little better by looking at a specific example where we call `solveHanoi` with 3 disks, where we wish to move the 3 disks from needle A to needle C while using needle B as a helper needle. (For simplicity, we'll leave off the double quotes around the strings A, B, and C that label the needles.)

$solveHanoi(3) \text{ from } (A) \text{ to } (C) \text{ using } (B) \Rightarrow$

$$\left\{ \begin{array}{l} solveHanoi(2,A,B,C) \Rightarrow \left\{ \begin{array}{l} solveHanoi(1)from(A)to(C)using(B) \Rightarrow moveDisk(1)from(A)to(C) \\ moveDisk(2)from(A)to(B) \\ solveHanoi(1)from(C)to(B)using(A) \Rightarrow moveDisk(1)from(C)to(B) \end{array} \right. \\ moveDisk(3)from(A)to(C) \\ solveHanoi(2)from(B)to(C)using(A) \Rightarrow \left\{ \begin{array}{l} solveHanoi(1)from(B)to(A)using(C) \Rightarrow moveDisk(1)from(B)to(A) \\ moveDisk(2)from(B)to(C) \\ solveHanoi(1)from(A)to(C)using(B) \Rightarrow moveDisk(1)from(A)to(C) \end{array} \right. \end{array} \right.$$

That is, a call to `solveHanoi(3,A,C,B)` results in calls to `solveHanoi(2,A,B,C)`, `moveDisk(3,A,C)`, and `solveHanoi(2,B,C,A)`. Each of these recursive calls of method `solveHanoi` gets expanded. For example the call to `solveHanoi(2,A,B,C)` results in calls to `solveHanoi(1,A,C,B)`, `moveDisk(2,A,B)`, and `solveHanoi(1,C,B,A)`. If we look at the order in which the `moveDisk` methods are executed, we see that the order of moves is:

- moveDisk(1) from (A) to (C)
- moveDisk(2) from (A) to (B)
- moveDisk(1) from (C) to (B)
- moveDisk(3) from (A) to (C)
- moveDisk(1) from (B) to (A)
- moveDisk(2) from (B) to (C)
- moveDisk(1) from (A) to (C)

This sequence of moves results in legally moving all 3 disks from needle A to needle C.

How do we know this method will work properly? Once more, we go back to our standard check list.

1. The base case of a single disk just moves it where it is supposed to go, and hence is correct.
2. The recursive case:
 - If you start with a positive number of disks, then the recursive calls to `solveHanoi` will be with one fewer disk. Hence recursive calls will eventually get down to 1 disk, the base case.
 - If we assume that the method works for $n-1$ disks, then it will work for n disks. (*Can you give a convincing argument?*)

Exercise 11.2.2 *Enhance the code for Hanoi presented above:*

- a. *We have not shown the code for the method `moveDisk`. It could either just print out a message (using `print`) describing which disk is moved from where to where, or it could result in altering a picture of needles and disks. Write out the code for the text-only version of `moveDisk`.*
- b. *(The following program requires arrays, covered in chapter 13) Write the code for an animated graphic version of Towers of Hanoi where there is a delay between each move of a disk. That is, let an `ActiveObject` control the animation of the algorithm. Interestingly, the code for the graphics is several times as long as the actual code for determining which disk should be moved next.*

Exercise 11.2.3 *Determine how many calls of `moveDisk` are required to run `solveHanoi(n,A,B,C)` to completion. Hint: first make a table of the number of moves for n ranging from 1 to 10. From the table, guess a formula involving n for the number of calls of `moveDisk`. Use an inductive argument similar to that given for the correctness of recursive programs to give a convincing argument that your formula is correct. If a robotic arm could move 1 disk per second, how long would it take to move all 64 disks from the start needle to the target needle using this algorithm. Do you now understand why the commercial version of the game only includes 8 disks?*

Exercise 11.2.4 *The recursive algorithm given in the method is the most efficient solution in terms of the number of disks moved. Try to find a convincing argument for this. Hint: Think about what has to be the configuration in order to move the biggest disk.*

11.3 Summary

In this chapter we introduced problem solving using recursion. In the first part of the chapter we described how to create recursive structures, classes that have instance variables of the same type as the entire class. Examples included collections of nested rectangles, lists of strings, and a broccoli plant. We presented the following set of rules for designing recursive structures and ensuring that the constructors and methods did what they were supposed to:

1. Define a type with all of the methods that both base case and recursive case classes must implement.
2. Define one or more objects or classes representing base cases. Make sure that all methods from the type are implemented. Convince yourself that the constructors and methods work correctly.
3. Define the constructors for recursive classes. Recursive calls of the class should only create objects that are simpler than the one being built. Similarly, instance variables whose types are the same as the object being constructed should only refer to objects simpler than the one being constructed. In particular, the construction of simpler objects in the class or object body should eventually end up at a base case. Convince yourself that the constructor will be correct under the assumption that the construction of simpler objects is correct.
4. Write each method under the assumption that it works correctly on all simpler objects. Convince yourself that the methods will be correct under the assumption that instance variables hold simpler objects.

In the second part of the chapter we looked at examples of recursive methods where the complexity was measured by an integer parameter. Examples included methods for raising numbers to powers and a solution to the Towers of Hanoi problem. A slightly different set of rules was presented to help in writing and ensuring correctness of these recursive methods:

1. Write the “base case”. This is the case where there is no recursive call. Convince yourself that this works correctly.
2. Write the “recursive case”.
 - Make sure all recursive calls go to simpler cases than the one you are writing. Make sure that the simpler cases will eventually get to a base case.
 - Make sure that the general case will work properly if all of the recursive calls work properly.

Recursive structures and methods are extremely important in computer science. We will encounter them again when we discuss algorithms for searching and sorting in Chapter 19. They also play a very important role in the study of data structures and algorithms in computer science.

11.4 Review Problems

Exercise 11.4.1 *What is wrong with the following proof by induction that in any group, all people are the same height:*

Base case: *If the group only has 1 person in it, then clearly every person in the group is the same height.*

Induction case: Suppose that in every group with n people in it, all of the people are of the same height. Show the same is true for all groups with $n + 1$ people. Let G be a group of $n + 1$ people. Remove 1 person, P , from the group. Let G' denote the group still remaining. By the induction hypothesis, because G' is now a group of n people, all of the people that remain in the group are of the same height. Put person P back in the group and remove a different person, P' . Let G'' denote the new group still remaining. Because there are n people in G'' , by induction all of them are the same height. Thus P is the same height as everyone else in that group. Meanwhile, we already determined that P' was the same height as everyone else in G' . Thus P and P' are the same height and everyone in the original group G is the same height.

11.5 Programming Problems

Exercise 11.5.1

- a. Define two classes `target` and `bullseye` that can be used to construct and drag around a target. The classes should take as parameters the center of the circle, the radius, and a `DrawingCanvas`. The classes should support `moveBy`, `moveTo`, `contains`, and `removeFromCanvas` methods. If the radius is less than 5 pixels, just draw a `FilledOval` as the bullseye. If the radius is greater, then draw a circle and then inside of it draw a target whose radius is 4 pixels smaller.
- b. Write code to test the `target` class so that when the user clicks on the canvas, it should draw a target with a radius of 18 pixels that is centered at the place the user clicked.

Exercise 11.5.2 Parsley is another plant that can be drawn recursively. Write a program to draw parsley as shown in Figure 11.14. New parsley branches are drawn at distances of $1/3$ and $2/3$ of the length of the stem, and at its end. The lowest branch is $3/4$ of the size of the main branch and is drawn at an angle of $\pi/6$ radians from that of the main branch. The next branch is drawn at $2/3$ of the size of the main branch and is drawn at an angle of $-\pi/5$ to the main. The top-most branch is $1/2$ of the size of the original and is at an angle of $\pi/4$ from the original.

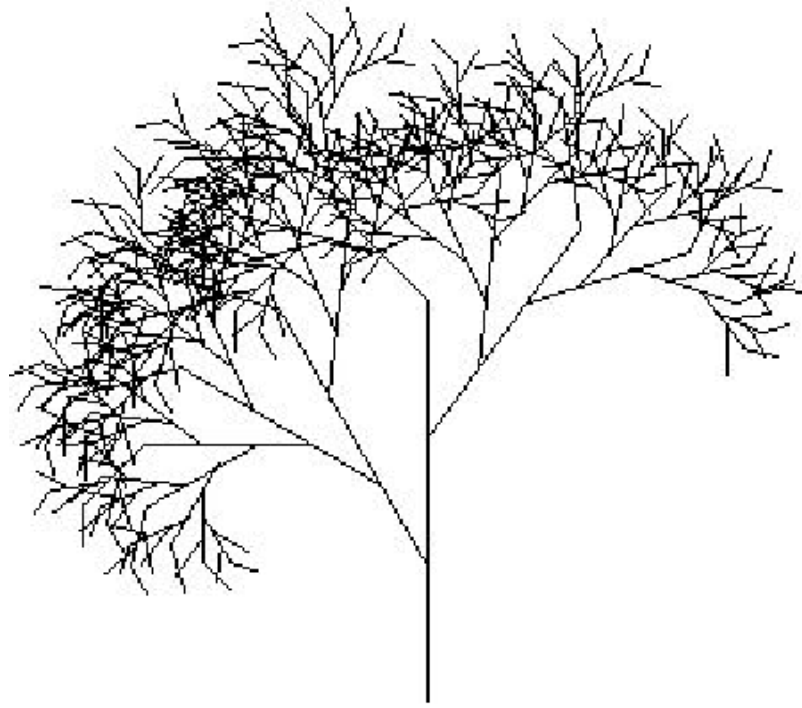


Figure 11.14: Parsley

Chapter 12

General Loops in Grace

In this chapter, we will take another look at loops in Grace. We will begin by identifying some patterns in the use of loops in examples we have already presented. Then we will introduce a new kind of loop, the `for` loop, that is useful in circumstances in which you know or can calculate in advance the number of times that the loop should be executed. We next discuss a minor variant of the `while` loop in which the exit test for the loop is placed at the end of the loop body rather than at the beginning. Finally, we will discuss common errors associated with loops and provide guidance on when to use each kind of loop.

12.1 Recognizing Patterns with Loops

A very important part of programming is recognizing patterns. Suppose you recognize a pattern of code usage and the context in which it is useful. Then when you recognize that you have a similar context, you can go back and reuse the pattern. For example, here is a pattern that you have seen many times:

```
method onMousePress (point: Point) -> Done {
  lastPoint = point
}
```

```
method onMouseDrag (point: Point) -> Done {
  if (dragging) then {
    // do something with point and lastPoint
    lastPoint := point
  }
}
```

We used this pattern when we were dragging things around (in that case the “do something” involved moving an object by the difference between `point` and `lastPoint`). We saw a similar pattern when we were scribbling on the screen. In that case we drew a line between `lastPoint` and `point`, but did not need the `if` statement.

Here is a very similar type of pattern that you have also seen:

```
lastTime := system.elapsed
while {...} pausing (delay) do {
  ...
  elapsedTime := lastTime - system.elapsed
}
```

```

screenObj.move (elapsedTime * xspeed, elapsedTime * yspeed)
...
lastTime := system.elapsed
}

```

This pattern showed up with animations when we wanted to make sure that we moved each item a distance proportional to the amount of time since it was last moved.

Actually, these two patterns are really the same. They involve remembering a value before a change happens, then doing something that involves both the new and old values after the change happens, and then remembering the current value so it can be used after the next change occurs. The main difference between the two examples is that in the first example the initialization takes place when the mouse is pressed, while the updates take place in repeated calls to `onMouseDown`. In the second example, by contrast, the initialization takes place just before entering the loop, while the updates all take place within the loop.

This notion of having to set a value before repeatedly performing an action is so common that it has even been given a name. Performing the initialization before the loop and then at the end of the loop is sometimes said to be “priming” the loop. The name has nothing to do with prime numbers, instead it has to do with the old notion of “priming a pump”. The dictionary definition of “priming” includes the notion of “making ready, preparing”. The idea here is that by assigning a value to `lastPoint` or `lastTime`, it can be used during the first iteration of the loop. All subsequent iterations depend on the value assigned at the end of the preceding iteration. As a result the needed value is always available at the beginning of each iteration of the loop.

If you learn to recognize these patterns in code, you will be able to apply them to your own programming. The idea is that most of the time, you can use familiar techniques to accomplish whatever it is you want to do. That way you can save your time and energy for the new hard problems when you encounter them. In many of the problems in this book, the first exercises attempt to get you to recognize and apply a pattern that has just been presented, while later problems attempt to stretch your creativity and problem-solving skills to accomplish something newer and more interesting.

12.2 Counting and for Loops

In this section we introduce Grace for loops. The `for` loop is included in Grace because it captures a very common pattern found in `while` loops.

The conditions of loops are often based on counting. Let’s look at some examples. In our brick wall example in Figure 7.11, we had code:

```

var brickCount: Number := 1

while {brickCount <= numBricksToDraw} do {
  ...
  brickCount := brickCount + 1
}

```

If we carefully examine the loop in the brick wall example above, we can see that it has the following structure:

```

var counter: Number = initialValue // initialize
while (counter <= lastValue) do { // test for termination
    ... // do stuff
    counter := counter+1 // increment
}

```

To set up a counting loop like this, the programmer must do four things:

1. Declare and initialize counter.
2. Set up the while header with a condition
3. Insert the body of the loop (labeled *do stuff* above).
4. Increment the counter

There are many, many examples that have very similar structure. In fact, this pattern is so common that most programming languages introduce a special loop construct to make it easier to express the pattern. In Grace this construct is called a for loop. It can replace the counting while loop above by writing the following:

```

for (initialValue .. lastValue) do { counter: Number ->
    <do stuff – but omit counter := counter + 1 at end>
}

```

The code in the parentheses describes a range of consecutive integer values. For example 1 .. 5 describes the range of values 1, 2, 3, 4, 5. When the for loop is executed, each time through it associates the next value in the range with the variable before the `->` and then executes the body of the for-loop. A very simple example is the following:

```

for (3 .. 6) do {n: Number ->
    print (n)
}

```

The first time through the loop, `n` takes on the value 3 and prints it, next time 4, then 5, and ends by printing 6.

It is easy to rewrite the brick example above to use a for loop:

```

for (1.. numBricksToDraw) do {brickCount: Number ->
    ...
}

```

From what you have seen, it should be obvious that you can't do anything more with a for loop than you can with a while loop. So why introduce it?

Essentially we have taken three of the four components of a counting while loop and combined them into one line of the for loop version. This is much simpler than the four-part counting while loop described above. It should also be apparent that its use makes it harder to forget initializing and incrementing the counter.

Aside from simplicity, another major benefit of the for loop is that it becomes evident, in one line of code, that this is a counting loop. This makes it easier for readers of your code to understand your intent.

We will refer to the variable introduced before the `->` in the loop body as the loop *index*. Be aware that because the declaration of this variable is located inside the loop body, it is only available inside that block. If you try to use `counter` outside of the loop, Grace will claim to have never heard of a variable with that name.

12.2.1 Examples of Using for Loops

Let's look at another example of a for loop. Suppose that we wanted to determine how much money we would have if we invested it for 10 years, gaining 5% interest each year (compounded annually). Let `amount` represent the amount of money we have invested. After one year, we would receive `amount * 5 / 100` in interest and hence we would have a total of:

```
amount + amount * 5 / 100
```

We can use a for loop to repeat this computation 10 times, each time updating the value of `amount` to reflect the total amount of money including the interest. Here is a method that calculates the final value of the investment for any starting value, interest rate, and number of years to invest:

```
// Return the value of startInvestment when invested at rate
// (specified as a percentage) for the given number of years.
// Parameter years must be an integer.
method investmentValue (startInvestment: Number, rate: Number, years: Number) -> Number {
  var amount: Number := startInvestment
  for (1 .. years) do {yearNum: Number ->
    amount := amount + amount * rate / 100.0
  }
  amount
}
```

Notice that `yearNum` is not actually used in the body of the for loop. However, your program will generate an error message if it is omitted from the block of code. Also notice that because the method is to return the value of `amount`, it is listed by itself on the last line of the method.

Exercise 12.2.1 *Suppose the interest for the investment is compounded quarterly (4 times per year) rather than annually. Rewrite the method to calculate the final amount. Assume the interest rate provided is the annual rate (e.g., it will need to be divided by four to obtain the quarterly rate). Hint: The index in the for loop should represent the number of quarters rather than years, and the upper bound will be an expression written in terms of years.*

Next, suppose we are considering different types of investments that have different interest rates associated with them. We want to determine what effect the different interest rates would have on the final value of the investment. The method will take as parameters a lower and an upper bound on interest rates rather than a single interest rate. As a result, we will need to provide multiple output statements, one for each interest rate.

We can write a new method that will use the method defined above to calculate the value of the investment for a series of interest rates starting at `startRate` and going up to `endRate`.

```
method printValues (startInvestment: Number, years: Number, startRate: Number, endRate: Number)
  -> Done {
  for (startRate .. endRate) do {rate: Number ->
    print("At {rate}%, the amount is: {investmentValue (startInvestment,rate,years)}")
  }
}
```

Each time through the loop, we calculate and print out the value of the investment using the current value of `rate`, then increment the rate, and continue until the last rate is `endRate`.

If we had not already written the method `investmentValue`, we could either write it so that it could be used as above, or we could instead insert the `for` loop to calculate the value of the investment inside of the `for` loop that iterates through the possible interest rates:

```
method printValues (startInvestment: Number, years: Number, startRate: Number, endRate: Number)
    -> Done {
  for (startRate .. endRate) do {rate: Number ->
    var amount: Number := startInvestment
    for (1 .. years) do { yearNum: Number ->
      amount := amount + amount * rate /100.0
    }
    print("At {rate}%, the amount is: {amount}")
  }
}
```

Note that the inner `for` loop is executed all the way through each time the outer loop goes through one iteration. Thus, when the initial rate is `startRate`, the inner `for` loop is executed a total of `years` times in order to calculate an amount, which is then printed. Next, `rate` is updated to the next value, the inner loop is executed `years` times again, and the new value is printed. Notice that we would get the wrong answer if we did not reset `amount` to `startInvestment` each time we begin the outer loop, because we would start with the wrong value for `amount` with all interest rates after the first.

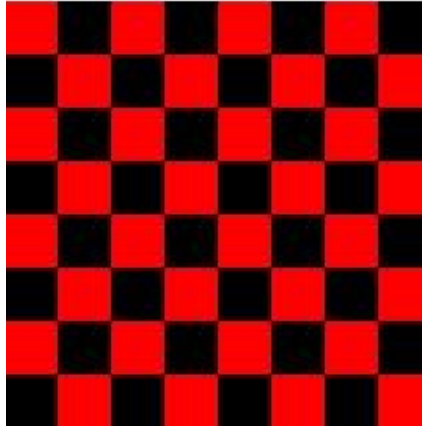
The example above illustrates the use of *nested* `for` loops. The notion of nesting loops should be familiar to you from Chapter 7 in which we introduced nested `while` loops. Indeed, any kind of loops may be nested. For example, we can have a `for` loop nested inside of a `while` loop or vice-versa.

Exercise 12.2.2 *With a little work some of the examples of while loops given in Chapter 9 can be rewritten using for loops. Rewrite either the Grass class from Figure 7.5 or the Bricks class from Figure 7.7 to use a simple counting for loop rather than a while loop. In each case the program will need to calculate the number of objects to be drawn.*

The previous example incremented the index `rate` each time through the outer loop, but it was not like our other counting loops because it didn't start at 0 or 1. In Figure 12.1 is a simple code fragment to draw a checkerboard on the screen that is an example of nested `for` loops that is more similar to our other counting loops: The output provided by the code is shown in the top part of the figure.

As the outer loop iterates through successive values of `row`, each time the inner loop runs to completion the computer will draw an entire row of cells. For example, when `row` is equal to 1, the inner `for` loop will draw the 8 squares of row 1. Each time the inner `for` loop body is executed, a new `FilledRect` is drawn, and then possibly colored red.

The key idea that allows us to decide which squares should be red has to do with the *parity* of the sum of the loop indices. Recalling that both the row and column numbers start at 1, note that when the sum of the row and column numbers is even, the squares are red, while when the sum is odd, the squares remain black. We determine whether a value is odd or even by looking at the remainder when we divide by 2, which is easily done with the “%” operator on integers.



```
for (1 .. 8) do {row ->
  for (1 .. 8) do {col ->
    def square: Graphic2D = filledRectAt ((left + (col - 1) * size) @ (top + (row - 1) * size))
      size (size @ size) on (canvas)
    if (((row + col) % 2) == 0) then {
      square.color := red
    }
  }
}
```

Figure 12.1: Nested for loops for drawing a checkerboard

Let us now examine how we compute the location of the squares. According to the program code, the location of the square in a particular row and column is $(\text{left} + (\text{col} - 1) * \text{size}, \text{top} + (\text{row} - 1) * \text{size})$. Thus, when *row* and *col* are both 1, the location of the square is given by $(\text{left}, \text{top})$. As we move across a row, the values of *col* increase by 1 for each new square. Thus the value of the x-coordinate increases by *size* for each successive square. Because the side of the square is *size* units wide, successive squares will be adjacent from left to right. Similarly, when moving to the next row, the value of *row* will increase by 1, resulting in a y-coordinate that also increases by *size*. Because the height of each square is *size*, again successive rows will be adjacent.

12.3 More loop constructs

Grace also has two more looping constructs that you may find helpful. They are `repeat()times()` and `while()do()`.

The first of these has the form

```
repeat (numExp) times {
  <code to repeat>
}
```

The behavior of this kind of loop is straightforward. It evaluates the *numExp* to a number *n* and then executes the body of the loop *n* times.

There are two important points about the syntax of the `repeat` loop:

1. the block of code to be repeated does not take an argument – it is a parameterless block.
2. the argument indicating the number of times the block is to be executed is surrounded by parentheses, not braces. This is because that expression is evaluated exactly once.

To add 7 to 93 a total of 8 times, we write:

```
var total: Number := 93
repeat (8) times {
  total := total + 7
}
```

This is a very handy construct if you need to repeat something a fixed number of times, and you do not need an index variable to keep track of what iteration you are in.

The syntax of the `do()while()` loop is as follows:

```
do {
  <code to repeat>
} while {condition}
```

It behaves very much like the `while()do()` loop except that the check of the condition is at the *end* of the loop rather than the beginning. It is convenient to use this looping construct when you will always execute the loop body at least once, and it is easier to evaluate the condition *after* rather than *before* the loop body is executed.

An example of its use is given in Figure 12.2, a rewrite of our previous program to play craps. In this version we use a nested `do()while()` loop to determine whether the player makes his “point”.

Exercise 12.3.1 *Some of our early examples of for loops don't actually use the index variable. Rewrite them as repeat loops.*

Exercise 12.3.2 *Suppose the inner do()while() loop were replaced with a while loop with exactly the same condition and the same loop body. Explain why the program would not work correctly.*

12.4 Avoiding Loop Errors

When a loop is executed, a block of code is generally executed multiple times without user intervention. As a result, it is easy to make errors that are hard to track down. In this section we enumerate some of the most common errors associated with loops.

Off by one errors. The counting for loops used as examples in this chapter all have the following structure:

```
for (1 .. maxCount) do { ... }
```

The body of such a loop is executed exactly `maxCount` times. It is easy to write a minor variation of this and end up with the count off by one. For example if the range starts at 0 rather than 1, the loop will be executed `maxCount + 1` times. On the other hand if the condition is written as `0 .. maxCount-1` then the loop will be executed `maxCount` times.

```

dialect "objectdraw"
import "random" as rand

object {
  inherit graphicApplicationSize (400 @ 400)

  var point: Number          // number trying to roll to win

  def maxVal: Number = 6    // max number on die

  // display instructions
  def welcomeMessage: Text = textAt (110 @ 50) with ("Let's play craps!") on (canvas)
  textAt (100 @ 90) with ("Click to roll the dice") on (canvas)

  // status of game
  def status: Text = textAt (100 @ 70) with ("") on (canvas)

  // For each click, roll the dice and report the results
  method onMouseClick (p: Point) -> Done {
    // get values for both dice and display sum
    var roll: Number := rand.integerIn (1) to (maxVal) + rand.integerIn (1) to (maxVal)
    welcomeMessage.contents := "You rolled a {roll}!"

    // Starting a new game
    if ((roll == 7) || (roll == 11)) then { // 7 or 11 wins on first throw
      status.contents := "You won!"
    } elseif {(roll == 2) || (roll == 3) || (roll == 12)} then { // 2, 3, or 12 lose on 1st throw
      status.contents := "You lose!"
    } else { // Set the roll as the new point to be made and continue game
      status.contents := "Try for your point!"
      point := roll
      var rollNum: Number := 1
      do {
        roll := rand.integerIn (1) to (maxVal) + rand.integerIn (1) to (maxVal)
        rollNum := rollNum + 1
      } while {(roll != point) && (roll != 7)}

      if (roll == point) then { // making the point wins!
        status.contents := "You win after {rollNum} rolls!"
      } else { // keep trying
        status.contents := "You win after {rollNum} rolls!"
      }
    }
  }
}

startGraphics
}

```

Figure 12.2: Craps game using do()while()

While both forms execute the loop body the appropriate number of times, most Grace programmers use the first form. The reasons for this will become most clear when using for loops with arrays. The main thing is to develop a consistent style for writing for loops, as then errors are less likely to result.

Infinite loops While loop bodies are designed to be executed repeatedly, it is usually not desirable to have them execute forever. Unfortunately, it is easy to make mistakes that result in a loop never terminating properly.

The most common mistake is to forget to update variables after the execution of a loop body. For example, suppose we tried writing a loop, but forgot to update a critical variable:

```
var dropCount: Number := 1
// generate specified number of balls
while (dropCount <= maxDrops) pausing (delayTime) do {
  var fb: FallingBall := fallingBallAt (xGenerator.nextValue @ 0) on (theCanvas)
}
```

Because we omitted the statement `dropCount := dropCount + 1` from the end of the loop body, the condition never becomes false. Errors like these are quite hard to track down, so always examine your loop constructions carefully before running your programs. Of course, using a for loop in this example makes it harder to forget to update the critical variable.

One simple mistake is to change an increasing index to be decreasing, but to forget to change the termination condition. For example the first example in this section might be changed to:

```
var dropCount: Number := maxDrops
// generate specified number of balls
while (dropCount <= maxDrops) pausing (delayTime) do {
  var fb: FallingBall := fallingBallAt (xGenerator.nextValue @ 0) on (theCanvas)
  dropCount := dropCount - 1
}
```

Because the programmer forgot to change the exit condition to `dropCount >= 1`, the condition will never become false, and the loop will continue until the user kills the program.

Using arithmetic comparisons in termination conditions. We indicated earlier that it is dangerous to use numbers in comparisons because of round-off errors. This warning is especially important in the use of comparisons involving fractional values as the condition in loops. Consider the following code:

```
var index: Number := 1
while (index < 2) do {
  print "Index is {index}"
  index := index + 1/3 {
}
```

This is a pretty trivial loop that starts with `index` having value 1. The value of `index` is increased by the value of `1/3` each time through the loop as long as `index` is less than 2. You would expect that the last value of `index` printed out would be something like 1.666667, but you would be wrong. Here is what was printed out during the execution of this loop:

```
Index is 1
```

```
Index is 1.333333
Index is 1.666667
Index is 2
```

What happened? Why did the 2 get printed before the loop terminated? The problem is round off error. The value computed for $1/3$ is $0.3333\dots3$ for some finite number of 3's because numbers are represented in Grace with only a finite amount of precision. When you add this number to 1 three times, you get something like $1.9999\dots9$, which is indeed less than 2, and so because the value of `index` is less than 2, it passes the test at the beginning of the while loop and executes the body of the while loop with `index` having value $1.9999\dots9$, which is not really what we intended. (Another example of the computer doing exactly what we said as opposed to what we wanted it to do!)

So why did it print `index is 2` on the last line? The reason is that the computer hold more places of the value than it prints out in a `print` statement. For example, if it held the value as 1.9999999 but only printed out the first 6 digits after the decimal point (as it did here), then it would round this up to 2.000000 before printing the corresponding string.

In summary, because the last value of `index` was a tiny fraction less than 2, the loop executed one more time than was expected.

What is the solution? First, try to avoid having the condition of a loop depend on the value of a Number if it might involve a fraction. If that must be the case, avoid using `==` (using that in the above example would have resulted in an infinite loop!), and instead use a comparison in which you have built in a “fudge factor” that will compensate for any round-off errors due to the representation of numbers in computer memory. For example, the condition of the while loop above could have been changed to `index < 1.999` because the expected last value will certainly be less than 1.999 , yet the difference between the last expected value ($1\frac{2}{3}$) and the test bound is much greater than any round-off error.

Do rest assured that integral values are held exactly, so these issues

12.5 Summary of Loops

In this chapter we covered Grace's three general loop structures. They are the while loop, which was also introduced in Chapter 7, the for loop, the repeat loop, and the do-while loop.

The general structure of the while loop is:

```
while ( <condition> ) {
  <code to repeat>
}
```

The condition is checked each time *before* executing the loop body. When it is false, execution of the loop terminates and the statement after the loop is executed. Thus if the condition is false the first time that it is checked then the loop body will not be executed at all.

The general structure of the do()while() loop is:

```
do {
  <code to repeat>
} while ( <condition> )
```

The condition is checked each time *after* executing the loop body. When it is false, execution of the loop terminates and the statement after the loop is executed. If the condition is false

the first time that it is checked then the loop body will have been executed exactly once.

The general structure of a repeat loop is the following:

```
repeat (exp) times {  
  <code to repeat>  
}
```

While the repeat loop does not provide an index that can be used in the loop body, the for loop does provide that index:

```
for (m .. n) do {count: Number ->  
  <code to repeat>  
}
```

In this case count iterates through the values from m to n, executing the body with count taking on the successive values from the range.

While any one of the loop constructs can be used to emulate any of the others, there are some general guidelines that can be helpful in determining which type of loop to use.

A repeat or for loop should be used instead of a while loop when

- The number of times that the loop body should be executed can be determined before beginning execution of the loop
- The index variable (if any) is not modified anywhere in the loop. (If the index variable is not needed than a repeat loop may be preferred over a for loop.)

While these conditions may seem very restrictive, there are many cases in which all of these hold and a repeat or for loop is the most appropriate loop construct.

In most other cases, the while loop is to be preferred. The one exception is when the loop body should always be executed at least once. In that case a do-while loop should generally be used.

12.6 Chapter Review Problems

Exercise 12.6.1 *The following method is meant to determine whether a given number, passed in as a parameter, is prime (i.e., divisible only by itself and 1).*

```
// determines whether an integer greater than 1 is prime}
```

```
method isPrime (n: Number) -> Boolean {  
  var divisor: Number := n  
  var evenlyDivisible: Boolean := false  
  while {(divisor > 1) && !evenlyDivisible} do {  
    evenlyDivisible := (n % divisor) == 0  
    divisor := divisor - 1  
  }  
  !evenlyDivisible;  
}
```

The main idea behind the method is to determine whether the number is evenly divisible by any number less than itself. Unfortunately, it doesn't quite accomplish what it is meant to do.

- Simulate the method to see what happens when the value passed in is 7.*
- Identify and correct the errors in the method.*

Exercise 12.6.2 The following method is meant to calculate $n!$ (n factorial), which is defined as follows: if n is 0, then $n!$ is 1. For all integers greater than 0, $n! = n(n-1)(n-2)\dots(1)$.

```
// returns n! for n >= 0
method factorial (n: Number) -> Number {
  var result: Number := 1
  for (1 .. (n-1)) do { counter: Number ->
    result := result * counter
  }
  result
}
```

Unfortunately, the method doesn't quite accomplish what it is meant to do.

- a. Simulate the method to see what happens when the value passed in is 4. The result returned by the method should be 24.
- b. What went wrong? Identify and correct the error in the method.

Exercise 12.6.3 Write a `for` loop that will print the two times table, from 2×1 through 2×10 .

Exercise 12.6.4 Write a pair of nested `for` loops that print the multiplication tables for 1 through 10.

Exercise 12.6.5 Write a pair of nested `for` loops that will print the following triangular pattern:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
```

12.7 Programming Problems

Exercise 12.7.1 Rewrite the program that draws a simple brick wall that is given in Figure 7.14. This time use nested `for` or `repeat` loops, rather than nested `while` loops.

Exercise 12.7.2 If you were asked to calculate the natural logarithm (\ln) of a number greater than 0 and less than or equal to 2, you would undoubtedly press the \ln button on your calculator. Grace also includes \ln in its math module. However, an alternative would be to find its value as follows.

$$\ln(n) = (n - 1) - 1/2(n - 1)^2 + 1/3(n - 1)^3 - 1/4(n - 1)^4 + 1/5(n - 1)^5 \dots$$

The more terms you include in your calculation, the more precise your answer will be. Write a method that will return $\ln(n)$ to a specified precision. That is, your method should take two parameters, n and a positive integer that gives the number of terms to be included in the calculation.

Exercise 12.7.3 Write a program that will simulate the roll of a die to determine which of two players goes first in a game. When the user clicks the mouse, the program should print a message on the canvas saying whether player 1 or player 2 is to go first. Since it is possible that both players will roll the die and get the same value, the process might have to be repeated before there is a clear winner. Use a `do ...while` loop here.

Chapter 13

Lists

If you look down at the bottom of the page you are reading, you will find a number. Page numbers are very handy. We don't think about them much, but without them, it would be harder to use a text book like this one. You wouldn't be able to look things up in the index. The table of contents would be much less useful as well.

Pages aren't the only things that are helpful to number. Obviously, in this text we also number the chapters and sections. If this was a really important book (like the Bible or a play by Shakespeare) then individual lines or verses would be numbered. If someone asked you what year it was or what time it was, the answers would involve numbers. If you were asked what day it was, on the other hand, you might answer "the 25th" or you might answer "Tuesday". Your answer would depend on whether you thought you were being asked about the day of the week or the day of the month. This reveals a little bit more about how we use numbers to identify things. When we are dealing with small collections, like the seven days of the week, we use a distinct name for each member of the collection. For larger collections, like the days of the month or the pages in this book, numbering seems much handier.

The same turns out to be true in computer programs. When writing programs that manipulate relatively small collections of data, we can associate each item with a distinct instance variable. On the other hand, when writing a program that manipulates a large collection of similar data items, it is often useful to be able to simply number the items instead of using a distinct name for each item. In Java, and many other programming languages, this is accomplished using the feature we will describe in this chapter, the *list*.

A list is a collection of objects. Each member of the collection is associated with a number called its *index*. To use a member of a list in a program, the index value associated with the desired member must be provided. This can be done by explicitly writing the number in the program or by using a formula involving variables, constants and arithmetic operations. We can vary which elements of a list are accessed as the program is running by varying the values of the expressions that specify list index values. This makes the list a powerful programming tool.

In this chapter, we will explain how to declare names that refer to lists, how to create lists, and how to make associations between index values and members of the list. Then we will look at a variety of common techniques for processing the information kept in a list.

Lists in Grace are similar to arrays in languages like C, C++, Java, and C#. However, they

are more flexible in that they can grow in size while being used. You will see in this chapter that they support a number of very powerful and flexible operations to help the programmer.

13.1 Declaring Lists

When we use numbers to identify members of a collection, we usually have to specify the collection we have in mind. For example, if we told you to read 12 of this book, you would not know what we were talking about. We would have to be more specific and tell you to read page 12 or chapter 12 (or even section 12 of a particular chapter). Similarly, in Grace, to use a list we need a name that will refer to the entire collection of elements in the list. We can access an element of a list using the `at` method and providing the location of the element in the list. For example, if `page` was the name of a list used to hold `String` values corresponding to the pages of a book, one could say

```
page.at (12)
```

to access the 12th element of the collection of pages. On the other hand if we had a list `canvasList` consisting of geometric objects on the canvas (listed from top-most to bottom-most), then `canvasList.at (1)` should return the top-most object displayed on the canvas.

As you will notice from these two examples, when you access an element of a list, you will get a different type of element, depending on the list. When we evaluated `page.at (12)`, a `String` was returned, while when `canvasList.at (1)` was evaluated, a `Graphic` object was returned. Thus we will have different types of lists, depending on the type of the elements they contain.

List types are specified by a term of the form `List[[T]]`, where `T` is the type of elements of the list. Thus in our first example, `page` will have type `List[[String]]`, because it contains elements of type `String`. In the second example, `canvasList` might have type `List[[Graphic]]` or `[[Graphic2D]]`, depending on what types of graphic objects it contains. Declaring each as instance variables would look like the following:

```
var page: [[String]]
var canvasList: [[Graphic]]
```

Note that choosing a good name for a list is a bit tricky. The declaration shown makes the name `page` refer to a whole collection of pages, suggesting that `pages` might be a better choice. When referring to a single element of the collection, however, you have to use the name of the collection together with an index value. In this context, saying `page.at (12)` makes more sense than saying `pages.at (12)`. The other strategy is to append the name with the word `List` as we did with `canvasList`.

The fact that the type of the list elements appears in list types raises an important point. In Grace, all the elements of a single list must be of the same type. You cannot have a list whose first element is a `Number`, whose second element is a `String`, and whose third element is a `Graphic`. On the other hand, you can work with lists of any type you need. Thus, in a program that worked with musical notes, each of which has type `Note`, you might define a song:

```
def songNote: [[Note]] = ...
```

to refer to the list of notes that represents the song.

13.2 Creating a List

The declaration of the list `page` shown above introduces a name that can be used to refer to a collection of `Strings`. The construction of a list is simple. All that Grace needs to know is the type of items that will be members of the list and any initial values to be held in the list. If the list is to be empty then we just write:

```
var page: List[[String]] := emptyList[[String]]
```

Of course there isn't much difference between an empty list of `String` and an empty list of `Number`, but the type specification ensures that only `Strings` can be added to the list.

Suppose on the other hand we wanted to create a non-empty list of `Strings`. Let's suppose we have the (very short!) pages consisting of the strings "Page 1", "Page 2", "Page 3", and "The end". The construction for our pages example would now be

```
var page: List[[String]] := list[[String]] ["Page 1", "Page 2", "Page 3", "The end" ]
```

In general this construction starts with `list.with`, followed by the type of item in the list in "double square brackets": "List[[...]]". Finally, all the elements that you want added initially are listed in a lineup. You can have as many of these elements as you like, but they must be surrounded by single square brackets. This is our first example of the use of "varargs" in Grace. It simply means that you can use the construction with as many parameters as are necessary, as long as they are all of the same type.

You can access any of the elements of a list with a method request of `at` with the position of the element you want to access. Elements are numbered 1 to the number of elements in the list.

```
def secondPage: Page = page.at (2)
```

After executing that statement, `secondPage` will have value "Page 2". If you make an `at` method request for an index that doesn't exist, then you will get a run-time error.

What if you don't have all the elements of the list at the beginning? Then you can build up the list one element at a time using the `add` method of `List`. For example, we can get the effect of the above declaration and initialization of `page` by writing the following sequence of commands:

```
var page: List[[String]] := emptyList[[String]]
page.add ("Page 1")
page.add ("Page 2")
page.add ("Page 3")
page.add ("The end")
```

Each time that the `add` method is requested of the `page`, the new `String` is added to the end of the list.

Of course you can request `add` on a list at any time and the element will be added to the end. We can also update elements in any position in a list with an `at()put()` method request.

```
page.at (1) put ("Page one")
```

The above method request will result in changing the first element of the list from "Page 1" to "Page one". Just like with an `at` method request, if you try to update a slot that does not exist, you will get a runtime error.

A list is a mutable object. That is, when you add a new element to a list, it remains the

same list. However, its contents change. The same is true of the other changes we can make to elements of type `List[[T]]`. Thus we can define a list identifier with a **def** and then apply methods to update its contents. You only need a variable definition if you are going to assign a new list to the identifier.

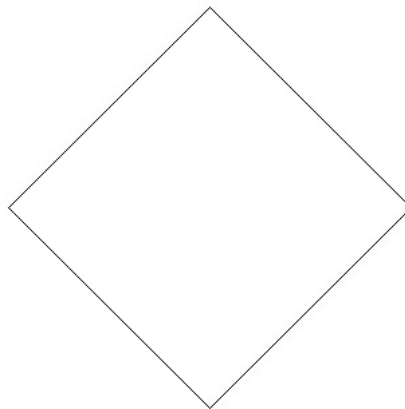
Once we obtain an element of a list of type `List[[T]]` using `at`, we can operate on it like any other element of type `T`. Going back to our example `canvasList` of type `List[[Graphic]]`, we can write code like `canvasList.at (2).color := red` or `canvasList.at (3).move (5, 7)`.

Exercise 13.2.1 *Provide an instance variable declaration for each of the names suggested in the following list. Include the specification of an initial value that associates the name with an appropriately constructed list. For example, if asked to declare `dailyTemp` as a name that could refer to a list of the high temperature readings for a range of days, an appropriate answer would be:*

def `dailyTemp`: `List[[Number]] = emptyList[[Number]]`

- a. *Provide a declaration for `cityPopulation`, a list used to hold the current populations of each of a collection of cities in the world.*
- b. *Provide a declaration for `scenes`, a list used to hold the text (held as a string) of each of the scenes in a play.*
- c. *Provide a declaration for `path`, a list used to refer to line segments in a route from one place to another on a map.*

Exercise 13.2.2 *Consider the drawing of a diamond shown below:*



- a. *Provide a **def** declaration for `diamond`, a name that could be used to refer to a list including the four `Lines` needed to make this drawing as elements. In your declaration, include a list construction that initializes the name `diamond` to refer to an empty list object.*
- b. *Write the code to create the diamond drawing and associate the four lines drawn with the first four elements of the list `diamond`. Assume that the diamond is drawn to just fit within a 200×200 pixel window.*
- c. *Suppose that we wanted to make the drawing a bit more colorful by making the top two `Lines` of the diamond red. Show the two statements that would be required to make this color change assuming the `diamond` list has been initialized as in part (b).*

13.3 Using Lists: A Triangle Class

In the preceding sections, we explained how to declare and construct an list, how to refer to a single element of an list, and how to update an element of an list. This is really all there is to learn about lists in Grace. At this point, however, you probably have little or no idea how to actually use an list in a program. To develop this ability, you need to become familiar with the patterns and strategies an experienced programmer employs when using lists. The best way to accomplish this is to examine some examples of programs that employ lists. In this and the remaining sections of this chapter, we will examine three extended examples of class definitions using lists that illustrate important techniques frequently used when programming with lists.

As a first example, suppose you wanted to write a program that involved drawing triangles on the screen. There are no classes included in `objectdraw` or the standard Grace libraries designed to draw triangles. We could simply draw triangles by creating three `Lines` for the edges of each triangle, but if triangles play a significant role in a program, it would make good sense to define a new class that makes it as convenient to work with a triangle as it is to work with a filled rectangle or framed oval.

Within such a triangle class, we would need instance variables to refer to the three `Lines` that make up the triangle. We could define three independent variables with names like `edge1`, `edge2`, and `edge3`. It would be better, however, to define an list of `Lines` to keep track of the edges. The following declaration could be used to introduce such an list:

```
def edge: List[[Line]] = emptyList[[Line]]
```

This declaration states that we plan to use the name `edge` to refer to a collection of `Lines`. While we have now created an empty list, additional code will be included in the `triangle` class to create three `Lines` and add them to the list. Once we have added the three edges, the expressions `edge.at(1)`, `edge.at(2)`, and `edge.at(3)` can be used to refer to the `Lines` in the list. *Note that in many programming languages, particularly those that derive from the language C, list indices start at 0 rather than 1. In Grace we follow the more intuitive idea of counting from 1.*

The simplest way to describe a triangle is to provide the coordinates of its three vertices. Therefore the `triangle` class will expect four parameters: three `Points` describing the vertices together with the `canvas`. These parameters will be used to create the `Lines` that make up the triangle and associate them with elements of the `edge` list. A fragment of the complete `triangle` class including this code is shown in Figure 13.1. Figure 13.2 shows the triangle that would be drawn if this constructor were invoked with the coordinates (100, 100), (50, 200), and (250, 100) as parameters. The figure also attempts to graphically represent the relationship between the list `edge` and the `Lines` displayed on the screen. A gray arrow is included to indicate which of the `Lines` would be associated with each list entry.

The `triangle` class should include implementations of methods like `moveBy`, `color:=`, and `visible:=` that are like those in the graphics classes included in the `objectdraw` library. The easiest of these methods to implement is `moveBy`. We can implement `moveBy` by simply using the three indexed variables `edge.at(1)`, `edge.at(2)`, and `edge.at(3)` much as we might have used three simple variables that refer to the edges of the triangle. The definition of `visible:=` we would produce in this way is shown here:

```
dialect "objectdraw"
```

```
class triangleAt (pt1:Point,pt2:Point,pt3:Point) on (canvas: DrawingCanvas) {  
  def edge: List[[Line]] = emptyList[[Line]]  
  edge.add (lineFrom (pt1) to (pt2) on (canvas))  
  edge.add (lineFrom (pt2) to (pt3) on (canvas))  
  edge.add (lineFrom (pt3) to (pt1) on (canvas))  
  
  // ... method declarations ...  
}
```

Figure 13.1: Constructor for the Triangle class

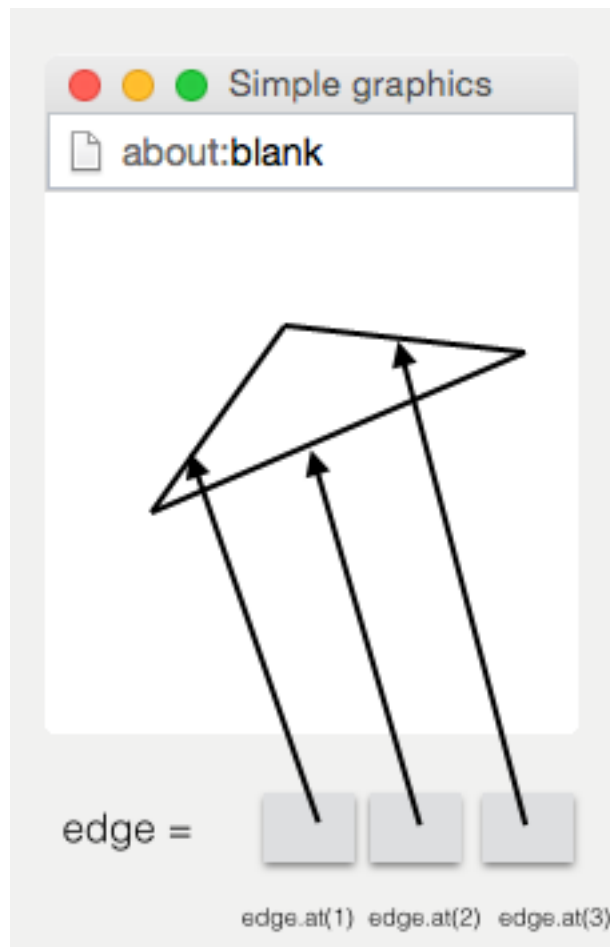


Figure 13.2: Elements of the edge list refer to components of a Triangle

```

method visible := (vis: Boolean) -> Done {
  edge.at (1).visible := vis
  edge.at (2).visible := vis
  edge.at (3).visible := vis
}

```

The value we include in an indexed variable does not have to be an numeric constant. We can instead use a variable, definition, or any other expression that describes a `Number` that identifies the desired element of the list. For example, if `edgeNum` is an `Number` variable, then we can say

```

edge.at (edgeNum).visible := false

```

We can use this fact to write an alternate version of `visible:=` that takes advantage of the fact that the edges are gathered together in an list. If we include the statement

```

edge.at (edgeNum).visible := vis

```

in a loop that varies the value of `edgeNum` from 1 to 3, execution of the loop will change the visibility of all three edges. A version of `visible:=` implemented using this approach is shown next:

```

method visible:= (vis: Boolean) -> Done {
  for (1 .. 3) {edgeNum: Number ->
    edge.at (edgeNum).visible := vis
  }
}

```

Comparing the two versions of `visible:=` above, it may not be immediately obvious that using a loop to write `visible :=` is desirable. The code that uses the loop is initially harder to understand than the first version of `visible:=`. Using a loop, however, makes the code more flexible. Suppose that instead of defining a triangle class, we had decided to define a class that drew hexagons. The name `edge` would have to be associated with an list of six `Lines` instead of three. The code constructing the hexagon class would have to create six `Lines` and associate them with the elements of the list. The definition of the `visible:=` method shown above, however, could be used in the hexagon class with the only change being replacing the 3 by 6 after the `for`. Furthermore, of course, a very similar loop could also be used for octagons or figures with even more edges. We could even create a `for` loop that works for both triangles and hexagons by using the `size` method for lists, which returns the number of elements in the list.

```

method visible:= (vis: Boolean ) -> Done {
  for (1 .. (edge.size)) do {edgeNum: Number ->
    edge.at (edgeNum).visible := vis
  }
}

```

Hopefully these brief examples help convince you of the advantage of using lists. Loops that are quite short and simple can describe the processing of large collections of objects.

You've seen `for` loops like the ones above before. Let's now introduce a feature of `for` loops that works for lists. If we don't need to know the actual indices of the elements of the list, we can just write a `for` loop that implicitly goes through them all in order:

```

method visible:= (vis: Boolean) -> Done {
  for (edge) do {e:Line ->
    e.visible := vis
  }
}

```

```

    }
  }

```

Like the earlier versions above, this goes through all the elements of `edge` and sets their visibility to correspond to the value of `vis`

```

  When we execute a loop of the form
  for (1 .. 3) do {i: Number ->
    // perform op on element edge.at (i)
  }

```

the values of the identifier `i` go through the values 1, 2, and 3. We had to refer to `edge.at (i)` to access the *ith* element of the `edge` list.

```

  On the other hand, when we write
  for (edge) do {e: Line ->
    // perform op on e
  }

```

the values of identifier `e` range through the values of the lines given by `edge.at (1)`, `edge.at (2)`, and `edge.at (3)`. This means we don't have to worry about writing the indices of the elements of the list, as the `for` loop takes care of accessing them for us.

To provide another example of the use of such loops, here is a definition of the `moveBy` method for our `triangle` class.

```

method moveBy (dx: Number, dy: Number) -> Done {
  for (edge) do {e:Line ->
    e.moveBy (dx, dy)
  }
}

```

Like the definition of the `visible:=` method, the loop in this method iterates over all the elements in `edge`. As a result, identical code could be used to define these methods in a `hexagon` or `octagon` class.

The graphic classes included in the `objectdraw` library provide methods through which one can obtain information about the position and dimensions of an object. For example, methods like `location`, `width`, and `height` allow one to obtain a complete description of a filled rectangle. To provide a similar feature for our `Triangle` class, we would like to define a method named `vertices` that will return the locations of all three vertices of a `Triangle`. A method, however, can only return a single result. Therefore, rather than returning the individual positions, our method will return a list containing the positions of the three vertices.

The code for `vertices` is here:

```

method vertices -> List[[Point]] {
  def vert = emptyList[[Point]]
  for (edge) do {e:Line ->
    vert.add (e.start)
  }
  vert
}

```

This method is yet another example in which we use a loop to perform a simple operation on every element of an list. In this case, the operation is to use the `start` method of the `Line` type to extract one of the triangle's vertices from each of the `Lines` that are its edges.

As noted above, `vertices` returns a list as its result. The `vertices` method is the first example

we have seen that uses this ability. In Grace, lists are objects. There are certain operations one can perform with objects of any type in Grace. You can define names and associate them with objects of any type. You can write methods that return objects of any type. Finally, it is also possible to pass any object, including an entire list, as a parameter to a method.

Exercise 13.3.1 *Provide a definition for a `color:=` method for the `triangle` class discussed in this section. Use a loop as shown in the implementations of the `moveBy` and `visible:=` methods in this section.*

Exercise 13.3.2 *In the preceding section, we suggested that many of the methods written for our `triangle` class could be used without change in classes that drew other kinds of polygons. The only difference between these classes and our `triangle` class would be the way in which the edge list was initialized.*

Next to a triangle, the simplest form of polygon is a rectangle. There already are rectangle classes included in `objectdraw`, but for this problem we would like you to consider how you could define a rectangle class of your own named `myOwnFramedRect`. In particular:

- a. *Show how to declare and initialize the edge list variable.*
- b. *Show the code to add the vertices to the list. Assume that the parameters to the constructor will be identical to those used by the `framedRect` class: the location of the upper-left corner, the width and height, and the canvas.*

Exercise 13.3.3 *We were able to implement all the methods in the `triangle` class using loops. In the code constructing the edges, however, it was not possible to use a loop because we passed the vertices of the triangle to the constructor as three separate `Point` parameters.. An alternate way to define the constructor is to have the vertices passed as a single list of `Points`. The header for this new class would be*

class `triangleVertices` (*vertex: List[[Point]]*) on (*canvas: DrawingCanvas*)

Show how to complete the definition of the constructor in this way. Hint: You can assign a list to a variable just like you would any other object..

Exercise 13.3.4 *The following problems involve lists of numbers.*

- a. *Write a method, `sum`, that takes a list of `Numbers` as a parameter and returns the sum of all elements in the list. For this part use the version of the `for` loop that uses a range of indices.*
- b. *Rewrite the `sum` method to use the enhanced `for` loop that suppresses the indices.*

13.4 Gathering Information from a List

There are many situations where it is useful to gather information about the entire collection of elements in a list rather than processing the members of the collection independently. Suppose, for example, that we wanted to determine the perimeter of a triangle represented using our `Triangle` class. We could not accomplish this by applying an operation independently

to each element as we did in the `moveBy` or `visible:=` methods. Instead we would have to write a loop which added together the lengths of all the edges. In this section we will examine several examples of loops that collection information from a collection of list elements in similar ways.

As a context for examining such loops, we will consider the construction of the Grace code needed to control the device shown in Figure 13.3.



Figure 13.3: Radar Trailer

These radar trailers have become very popular with local police forces. Our town purchased one recently and we get to test its accuracy regularly as we drive around town. We suspect, however, that many drivers who pass such radar trailers underestimate the sophistication of these devices. The trailer shown can be purchased with an option that includes a traffic statistics computer that provides traffic counts and an analysis of speed categories. Basically, there is a computer in the radar cart running a program that keeps statistical summaries of the traffic that passes by.

It isn't hard to imagine how the statistics collected by such a radar unit could assist police efforts to enforce speed limits. If it simply kept a count of the number of speeders detected, the radar trailer could be used to identify those roads on which speeding is a problem. This information could be used to determine where to assign police officers to monitor traffic.

Let's consider how we might collect more specific information about the pattern of speeding violations. In particular, we will consider how to write a program that determines the number of speeders that pass the trailer during each of the 24 hours of a single day.

The 24 numbers used to count speeders will be kept in a list. The necessary list can be declared and initialized by the instance variable declaration

```
def speeders: List[[Number]] = emptyList[[Number]]
```

The indexed variable `speeders.at (hour)` will be used to access the number of speeders detected during a given “hour”. Thus, `speeders.at (9)` will refer to the number of speeders detected at 9 o’clock (i.e. between 9:00 and 9:59). We will use a 24 hour clock so that a unique index value is associated with each hour. This will enable us to avoid using “a.m.” and “p.m.” to distinguish morning from evening. Instead, we will add 12 to the hour of any “p.m.” time. Rather than saying 3 p.m. we will say 15, and for 10 p.m., we will use 22. Of course, 2 a.m. will still be 2 and 5 a.m. will still be 5.

One special case is worth noting. In a 24 hour clock, midnight can be referred to as either 00:00 or 24:00. However, one minute after midnight is 00:01 and so on. For the purposes of simplifying our code, we will refer to that first hour of the day with the hour designation of 24. Thus, we differ from the standard, by writing 1 minute after midnight as 24:01. (But see the homework problem below on how to fix this!) Thus, `speeders.at (24)` will be the element of the list used to keep track of the number of speeders seen between midnight and 12:59 a.m. Obviously, this is convenient given that Grace starts numbering the elements of all lists with 1 rather than 0.

We know that the list `speeders` will have 24 items in it, one for each hour of the day. Thus we will need to initialize it in the class definitions:

```
def speeders: List[[Number]] = emptyList[[Number]]
for (1 .. 24) do {
  speeders.add (0)
}
```

Thus we initialize `speeder` to have 24 elements, each of which has value 0, corresponding to the fact that we have not yet detected any speeders at any time of the day.

While we don’t actually know any details about the organization of the software sold with these radar trailers, it is not difficult to imagine a possible structure for such a program. There would be (at least) two classes. One program/object would act as the “controller”. We will assume it is named `radarController`. Just as our classes inheriting `graphicApplication` defined event-handling methods to respond to mouse or keyboard events, the `RadarController` would define an event-handling method that would be invoked whenever a vehicle was detected. This method would include code to use the trailer’s radar system to determine the vehicle’s speed.

Another class would be responsible for recording statistics. We will assume it is called `trafficStats`. The `RadarController` would invoke an appropriate method on an object of the `RadarStats` class each time it detected a vehicle and measured its speed. Within the code of this method, we would update the list when a new speeder was detected. Accordingly, the declaration and initialization of the `speeders` list would be included in the `radarStats` class. The class would also have to provide methods to enable someone to access the statistics that had been collected. There might be methods to draw a graph showing the number of speeders detected at each hour of the day or to report the hour during which the highest number of speeders had been detected.

```

method vehicleReport (speed: Number, hour: Number, minute: Number) -> Done {
  if (speed > speedLimit) then {
    speeders.at (hour) put (speeders.at (hour) + 1)
  }
}

```

Figure 13.4: The vehicleReport Method

13.4.1 Counting Speeders

Let's start by seeing how we might write the method of the `radarStats` class that the `RadarController` will invoke to handle the detection of a new vehicle. We will assume that this method is named `vehicleReport` and that it takes three parameters: a number giving the vehicle's speed and two numbers reporting the time of day. The first integer will be the hour (encoded using a 24 hour clock as described above). The second integer will be the minute within the hour. Thus the header for this method might be:

```

method vehicleReport (speed: Number, hour: Number, minute: Number) -> Done {

```

Figure 13.4 shows a version of the `vehicleReport` method designed to add one to the appropriate element of the `speedersAt` list each time a speeder is detected. The if statement included in the method determines whether or not the detected vehicle is speeding. Like the loops in the methods of the `Triangle` class, this method depends upon the fact that a list index can be specified using a variable or parameter name. Since the parameter `hour` contains the hour at which the speeder was detected, `speeders.at (hour)` will refer to the list element that should be incremented. The body of the if statement increments this indexed variable.

Exercise 13.4.1 *Suppose that we want to determine the fraction of the cars that pass by the radar trailer that are speeding during each hour of the day. To do this, we would have to count both the total number of cars that pass by during each hour and the number of speeders and then divide the second number by the first to determine the fraction of the cars that are speeding.*

Assuming that the `radarStats` class includes the following declarations:

```

def speeders: List[[Number]] = emptyList[[Number]]
def drivers: List[[Number]] = emptyList[[Number]]
def speedingFraction: List[[Number]] = emptyList[[Number]]

```

show how to rewrite the `vehicleReport` method so that `drivers.at (h)` refers to the total number of drivers seen at hour `h` and `speedingFraction.at (h)` refers to the fraction of cars that were speeding at hour `h`. Your code should update these lists appropriately each time `vehicleReport` is invoked.

13.4.2 Drawing a Histogram

Of course, to be useful, our program has to display or report the data it has collected. One way to do this would be to draw a histogram or bar graph like the one shown in Figure 13.5, showing how many speeders were seen during each hour of the day. The code to draw such

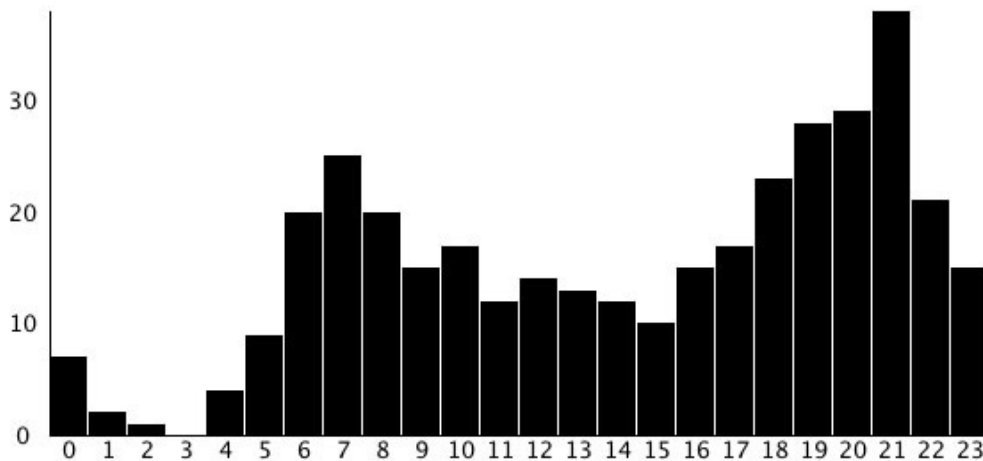


Figure 13.5: A graph displaying numbers of speeders detected at different times of the day

a graph will provide yet another example of a loop that performs some operation on each element of a list. In this case, the operation will be to draw a bar corresponding to each value found in the `speeders` list. In addition, however, drawing a histogram will also require the use of a loop that gathers information about all the elements in the list.

One decision we must make before drawing a bar graph is what scale to use on the y axis. If the radar trailer is placed on a busy highway, the number of speeders detected might be in the thousands. To ensure that all the bars drawn fit on the screen, we would have to make each pixel on the display correspond to 10 or more speeders. Unfortunately, using the same scale when the trailer is placed on a quiet road would lead to a graph where all the bars are too short to see. Thus, we have to look at the data collected to pick an appropriate scale before drawing the graph.

We will consider two ways to set the scale based on the data collected. The first is to graph the percentage of speeders seen in each hour rather than the actual number of speeders seen. If 20% of the speeders were seen at 7 a.m., we would draw a bar whose height was equal to 20% of the vertical space available for the graph. If only 10% of the speeders were seen at some hour, the bar drawn for that hour would be one tenth of the maximum. In general, to determine the size of the bar to be drawn for a given hour we would multiply the maximum bar size by the result of dividing the number of speeders seen in that hour by the total number of speeders seen. To do this, we need to compute the total number of speeders seen.

Summing the Values in a Numeric List

We can compute the total number of speeders by adding up all the values in the `speeders` list. The method in Figure 13.6 shows the code required. This method would be added to the `radarStats` class. The loop in the method adds the number of speeders associated with each

```

method totSpeeders -> Number is confidential {
  var total: Number := 0

  for (speeders) do {speedersAt: Number ->
    total := total + speedersAt
  }

  total
}

```

Figure 13.6: The speedersSeen method

element of the list (i.e., the number of speeders each hour of the day) to a variable named `total`. Before the loop is executed, `total` is initialized to the value 0.

The body of the loop in the `totSpeeders` method depends on the identifier `speedersAt`. Each time through the loop it takes on the value of the next item in the list `speeders`. After each repetition of the loop, `total` will equal the sum of all of the entries in the `speeders` list up to and including the most recent element of the list represented by `speedersAt`. Therefore, when all the iterations are complete `total` will be equal to the sum of all the entries.

Drawing a Simple Histogram

With this confidential method available, we can write a method to draw the desired graph. To keep things simple, we will just worry about drawing the bars without adding any labels. That is, the code we write will draw something that looks more like Figure 13.7 than Figure 13.5.

The loop that draws the histogram is structurally similar to the loops used in the `Triangle` class. The body of the loop will contain code to independently process each of the elements of the list. This time, the operation performed will be to draw the bar in the graph that corresponds to a particular hour.

We will assume that a number of instance variables describing the dimensions of the graph are declared in the `trafficStats` class and initialized by its constructor. In particular, we assume that:

`graphHeight` is the height of the area in which the graph is to be drawn,

`graphLeft` is the x coordinate of the left edge of the graph, and

`graphBottom` is the y coordinate of the bottom edge of the graph.

`graphTop` is the y coordinate of the top of the graph.

`barWidth` is the width of a single bar.

In addition, we use the `totSpeeders` method described above to set the variable `totalSpeeders` equal to the total number of speeders detected.

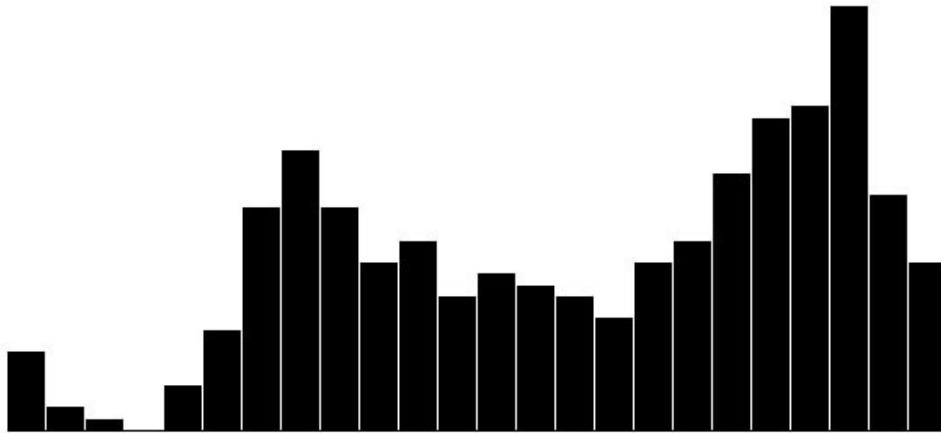


Figure 13.7: A drawing of the essence of a histogram

The method that draws the bars of the histogram is shown in Figure 13.8. The header of the loop that draws the bars is identical to the header of the loop in `speedersSeen`. The body of this loop contains two statements. The first:

```
barHeight = (speeders.at (hour) / totalSpeeders) * graphHeight;
```

computes the correct height for the bar representing the speeders seen during `hour`. It does this by dividing `speeders.at (hour)`, the number of speeders seen in the hour being processed, by the total number of speeders seen. The result is then multiplied by the height of the longest possible bar to determine the height of the bar that should be drawn.

We found it convenient to write the `for` loop in this method with `for (1 .. 24) do {...}` rather than `for (speeders) do {...}` as we did with the previous methods. We did this because we needed to use the value of the index `hour` in calculating the left edge of the filled rectangle. If we had not needed that value, we could have just used `speeders`.

Finding the Largest Value in a Numeric List

Unfortunately, if we use this method to draw a histogram of the speeder data collected, the result is likely to look like the image shown in Figure 13.9. The bars in this graph are all rather short. The problem is that the number of speeders seen at any hour will on average be 1/24th of the total number of speeders seen. Therefore, the average bar our program draws will be 1/24th of the total vertical space available for the graph.

To make the bars taller, we should make the bar for the hour with the largest number of speeders as tall as space allows and then adjust the sizes for all the other bars accordingly. To do this, we have to determine how many speeders were counted during the hour at which the largest number of speeders were seen. This involves writing another simple loop that

```

method drawSimpleHistogram -> Done {
  def topNumSpeeders: Number = totSpeeders
  canvas.clear

  lineFrom (inset @ graphTop) to (inset @ (graphTop + graphHeight)) on (canvas)
  lineFrom (inset @ (graphTop + graphHeight)) to ((inset + 24 * barWidth) @ (graphTop + graphHeight))
  on (canvas)

  def graphLeft: Number = inset
  def graphBottom: Number = graphTop + graphHeight

  for (1 .. 24) do {hour: Number ->
    def barHeight: Number = graphHeight * speeders.at (hour) / topNumSpeeders
    bar.add(filledRectAt ((graphLeft + (hour - 1) * barWidth) @ (graphBottom - barHeight))
              size ((barWidth - 1) @ barHeight) on (canvas))
  }
}

```

Figure 13.8: Method that draws histograms



Figure 13.9: A bar graph with rather short bars

gathers information as it processes all of the elements of the `speedersAt` list.

To understand how this loop will function, it helps to think carefully about the list-processing loop in `totSpeeders`.

```

method totSpeeders -> Number is confidential {
  var total: Number := 0

  for (speeders) do {nextSpeeders: Number ->
    total := total + nextSpeeders
  }

  total
}

```

In particular, recall how the variable `total` is used. Each time an iteration of the loop is completed, the value of this variable has a simple connection to the elements of the list. The

```

// return the max number of speeders at any hour
method maxSpeeders -> Number is confidential {
  // the current max number of speeders for hours already processed
  var maxV: Number := 0

  for (speeders) do {nextSpeeders: Number ->
    if (maxV < nextSpeeders) then {
      maxV := nextSpeeders
    }
  }
  maxV
}

```

Figure 13.10: Method to find largest value in speedersAt list

value of `total` is always the sum of all of the list elements that the loop has processed so far. That value is returned as the value of the method once the loop has completed.

Similarly, to find the largest value in the list, we will need a loop that manipulates a variable that keeps track of the largest list element processed by the loop so far. The loop in the method shown in Figure 13.10 uses such a variable named `max` to find the largest element in the list and return its value. We begin by setting `max` equal to 0 as all the values in the list are at least 0. Each of the elements in the list is then compared to `max` as the loop executes. as in `totSpeeders`, the loop uses the variable `nextSpeeders` to sequentially select and process each element in the list. The variable `max` will always be associated with the largest element found in the list before the entry currently being processed by the loop, `nextSpeeders`. If `nextSpeeders` is greater than the value of `max`, then the value of `max` is changed to make it equal to this element.

If the `maxSpeeders` method is added to our `RadarStats` class, it is quite easy to modify the method shown earlier to draw histograms like those shown in Figure 13.7. The modified version of `drawHistogram` is presented in Figure 13.11. The only change is that the variable `totalSpeeders` used in the original version has been replaced by a variable named `topNumSpeeders` which is set using the `maxSpeeders` method.

Exercise 13.4.2

- a. Write a `minSpeeders` method that will return the number of speeders detected during the hour in which the fewest speeders were detected.
- b. Write a `minSpeederHour` method that returns the hour during which the smallest number of speeders was seen. If there is a tie, your method can return any hour during which the smallest number of speeders was seen.

Exercise 13.4.3 A histogram or bar graph is just one of many ways we could display the counts of speeders stored in the `RadarStats` class. A simple alternative would be to draw lines

```

// Draw histogram so largest bar is same size as graph height
method drawHistogram -> Done {
  // max number of speeders at any hour
  def topNumSpeeders: Number = maxSpeeders
  canvas.clear

  textAt (50 @ inset) with ("max speeders in any slot: {topNumSpeeders}") on (canvas)

  // height represented by each speeder in an hour
  def numSpacing: Number = graphHeight / topNumSpeeders

  // mark vertical axis for each 10 speeders
  for (1 .. (topNumSpeeders / 10)) do {count: Number ->
    def count10 = 10 * count
    textAt (0 @ (graphTop + graphHeight - numSpacing * count10)) with (count10.asString) on (canvas)
    print "{count10} at {(graphTop + graphHeight - numSpacing * count10)}"
  }

  // draw graph left and bottom edges
  lineFrom (inset @ graphTop) to (inset @ (graphTop + graphHeight)) on (canvas)
  lineFrom (inset @ (graphTop + graphHeight)) to ((inset + 24 * barWidth) @ (graphTop + graphHeight))
    on (canvas)

  // label horizontal axis with hours
  for (1 .. 24) do {hour ->
    textAt ((inset + barWidth * (hour-1)) @ (graphTop + graphHeight + 20))
      with (hour.asString) on (canvas)
  }

  // left and bottom edges of graph
  def graphLeft: Number = inset
  def graphBottom: Number = graphTop + graphHeight
  // draw bars for each hour
  for (1 .. 24) do {hour: Number ->
    bar.add (filledRectAt ((graphLeft + (hour - 1) * barWidth) @
      (graphBottom - graphHeight * speeders.at (hour) / topNumSpeeders))
      size ((barWidth - 1) @
      (graphHeight * speeders.at (hour) / topNumSpeeders))
      on (canvas))
  }
}

```

Figure 13.11: Method that draws well-scaled histograms

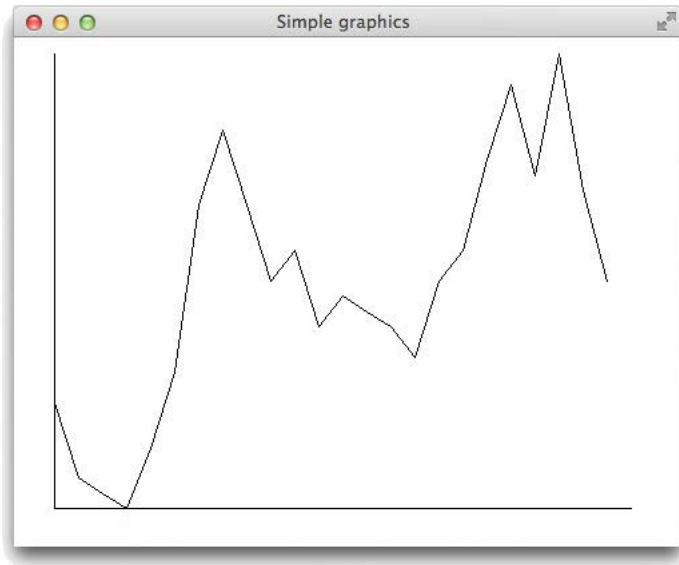


Figure 13.12: Sample of output expected from the `drawLineGraph` method

connecting the points with coordinates $(\text{hour}, \text{speeders.at} (\text{hour}))$ for all the values of hour from 1 to 24. A picture of what we have in mind is shown in Figure 13.12.

Write a method `drawLineGraph` that would draw a graph like that shown in the figure. Assume that your method is included within the `trafficStats` class and can therefore access the names `barWidth`, `graphTop`, and `graphHeight`. Although you aren't drawing any bars, you should use the value of `barWidth` to determine how to space the points you connect horizontally.

13.5 Collections with Variable Sizes

Let's consider a new set of examples using lists while sticking with the theme of speed.

Over the last few years, one of the authors has had the pleasure of spending a lot of time watching and sometimes helping to run long distance races involving teams from our local high school. In the fall, there is cross country running. Then, in the winter, the sneakers get replaced by skis and poles as the cross country skiing season starts. The examples we will discuss in this section are based on the process of timing and scoring a cross country race.

The timing part is pretty simple. When the race starts, several volunteer timers start their stop watches. Each racer is assigned a number and wears a bib displaying that number. As the racers cross the finish line, the timers write down each racer's bib number and the elapsed time when the racer finished. At the end of the race, the timers have a list that looks something like the example shown in Figure 13.13.

From this list, it is easy to determine where each racer placed. A bit more work is required, however, to determine a team's score. A team's score is determined by adding together the placements of that team's four fastest racers. The team with the smallest score wins the race.

To make it easy to determine which runners are from which teams, the local high school league assigns bib numbers in such a way that the last digit on each runner's bib is the number

Place	Bib No.	Elapsed Time
1	81	20:16
2	71	21:32
3	170	22:34
4	31	23:06
5	200	23:08
6	41	23:10
7	73	23:16
8	83	23:29
9	189	23:53
10	20	23:54
11	9	23:56
12	21	24:00
13	259	24:07
14	60	24:20
15	111	24:33

Figure 13.13: List of finishing times for racers

of that runner's team. (Obviously, the county league has less than 11 teams.) Looking at the sample results in Figure 13.13, the bib numbers for the first two runners, 81 and 71, indicate that they belong to team number 1. The fourth and sixth place finishers, numbers 31 and 41, also ran for team 1. Accordingly, team one's score for the race is $1 + 2 + 4 + 6$ for a total of 13. Team zero's runners took third place (bib number 170), fifth place (bib number 200), tenth place (bib number 20), and fourteenth place (bib number 60). Their team score is $3 + 5 + 10 + 14$ for a total of 32.

To explore more ways of manipulating lists, we will consider how we might write a program to assist in the process of compiling race results when given a list of finishing times as shown in Figure 13.13. We assume that there will be a main program, `raceController`, and a `raceStatistics` class. The `raceController` object will inherit `graphicApplication` and provides a user interface designed to enable race officials to enter the data collected by the timers. The `raceStatistics` class uses a list to keep track of the data entered and provides methods to perform operations like determining a particular team's score. We will describe the implementation of the `raceStatistics` class in some detail but we will not explore the implementation of the associated `raceController`.

Our motivation for largely ignoring the contents of the `raceController` class is the desire to focus attention on the subject of this chapter, lists. Our ability to do this, however, illustrates a more general design principle. When writing a program it is desirable to separate those components of the code that determine the interface presented to the user from those that organize and process the underlying data. User interfaces change frequently. If the league purchased an electronic timing system, manual entry of finishing times might be eliminated entirely. The rules for determining which team won, however, would likely remain the same. Separating the implementation of the user interface from the code that manipulates the data collected makes it easier to change the interface without having to revise the entire program.

While we will not describe the details of the user interface provided by the `raceController`, we must discuss its functionality in order to understand the methods that will be required in the `raceStatistics` class. The idea is that as soon as the race ended, the lists of finishing times would be entered into the program. To make this possible, `raceController` might provide two text fields where the user could type a bib number and a time along with an “Enter” button to press after each racer’s time is entered. The `raceStatistics` class would provide an `addRacerBib()time()` method that would take a bib number and time as parameters and add the information to the collection maintained by the `raceStatistics` object.

Once all the data have been entered, the user should be able to press a button to request race results including a list of all the finishing times and bib numbers and a list of team scores. Since there might be situations where one of these reports was desired without the other, the `raceStatistics` class should probably include one method to list individual finish times and another to list team scores. Also, it might be useful to be able to ask how a particular racer did. That is, there should be a method which would take a bib number as a parameter and return the place that runner finished.

13.5.1 Parallel lists vs. lists of Objects

To start, we must consider how to represent the data found in the list of finishing times. In the `radarStats` class, each element we wanted to keep track of was a simple integer, so a list of ints was sufficient. Now the list of items we want to manipulate is not a list of integers but a list of pairs. Each item includes both a bib number and a finishing time.

We could use two lists to represent the list of finishers. One list would be used to hold the bib numbers. The other list would be used to hold finishing times. The lists might be declared as:

```
def bibNumber: List[[Number]]
def elapsedTime: List[[Number]]
```

with the understanding that `bibNumber.at(i)` and `elapsedTime.at(i)` would hold the bib number and finishing time for a single racer. We have two potentially independent lists, but the code that manipulates these lists is designed to ensure that the item associated with a given index in one list is associated with the same index in the other list. When used in this way, we say that the lists are *parallel lists*.

An alternate approach is to define a new class whose objects will represent the complete state of a single item in the list and then to create a list whose elements belong to this new class. Taking this approach, we would define a `racerInfo` class creating objects of type `RacerInfo` to represent racers and make a list of `RacerInfos`.

The code to define such a `RacerInfo` type and `racerInfo` class is shown in Figure 13.14. The class takes as parameters a racer’s bib number and finishing time. These values are associated with defs in the object constructed. Because they are declared to be public, methods of the same name are implicitly created to provide access to their values. In addition, there is a `team` method that determines a runner’s team by extracting the last digit of the runner’s bib number, unless that number is 0, in which case the team is 10.

Given this definition, we can then include in our `raceStatistics` class a definition of the form

```
def racer: List[[RacerInfo]] = emptyList[[RacerInfo]]
```

 to refer to an empty list in which descriptions of all the racers’ finishes may be collected.

```

type RacerInfo = {
  bibNumber -> Number
  finishingTime -> Number
  team -> Number
}

// class to hold info on single racer
class racerInfoNumber (bibNumber!: Number) time (finishingTime!: Number) -> RacerInfo {
  def bibNumber: Number is public = bibNumber!
  def finishingTime: Number is public = finishingTime!
  method team -> Number {
    def teamNumber: Number = bibNumber % 10
    if (teamNumber == 0) then {10}
    else {teamNumber}
  }
}

```

Figure 13.14: Definition of racerInfo class

The racerInfo class provides a way to combine the two pieces of information that describe a racer's finish into a single object so that a collection of such objects can be placed in a single list. In addition, it incorporates in our program's design a concrete specification of the aspects of a racer that are significant. This would be an appropriate design decision even if the new class did not also facilitate the collection of information about many racers into a single list. In situations like this, it is generally much better to define a new class and use a single list than to use parallel lists.

With this scheme in mind, it is now easy to write a method named `addRacerBib()time()` that will take a bib number and a time, create a RacerInfo object to hold this information, and associate this new object with the appropriate list index:

```

method addRacerBib (bib: Number) time (finishTime: Number) -> Done {
  def newRacer: RacerInfo = racerInfoNumber (bib) time (finishTime)
  racer.add (newRacer)
  done
}

```

Exercise 13.5.1 *The code in the following sections assumes that the entries in the racer list are added in the order that the racers finished. Show how to modify the `addRacerBib()time()` method to ensure that this is true. Your version should print an error message if an attempt is made to add a racer out of order. It should then return without actually adding information about the new racer.*

13.5.2 Displaying the Results

We described three ways in which the raceStatistics class should be able to provide information about race results: by listing all the runners in the order that they finished, by returning the

```

method individualResults -> String {
  var results: String := ""

  for (1 .. racer.size) do {place: Number ->
    results := results ++ "{place}. Racer {racer.at (place).bibNumber} with time" ++
      " {racer.at (place).finishingTime}\n"
  }
  results
}

```

Figure 13.15: The `individualResults` method

placement for a particular runner given that runner's bib number, and by listing team scores. We will show methods that perform each of these functions.

Listing the elements in a list

We would like the method that lists all the runners to produce a display that looks something like:

```

1. Racer 81 with time 20:16
2. Racer 71 with time 21:32
3. Racer 170 with time 22:34
4. Racer 31 with time 23:06
5. Racer 200 with time 23:10
6. Racer 41 with time 23:16
7. Racer 73 with time 23:16
.
.
.

```

There are several ways that the program might present this information. It might simply use `print` to write the text on the system output or it might want to include a component in the program's GUI and present the text there. To provide a method flexible enough to support either of these approaches, we will write code that creates a `String` holding the entire display and then returns this `String`.

The code for such a method named `individualResults` is shown in Figure 13.15. The body of the method contains a loop that resembles the loops we used earlier in this chapter. Rather than a loop that used the enhanced `for` to iterate over the elements (e.g., with code like `for (racer) do ...`), we iterate over the indices in the array. The reason is that we need the value of the index, `place`, to print the place of the racer.

Before the loop, we declare and initialize the variable `results` to be an empty string. The body of the loop consists of a single assignment statement that adds the text of one line of the list of results to the `String` associated with name `results`. Each time the loop is executed,

the loop variable `place` refers to an index identifying the entry in the list `racer` that should be added to the list. The code that describes the text that is added to `result` on each iteration is divided into two lines to make it more readable. The material in curly braces inside the string is evaluated and then converted to a string. The `\n` at the end of the string will be converted to new line character when it is printed. Thus each finisher's results will be printed on a separate line if we write `print(stats.individualResults)` when `stats` is generated by `raceStats`

The most important thing to understand about this loop is the degree of similarity between this loop and the list processing loops seen earlier in this chapter. The bodies of these loops are all designed to perform some operation on one element of a list. The loop header ensures that the loop body is executed once for each relevant entry in the list.

Finding a Single Element in the list

We would also like to provide a method to determine where a particular racer placed without displaying the entire list. This method will take a bib number as a parameter and return that racer's placement in the race. To do this, we will write a loop that searches through the elements of the list until it finds the entry with the desired bib number. This loop will be a bit different from the others we have considered. It doesn't always have to process every element of the list. It can stop as soon as the requested bib number is found.

There is one unexpected possibility that we have to account for when writing this loop. The person using the program might make a mistake when typing in the bib number and therefore ask our code to search for something it will never find. We have to decide what value our method should return in this case. We should return something that will make it clear that something went wrong. The method is supposed to return a racer's place. Accordingly, if we can't find the racer, returning something that couldn't possibly be someone's placement is an effective way to signal that a problem occurred. Returning 0, -1 or any other negative value would do. We will use -1.

We will name this method `placement`. This differs from our previous loops in that we can exit the loop as soon as we find the element. After all, once we find it there is no sense continuing to look. Because our `for` loops are designed to execute a fixed number of times, that seems difficult. Of course we could write a `while` loop. However, we show how a `for` loop can be combined with a `return` statement to get the same effect. The code for this method is shown in Figure 13.16.

This version takes advantage of the fact that a `return` statement can be used to terminate execution of a method, returning its argument as the value of the method. Accordingly, when the `if` statement within the body of the `for` loop finds the desired entry in the list, it immediately executes a `return` that specifies the appropriate value for that racer's placement. On the other hand, if the element is not found before the loop goes through all the elements of the loop, then the value -1 will be returned.

Exercise 13.5.2 *Write another version of method `placement` using a `while` loop (and no `return` in the loop) rather than a `for` loop.*

```

method placement (bib: Number) -> Number {
  for (1 .. racer.size) do {place: Number ->
    if (racer.at (place).bibNumber == bib) then {
      return place
    }
  }
  -1
}

```

Figure 13.16: The placement method

Computing Team Scores

We will consider how to write two methods dealing with team scores. First, we will work on a method named `teamScore`. It will take a parameter named `teamNo` that identifies a particular team and return that team's score. Next, we will construct a `teamStandings` method that will compute the scores of all the teams that participated in a meet.

In the local high school cross country meets, a team's score is determined by adding together the places of the team's top four finishers. With this in mind, we will assume that the constant

```
def numberOfScorers: Number = 4
```

has been declared. The exact number of finishers counted actually varies from sport to sport. The value associated with `numberOfScorers` could easily be changed to make our program work correctly for another sport.

Computing a team's score requires a loop that combines aspects of several of the loops we have considered. We will be adding together place values, just as we added together numbers of speeders in Section 13.4.2. We will be looking for specific entries, just as we did when looking for a bib number in the `placement` method. Finally, like the loop in `placement`, this loop does not have to examine every element in the list. It can stop as soon as it has found four racers for the team specified.

We will again use variables to hold information determined by previous iterations of the processing loop. One variable, `racersCounted`, will keep track of how many racers from the specified team have been found so far. The other variable, `score`, will hold the sum of the place values for those racers.

A complete version of `teamScore` is shown in Figure 13.17. The header of the `for` loop is set up to examine all of the racers in `racer`. The `if` statement that forms the body of the loop ensures that only racers associated with the specified team are processed. For each racer that is processed, we add 1 to `racersCounted` and add the racer's place to `score`. If we have found a match, we also check to see if this is the last scorer needed for the team. If so, we return the total score to this point.

Of course, it is possible that for some reason fewer than 4 racers on a team will actually finish. If that is the case – and that will happen if the `for` loop completes examining all of the finishers – the method will return -1, a value that can be recognized as an error indicator,

```

method teamScore (teamNum: Number) -> Number {
  // number of racers in team seen so far
  var racersCountedSoFar: Number := 0
  // Sum of finishing places
  var score: Number := 0

  // add up all the scores that count on this team
  for (1 .. racer.size) do { place: Number ->
    if (racer.at (place).team == teamNum) then {
      racersCountedSoFar := racersCountedSoFar + 1
      score := score + place
      if (racersCountedSoFar == numberOfScorers) then {
        // got places for all that can count
        return score
      }
    }
  }
  // didn't get enough finishers
  -1
}

```

Figure 13.17: The `teamScore` method computes the score for one team

since it could not possibly be an actual team's score.

Our next task is to write a method named `allTeamScores` that will determine the scores for all the teams that competed. That method will take a parameter `numberOfTeams`, indicating the number of teams who will be competing in the race.

Obviously, this method cannot return a single score as its result. Instead, we will design it to return a list of integers containing one element for each of the ten possible teams. The element of the list associated with the team number will refer to that team's score.

We could use `teamScore` to write the `allTeamScores` method. We would simply write a loop that would call `teamScore` once for each team number from 1 to `numberOfTeams`. If you think about it for a moment, you will realize that this would make the computer do a lot of unnecessary work. First, we would call `teamScore` to determine the score for team 1. This would involve checking the team number of all of the racers up to the fourth racer from team 1. Next, we would do the same thing for team 2. Unless the first four racers for team 2 came in before all the racers from team 1, the computer will again check some of the team 1 racers to see if they are from team 2, even though they have already been identified as being from team 1.

The source of inefficiency here is the fact that such code would instruct the computer to make a separate pass through all the entries at the beginning of the `racer` list for each team. This is unnecessary. All of the team scores can be computed at the same time during one pass through the list. To do this, we need to keep track of how many racers we have found from each of the 10 teams and the accumulated score for each team. That is, we need a

```

method allTeamScores (numberOfTeams: Number) -> List[[Number]] {
  def numTeamScorers: List[[Number]] = emptyList[[Number]]
  def teamScoreTotal: List[[Number]] = emptyList[[Number]]

  // create a slot in each list for each team and initialize to 0
  for (1 .. numberOfTeams) do {teamNumber: Number ->
    numTeamScorers.add (0)
    teamScoreTotal.add (0)
  }

  // add up all the scores that count on this team
  for (1 .. racer.size) do {place: Number ->
    def teamNumber: Number = racer.at (place).team
    if (numTeamScorers.at (teamNumber) < numberOfScorers) then {
      // more scorers can count so update scores and count
      def newTotal: Number = teamScoreTotal.at (teamNumber) + place
      teamScoreTotal.at (teamNumber) put (newTotal)
      def newScorers: Number = numTeamScorers.at (teamNumber) + 1
      numTeamScorers.at (teamNumber) put (newScorers)
    }
  }

  // check got enough scorers!
  for (1 .. numberOfTeams) do { teamNumber: Number ->
    if(numTeamScorers.at (teamNumber) < numberOfScorers) then {
      teamScoreTotal.at (teamNumber) put (-1)
    }
  }
  teamScoreTotal
}

```

Figure 13.18: The teamStandings method

collection of team scores and “team members counted so far” variables. We can do this easily by replacing each of these simple variables used in `teamScore` with lists containing `teamNum` elements:

```

var racersCountedSoFar: Number := 0
var score: Number := 0

```

can be replaced by the declarations

```

def numTeamScorers: List[[Number]] = emptyList[[Number]]
def teamScore: List[[Number]] = emptyList[[Number]]

```

in the `teamStandings` method.

The definition of the `allTeamScores` method is shown in Figure 13.18. The method has three loops. The first adds a slot with value 0 for each team competing. The second looks through the list of finishers and tries to compute scores for all the teams that competed.

Three statements in the body of the `this` loop do the bulk of the work. The assignment `def teamNumber: Number = racer.at (place).team`

extracts the team number for the current racer. The two statements

```
def newTotal: Number = teamScore.at (teamNumber) + place
teamScore.at (teamNumber) put (newTotal)
def newScorers: Number = numTeamScorers.at (teamNumber) + 1
numTeamScorers.at (teamNumber) put (newScorers)
```

add this racer's placement to the team score and then increase the count of the number of racers from this team that have been seen. These statements are placed in an `if` statement to ensure that at most `numberOfScorers` racers from each team are considered when determining the team's score.

The final loop then checks the elements of the `numTeamScorers` list looking for any team that did not have at least `numberOfScorers` racers finish. Such teams receive a score of -1. Finally, the list containing the scores is returned as the method's result.

Exercise 13.5.3 *The version of the `allTeamScores` method shown in Figure 13.18 may do a bit more work than necessary. The `for` loop in this method looks at every single entry in the `racer` list. Typically, there will be many racers who don't score at all who finish after the slowest racer who scores. That is, once the loop has found enough racers from each of the teams to compute a score for every team it could stop without looking at the remaining entries in the list.*

Show how to modify our definition of `allTeamScores` to take advantage of this observation. You will have to change the first loop so that it keeps track of how many team scores have been completely determined and stops when this number equals `teamCount`.

13.6 Adding and Removing Elements

In the preceding section, we assumed that the finishing times and bib numbers were supplied to our program in the desired order and that they were entered correctly. It is not always possible to make such assumptions about the data processed by a program that uses lists. In fact, it is not at all clear that it is reasonable to make this assumption about our finishing time data.

Having been volunteer timers, we know that mistakes get made at races. At the end of one race, for example, someone noticed that one racer somehow finished twice! (i.e. that person's bib number was misread and written down incorrectly so that it was actually the number of another racer). To preserve faith in the system, the coaches do what they can in the back room to repair such recording errors. Therefore, if we want a program that really assists in the process of compiling race results, it should facilitate making corrections to deal with errors made by the timers and errors made while entering the data they recorded. In the context of learning about lists, the most interesting types of corrections to consider are adding a new racer to the list and deleting an incorrect entry from the list.

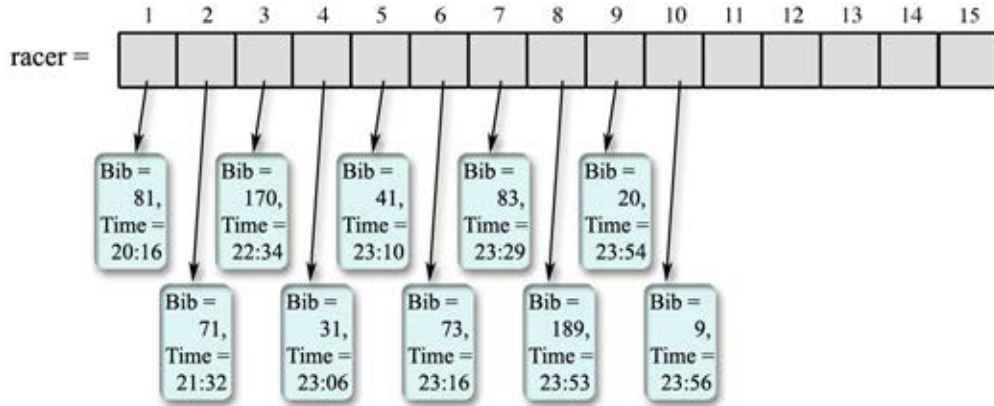


Figure 13.19: racer list with a missing element.

13.6.1 Adding an Element to an Ordered list

Suppose that a person using our program gets distracted while entering the finishing times and skips one of the racers listed. Such mistakes are almost certain to happen, so we should think about how our program could provide the ability to add an omitted entry at the correct position in the list.

We will not worry about the user interface we would have to implement to make such insertions possible. Instead, we will focus on the implementation of a new method of the `RaceStatistics` class designed to support the addition of additional entries to the list. Let us assume this method is named `addRacerBib(time())position()` to highlight how it differs from the `addRacerBib(time())` method already included in the class.

It will help to think about the steps required to perform a specific insertion before trying to construct the code for this method. Suppose that while entering the times shown in Figure 13.13, a person using our program skips the fifth finisher and does not notice the mistake until six more finishing times have been entered. Figure 13.19 depicts the information that would be recorded in the list.

This figure is very similar to Figure 13.20 which depicts the state of the list after the first 10 finishing times are entered correctly. If one compares the two diagrams closely, however, it becomes clear that the `RacerInfo` objects associated with list elements 4, 5, 6, 7, 8, and 9 are all wrong! We can't fix the problem by just adding the correct entry at position 4. We have to fix all the incorrect entries. The racer associated with the 4th position of the list should be associated with the 5th position in the list, the racer associated with the 5th position should be associated with the 6th position, and so on.

The task of inserting the new element can be broken into two subtasks. First, we have to move all the entries associated with list indices greater than or equal to 4 over one position. The diagram shown in Figure 13.21 suggests the state of the list after this step is complete. Next, we would create a new `RacerInfo` object describing the omitted racer and associate it with index value 4. Figure 13.22 describes the desired state of the list and variable after the insertion is complete.

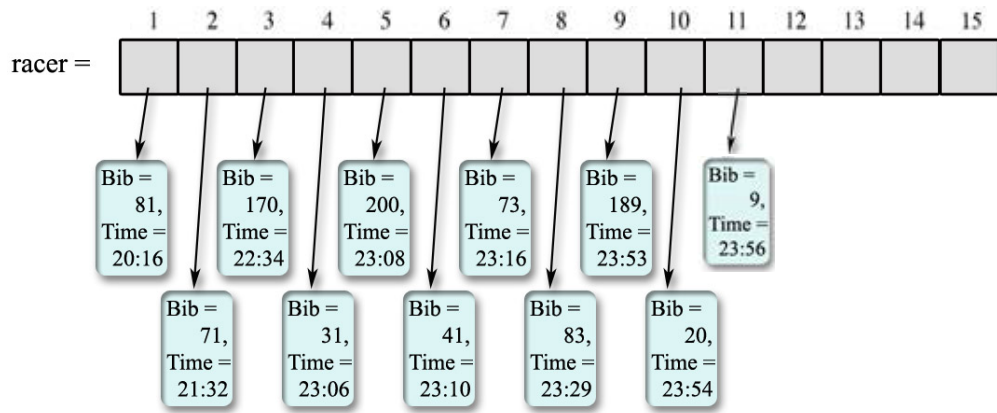


Figure 13.20: State of racer list after adding all 11 racers

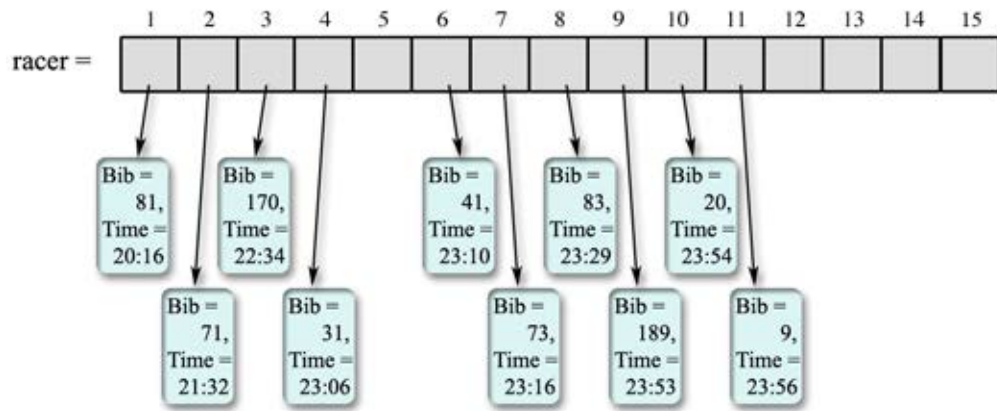


Figure 13.21: racer list with entries after omission shifted right.

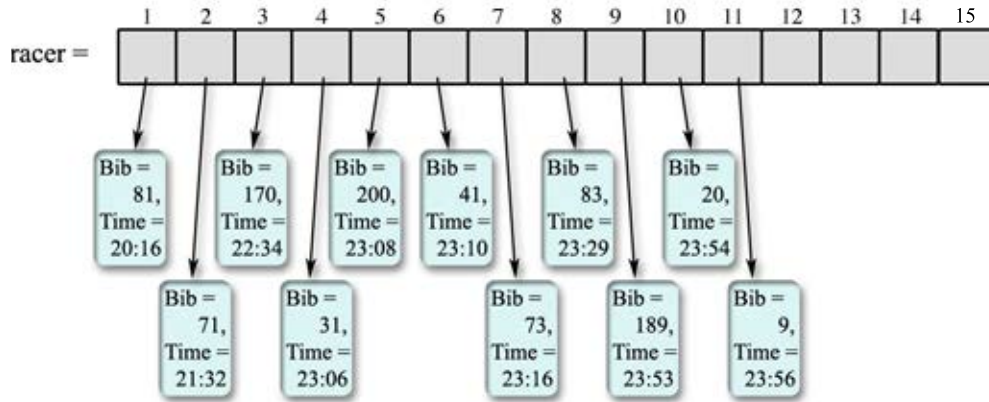


Figure 13.22: racer list after insertion is complete.

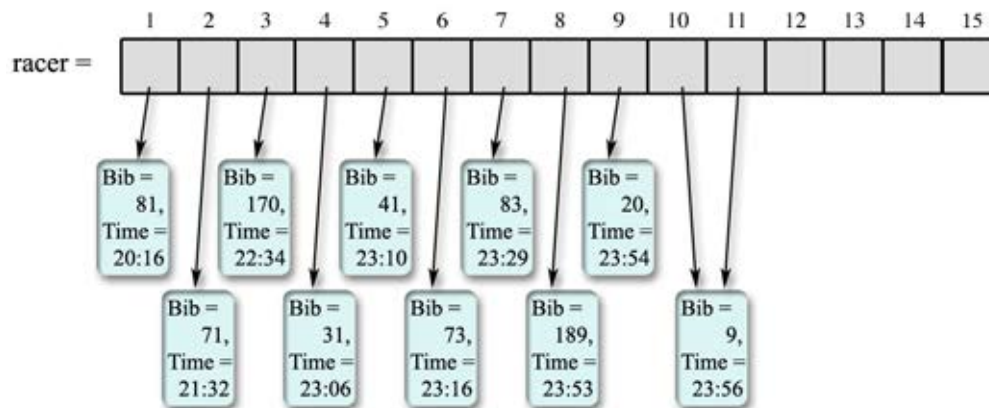


Figure 13.23: racer list after moving element 10.

Moving all the misplaced racers over one position in the list is the hard part. There is no special Grace instruction to move a whole collection of list entries at once. Instead, the RacerInfo objects will have to be associated with new list indices one by one using `at(put())` method requests.

Let's start by seeing how we can use an `at(put())` method to move the last element in the list, the RacerInfo object for the racer with bib number 9, from its initial position in the list, position 10, to the next position to the right, position 11. To do this, we would use the assignment

```
racer.at(11) put (racer.at(10))
```

This assignment tells Grace to make the 11th position of the list refer to the racer already associated with the 10th position of the list. As a result, after this assignment, both the 10th and 11th positions of the list would refer to the same racer. The diagram in Figure 13.23 depicts the state of the list after this assignment.

This situation, shown in Figure 13.23, is probably not quite what you thought we had

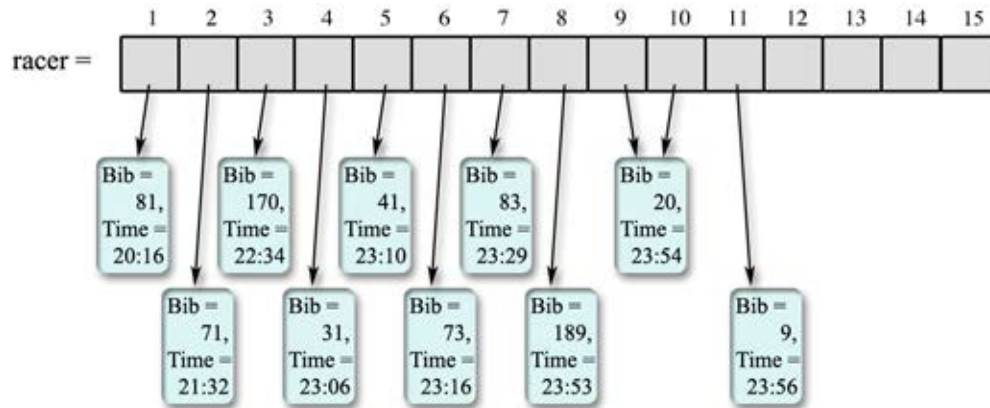


Figure 13.24: racer list after moving elements 9 and 10.

in mind when we said we wanted to move the entry from position 10 to position 11. You probably were expecting only the entry in position 11 to refer to the racer when we were done.

To see how we reach the expected state, consider the next step in the process. The assignment

```
racer.at (10) put (racer.at (9))
```

would be used to move the object describing the racer wearing bib 20 from entry 9 to entry 10. At the same time, this assignment would have the effect of ensuring that the racer with bib number 10 is referred to only by list entry 11. After executing this assignment, the list will look like the picture in Figure 13.24. While only list element 11 refers to the racer with bib number 9, racer 20 is temporarily associated with two list index values, 9 and 10. The association of racer 20 with list element 9 is undesired, but the next step,

```
racer.at (9) put ( racer.at 8 )
```

will eliminate this unwanted association.

The complete set of at()put() method requests needed to move the last six racers over one position is

```
racer.at (11) put (racer.at (10))
racer.at (10) put (racer.at (9))
racer.at (9) put (racer.at (8))
racer.at (8) put (racer.at (7))
racer.at (7) put (racer.at (6))
racer.at (6) put (racer.at (5))
```

Executing these assignments would move all the necessary elements of the list over by one position as shown in Figure 13.25. Note that this figure is slightly different from the goal depicted in Figure 13.21. Entry 4 still refers to the racer that has been moved to entry 5. This reference must be replaced by a reference to a new RacerInfo object for the added racer to complete the insertion process.

The assignments shown in the list above are quite similar to one another. They are all of the form

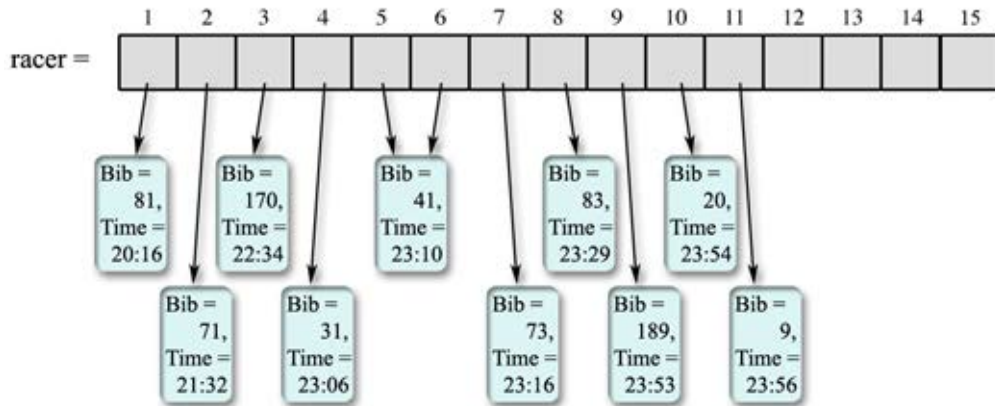


Figure 13.25: racer list after moving 6 elements over by one.

```
racer.at (racerIndex + 1).put (racer.at (racerIndex))
```

This makes it easy to write a loop to perform the assignments rather than actually writing the assignments out in our code. The loop will simply execute the `at()put()` method requests shown above for all values of the variable `position` from the last element of the list down to but not including the index of the position where the new element will be inserted. The index where the new element is to be inserted will be provided as a parameter to the `addRacerBib()` `time()position` method. Assuming we name that parameter `insertionPos`, the loop needed to move the required elements over by one position would be:

```
for ((insertionPosition .. (racer.size)).reversed) do {racerIndex: Number ->
  racer.at (racerIndex + 1).put (racer.at (racerIndex))
}
```

This loop is different from the other list loops we have seen in this chapter. It goes backwards. The other loops have all started with the smallest index and worked toward the largest by increasing the index variable by 1 with each iteration. Now we want to go from the largest to the smallest.

Rather than providing the `for` loop with a range like `5 .. 10`, we are using a new construct to build a range object that goes backwards. In general `(small .. big).reversed` gives the reverse of the range `small .. big`. Thus `r(10 .. 5).reversed` provides the numbers 10, 9, 8, 7, 6, 5, in that order. As a result, the loop in our code to move the racers starts at `racer.size` and continues decreasing the loop variable `racerIndex` by one with each iteration until it gets to `insertionPosition` for the last iteration.

Consider what would happen if we performed the assignments with the list indices in increasing order. We would first execute the assignment

```
racer.at (6) put (racer.at (5))
```

This would make the two list indices, 5 and 6, refer to the same `RacerInfo` object. Assuming we started with the list in the configuration shown in Figure 13.19, the diagram in Figure 13.26 shows the result of starting with this assignment. The list indices 5 and 6 are both associated with racer 41.

This isn't too surprising. When we started with the assignment

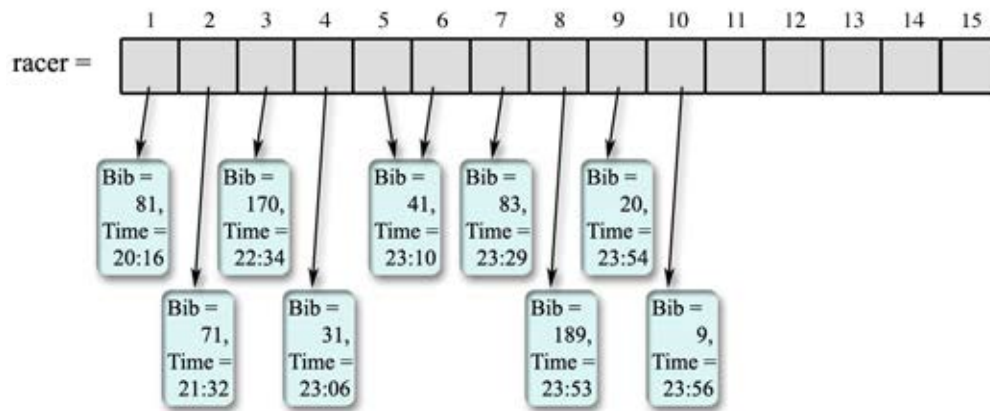


Figure 13.26: racer list after executing racer.at (6) put (racer.at (5))

racer.at (10) put (racer.at (9))

we also ended up with two list entries associated with a single racer, but a surprising thing has happened. The racer object that entry 5 referred to before the assignment, the object for racer with bib 73, is lost. No list entry refers to this object.

Things get worse if we continue to execute the at()put() method request for the next larger pair of index values:

racer.at (7) put (racer.at (6))

This makes list entry 7 refer to the same object as entry 6 as shown in Figure 13.27. Now three entries refer to the same object and two racers have been lost!

If we continue executing the assignments in increasing order of index values we will end up leaving the list in the state shown in Figure 13.28. All of the entries after the insertion point now refer to the element initially found at the insertion point, and all of the objects after this point have been lost. Clearly, we have to use a loop that works from the largest subscript to the smallest.

To complete the addRacerAtPosition method, we need to combine the loop that moves each of the elements after the insertion point with code to add the new entry. Adding this code yields a complete implementation for addRacerAtPosition as shown in Figure 13.29. Once the elements have been moved out of the way, the code to insert the new element is similar to that seen in addRacerBib()time() except that the parameter insertionPos is used as the index for the new element's position. We also make sure that the new racer's position is contiguous to the existing racers.

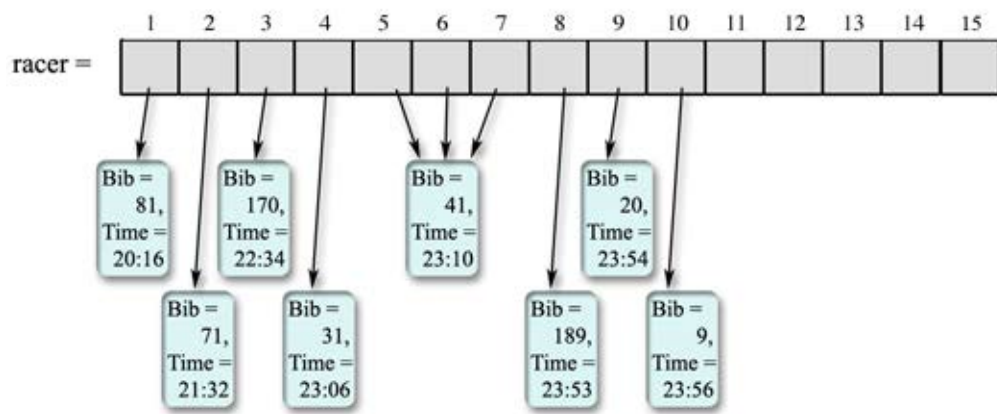


Figure 13.27: list after executing racer.at (7) put (racer.at (6))

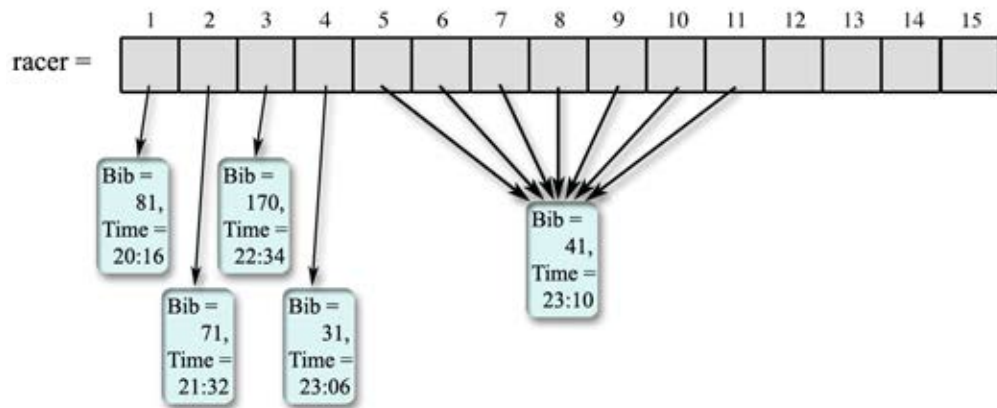


Figure 13.28: Where did all the racers go?

```

method addRacerBib (bib: Number) time (finishTime: Number) position (insertionPosition: Number) ->
Done {
  if ((insertionPosition >= 1) && (insertionPosition <= (racer.size + 1))) then {
    for ((insertionPosition .. racer.size).reversed) do {racerIndex: Number ->
      racer.at (racerIndex + 1).put (racer.at (racerIndex))
    }
    def newRacer: RacerInfo = racerInfoNumber (bib) time (finishTime)
    racer.at (insertionPosition) put (newRacer)
  } else {
    print "Insertion position {insertionPosition} illegal"
  }
}

```

Figure 13.29: The addRacerAtPosition method

13.6.2 Removing an Element from a list

Programs that manipulate data in lists often need to update the collection by removing a specific item. It is not hard to imagine such a need arising in the context of our race results program. An error might be made in which the information for a single racer was entered twice, or after the results were entered, a racer might be disqualified for some infraction reported after the finish. To address these situations, we will consider how to implement a `removeRacerAt` method which will remove the racer at a particular position from the list.

Like the process of adding an element, the task of removing an element requires the ability to shift a sequence of elements over one position. In Figure 13.20, we showed a diagram of the state of the list after the first 10 recorded finishing times from Figure 13.13 were correctly entered. Suppose, starting from that configuration, we wanted to delete the entry in position 7.

We could try just replacing that value by some dummy value that didn't correspond to a racer. This would leave the list in the state shown in Figure 13.30. The code we have written thus far, however, depends on the assumption that all of the elements of the list that refer to racers will be contiguous. Leaving element 7 empty violates this assumption. So to complete the removal we have to shift all of the entries after position 7 to the left one place.

In this case, the required shift can be performed by the statements

```

racer.at (7) put (racer.at (8))
racer.at (8) put (racer.at (9))
racer.at (9) put (racer.at (10))

```

On the other hand, the last assignment in the list above sets element 9 to have the same value as element 10. If this last assignment was omitted from the list above, then the diagram in Figure 13.31 would depict the final state of the list. Both elements 9 and 10 refer to the same entry. In most cases, this would not effect the correctness of our program. If we wish to avoid having the last two slots refer to the same element, we must invoke method `removeLast` to eliminate the last slot in the list.

As in the method to add a racer, we will want to create a loop to execute the sequence of

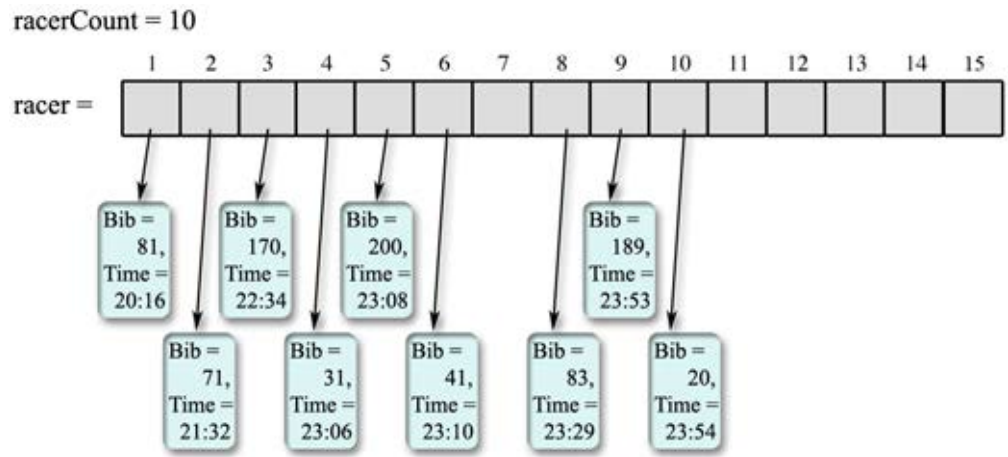


Figure 13.30: list after putting dummy value in slot 7

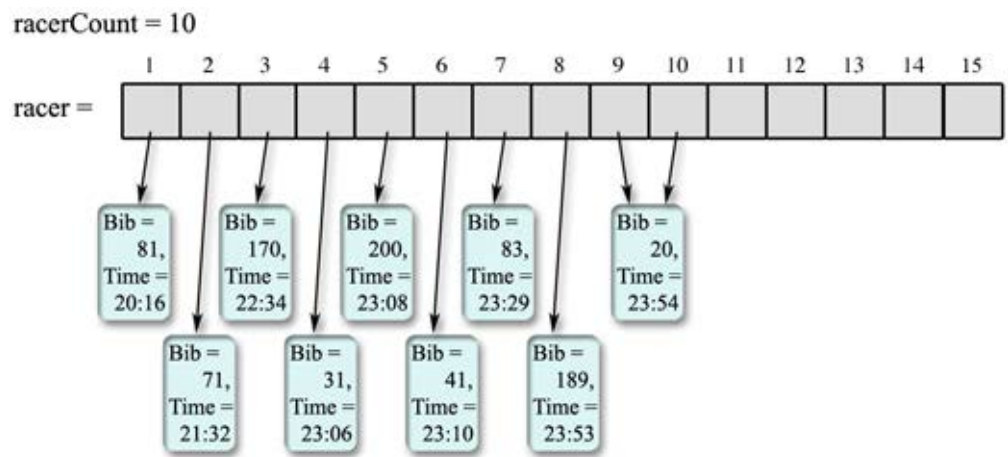


Figure 13.31: racers list with last element duplicated

```

method removeRacerAtPosition (position: Number) -> Done {
  if ((position >= 1) && (position <= racer.size)) then {
    for (position .. (racer.size - 1)) do {place: Number ->
      racer.at (place) put (racer.at (place + 1))
    }
    racer.removeLast
  }
}

```

Figure 13.32: The removeRacerAtPosition method

```

method removeRacerAtPosition (position: Number) -> Done {
  if ((position >= 1) && (position <= racer.size)) then {
    for ((position .. (racer.size - 1)).reversed) do {place: Number ->
      racer.at (place) put (racer.at (place + 1))
    }
    racer.removeLast
  }
}

```

Figure 13.33: The removeRacerAtPosition method with the loop (incorrectly) reversed

assignments necessary to shift the appropriate elements over by one when an entry is deleted from a list. The list element set by the first assignment should be the element associated with the index of the element being deleted and the index of the last element set should be equal to the size of the list *after* the element is removed, which is the value (before deletion) of `racer.size - 1`. Just as in the code that shifted elements during an addition, the order in which these assignments is executed is critical. The loop must first execute the assignment with the smallest subscript values and then increase the values used by one on each iteration.

The complete code for `removeRacerAtPosition` is shown in Figure 13.32. The entire body of this method is included within an if statement that tests to make sure the operation requested is possible. In this case, the test made is that the position provided as a parameter actually refers to an existing element in the list.

Exercise 13.6.1 *In Section 13.6.1, we explained that it was important that the loop in the `addRacerBib()time()position()` method start at the end of the list and work its way down to the point of insertion. In the `removeRacerAtPosition` method, however, we have used a loop that steps through the elements to be moved in the opposite direction. To appreciate why this is necessary, draw a diagram like the one shown in Figure 13.20 showing how the list would look if we invoked `removeRacerAtPosition` with parameter value 7 after redefining `removeRacerAtPosition` as shown in Figure 13.33.*

13.7 Summary

Lists provide a convenient and efficient mechanism for manipulating collections of related data items. The features of the Grace language that support lists are actually fairly simple. We can declare list variables and parameters, create lists using list classes or methods, associate lists with names through assignments or parameter passing, write methods that return lists, and, finally and most importantly, access individual members of an list using indexed variables. These simple features, however, can be used to organize collections of data in ways that range from simple to quite complex.

The challenge for the beginner is to learn to use Grace's list features effectively. In this chapter, we have introduced several common and important techniques for using lists to organize data. We have seen how to write loops that process all the elements of an list independently and examples of loops that instead collect information about the collection as a whole. We also showed how to search a list to find a particular element of the collection. We learned how to add and remove elements from such collections.

Our exploration of lists continues in the next chapter where we discuss lists whose elements are themselves lists.

13.8 Chapter Review Problems

Exercise 13.8.1 *The following table shows the percentage of the total land area of the Earth located in each of the seven continents.*

<i>Continent</i>	<i>Percent of Total Land Area</i>
<i>Africa</i>	<i>20.3 %</i>
<i>Antarctica</i>	<i>8.9 %</i>
<i>Asia</i>	<i>30.0 %</i>
<i>Australia</i>	<i>5.2 %</i>
<i>Europe</i>	<i>6.7 %</i>
<i>North America</i>	<i>16.3 %</i>
<i>South America</i>	<i>8.9 %</i>

- a. *Provide declarations for two parallel lists named `continentalArea` and `continentName` designed to hold the information in this table. Include list initializers in each declaration so that the elements of the list are associated with the data provided in the table.*
- b. *Write a loop which will print a table of the actual area of each continent given an estimate of the total land area of the Earth. Assume that the total land area is associated with a constant named `totalArea`. Don't bother to format your output as a table with a heading or to deal with the fairly ugly way Grace displays fractional values. For example, using a fairly accurate value of 57 million square miles for `totalArea` your loop's output might look like:*

```
Africa 1.1571e+7
```

```

type Continent = {
    name -> String
    area -> Number
}

class continent.named(aName: String, theArea: Number) -> Continent{
    method name -> String { aName }
    method area -> Number { theArea }
}

```

Figure 13.34: A class to represent continents

```

Antarctica 5073000.000000001
Asia 1.71e+7
Australia 2964000.0000000005
Europe 3819000.0
North America 9291000.0
South America 5073000.000000001

```

- c. Assuming that the class named `continent` is defined as shown in Figure 13.34, provide a declaration for a list of continents to hold the information in the table above. Include the initialization code to associate the elements of the list with appropriate `Continent` objects.
- d. Write a loop to print the estimated areas of each of the continents using the list of `Continent` objects rather than the two parallel lists defined for part (a).

Exercise 13.8.2 Show the output that would be produced when the method `mystery` shown in Figure 13.35 is invoked.

Exercise 13.8.3 Consider how one might represent a polynomial like

$$4x^3 + 2.3x^2 + x + 7$$

by placing its coefficients in a list of doubles. A natural approach is to place the coefficient of the term including x^i in the $i + 1$ st element of the list. For example, the polynomial above could be represented by a list declared as

```

def coef: List[[Number]] = list[[Number]] [7, 1, 2.3, 4 ]

```

The skeleton of a class named `polynomial` designed to represent polynomials using this scheme is shown in Figure 13.36. The constructor for this class takes the degree of the polynomial, the highest power of x included in the terms of the polynomial, as a parameter. The `setCoefAt()``put()` method can be used to specify the coefficients of each term. For example, to create and initialize a `Polynomial` object representing the polynomial used as an example above we would declare

```

def sample: Polynomial = polynomialDegree (3)

```

and then execute the invocations

```

method mystery -> Done {

  def report: List[[Number]] = list[[Number]] [5, 4, 10, 4, 6, 3, 4]

  var turn: Number := 1
  var shot: Number := 1
  var lastTotal: Number := 0

  while {(shot <= report.size) && (turn <= 10)} do {
    var increment: Number := report.at (shot) + report.at (shot+1)

    if (increment >= 10 ) then {
      increment := increment + report.at (shot + 2)
    }

    if ( report.at (shot) < 10) then {
      shot := shot + 1
    }

    lastTotal := lastTotal + increment

    print( "At {turn} you have {lastTotal}")

    shot := shot + 1
    turn := turn + 1
  }
}

mystery

```

Figure 13.35: Method `mystery` for exercise 13.8.2

```

type Polynomial = {
  setCoefAt (i: Number) to ( value: Number) -> Done
  ...
}

class polynomialDegree(d: Number) -> Polynomial {

  // coef.at (i+1) is the coefficient of x to the ith power
  def coef: List[[Number]] =emptyList

  // initialize all values to 0
  for (0..d) do {i: Number -> coef.add(0)}

  // set the value of the coefficient of (x to the ith power)
  method setCoefAt(i: Number) to ( value: Number) -> Done {
    if ((i >= 0) && (i < coef.size)) then {
      coef.at (i + 1)put(value)
    }
  }
  // other methods would be included below
  ...
}

```

Figure 13.36: Skeleton of a `Polynomial` class for Exercise 13.8.3


```

sample.setCoefAt (0) to (7)
sample.setCoefAt (1) to (1)
sample.setCoefAt (2) to (2.3)
sample.setCoefAt (3) to (4)

```

One method that should be included in the definition of such a polynomial class is a method to evaluate the polynomial at a given value of x . Four possible definitions of such a method are provided below. Some of them function correctly, while others contain errors. Identify the correct method(s) and indicate how the others would fail.

a. **method** `eval (x: Number) -> Number {`
 var `result: Number := coef.at (1)`
 var `power: Number := x`

```

    for (2 .. coef.size) do {i: Number ->
      result := result + power * coef.at (i)
      power := power * x
    }
    result
  }

```

b. **method** `eval (x: Number) -> Number {`
 var `result: Number := 0`
 var `power: Number := x`

```

    for (1 .. coef.size) do {i: Number ->
      result := result + power * coef[i]
      power := power * x
    }
    result
  }

```

c. **method** `eval (x: Number) -> Number {`
 var `result: Number := 0`
 var `power: Number := 1`

```

    for (1 .. coef.size) do {i: Number ->
      result := result + power * coef[i]
      power := power * x
    }
    result
  }

```

d. **method** `eval(x: Number) -> Number {`
 var `result: Number := 0`
 var `power: Number := 1`

```

    for (coef) do {cVal: Number ->
      result := result + power * cVal
      power := power * x
    }
    result
  }

```

13.9 Programming Problems

Exercise 13.9.1 In exercise 13.8.3 we described parts of a *Polynomial* class. In that exercise we provided code for a constructor, a method to set its coefficients, and a method to evaluate a polynomial at a particular value.

Complete the *Polynomial* class by implementing the methods described below. For the purpose of presenting examples to illustrate how the methods should behave, assume that we have declared two *Polynomial* variables *p* and *q* and executed the code

```
def p: Polynomial = polynomialDegree (2) //Create a 2nd degree polynomial}
def q: Polynomial = polynomialDegree (3) // Create a 3rd degree polynomial}
```

```
p.setCoef.at (0) to (3)
p.setCoef.at (2) to (1)
```

```
q.setCoef.at (3) to (4)
q.setCoef.at (2) to ( 2.5)
q.setCoef.at (1) to (1)
q.setCoef.at (0) to (7)
```

so that *p* represents the polynomial $x^2 + 3$ and *q* represents

$$4x^3 + 2.5x^2 + x + 7$$

The methods you should define are

method *asString* \rightarrow *String*

This method should return a *String* representing the polynomial in a form suitable for human consumption. Since superscripts are not available, use the character “^” to denote exponentiation. Thus, invoking the *asString* method on the *Polynomial* *q* should produce a string like

$$4.0x^3 + 2.5x^2 + 1.0x^1 + 7.0x^0$$

When possible, simplify the *String* returned by eliminating unnecessary terms. For example, it would be best if *q.asString* returned

$$4.0x^3 + 2.5x^2 + x + 7.0$$

and *p.asString* returned

$$x^2 + 3.0$$

method *plus* (*addend* : *Polynomial*) \rightarrow *Polynomial*

This method should return the *Polynomial* which is the sum of the polynomial represented by the *Polynomial* object on which it is invoked and the *Polynomial* passed as a parameter to the method. For example,

```
def r: Polynomial = p.plus (q)
print (r.asString )
```

should display

$$4.0x^3 + 3.3x^2 + x + 10.0$$

method *times* (multiplicand: Polynomial) → Polynomial

This method should return the Polynomial which is the product of the polynomial represented by the Polynomial object on which it is invoked and the Polynomial passed as a parameter to the method. For example,

```
def r: Polynomial = p.times(q)
print(r.asString)
```

should display

$$4.0x^5 + 2.5x^4 + 13.0x^3 + 14.5x^2 + 3.0x + 21.0$$

Hint: In order to write these you will find it useful to first define the following methods:

```
// return the degree of the polynomial
degree → Number
```

```
// return the coefficient of the ith power of x
coefAt (i: Number) → Number
```

You will find these useful in implementing the methods *plus* and *times*.

Exercise 13.9.2 Write a program that implements the following puzzle-like test of a user's visual acuity. The program should display an eight by eight grid of rectangles of different shades of gray as shown in Figure 13.37 (if you want to make it look prettier than our version, you can use different colors instead of shades of gray). Each shade that appears in your grid should appear exactly twice. Each rectangle in your grid should be surrounded by a black framed rectangle so that the gray patches are clearly separated from one another.

The goal of this puzzle is to identify the pairs of rectangles that are the same color. The user of your program will do this by using the mouse to identify rectangles that appear to match. When the user clicks on a rectangle to identify it as the first member of a matching pair, your program should highlight the selected rectangle by turning the framed rectangle that surrounds it red. In Figure 13.37, the rectangle containing the mouse cursor has just been selected and the frame has been painted red (making it appear light gray in the figure).

If the next rectangle the user clicks on is exactly the same shade as the first, they should both be removed from the display, revealing a pattern drawn underneath the rectangles as shown in Figure 13.38. The pattern we have drawn is simply a set of vertical lines. You could use any pattern you want or even place an image in the background. If the second rectangle the user selects does not exactly match the first, the frame around the first rectangle selected should be set back to black and the user should be allowed to try again.

Each time the user clicks, the program should add one to the user's "score" and display this score at the bottom of the window. Figure 13.39 shows the state of the display after many matches have been found. The goal is to find all the matching pairs revealing the entire background with as few clicks as possible.

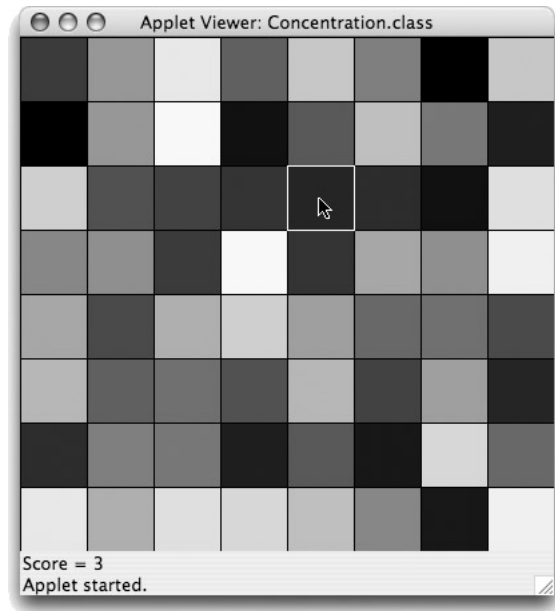


Figure 13.37: Matching patches program after one patch is selected

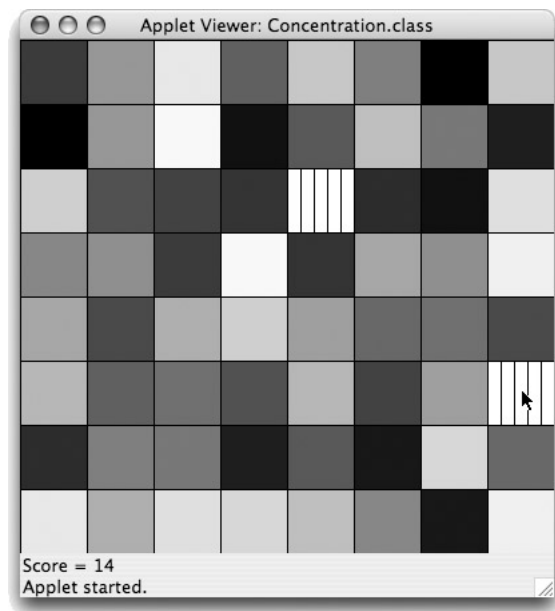


Figure 13.38: Matching patches program after one pair has been identified

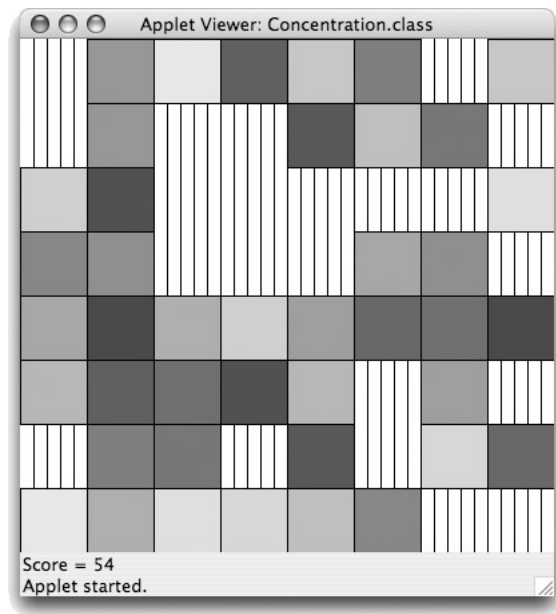


Figure 13.39: Matching patches program after many pairs have been identified

Chapter 14

Multidimensional lists

In Chapter 13 you learned that a list is a collection of values or objects, where each member of the collection is numbered by an index value. Lists give us a natural way to represent collections of objects or values in the real world – for example, pages in a text book, speeders per hour in a day, or the order of racers completing a race.

In this chapter, we explore *lists of lists*, commonly referred to as *two-dimensional lists*. You already know that a list can represent a collection of any type of value or object. So, naturally, a list can represent a collection of lists. The world is filled with examples of collections such as these. Consider, for instance, a monthly magazine. We number the monthly editions as well as the pages of each. Now consider a calendar. We number the months as well as the days of each month.

In the next section, we introduce a calendar program for which a two-dimensional list is a natural data structure. In the context of this example, we discuss the declaration and construction of two-dimensional lists, and how to make associations between index values and members of a two-dimensional list. We then go over common algorithms for traversing, i.e., walking through, a two-dimensional list in order to process the information contained there.

We then turn our attention to *matrices*. In a two-dimensional list, the lists in the collection need not be of the same size. In some situations, however, all lists in the collection do have the same size. A chess board is a good example. If you think of each row of a chess board as a list of eight elements, then the entire board can be represented by a list of these rows. This type of two-dimensional list is called a *matrix*. While matrices are really just two-dimensional lists, they are conceptually a bit different and deserve separate treatment.

14.1 General Two-Dimensional Lists

We will explore two-dimensional lists through an example of an interactive calendar manager. You have probably seen or used one of these. A sample user interface for such a calendar manager is shown in Figure 14.1. This program allows the user to enter the description of an event that will occur on a given date.

If our calendar is meant to represent a full year of days, then we need the ability to represent up to 366 (in case of leap years) strings that describe daily events. You might imagine using a 366-element list to do so. But is this conceptually the best choice? While it

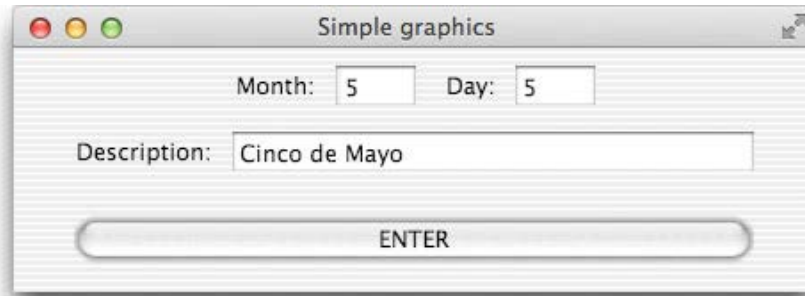


Figure 14.1: Interface used to enter new calendar events

is true that a year is made up of 365 or 366 days, we tend not to describe any individual date as “day 364” or “day 32”. Instead, we describe a date by giving the month and then the day within that month, as in 12/30 or 2/1. So let’s develop a data structure that serves as a better representation of this idea. Let’s think of each month as a list of strings that represent daily events. A year then is a 12-element collection of months. That is, a year is a list of months, each of which is a list of `Strings`.

14.1.1 Declaring a list of lists

In writing our calendar program, we will define a class `yearlyCalendar` that will describe a full year of daily events. As we just discussed, we will think of a year as a 12-element list of months, each of which is a list of `Strings` that describe the daily events. To declare `dailyEvent` as the name of this list of lists of `Strings`, we say:

```
def dailyEvent: List[[List[[String]]]] = emptyList[[List[[String]]]]
```

Since `dailyEvent` is meant to be a list of `String` lists, you simply wrap an extra `List[[]]` around the element type `List[[String]]`.

14.1.2 Creating a list of lists

To construct a list of months, each of which will refer to a list of events corresponding to the days of the month, will take a bit of work. See Figure 14.3.

We know that there are 12 months in a year, so `dailyEvent` should represent 12 months (i.e., lists of `Strings`). We initialize it as an empty list holding elements that are lists of strings. To create and insert initial values in the actual 12-element list we write a `for` loop. Each time through the loop we create a list of default events (“-”), one for each day of the month, and then add the list representing days to the month list. We use the confidential method `getDays` to return the number of days of each month.

After this construction, we have a 12-element list, each of which contains a list of “-”s, one for each day of that month, as illustrated in Figure 14.4. Notice that there are 12 rows



Figure 14.2: A 12-element list. Each element in the list has the potential to refer to a list of strings.

in the figure, and each row has from 28 to 31 elements, representing the number of days in each month.

14.1.3 Indexing a list of lists

Say that a user of our calendar program wishes to enter the following information

1/15 Spring semester starts

That is, the month entered is 1; the day entered is 15; and the event description is "Spring semester starts". We want to record this information in the two-dimensional list, `dailyEvent`. Specifically, we want the fifteenth entry in the first row of `dailyEvent` refer to the string "Spring semester starts".

The slot we wish to update is the 15th element of the first list. The first list can be referenced by `dailyEvent.at(1)`. We can access the 15th element of this list by writing `dailyEvent.at(1).at(15)`. However, we don't want to just access this element, we want to update the value of that element. We can accomplish the update by writing

`dailyEvent.at(1).at(15).put("Spring semester starts")`

That is, writing `dailyEvent.at(1)` gives us that first list. If we make the method request `at(15).put("Spring semester starts")` to that element, then we will have accomplished the update. Make sure you understand why there is a "." before each `at`, but not before the `put`.

```

def dailyEvent: List[[List[[String]] ]] = emptyList[[List[[String]] ]]

// Initialize each month of calendar
for (1 .. 12) do {month: Number ->
  def numDays: Number = getDays (month)
  def monthOfDays: List[[String]] = emptyList[[String]]

  // Initialize days in the month
  for (1..numDays) do {day: Number ->
    monthOfDays.add (" - ")
  }

  dailyEvent.add (monthOfDays)
}

```

Figure 14.3: Creating a calendar as a list of lists

Figure 14.5 shows `dailyEvent` after two assignments have been made to list elements – the one above, as well as

```
dailyEvent.at (4).at (30) put ("Mom's birthday")
```

This assignment indicates that April 30 is “Mom’s birthday”.

More generally, the `yearlyCalendar` class should have a method `insertEvent` that takes three parameters – a month, a day, and an event description – and should set that entry appropriately. The method is as follows:

```

method insertEventOn (month: Number, day: Number) with ( description: String) -> Done {
  dailyEvent.at (month).at (day) put (description)
}

```

A similar method can be written to retrieve the event associated with a particular date. Given a month and a day, the method simply returns the corresponding entry in `dailyEvent`:

```

method eventOn (month: Number, day: Number) -> Done {
  dailyEvent.at (month).at (day)
}

```

Now let’s take another look at the `dailyEvent` list. A different view is shown in Figure 14.6. In this figure we have “squashed” the rows together so that the list of lists looks more like a table with rows and columns. When you think of the data structure in this way – i.e., as a *two-dimensional list*, it is then natural to think of the indices for accessing elements as specifying a row and a column in a table. Therefore, if `somelist` is a two-dimensional list, then

```
somelist.at (rowNum).at (colNum)
```

refers to the element in the row numbered `rowNum` and the column numbered `colNum`.

Exercise 14.1.1 Write the statement that records that July 4 is Independence Day.

14.1.4 Traversing a Two-Dimensional list

As you saw in Chapter 13, we often want to do something with every element in a list. If so, we use a loop to iterate through the elements, performing on each one the specific action

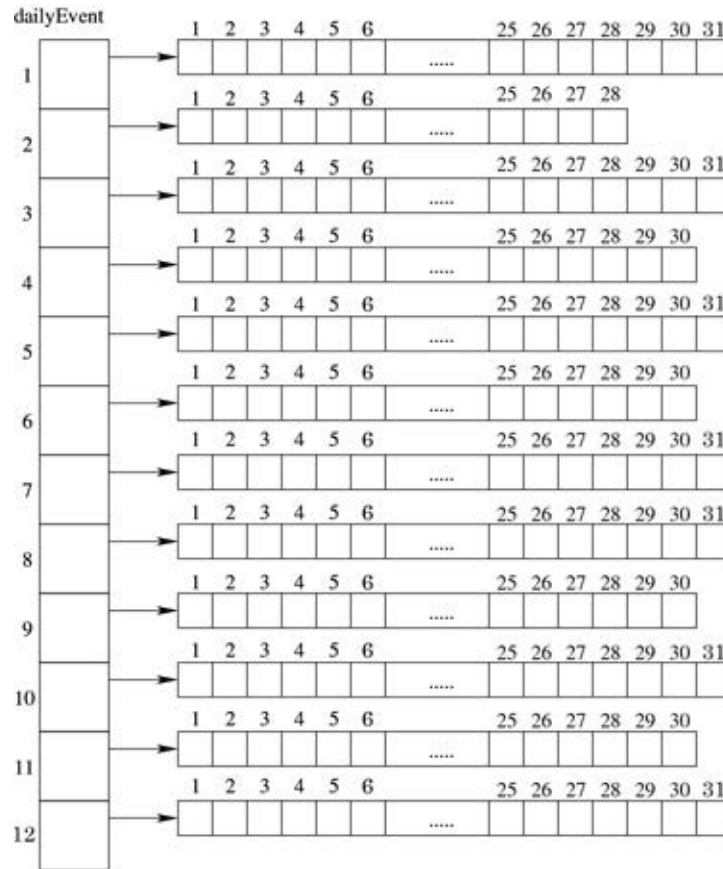


Figure 14.4: Daily events organized as a 12-element list (months) of lists (days of each month).

desired. We will continue to follow this basic pattern even with two-dimensional lists, but with a little extra work to handle the extra dimension.

Say that in our calendar program, we want to clear the calendar in order to reset each entry to "-". The general structure will be similar to that we used to initialize the calendar. We will need to reset each element in each month. If month is a variable that refers to the list for a specific month, then we initialize the entries for that month with a simple loop:

```
for (1 .. month.size) do {day: Number ->
  month.at (day) put ("-")
}
```

Notice that the for loop iterates over the correct number of days in each month.

We needed a for loop over the indices in the list because we needed to specify where to put the "-". On the other hand, we can just iterate over the list of months without worrying about the indices, because we don't need the index when requesting the method:

```
for (dailyEvent) do {month: List[[String]] ->
  for (1 .. month.size) do {day: Number ->
    month.at (day) put ("-")
  }
}
```

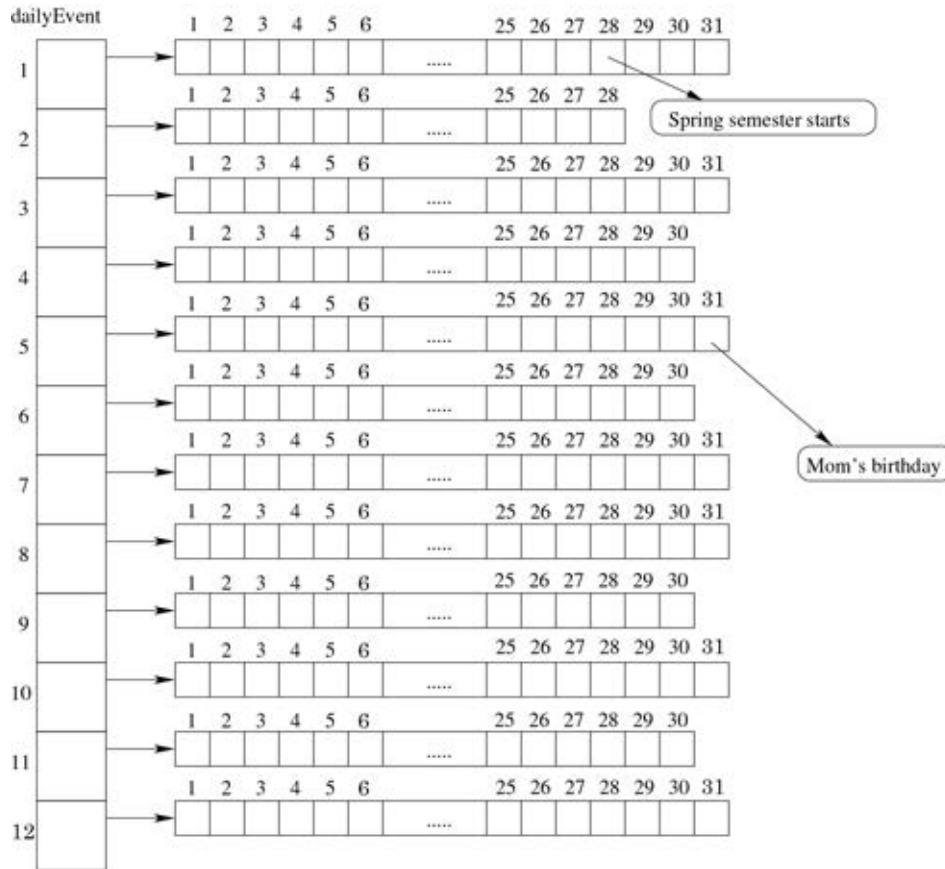


Figure 14.5: Yearly calendar after two events have been entered.

Exercise 14.1.2 Try to write the above code using nested for loops where each of the loops iterates over the list instead of the indices. E.g.,

```

for (dailyEvent) do {month: List[[String]] ->
  for (month) do {dayEvent: String ->
    ...
  }
}

```

What goes wrong?

In general, traversal of a two-dimensional list is accomplished with nested for loops, one of which handles the rows and the other the columns. Figure 14.7 outlines two ways of doing this.

Figure 14.8 shows the definition of the YearlyCalendar class that we have been discussing. This class has two instance variables, `dailyEvent`, the two-dimensional list, and a number, `year`. This variable can be used to determine leap years, so that February has the appropriate number of days. The constructor takes a single parameter that is the year of the calendar to be constructed. It then constructs the two-dimensional list and initializes each daily event to "-". The method `setEvent` allows a user to add an event to the calendar; the method `getEvent`

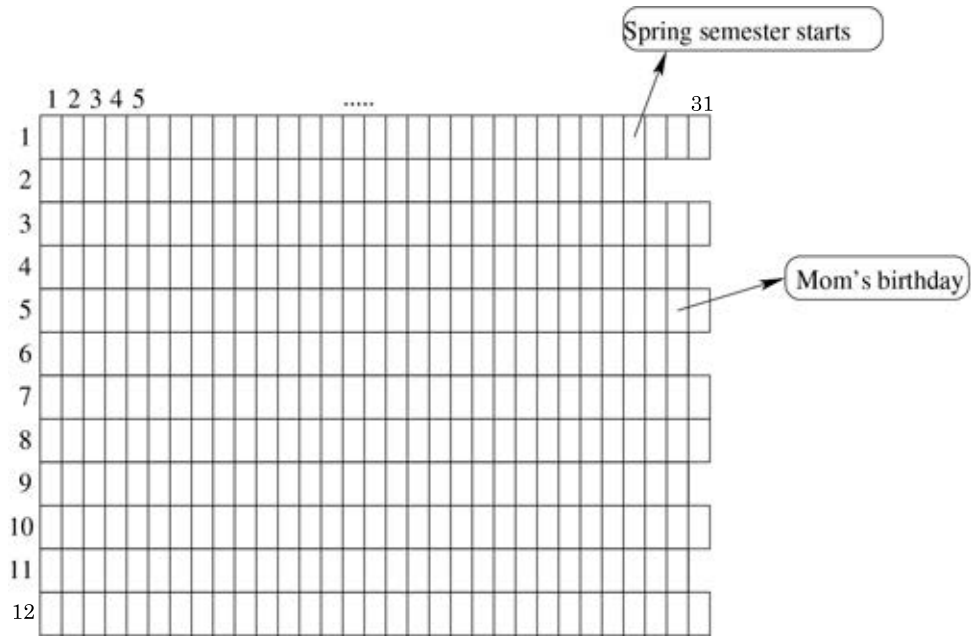


Figure 14.6: Another view of daily events as a jagged table

returns the event description associated with a particular date.

Exercise 14.1.3 Add a method to the `YearlyCalendar` class called `printYear`, with the following header

method `printYear` → Done

This method should print the entire year's activities. That is, for each day of the year it should print the date followed by the activity for that day. The format of each printed line should be as follows:

month/day: activity

14.1.5 Beyond Two Dimensions: Extending the Calendar Program

The class `yearlyCalendar` that we have been discussing is admittedly quite limited. Any good calendar manager should allow a user to enter more than one event for any given date. In this section, we briefly examine ways in which the `yearlyCalendar` class can be extended.

Any given date in your calendar has the potential to be fairly complex. As Figure 14.9 shows, you might have general event descriptions for a day as well as hourly appointments or meetings. Since any single day is filled with a variety of information, let's define a new class, `calendarDay`, that can be used to describe the events in any single day.

As illustrated in Figure 14.9, there are at least two different kinds of event descriptions you might find on a single date. These include descriptions for the day as a whole and a listing of appointments. In order to represent these two related, but distinct, types of information, we'll define two lists in the `calendarDay` class. One of these will be the collection of full-day events,

```

// First alternative
for (my2dList) do { row: List[[T]] ->
  for (1..row.length) do { col:T ->
    // do something with list element row.at (col)
  }
}

// Second alternative
for (my2dList) do { row: List[[T]]
  for (row) do { elt: T
    // do something with list element elt
  }
}

```

Figure 14.7: General structure of nested for loops to traverse a two-dimensional list

and the other will be the collection of hourly appointments. Both lists will be collections of type `String`. A sketch of some of the type and class descriptions can be found in Figure 14.10.

We can now modify `yearlyCalendar` to make use of this new class. The first thing we need to do is declare that a year of daily events is a list of list of `CalendarDays`, rather than simply `Strings`:

```
def dailyEvent: List[[List[[CalendarDay]] ]] =emptyList[[List[[CalendarDay]] ]]
```

So, in essence, `dailyEvent` is a list of months, each of which is a list of days, each of which is two lists of daily and hourly events.

We leave the remainder of the implementation of this class as an exercise. The important idea to take from this section is that it can be very useful to construct lists which have, as their elements, other objects containing lists..

If we now consider the overall structure of our calendar, we can see that it is quite complex. It is a list of months where each month is a list of days, while each day is two lists: a list of full day events, and a list of timed events. Luckily, by breaking up the definitions, we can keep a handle on the complexity.

Exercise 14.1.4 Complete the class `calendarDay` by adding the following methods:

```

// Clear all event entries for this day
// All event entries are set to "----"
method clearDay -> Done

// Return the event for a specific hour on this day
method getHourlyEvent (hour: Number) -> String

// Print all hourly events}
// Each hourly event on a separate line in the form
method printHourlyEvents -> Done

// Print all full day events
method printFullDayEvents -> Done

// Print all events for this day

```

```

class yearlyCalendar (theYear: Number) {
  def year: Number = theYear
  // List of events in the year
  def dailyEvent: List[[List[[String]] ]] = emptyList[[List[[String]] ]]

  // Initialize each month of calendar
  for (1 .. 12) do {month: Number ->
    def numDays: Number = getDays (month);
    def monthOfDays: List[[String]] = emptyList[[String]]

    // Initialize days in the month
    for (1..numDays) do {day: Number ->
      monthOfDays.add (" -")
    }
    // add to the list of months
    dailyEvent.add (monthOfDays)
  }

  // return number of days in the month
  method getDays (monthNumber: Number) -> Number is confidential {
    if ((monthNumber == 9) || (monthNumber == 4) || (monthNumber == 6) || (monthNumber == 11))
      then { 30
    } elseif {monthNumber == 2} then { // special case for February
      if (leapYear (year)) then { 29 } else { 28 }
    } else { 31 }
  }

  // Determine whether a year is a leap year
  method leapYear (year!: Number) -> Boolean {
    ((year! % 4) == 0) && ((year! % 100) != 0)
  }

  // insert an event with description on calendar for month, day
  method insertEventOn (month: Number, day: Number) with (description: String) -> Done {
    dailyEvent.at (month).at (day) put (description)
  }

  // return the events on month, day
  method eventOn (month: Number, day: Number) -> Done {
    dailyEvent.at (month).at (day)
  }

  // print all the events for each day of the year
  method printYear {
    for (1 .. 12) do {monthNum: Number ->
      for (1 .. dailyEvent.at (monthNum).size) do {dayNum: Number ->
        print ("{monthNum} / {dayNum} : " ++ eventOn (monthNum,dayNum))
      }
    }
  }
}

```

Figure 14.8: The YearlyCalendar class

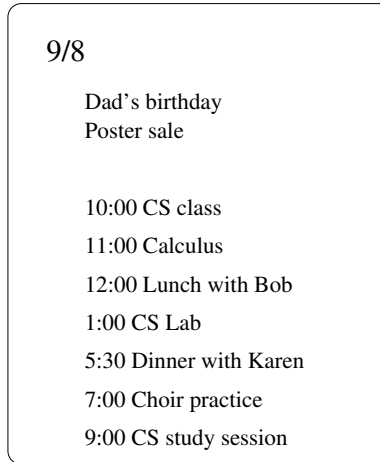


Figure 14.9: Calendar entries for one day

method `printDay` → *Done*

In addition, modify the class code so that it calls the method `clearDay` to initialize all of the event entries for the day.

14.2 Matrices

We now turn our attention to *matrices*. As we said in the introduction to this chapter, a *matrix* is simply a two-dimensional list in which all the rows have the same length. If all the months of the year had 31 days, for example, our two-dimensional list, `dailyEvent` could be called a matrix.

Let's consider several additional examples to illustrate the structure of a matrix. First, consider the image in Figure 14.11. The image is a magnified square region of pixels from a drawing (like one you might create with primitives from the `objectdraw` library). If you were to consider what data structure might nicely represent this, a two-dimensional list should come to mind. This is a nice conceptual fit, because the image has a two-dimensional structure. Each pixel can be described by its row and column position, as well as its color value.

There are some interesting and important differences between Figure 14.11 and the two-dimensional `dailyEvent` list we discussed in the preceding sections. First, each row has the same length. More importantly, however, there is no special meaning we can ascribe to any given row or column. Whereas a calendar can be described as a list of months, each of which is a list of days, this is simply a collection of pixels in a particular configuration. Other similar examples include the chess board and sliding block puzzle shown in Figure 14.12.

We'll begin our discussion of two-dimensional lists like these, i.e., matrices, with a puzzle not pictured in Figure 14.12, magic squares. We'll then return to the example of pixels in an image.


```

type TimedEvent = {
  hour -> Number
  minute -> Number
  duration -> Number
  description -> String
}

type CalendarDay = {
  enterFullDayEvent (...) -> Done
  fullDayEvents -> List[[String]]
  enterAppointmentAt (hour: Number, minute: Number)
    duration (length: Number) with (description: String) -> Done
  appointmentAt (hour: Number, minute: Number) -> TimedEvent
  ...
}

class calendarDay -> CalendarDay {
  def fullDayEvents: List[[String]] is public = emptyList[[String]]
  def appointment: List[[TimedEvent]] = emptyList[[TimedEvent]]

  method enterFullDayEvent(description: String) -> Done {...}

  // enter time based on 24-hour clock
  method enterAppointmentAt(hour: Number, minute: Number)
    duration(length: Number) with (description: String) -> Done { ... }

```

Figure 14.10: Sketch of calendar-related class to hold daily events.

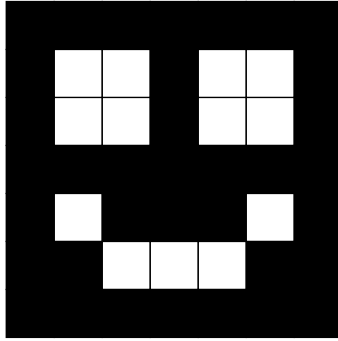


Figure 14.11: Magnified pixels from an image

14.2.1 Designing the Matrix class

Because we typically start by initializing all the entries to a matrix with a fixed value, we will find it helpful to define a `Matrix` class that will perform that initialization. It will also contain values for accessing and updating elements based on their row and column. See Figure 14.13.

The only tricky code is the code for initializing all the entries to have the default value. As we did with our calendar example, we simply construct a list for each row and then add that row to the list of rows. Notice that the methods for setting and getting elements are also a bit simpler to write.

14.2.2 Magic Squares

A square of numbers is said to be a *magic square* if all of the rows, columns, and diagonals add up to the same number. Figure 14.14 shows two magic squares. In the first, each of the rows, columns, and diagonals adds up to 15. In the second, each row, column, and diagonal adds up to 65. Notice that the first is filled with the integers 1-9 and the second with the integers 1-25. It turns out that as long as n is odd, there is a simple algorithm for creating an $n \times n$ magic square from the numbers 1 through n^2 . We'll get back to the algorithm a bit later. For now, let's focus on the way we would check whether a given $n \times n$ square is a magic square.

14.2.3 Traversing a Matrix

In order to determine whether a square is a magic square, we need to find the sum of each row, each column, and each diagonal. We will consider the traversals of these – i.e., the rows, columns, and diagonals – separately.

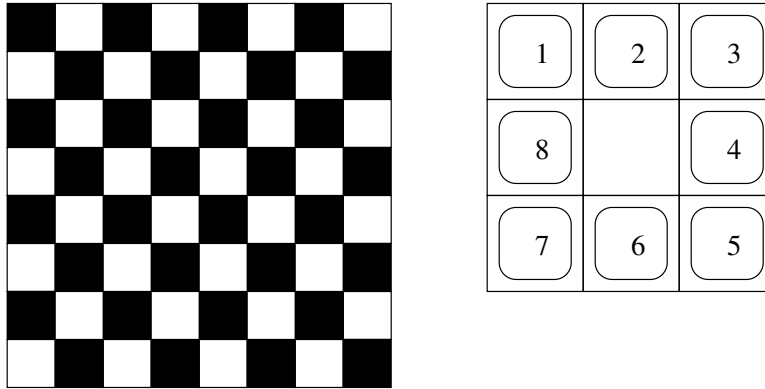


Figure 14.12: A chessboard and sliding block puzzle: examples of objects that can be represented by two-dimensional matrices

Row by Row Traversal

When we are given a square to examine, we don't know at the outset what the sum of any row, column, or diagonal should be. So before we can confirm that all the sums are the same, we need to find a sum that will be our target. We do this by computing the sum of the elements in the first row of our square:

```
// Compute sum of elements in row 1
var targetSum: Number := 0
for (1 .. size) do {col: Number ->
  targetSum := targetSum + magicSquareMatrix.at (1, col)
}
```

We declare a variable `targetSum` that will be the sum of the first-row elements. We initialize it to 0, and then add to it the value of each element we examine. In order to access a cell of the matrix, we need to specify both a row and a column. Since the first row is being examined here, the row index is always 1. The column index, on the other hand, begins at 1 and goes sequentially through the columns until the index is `size`.

After `targetSum` is computed, we can examine the rest of the matrix. First, we consider each of the rows. We compute the sum of each row, in turn, and compare it to the target. If the sum of any row is different from the target, we return the value **false**. Otherwise, we go on to check the next rows until we have checked them all, in which case we return **true**.

```
for (2 .. size) do {row: Number -> // check sum of each row
  var sum: Number = 0
  for (1 .. size) do {col: Number ->
    sum := sum + magicSquareMatrix.at (row, col)
  }
  if (sum != targetSum) {
    return false
  }
}
```

```

// type of two dimensional matrix holding elements of type T
type Matrix[[T]] = {
  at (r: Number, c: Number) -> T
  at (r: Number, c: Number) put (value:T) -> Done
}

// Create matrix of elements of type T with given number of rows and columns,
// initialized with all slots holding dvalue
class matrixSize[[T]] (rows: Number,cols: Number) defaultValue (dvalue:T) ->
  Matrix[[T]] {
  // matrix representation as list of lists of elements of type T
  def values: List[[List[[T]] ]] = emptyList[[List[[T]] ]]

  // initialize all to value dvalue
  for (1..rows) do {rowNum: Number ->
    def rowList: List[[T]] = emptyList[[T]]
    for (1 .. cols) do {colNum: Number ->
      rowList.add (dvalue)
    }
    values.add (rowList)
  }

  // return element at row r and column c
  method at (r: Number,c: Number) -> T {
    values.at (r).at (c)
  }

  // update element at row r and column c to have value value.
  method at (r: Number,c: Number) put (value: T) -> Done {
    values.at (r).at (c) put (value)
  }
}

```

Figure 14.13: A class for representing matrices

4	9	2
3	5	7
8	1	6

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Figure 14.14: Magic squares

```
}
true
```

The inner loop that traverses each row should look very familiar; it's almost identical to the loop we wrote to compute `targetSum`. Around it is the loop that controls movement through the matrix row by row. Note that the first row index is 2, rather than 1, this time. This is because row 1 was used to compute the initial target sum.

Notice that we return a value of `false` as soon as we discover a row whose sum is different from `targetSum`. If we get all the way to the end of the outer for loop, then we know all the rows have the same sum, so the result is `true`.

Column by Column Traversal

To check that the sum of each column is equal to the target, we again write a pair of nested loops. The loops will be virtually identical to those that check the rows, but we will reverse the order of the nesting. The inner loop will check a complete column by going through all of the rows in a given column, while the outer loop will control movement through the list column by column:

```
for (1 .. size) do {col: Number -> // check sum of each row
  var sum: Number = 0
  for (1 .. size) do {row: Number ->
    sum := sum + magicSquareMatrix.at (row, col)
  }
  if (sum != targetSum) \{
    return false
  }
}
true
```

The works as desired because the inner loop is executed entirely for each iteration of the outer loop. So when the variable `col` has the value 1, the inner loop goes through each row number in column 1. When the variable `col` has the value 2, the inner loop goes through each row number in column 2; and so on.

Diagonal Traversal

Once we have checked the rows and columns, we can check each of the diagonals. We'll do this by writing two separate loops. (No nesting of loops is required this time!) The key to traversing a diagonal is to recognize that both the row and column indices need to change with each iteration of the loop.

Let's consider the major diagonal first. Here we need to examine the elements in positions (0, 0), (1, 1), (2, 2), etc. The row and column indices certainly change, but each time they remain equal to each other. As a result, we only need a single variable to help us iterate over the elements on the diagonal:

```
var sum: Number := 0
for (1 .. size) do { index: Number -> //check sum of major diagonal
  sum := sum + magicSquareMatrix.at (index, index)
}
sum == targetSum
```

When we have computed the sum, we simply return whether or not it is equal to `targetSum`.

The minor diagonal, which starts in the upper right corner and moves to the lower left, is a bit more tricky. This time we notice that when the row value is 1, the column value is the maximum column value, `size`. When the row value is 2, the column value is 1 less than the maximum. When the row value is 3, the column value is 2 less than the maximum. That is, if the row value is given to us by the variable `row`, then the appropriate column is:

```
size + 1 - row
```

We use this information to help us write a `for` loop to traverse the minor diagonal as follows:

```
sum := 0
for (1 .. size) do {row: Number // check sum of minor diagonal
  sum := sum + magicSquareMatrix.at (row, size + 1 - row)
}
sum == targetSum
```

14.2.4 Filling a Magic Square

Now we turn to filling a magic square. As we indicated earlier, as long as n is odd, there is a simple algorithm for creating an $n \times n$ magic square. The algorithm is as follows:

- Place a 1 in the center of the bottom row. Then fill in the remainder of the square by following these rules:
- Try to place the next integer (one greater than the last one you placed) in the cell one slot below and to the right of the last one placed. If you fall off the bottom of the list, go to the top row of the same column. If you fall off the right edge of the list, go to the leftmost column of the same row. If that cell is empty, write the next integer there and continue.

If the cell is full, go back to where you wrote the last integer and write the next one in the cell directly above it.

Try this yourself. Consider a small example, like a square with three rows and three columns.

Exercise 14.2.1 *To ensure that you really see how the magic square filling algorithm works, try it on a larger square. This time use the algorithm to fill in a square with five rows and five columns.*

Now let's write Grace code to implement this algorithm. Let's begin by initializing each cell in the square to 0. Luckily we already wrote a class definition for matrix that allows us to create the elements and initialize them to 0.

```
// Array holding entries in magic square
def magicSquareMatrix: Matrix[[Number]] = matrixSize[[Number]] (size, size) defaultValue (0)
```

Next we'll declare and initialize two integer variables, `currRow` and `currCol`. These will refer to the row and column of the current cell to be filled. Since the first cell to be filled is in the bottom row of `magicSquareMatrix`, the initial value of `currRow` will be `size`. Since the cell is in the middle of the bottom, our column index needs to refer to the middle element. To find the middle column, we simply divide `size+1` by 2. The reason is that we are looking for the value half-way between 1 (the starting index) and `size` (the ending index). Thus we just add them together and divide by 2. For example if `size` is 5, the middle column is $(5 + 1) / 2 = 3$. So

```
var currRow: Number := size
var currCol: Number := (size + 1) / 2
```

Now we can begin to fill the matrix. Since filling `magicSquareMatrix` is a matter of repeatedly applying the rules above, we'll place the code in a loop. The loop will control the values placed in the matrix, so we can write it as:

```
for (1 .. (size * size)) do { nextInt: Number ->
  // fill a cell with nextInt
}
```

As we enter the loop, we know the row and column indices of the cell to fill, so we say:

```
magicSquareMatrix.at (currRow, currCol) put (nextInt)
```

Now we need to find the next cell to fill, so that the next loop iteration will fill it properly. First we try to move one space to the right and one space down:

```
def nextRow: Number = currRow + 1
def nextCol: Number = currCol + 1
```

But we need to take into account the possibility that we'll fall off the right edge or the bottom edge of the matrix. For instance, if `magicSquareMatrix` is a three-by-three matrix and we just finished filling the first cell (i.e., the cell in the middle of the bottom row), then the next cell to fill would be the one with row index 4 and column index 3. But, as illustrated in Figure 14.15, this is not a valid cell in the matrix. In this case, we need to move to the first row in the same column, i.e., to the row with index 1. We can handle this easily by modifying the lines that update `nextRow` and `nextCol` as follows:

```
def nextRow: Number = (currRow % size) + 1
def nextCol: Number = (currCol % size) + 1
```

Remember that the `%` operator gives us the remainder when dividing one integer by another. So if `currRow` is 3, then next row is $(3 \% 3) + 1$, which evaluates to $0 + 1$ or 1.

We still have to take care of one more thing before we can fill the cell. We need to be sure that it is actually available, i.e., that it hasn't already been filled. For example, Figure 14.16 shows `magicSquareMatrix` after three cells have been filled. The next one to fill would be the center bottom cell, which already has a value. In this case, the algorithm tells us to go

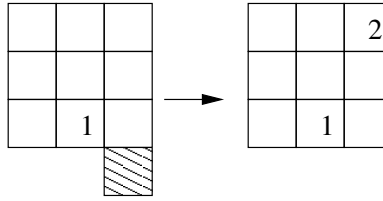


Figure 14.15: Wrapping around after falling off the bottom of the square

back to the cell just filled (the cell with value 3) and move one cell up. We can accomplish the check for availability and possible index adjustment as follows:

```

if (magicSquareMatrix.at (nextRow, nextCol) == 0) then {
    currRow := nextRow
    currCol := nextCol
} else {
    // The cell was full – Use the cell above the previous one
    if(currRow == 1) then {
        currRow := size
    } else {
        currRow := currRow - 1
    }
}

```

The final statement makes an adjustment in the case that moving up moves us out of the matrix (i.e., up from row 1). It wraps the row around to the bottom.

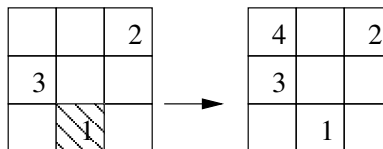


Figure 14.16: Moving up if the next cell to be filled is already full

Now that we know the next cell, the next iteration of the loop can fill it. A method called `fillSquare` with the complete implementation is shown in Figure 14.17.

14.3 Summary

In this chapter we introduced multi-dimensional lists, focusing on two-dimensional lists. Conceptually, we can consider two distinct types of two-dimensional lists. In the first case, we can think of the two-dimensional list as a list of lists. That is, we can view each of the


```

// Place values to make a magic square
method fillSquares -> Done is confidential {

  // Set the indices for the middle of the bottom row
  var currRow: Number := size
  var currCol: Number := (size + 1) / 2

  // Fill each cell
  for (1..(size * size)) do { nextInt: Number ->
    magicSquareMatrix.at (currRow, currCol) put (nextInt)
    // Find the next cell, wrapping around if necessary
    def nextRow: Number = (currRow % size) + 1
    def nextCol: Number = (currCol % size) + 1

    // If the cell is empty, remember those indices for the
    // next assignment
    if (magicSquareMatrix.at (nextRow, nextCol) == 0) then {
      currRow := nextRow
      currCol := nextCol
    } else {
      // The cell was full - Use the cell above the previous one
      if(currRow == 1) then {
        currRow := size
      } else {
        currRow := currRow - 1
      }
    }
  }
}

```

Figure 14.17: A method to fill a magic square

rows as having a significant meaning. In the second form, i.e., the matrix, the structure is simply a two-dimensional grid, with no special significance to the rows or columns. While it is useful to think of these as different conceptually, they really are the same, and, as a result, the mechanisms for declaring them, constructing them, indexing, and traversing them are the same:

- To declare a two-dimensional list, we define the type by applying the List operator twice to a type.

```
type listName = List[[List[[T]] ]]
```

- We defined a class to represent a two-dimensional matrix. To create a matrix we need to specify the type of elements it holds, the number of rows and columns, and a default value to fill all the slots with:

```
def myMatrix: Matrix[[T]] = matrixSize[[T]] (numRows,numCols) defaultValue (dv)
```

- To refer to an individual element within a matrix, we provide its row and column indices to method at:

```
myMatrix.at (myRow, myCol)
```

- Typically, working with a matrix involves traversing it row by row, column by column, or along diagonals. To traverse an entire matrix, we write nested loops.

14.4 Chapter Review Problems

Exercise 14.4.1 *A particular university enrolls 5,000 students. In a semester, a student can take up to six classes. The grades assigned at this university are numbers in the range 0.0 to 4.0, where 4.0 corresponds to the letter grade A.*

- Declare a list of lists called **grades** that can hold all of the grades for a single semester. You can think of each row representing the grades for one of the students. Remember that some rows will be longer than others, depending on the number of courses taken by each student.*
- Write a method **printAvg**s that will print the average grade for each of the students. Remember that each student can from one to six courses. (You may assume that each student takes at least one course.)*

Exercise 14.4.2 *Company X pays its employees every four weeks. Their payroll system is implemented in Grace. This exercise will ask you to help develop one class, **employeeTime** to manage the time worked by an employee during a four-week period.*

- Declare an instance variable **hoursWorked** that can keep track of the number of hours worked per day in a four-week pay cycle. This should be a matrix. You can think of each row as a week. The number of hours worked in a single day should be recorded as a **Number**.*

- b. Write the constructor for this class. The constructor should take two parameters: a `String` that is the employee ID and a `Number` that is the employee's hourly wage.

The constructor should construct `hoursWorked`. Remember that it is good style to define constants for the number of weeks and the number of days (i.e., the number of rows and columns in `hoursWorked`). In addition, the constructor should remember the employee ID and hourly wage in instance variables.

- c. Define constants (`defs`) `saturday` and `sunday`, with values 6 and 7, respectively. These will be the index values for Saturday's hours worked and Sunday's hours worked.

What is the index of the third Saturday in the four-week cycle? What is the index of the first Tuesday?

- d. Employees get paid double on Saturdays and Sundays. Write a method that will calculate the pay earned by an employee for regular hours worked, i.e., hours worked Monday through Friday.

- e. Write a method that will calculate the pay earned by the employee for working on Sundays.

Exercise 14.4.3 Your friend has written a tic-tac-toe program. They have declared the game board to be a matrix with three rows and three columns. To represent the players X and O, they have defined constants with values 1 and 2, respectively. The matrix, then, is a matrix of `Number`.

Your friend has written the following method that is meant to return true if the player passed in as a parameter is a winner along the major diagonal (from upper left to lower right). Unfortunately, it doesn't work. Fix the method.

```
// the number of rows and columns
def size: Number = 3

method diagWin (player: Number) -> Boolean {
  for (0 .. size) do {
    if (board.at (i, i) != player) then {
      return false
    }
  }
  true
}
```

Exercise 14.4.4 You are on a team that is developing a game in which the player must escape from a monster. Both the monster and the player move through a space that looks like a grid. In a single move, the monster can move one space vertically, horizontally, or diagonally. The player can only move vertically or horizontally. This certainly appears to give the advantage to the monster. Fortunately, the player can do something that the monster cannot do. The monster can never move off the grid. If it is at the right edge of the grid, for example, it cannot move to the right. The player, on the other hand, can move to the right. If the player does so, they re-enter the grid on the opposite side, as indicated in Figure 14.18. The player can make analogous moves off the left edge of the grid.

- a. Write a method `moveRight` that takes a `Number` parameter `col` that is the player's current column. The method should return a `Number` that is the column into which the player should move if they are going right. You may assume that a constant `numCols` has been defined that gives the total number of columns in the grid.
- b. Now write a method `moveLeft`.

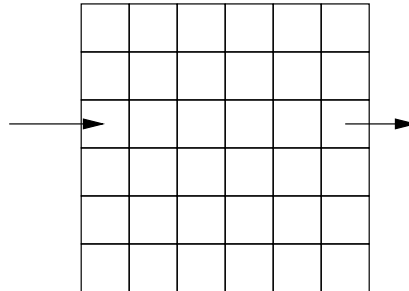


Figure 14.18: Player moving off the right of the game grid re-enters from the left

14.5 Programming Problems

Exercise 14.5.1 Write a program that will let a user play with ice blocks on a canvas. Your program should begin by drawing a wall of rectangular ice blocks on the canvas. When the user clicks on an ice block, it should be removed. When all the ice blocks in a level of the wall have been removed, the layers above it should fall down on top of the remaining levels below.

In addition to your main program, you should implement three classes called `IceWall`, `IceLayer`, and `IceBlock`. The constructors and methods for these classes should be as follows:

// A wall of ice blocks

type `IceWall` = {...}

// Construct an ice wall
// upperLeft – location of the upper left of the wall
// width – number of ice blocks on each level
// height – number of layers
// iceWidth – width in pixels of a block of ice
// iceHeight – height in pixels of a block of ice
// canvas – canvas on which to draw ice wall

class `IceWallAt` (`upperLeft: Point`) `size` (`width: Number`, `height: Number`)
`blockSize` (`iceWidth: Number`, `iceHeight: Number`) `on` (`canvas: DrawingCanvas`) \rightarrow `IceWall` {

```

...

//Hit the ice block at hitPoint (if any)
//If a block is hit, it is removed.
//If removing this block empties the layer, the layers above it fall
//into their new positions.
method hit (hitPoint: Point) -> Done {...}
}

type IceLayer = {...}

// Construct a layer of ice blocks
// layerX - the x coordinate of the upper left of the layer
// layerY - the y coordinate of the upper left
// blockHt - the height of a block of ice
// blockWidth - the width of a block of ice
// numBlocks - number of blocks initially
// canvas - canvas on which to draw the ice layer

class iceLayerAt (upperLeft: Point) blockSize (blockDimension: Point)
  with (numBlocks: Number) on (canvas: DrawingCanvas) -> IceLayer {

...

//Returns true if a point is contained in one of the ice blocks
//in this layer; false otherwise}
method isHit (point: Point) -> Boolean {...}

//Returns true if the layer is empty; false otherwise.
method isEmpty -> Boolean {...}

//Make the layer of ice blocks fall
method fall -> Done {...}
}

type IceBlock = { ... }

// Construct a block of ice
// upperLeft - the location of the upper left of the ice block
// deimension - the height & width of the ice block
// canvas - canvas on which to draw the ice block
class iceBlockAt (upperLeft: Point) size (dimension: Point) on (canvas: DrawingCanvas) -> IceBlock {

...

//Move the ice block dx horizontally and dy vertically
method moveBy (dx: Number, dy: Number) -> Done {...}

//Return true if the ice block contains point; false otherwise
method contains (point: Point) -> Boolean { ... }

```

```
//Remove the ice block from the canvas  
method removeFromCanvas → Done { ... }  
}
```

Chapter 15

Strings

Up to this point, we have focused much of our attention on programs that manipulate graphical or numeric information. You're undoubtedly aware, of course, that there are also many computer applications that manipulate textual data. These include word processors, on-line dictionaries, web browsers, and so on. In this chapter we explore Grace's `String` class, which will allow us to refer to and manipulate textual information.

We briefly introduced the `String` class in Section 5.7. There you learned that strings are, in their simplest form, quoted text. You have seen that you can define variables that refer to strings and that you can use them in various contexts. You have used them often, in fact, as parameters to the `text` class and `print`.

You have also learned that strings can be manipulated in a variety of ways. You can concatenate `Strings` using the `++` operator. You can also manipulate them with a variety of methods. For example, you can compare them using `==`, and you can find their size with the `size` method.

The type `String` is like the types `Number` and `Boolean` in that they are immutable and we can write “literal expressions” to create elements of the type rather than using a class definitions. With `Number`, we just write values like 47 and 3.14159, with `Boolean` we write the two predefined values `true` and `false`. With strings we just write a sequence of characters surrounded by double quotes, such as “Programming with Grace!”.

We begin this chapter with a discussion of a small but very useful string – the empty string. We then turn our attention to `String` methods. Grace's `String` type supports a large number of interesting methods, and we will discuss most of them here.

We end with a description of strings of length 1. While it may seem odd to focus on these small strings, they are useful to consider as pieces of larger strings.

15.1 Little Strings and Big Strings

15.1.1 The Empty String

The smallest possible string is one that is made up of no characters at all. This string of size 0 is called the *empty string*. Expressed as a literal, the empty string is written as two double quotes with nothing between them: `""`

Typically, the empty string is used in contexts where we want to build a string “from nothing”. As an example, recall the Morse Code program from Figure 5.11. That program displays a series of dots and dashes, based on a user’s mouse clicks. A long click signifies a dash, while a short click signifies a dot. In that program, the resulting Morse code message is a `String` named `currentCode`. We initialize `currentCode` to `"Code = "` and then add either a dot or dash to it with each mouse click. But, what if we want `currentCode` to refer only to the series of dots and dashes entered by the user? That is, what if we don’t want the string to have the prefix `"Code = "`? We can accomplish this by simply initializing `currentCode` to the empty string, as in Figure 15.1. This initialization says that variable `currentCode` refers to a `String` that is empty until the first dot or dash is concatenated to it.

The empty string serves a purpose that is in many ways analogous to the purpose of the number 0 in arithmetic. It provides us with a way to begin with nothing so that we can accumulate a “bigger” value over time.

15.1.2 Long Strings

As we have just seen, strings in Grace can be extremely short. They can also be arbitrarily long. The text of this chapter, for instance, could be treated as a single `String`. While long strings don’t require any special handling, there is one aspect of them that deserves a bit of discussion – readability. We have already said that this chapter could be a single `String`. What if we wanted to print that very long string using `print`? If our `String` was comprised of just the characters in the chapter, without any special attention given to the line breaks, the resulting printed text would be one long line and as a result be unreadable.

There are many times when it is useful to place line breaks into a long string by including the special character `\n`, which is the newline character. Say, for example, that we want to print instructions for the Morse Code program described in Section 15.1.1. Sample instructions are given in Figure 15.2.

We could obviously print the instructions in Figure 15.2 with a series of five `print` statements. As an alternative, we could define a `String` constant named `instructions` to be all of the instructions, as in Figure 15.3 and then simply print them with the single statement

```
print(instructions)
```

Note the use of `\n` in Figure 15.3 to control the formatting of the instructions by forcing line breaks. You’ll also notice that the `String` `instructions` is split over five lines, with the concatenation operator (`++`) at the end of each line to tie the pieces together. The reason for writing the instructions on five lines is to improve readability. This way, the format in the program mimics the format of the actual instructions. This will be useful to anyone reading our program. Finally, it is useful to note that the `String` `"\n"` is simply a newline string. While it appears to have size two, it really has size one, since it contains only the newline symbol.

There are many times when it is useful to express a long `String` literal as the concatenation of two or more shorter `Strings`. One place where this can be done is in the context of `print`. Say that we had implemented a Morse code translator (which we will do later in this chapter), and that a user of our program had entered an inappropriate (i.e., untranslatable) character, and say that we wanted to print a


```

dialect "objectdraw"
import "sys" as sys

object {
  inherit graphicApplicationSize (400 @ 400)

  // Minimum time (in seconds) for a dash
  def dashTime: Number = 0.2

  // String to hold sequence entered so far
  var currentCode: String := ""

  // Time when mouse was last depressed
  var pressTime: Number

  // Text used to display Morse code on canvas
  def display: Text = textAt (30 @ 30) with ("Code = ") on (canvas)

  // Record time at which mouse was depressed
  method onMousePress (point: Point) -> Done {
    pressTime := sys.elapsedTime
  }

  // Add . or - depending on how long mouse held before release
  method onMouseRelease (point: Point) -> Done {
    if ((sys.elapsedTime - pressTime) > dashTime) then {
      currentCode := currentCode ++ " -"
    } else {
      currentCode := currentCode ++ "."
    }
    display.contents := currentCode
  }

  startGraphics
}

```

Figure 15.1: A program to display Morse code as dots and dashes

This program will allow you to enter a message in Morse Code.

To enter your message:
Click the mouse quickly to generate a dot;
Depress the mouse longer to generate a dash.

Figure 15.2: Instructions for the Morse code program

```
def instructions: String =  
  "This program will allow you to enter a message in Morse Code.\n" ++  
  "\n" ++  
  "To enter your message:\n" ++  
  "Click the mouse quickly to generate a dot;\n" ++  
  "Depress the mouse longer to generate a dash."
```

Figure 15.3: Using `\n` to force line breaks

useful one-line error message in response to this. We could do so with the following statement:

```
print("The message that you have entered contains "  
      "characters that cannot be translated.")
```

This would print our error message on a single line as desired, but it also makes our program more readable than it would be if we did not split the string over two lines.

Grace does not allow us to write a single string literal with actual line breaks in it, as in:

```
print ("The message that you have entered contains  
      characters that cannot be translated.")
```

If you want to split a string literal over several lines, you will need to use the concatenation operator to join substrings of the larger string, as above.

15.1.3 Interpolation in Strings

As we saw earlier, we can insert computed values into strings by enclosing the expression to be evaluated in curly braces. Consider the following program:

```
def count: Number = 12  
print "{count} squared is {count * count}"
```

Executing this code will result in printing "12 squared is 144". Enclosing values in curly braces inside a string is known as string interpolation. There is no similar notation in languages like Java or C++, where the programmer would instead write something like `count + " squared is " + (count * count)`. (In those languages `+` is used for string concatenation.)

When an expression inside a string literal is contained in curly braces, Grace will evaluate the expression, compute the results of sending `asString` to the value of the expression, and then insert that string in the appropriate part of the string. In the above example, `count` evaluates

to 12 and is converted to the string "12", `count * count` evaluates to 144 and is converted to the string "144". Then those values are inserted at the corresponding parts of the string literal, resulting in the string "12 squared is 144", which is then printed.

Suppose you want to print out a string that contains curly braces. You can get those to print out by replacing "{" by "\{" and "}" by "\}". For example we can print out the string {1,2,3} by writing `print "\{1,2,3\}"` whereas writing `print "{1,2,3}"` would generate an error message because Grace doesn't know how to evaluate numbers separated by commas.

15.2 A Collection of Useful String Methods

Grace's `String` class provides many useful methods. In Section 5.7, we introduced the method `size`. The invocation

```
someString.size
```

returns a `Number` that is the number of characters in `someString`.

In this section we introduce several additional useful `String` methods. We begin by building a class for maintaining and searching a URL history list, which will serve as an example throughout this section.

15.2.1 Building a String of URLs

In Section 11.1.2 we discussed one implementation of a URL list. Many web browsers make it easy for you to type in the URL (universal resource locator) for a web page by offering automatic address completion. If I type "http://www.p", my browser suggests "http://www.pomona.edu" as a completion. The browser does this by keeping track of URLs typed in by users. It can then provide a sublist of URLs that includes all of the strings in the list that match a given prefix. In this section, let's imagine a similar type of functionality. We will implement a class that will maintain a URL history. Based on that history, we will add methods to do automatic address completion.

The implementation that will be given here is different from that in Section 11.1.2. As discussed in Chapter 20, there are often many possible ways to implement a data structure or a particular type of functionality.

Figure 15.4 shows a type `URLHistory` representing objects that maintain a history of URLs. It includes the method `addURL` to add a URL to the list, `asString` to convert the history to a string, `contains` to determine if the history contains a URL, and `findCompletions` that takes a URL prefix and then returns a string representing all URL's in the history that start with that prefix.

We will create a class `uRLHistory` to implement the type `URLHistory`. This class will maintain the URL history as one long string called `urlString`. We will create an empty history by initializing `urlString` to the newline character, `\n`. We will see later that this will make some of our searches simpler.

To add a new URL to the existing history, we simply concatenate it to the front of `urlString`, with a newline character to separate it from the existing history. We choose the newline character as a separator since we are guaranteed that it will not also appear in a given URL. Figure 15.5 shows the class definition.

```

// type of objects holding history of URL's seen recently
type URLHistory = {

    // add aURL to collection saved
    addURL (aURL: String) -> Done

    // return a string representing all the URL's
    asString -> String

    // Determine if aURL is in the history list
    contains (aURL: String) -> Boolean

    // return string representing all URL's in history w/ aURL as prefix
    findCompletions (aURL: String) -> String
}

```

Figure 15.4: URLHistory type

We put off the discussion of the `contains` and `findCompletions` methods until after the next section, where we will develop a very useful helper function called `indexOf`.

15.2.2 Adding a `contains` method to `uRLHistory`

The `contains` method can be used to determine if one string occurs as a contiguous substring of another. It simply returns a `Boolean` value corresponding to whether or not it occurs anywhere in the receiver. Thus `"His toy is here".contains("is")` would return `true`, while `"Her blouse is red".contains("sis")` would return `false`.

Figure 15.5 includes a method of the `URLHistory` class also called `contains`. It takes a `String` as a parameter that is a URL to be found in the current URL history. If that URL is found, it returns `true`. Otherwise, it returns `false`. As you might guess, this method will use the `contains` method for `String`. However, we have to do a bit of extra work to get the desired answer. Rather than searching for the URL immediately, we first tack on a newline character to both the beginning and the end. The reason for doing this is that we want to be certain that we've found exactly the URL that is desired. If we were looking for

```
"http://www.cs.pomona.edu"
```

in a URL history that contained only

```
"http://www.cs.pomona.edu/~cs51"
```

for example, we would not want `contains` to return `true`, because our URL was not found. Once this is done, we can use the `contains` method for `String` on this longer string. If it returns `true` then we can be confident that the exact URL occurs in the list. See the code in Figure 15.5.

Exercise 15.2.1 *Modify the method `addURL` so that it only adds a URL to the history if it is not already there.*

```

class URLHistoryEmpty -> URLHistory {

  var urlString: String := "\n"

  // add aURL to collection saved
  method addURL(aURL: String) -> Done {
    urlString := aURL ++ "\n" ++ urlString
  }

  // return a string representing all the URL's
  method asString -> String {
    urlString.substringFrom (2) to (urlString.size)
  }

  method contains (aURL: String) -> Boolean {
    // look for URL terminated by newline separator
    urlString.contains ("\n" ++ aURL ++ "\n")
  }

  // return string representing all URL's in history w/ aURL as prefix
  method findCompletions (aURLPrefix: String) -> String {
    var allCompletions: String := ""

    var urlStart: Number := urlString.indexOf ("\n" ++ aURLPrefix)

    while {urlStart >= 1} do {
      def urlEnd: Number = urlString.indexOf ("\n") startingAt (urlStart + 1)
        ifAbsent {urlStart}
      def possCompletion: String = urlString.substringFrom (urlStart + 1) to (urlEnd)
      allCompletions := allCompletions ++ possCompletion
      urlStart := urlString.indexOf ("\n" ++ aURLPrefix) startingAt (urlEnd)
        ifAbsent {0}
    }
    allCompletions
  }
}

```

Figure 15.5: uRLHistory class

15.2.3 Finding the Position of a Substring with `indexOf`

Suppose we want to determine whether a particular URL is in the URL history. To do so we can use the `String` method `indexOf`. If `source` is the string we are searching and `sub` is the string we are looking for inside it then

```
source.indexOf (sub)
```

returns the positive number `n` if `sub` is a contiguous substring of `source` whose first occurrence starts at character `n` of `sub`. For example, if `source` is

```
"Strings are objects in Grace."
```

and if `sub` is `"in"` then

```
source.indexOf (sub)
```

should return the Number 4. In this example `indexOf` returns the start of the first occurrence of `"in"`, even though it appears twice. The value 0 is returned if the pattern is not found within the string to be searched. So if `sub` is `"classes"` then

```
source.indexOf (sub)
```

yields the value 0.

The return value is 0 when there is no occurrence because the method must return a Number, and 0 is not a possible answer if there is a match. Thus we can easily determine if there was a match by checking to see if the value returned is positive.

It is important to note that `indexOf` is case-sensitive. Since the upper and lower case characters are distinct from each other, the string `"IN"` is completely different from `"in"`. Thus

```
source.indexOf ("IN")
```

would return 0. We will see how to get around this in the next section.

However, we want to finish this section by introducing a useful variant of `indexOf`. The method `indexOf()startingFrom()` allows you to start searching for the substring wherever you would like in the source string. Thus if the value of `source` is again `"Strings are objects in Grace"` then `source.indexOf ("in") startingFrom (1)` is 4, just as before. However `source.indexOf ("in") startingFrom (5)` is 20 because we started searching after the first occurrence of `"in"`.

Exercise 15.2.2 *The method `contains` in the `String` class is a bit redundant as it is easily definable from `indexOf`. Please write a possible definition of method `contains` in the `String` class using the method `indexOf`.*

15.2.4 Dealing with Lower and Upper Case

While there are certainly contexts in which it is useful to distinguish between lower and upper case characters, there are other times when it is unnecessary (or even wrong) to do so. For example, if the URL

```
http://www.cs.pomona.edu
```

were in our history, we would surely want to recognize

```
HTTP://www.cs.pomona.edu
```

as being the same thing.¹

The equality relation `==` determines whether one string is equal to another in the sense of being composed of the same sequence of characters. Sameness is taken to be a strict equality with respect to case. "A" is not the same as "a", "R" is not the same as "r", and so on. If we want to test for equality but also want to ignore differences in case, we can accomplish this by first converting the strings to all upper or all lower case, and then compare them.

Two useful methods for helping us deal with issues of case are `asLower` and `asUpper`. An invocation such as `someString.asLower` returns a new *copy* of `someString`, with all upper case characters replaced by their lower case counterparts. An invocation of `asUpper` does the opposite. Characters that are not letters of the alphabet are copied unchanged by `asLower` and `asUpper`. Thus `"Pomona47?".asLower` returns `"pomona47?"`.

It is important to remember that `asUpper` and `asLower` are not mutator methods. They return a new string rather than modifying the receiver. In fact, the `String` type *has no mutator methods*. Grace Strings are *immutable*.

The methods `asLower` and `asUpper` are extremely useful in many contexts. For example, in Figure 15.5, we showed a method that used the string method `contains` to determine whether a URL history contained a specific URL. Our method would be improved considerably if we were to make it insensitive to case.

To improve our `contains` method of class `uRLHistory`, we can do the following. Before invoking the `contains` method of `String`, we'll make both the URL history and the URL to be found lower case (or equivalently, we could make them upper case). We'll make sure we only add lower case URL's to the history in `addURL` and then modify the `contains` method of `uRLHistory` so that it first creates a copy of the URL in lower case before doing the search. The new version of the class can be found in Figure 15.6.

Exercise 15.2.3 *What is printed by each of the following code fragments:*

- a.

```
def bookTitle: String = "Programming with Grace"
bookTitle.asUpper
if (bookTitle.indexOf ("GRACE") == 0) then {
  print("No GRACE here!")
} else {
  print("We found GRACE!")
}
print(bookTitle)
```
- b.

```
var bookTitle: String := "Programming with Grace"
bookTitle := bookTitle.asUpper
if (bookTitle.indexOf ("GRACE") == 0) then {
  print ("No GRACE here!")
} else {
  print ("We found GRACE!")
}
print(bookTitle)
```

¹The part of a URL following the domain name is, in fact, case sensitive. For purposes of this example, we will ignore that issue.

```

class uRLHistoryEmpty -> URLHistory {

  var urlString: String := "\n"

  // add aURL to collection saved
  method addURL (aURL: String) -> Done {
    urlString := aURL.asLower ++ "\n" ++ urlString
  }

  // return a string representing all the URL's
  method asString -> String {
    urlString.substringFrom (2) to (urlString.size)
  }

  // Determine if aURL is in the history list
  method contains(aURL: String) -> Boolean {
    def lowerAURL: String = aURL.asLower
    urlString.contains ("\n" ++ lowerAURL ++ "\n")
  }

  // return string representing all URL's in history w/ aURL as prefix
  method findCompletions (aURLPrefix: String) -> String {
    var allCompletions: String := ""
    def lcPrefix: String = aURLPrefix.asLower

    var urlStart: Number := urlString.indexOf ("\n" ++ lcPrefix)

    while {urlStart >= 1} do {
      def urlEnd: Number = urlString.indexOf ("\n") startingAt (urlStart + 1)
      ifAbsent {urlStart}
      def possCompletion: String = urlString.substringFrom (urlStart + 1) to (urlEnd)
      allCompletions := allCompletions ++ possCompletion
      urlStart := urlString.indexOf ("\n" ++ lcPrefix) startingAt (urlEnd)
      ifAbsent {0}
    }
    allCompletions
  }
}

```

Figure 15.6: Case-insensitive uRLHistory class

15.2.5 Cutting and Pasting Strings

You already know that you can paste strings together with the concatenation operator (`++`) and with interpolation. In this section we will show you how to pull strings apart. In particular, we will show you how to extract a substring of a given string. To do this, you can use the method `substring`. The invocation

```
someString.substringFrom (startIndex) to (endIndex)
```

returns the substring of `someString` beginning at index `startIndex` and up to, and including, the character at position `endIndex`. That is, `endIndex` should be the position of the last character in the desired substring. If `endIndex` is beyond the last character in the string, then `substring` will take the characters up to and including that last character.

Thus if `urlString` is `"http://www.cs.pomona.edu"` then

```
urlString.substringFrom (8) to (10)
```

returns `"www"` and

```
urlString.substringFrom (1) to (7)
```

returns `"http://"` and

```
urlString.substringFrom (8) to (urlString.size)
```

returns `"www.cs.pomona.edu"`. while

```
urlString.substringFrom (8) to (50)
```

also returns `"www.cs.pomona.edu"`.

Let's look at the last two examples in a bit more detail. In both we gave the value 8 as the index of the first character of our substring. We then gave the first value `urlString.size` as the position of the last character to be included in the substring. The size of `urlString` is 26. Thus everything from index 8 to the end is included in the substring, because the last index to be included refers to the last element in the string. In the last example, we got exactly the same answer because 50 is beyond the end of the string, and it only takes characters up to the end of the string.

Now let's see how we can use `substring` to help us find possible completions for a partially entered URL. This is the last method in our `uRLHistory` class in Figure 15.6.

We begin by making a copy of `aURLPrefix` that is entirely lower case called `lcPrefix`. As before, we do this because we want our URL history methods to be case-insensitive, and `urlString` is all lower case. We then use `indexOf` to find the first occurrence of `"\n"++lcPrefix` in the history. Our aim is to extract the entire URL from this position to the closest following newline character, which is acting as a separator between URLs. To find the newline character, we again use `indexOf`, this time beginning our search from the position just past where we just found `lcPrefix`. We then get the substring from the history, beginning with the first position of our URL and ending with (and including) the position where we found the newline character. We keep the newline character so that it can continue to serve as a separator. We then concatenate this to `allCompletions`, which is a `String` consisting of all matching URLs found. Once we have extracted a URL, we begin to look for the next possible completion. Again, we use the method `indexOf`, but this time we begin the search from the point where we just ended. We continue to extract URLs as long as the prefix can be found in the history.

15.2.6 Trimming Strings

Throughout this chapter we have been making a very important assumption about the strings passed as parameters to our methods. We have assumed that they have no leading or trailing blanks. This is an important and perhaps overly optimistic assumption. What if the string passed to the `add` method is

```
"http://www.cs.pomona.edu "
```

Do we really want to treat this as if it is different from

```
"http://www.cs.pomona.edu"
```

Of course not. The method `trim` is very useful in cases such as this one. The invocation `someString.trim`

returns a copy of `someString` with the white space removed from both ends. Like `asLower` and `asUpper`, this is not a mutator method. In the invocation above, `someString` is not modified.

If we wanted to modify `someString` to be the string with leading and trailing blanks removed, we would say

```
someString := someString.trim
```

We show below a modified version of the `addURL` method of our `uRLHistory` class in which we throw away the spaces around a URL before adding it to `urlString`.

```
// add aURL to collection saved
method addURL (aURL: String) -> Done {
  urlString := (aURL.trim.asLower) ++ "\n" ++ urlString
}
```

This time, the `addURL` method takes care of both trimming new URLs and converting them to lower case. When we apply `trim` to `aURL`, we get a copy with all leading and trailing blanks removed. To this string we apply method `asLower`, which returns yet another `String`, this time with all letters converted to lower case. As a result, all the URLs in `urlString` are remembered as lower case, with no extraneous blanks at the beginning or end.

The `contains` method does something similar to `add`. Before checking whether a URL is found in `urlString`, we trim `aURL` and then send the message `asLower` to the new trimmed `String`.

Exercise 15.2.4 *Modify the method `findCompletions` in Figure 15.6 so that the prefix passed in to the method as a parameter is trimmed before any searching is done for completions of that prefix.*

15.2.7 Comparing Strings

You already know that you can compare `Strings` for equality with the method `==`. Grace also provides operations that allow you to compare strings using *dictionary ordering*, so one string will come before another if the first would appear before the second in a dictionary.

There is one complication in our ordering, however. That is that all of the capital letters come before all of the lower case letters in the standard ordering of characters on computers. Thus `"MAT"` comes before `"cAT"` in this ordering. If all the characters in the strings to be compared are lower case (or equivalently all upper case) then the ordering is as expected. Often we will use the methods `asUpper` and `asLower` before comparing elements.

Suppose we would like to maintain our URL history in alphabetical order. If so, we could rewrite the method `addURL` as in Figure 15.7 so that all URL's are in order.

```

// add aURL to collection saved
method addURL (aURL: String) -> Done {
  def trimmedURL: String = aURL.trim.asLower ++ "\n" ++ urlString
  var urlNotPlaced: Boolean := true
  var urlStart: Number := 2
  var urlEnd: Number := urlString.indexOf ("\n") starting (urlStart)

  while {(urlEnd > 0) && urlNotPlaced} do {
    def nextURL: String = urlString.substringFrom (urlStart) to (urlEnd-1)
    if (trimmedURL > nextURL) then { // get next URL
      urlStart := urlEnd + 1
      urlEnd := urlString.indexOf ("\n") starting (urlStart)
    } else { // found right position so insert it
      urlString := urlString.substringFrom(1) to (urlStart-1) ++ trimmedURL ++ "\n"
        ++ urlString.substringFrom(urlStart) to (urlString.size)
      urlNotPlaced := false
    }
  }
  if (urlNotPlaced) then { // put new URL at end
    urlString := urlString ++ trimmedURL ++ "\n"
  }
}

```

Figure 15.7: Adding a URL to a history of URLs maintained in alphabetical order

The method `addURL` works as follows. First we trim `aURL`, the URL to be added, and save a copy that is all lower case in `trimmedURL`. Now we need to compare `trimmedURL` with the other URLs to determine where the new one should be placed. Figure 15.8 helps to illustrate how we'll go about this. The figure shows a list of strings that are names of fruits and nuts. A new string, `grape`, is to be added to the list. We consider the first string, `apple`, so that we can compare `grape` to `apple`. The expression `"grape" > "apple"` returns `true`, indicating that `"grape"` is bigger than `"apple"`. In other words, if placed in dictionary order, `"grape"` would come after `"apple"`. So we go on to compare `"grape"` to the next string, `"banana"`. Once again, it will be bigger, so we compare `"grape"` to `"grapefruit"`. This time the comparison will fail as `"grape"` comes before `"grapefruit"` in the dictionary. Since this is the first time it fails, we know that this is the correct spot to insert the new string.

apple	←	grape
banana	←	grape
grapefruit	←	grape
mango		
peanut		
pear		

Figure 15.8: Inserting a string into a list in alphabetical order

The while loop in Figure 15.7 does exactly this. It extracts a URL for comparison from `urlString`. The variables `urlStart` and `urlEnd` give the starting and ending indices of the URL

to be extracted. Initially the values of these variables are 2 and n , where n is the position of the newline character after the first URL in the list. (We start at 2 because the first character is the newline character.) Once the comparison string has been extracted, the method compares `trimmedURL` to the comparison string. If `trimmedURL` is bigger, we know it should appear later in the list of strings, so we keep searching for the appropriate position. To do this, we update the values of `urlStart` and `urlEnd`. However, if `trimmedURL` is not larger, then we know that we have found the right place to insert `trimmedURL`. We construct a new string by concatenating the part of `urlString` up to the position where `trimmedURL` is to be placed, together with `trimmedURL`, a newline to serve as a delimiter, and the rest of `urlString`. We then remember this as the new value of `urlString`.

There is one more thing that the method `addURL` needs to handle. It is possible that `trimmedURL` needs to be inserted as the final string in `urlString`. If so, the `while` loop will complete without placing `aURL` where it belongs. Before the method terminates, we check for this case and, if necessary, concatenate `trimmedURL` to the end of `urlString`.

Exercise 15.2.5 *The condition that is checked at the end of `addURL` in Figure 15.7 takes care of the case where `aURL` is the first URL to be inserted into `urlList`. Trace through the method to convince yourself that this happens correctly.*

15.3 Characters

Unlike some languages, Grace does not treat single characters any differently from strings of length 1. However, there are some operations that only make sense for strings consisting of a single character.

Each character is represented within the computer's memory as an integer. There are various standard codes that can be used to represent characters. ASCII (American Standard Code for Information Interchange), for example, can represent 256 characters. Characters in Grace follow the Unicode encoding scheme. Unicode can represent many more characters than ASCII. The first 256 of these are the same as ASCII codes, for compatibility. You can find out the code for a character by requesting the `ord` method on a string consisting of that character.

For example, `"c".ord` returns 99, while `π.ord` returns 960. These numerical codes determine the ordering of strings. For example, `"A"` comes before `"a"` in the ordering of strings because `"A"` has code 65 while `"a"` has 97. The main important thing you need to know about these codes are that the codes for `"a"` through `"z"` are consecutive, as are those for `"A"` to `"Z"` and `"0"` through `"9"`. Generally that is all you need to know about these codes in order to use them in your programs.

15.3.1 Extracting Characters from Strings

Just as we can access individual elements of a list, we can also access the characters that make up a string. The Grace `String` class includes a method `at`, which takes a `Number` as an index and returns the character at that location. For example, if `aString` is `"Coffee"`, then `aString.at(2)` is the string `"o"`.

One common use for this method is to check whether the characters in a string all have some property, like being numeric or upper case. Consider, for example, a medical record management program that would allow a physician to enter patient information. One screen from such a program might look like that in Figure 15.9. This interface has, among other things, a `TextField` for entering the patient's weight. Say that this information is used in various parts of the program to perform a variety of calculations and that we want to be able to treat the patient's weight as an `Number`.

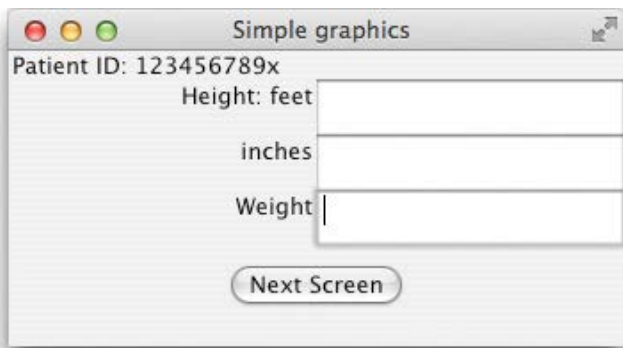


Figure 15.9: User interface for a medical record system

We can extract the information from the `TextField` with the method `text`. So, for example, if the name of the `TextField` is `weightField`, we can say:

```
def weight:String = weightField.text
```

To convert this `String` into an `Number`, we can use the `asNumber` method of the `String` type as follows:

```
def weightValue: Number = weight.asNumber
```

However, this will only work if the weight that was entered *looks like* a `Number` value.² So "154" can be made into a `Number`, but "154lbs" cannot. Furthermore, even extraneous blanks in a string, as in " 12" cause it to be rejected for conversion. (On the other hand, leading zeros, as in "00012" are allowed.)

To check whether all of the characters in `weight` are numeric, we could write a method `validNumber` as in Figure 15.10. In the method `validNumber`, the `for` loop considers the characters in `number`, one at a time. If a non-numeric character is encountered, the method returns `false`. If the `for` loop terminates after examining all of the characters, then a non-numeric character was not found, and the method returns `true`.

In the same figure is an alternative version of the method `validNumber2`, where the `for` loop takes us directly through the characters of the string. This is preferable over `validNumber` as we have no real need for the index of the elements in the body of the `for` loop.

Finally, we can take advantage of some methods defined on `String` that we have not yet introduced to make this even simpler. The method `validNumber3` in Figure 15.10 takes advantage of the built-in `String` methods, `startsWithDigit`, and also uses the fact that we can iterate through the elements of a string in a `for` loop, just as we could with lists.

²In fact, if we attempt to convert a string that is not of the proper form, an exception will be raised.

```

method validNumber (number: String) -> Boolean {
  for (1 .. number.size) do {index: Number ->
    if ((number.at(index) < "0" || (number.at(index) > "9" )) then {
      return false
    }
  }
  true
}

method validNumber2 (number: String) -> Boolean {
  for (number) do {digit: String ->
    if ((digit < "0" || ( digit > "9")) then {
      return false
    }
  }
  true
}

```

Figure 15.10: Method to check whether a String looks like a valid integer

Method `startsWithDigit` returns true if the first character of the receiver is a digit from 0 to 9. When we use this on a string of size 1, it simply tells us whether the only character in the string is a digit. The `for` loop in this method iterates over the letters in the string `number`. Thus if `number` is the string "572", then the values of `letter` in the `for` loop will run through "5", then "7", and then "2".

```

method validNumber3 (number: String) -> Boolean {
  for (number) do {letter: String ->
    if (!letter.startsWithDigit) then {
      return false
    }
  }
  true
}

```

Figure 15.11: Alternative method to check whether a String looks like a valid integer

Exercise 15.3.1 Write a method called `isAlphabetic` that returns true if and only if all the characters in a String are either lower case or upper case letters of the alphabet. Write two versions of this method. The first should be like the method `validNumber` above, using indices to access the elements of the string and using inequalities to determine if the element is alphabetic. For the second, emulate `validNumber3` above by iterating directly over the characters in the string and using the built-in String method `startsWithLetter`.

15.3.2 Performing Calculations based on Characters

We mentioned above that in Grace the `ord` method of strings returns the integer code of a character; `ord` actually returns the code for the first character of the string. As you will see in

this section, the ability to perform arithmetic operations on the codes of characters can be extremely useful.

A	. -	N	- .
B	- . . .	O	- - -
C	- . . .	P	. - - .
D	- . .	Q	- - - .
E	.	R	. - .
F	. . - .	S	. . .
G	- - .	T	-
H	U	. . -
I	. .	V	. . . -
J	. - - -	W	. - -
K	- . -	X	- . . .
L	. - . .	Y	- . - -
M	- -	Z	- - . .

Figure 15.12: A partial Morse code alphabet

To illustrate the utility of these operations, let's consider a Morse code program. This time, rather than simply allowing a user to generate a message of dots and dashes, as in Figure 15.1, our program will translate messages into Morse code. Let's keep things simple for the moment and concentrate on translating only alphabetic messages. That is, we won't allow numbers or punctuation. In fact, let's assume that all letters are upper case.

Each letter in the alphabet is represented by a unique series of dots and dashes, as in Figure 15.12. This means that it is conceptually quite straightforward to translate a message that is a string. All we have to do is consider each of the characters in the message and find its equivalent Morse code representation. So, for example, the message

I LOVE GRACE

becomes

.. .-.. - - - - - . - . - - - . - . - - - . .

The method `toMorseCode` in Figure 15.13 does exactly that. For each letter in `message`, it first converts it to lower case and then sends it off to the confidential method `morseCode` to translate the character. Numbers and letters will be handled in a general way to be discussed next. If it is a blank or punctuation, it is handled specially to provide an appropriate code.

The interesting question here is how the method `morseCode` determines the sequence of dots and dashes that corresponds to a letter or number. Let's handle the case of letters first. Numbers will be very similar.

The `morseCodeTranslator` object includes a definition for `letterCode` that sets it equal to the list of the codes for all letters (held in order).

That is, the first element of the list, the element at index 1, is the code for the letter A, ". -", the second element, at index 2, is the code for the letter B, "- . . ." and so on. All we need to do is figure out how to use character information to help us find its corresponding index in the list.

We have mentioned a number of times that characters are associated with integers, obtained by sending the message `ord` to the string. These numbers are exactly what we need for indexing the list. It would certainly be convenient if the `ord` of `"a"` was 1, with the `ord` of `"b"` equivalent to 2, and so on, but this is not the case.³ However, we can calculate the appropriate index by determining how far away the given letter is from the beginning of the alphabet. Look at the first branch of the `if` statement for method `morseCode`. Having determined that the letter `startsWithLetter`, it subtracts `"a".ord` from `letter.ord` and then adds 1. If we subtract the value of `"a".ord` from `"a".ord` and then add 1, we get 1; subtracting `"a".ord` from `"b".ord` and then adding 1 yields 2. Subtracting `"a".ord` from `"z".ord` and then adding 1 yields 26, since their `ords` are different by a value of 25. Note that this works because the `ords` of the characters `"a"` to `"z"` are consecutive. (The same is true for the characters `"A"` to `"Z"` and `"0"` to `"9"`.)

The same trick works for numbers. The identifier `numberCode` is associated with the list of codes for the numbers from 0 to 9. If `letter` represents a digit, the value of `letter.ord - "0".ord + 1` is the numeric value of the digit and hence can be used as a subscript to get the corresponding morse code for the digit.

Exercise 15.3.2 *Write a method to check whether a `String` is valid for translation to Morse code. Check whether the characters in the string are letters, numbers, or the punctuation symbols `'.'`, `','`, `'?'`. You should also allow blanks.*

³The `ord` associated with `"a"` happens to be 97, but the specific value doesn't matter.

```

type String = {
  * (n: Number) -> String
  // returns a string that contains n repetitions of self, so "abc" * 3 = "abcabcabc"

  ++ (other: Object) -> String
  // returns a string that is the concatenation of self and other.asString

  < (other: String)
  // true if self precedes other lexicographically

  <= (other: String)
  // self == other || self < other

  == (other: Object)
  // true if other is a String and is equal to self

  != (other: Object)
  \= (other: Object)
  // (self == other).not

  > (other: String)
  // true if other precedes self lexicographically

  at (index: Number) -> String
  // returns the character in position index (as a string of size 1)

  asLower -> String
  // returns a string like self, except that all letters are in lower case

  asNumber -> Number
  // attempts to parse self as a number; returns that number, or NaN if it can't

  asString -> String
  // returns self, naturally

  asUpper -> String
  // returns a string like self, except that all letters are in upper case

  capitalized -> String
  // returns a string like self, except that the initial letters of all words are in upper case

  compare (other:String) -> Number
  // a three-way comparison: -1 if (self < other), 0 if (self == other), and +1 if (self > other)
  // This is useful when writing a comparison function for \code{sortBy}

  contains (other:String) -> Boolean
  // returns true if other is a contiguous substring of self

```

Figure 15.14: Useful methods of type String

15.4 Summary

In this chapter we expanded on your earlier understanding of `Strings`, including many useful methods. The main points about strings that you should take from this chapter are as follows:

- The empty string, `""`, is a very useful string, particularly in cases where a string is to be built incrementally from nothing.
- `String` objects are immutable. There are no mutator methods that modify strings. Instead, all `String` methods that would make changes to a string return a new copy with the appropriate modifications.
- `String`'s `String` class provides many useful methods, including those listed in Figures 15.14, 15.15, and 15.16.
- `String` does not support a separate data type for individual characters. Instead it provides methods that are designed for strings of length one. The `ord` method, in particular, allows us to perform many useful operations on individual characters in strings.

15.5 Chapter Review Problems

Exercise 15.5.1 *What do we mean when we say that `Strings` are immutable?*

Exercise 15.5.2 *What is returned by each of the following method invocations when `bigString` is*

```
"I drank java on the island of Java."
```

- `bigString.indexOf("java")`
- `bigString.indexOf("java")` startingAt (9)
- `bigString.indexOf("java")` startingAt (10)
- `bigString.indexOf("an")`
- `bigString.indexOf("an")` startingAt (5)
- `bigString.indexOf("an")` startingAt (6)

Exercise 15.5.3 *What is returned by each of the following method invocations when `firstString`, `secondString`, and `thirdString` are initialized as follows:*

```
def firstString: String = "sunshine"  
def secondString: String = "Sunshine"  
def thirdString: String = " sunshine"
```

- `firstString == secondString`
- `firstString == thirdString`

- c. `firstString == secondString.asLower`
- d. `firstString == thirdString.asLower`
- e. `firstString.trim == thirdString.trim`

Exercise 15.5.4 *What is returned by `bigString.endsWith("Java")`, when `bigString` is `"I drank java on the island of Java."`*

Exercise 15.5.5 *What is returned by each of the following method invocations when `bookTitle` and `isbn` are initialized as follows:*

```
def bookTitle: String = "Artificial Intelligence: A Modern Approach"
```

```
def isbn: String = "0-13-790395-2"
```

- a. `bookTitle.asUpper`
- b. `isbn.asUpper`
- c. `isbn.substringFrom (6) to (12)`
- d. `bookTitle.substringFrom (26) to (bookTitle.size)`
- e. `bookTitle.substringFrom (26) to (bookTitle.size - 1)`

Exercise 15.5.6 *What is returned by each of the following method invocations when `bigString` is*

`"I drank java on the island of Java."`

- a. `bigString.at (6)`
- b. `bigString.at (1)`
- c. `bigString.at (0)`
- d. `bigString.at (bigString.size)`

Exercise 15.5.7 *The following method is meant to return the number of times that `pattern` occurs in the string `toSearch`. For example, if `toSearch` is*

`"ATCGGTGGGTTCCAAGGG"`

and `pattern` is `"GG"` then the method should return 5. In particular, overlapping patterns should be counted as `"GGG"` has two copies of `"GG"`

```
method patternCountOf (pattern: String) in (toSearch: String) {
  // number of times pattern found
  var count: Number := 0
  // index of the first occurrence of pattern, if any
  var index : Number := toSearch.indexOf (pattern)
  // number of characters to skip before looking for next pattern
  def skipDist: String = pattern.size
  while {index > 0} do {
    count := count + 1
  }
```

```

    index := toSearch.indexOf (pattern) startingAt (index + skipDist)
  }
  count
}

```

- a. Unfortunately, the method does not do what it is meant to do. What does the method return when pattern is "GG"?
- b. Fix the method so that it does what it was intended to do.

Exercise 15.5.8 The following method is meant to return a boolean, indicating whether pattern can be found in the string toSearch. It is meant to be case insensitive. For example, if toSearch is

```
"ATCGGTGGGTTCCAAGGG"
```

and pattern is "atcg", the method should still return true.

```

method find (toSearch: String, pattern: String) -> Boolean {
  // make sure both strings are lower case for comparison purposes
  toSearch.asLower
  pattern.asLower
  toSearch.indexOf (pattern) > 0
}

```

The method, however, does not work as expected. Please fix it.

Exercise 15.5.9 Write a method that takes a list of words (i.e., Strings) as a parameter and returns true if and only if the list is in alphabetical order. Mixed case (i.e., both lower and upper case) can be used in the words, so be sure to take that into account.

Exercise 15.5.10 Write a method that takes a name in the format "First Middle Last" and returns the name in the format "Last, First MiddleInitial". That is, given the name string "Andrea Pohoreckyj Danyluk", the method should return "Danyluk, Andrea P."

15.6 Programming Problems

Exercise 15.6.1 Write a program that allows someone to draw ovals in a window. Each time the user clicks the mouse, a small oval (having width and height of 20) should be drawn at the location of the mouse click. If the user exits the window, the program should print the locations of all the ovals drawn so far, with the following format:

```

(x1, y1)
(x2, y2)
⋮
(xn, yn)

```

That is, if four ovals have been constructed at Points (10.0, 20.0), (45.0, 23.0), (22.0, 105.0), and (150.0, 120.0), the the following should be printed:

```

(10.0, 20.0)
(45.0, 23.0)

```

(22.0, 105.0)
(150.0, 120.0)

You can keep track of the locations to be printed as a `String` that becomes longer each time the user clicks the mouse to construct a new oval.

Exercise 15.6.2 One strand of DNA can be described by a sequence of bases. Four different bases are found in DNA: adenine (A), guanine (G), cytosine (C), and thymine (T). Two strands are twisted together in the structure of a double helix. Interestingly, if you know the sequence of bases on one strand, you can infer the sequence on the other. A always pairs with T and G always pairs with C. So if one strand of DNA has a partial sequence AGCCTGG, then its complementary strand has the bases TCGGACC.

Write a program that allows a user to type a sequence of bases on one DNA strand. The string of bases should be entered in a `textField`. You should also provide a button that, when pressed, will give the user the complementary sequence, displayed in another `textField`.

Now expand your program in the following ways:

- a. The generated complementary sequence should always be displayed in upper case. However, you should allow the user to type their initial sequence in either lower or upper case (or mixed case).
- b. Include a method that will check whether the user's sequence is legitimate, i.e., whether it contains only the characters A, G, C, and T. If the sequence to be converted is not entered properly, you should print an error message, rather than the complementary sequence.

```

endsWith (possibleSuffix: String)
// true if self ends with possibleSuffix

indexOf (sub:String) → Number
// returns the leftmost index at which sub appears in self, or 0 if it is not there.

indexOf(pattern:String)startingAt (offset)
// like the above, except that it returns the first index ≥ offset.

indexOf (sub: String) ifAbsent (absent:Block0[[W]]) → Number | W
// returns the leftmost index at which sub appears in self; applies absent if it is not there.

indexOf (pattern:String) startingAt (offset) ifAbsent (action: Block0[[W]]) → Number | W
// like the above, except that it returns the first index ≥ offset.

isEmpty → Boolean
// true if self is the empty string

lastIndexOf (pattern: String) → Number
// returns the rightmost index at which sub appears in self, or 0 if it is not there.

lastIndexOf (pattern: String) ifAbsent (absent:Block0[[W]]) → Number | W
// returns the rightmost index at which sub appears in self; applies absent if it is not there.

lastIndexOf (pattern :String) startingAt (offset) ifAbsent (action:Block0[[W]]) → Number | W
// like the above, except that it returns the last index ≤ offset.

ord → Number
// a numeric representation of the first character of self, or NaN if self is empty.

replace(pattern: String) with (new: String) → String
// a string like self, but with all occurrences of pattern replaced by new

size → Number
// returns the size of self, i.e., the number of characters it contains.

startsWith (possiblePrefix: String) → Boolean
// true when possiblePrefix is a prefix of self

startsWithDigit → Boolean
// true if the first character of self is a (Unicode) digit.

startsWithLetter → Boolean
// true if the first character of self is a (Unicode) letter

startsWithPeriod → Boolean
// true if the first character of self is a period

```

Figure 15.15: Useful methods of type String (continued)

```

startsWithSpace -> Boolean
// true if the first character of self is a (Unicode) space.

substringFrom (start: Number) size (max: Number)
// the substring of self starting at index start and of length max characters,
// or extending to the end of self if that is less than max. If start = self.size + 1, or
// stop < start, the empty string is returned. If start is outside the range
// 1..self.size+1, BoundsError is raised.

substringFrom (start: Number) to (stop: Number) -> String
// returns the substring of self starting at index start and extending
// either to the end of self, or to stop. If start = self.size + 1, or
// stop < start, the empty string is returned. If start is outside the range
// 1..self.size+1, BoundsError is raised.

trim -> String
// a string like self except that leading and trailing spaces are omitted.
}
...

```

Figure 15.16: Useful methods of type String (continued)

Chapter 16

Inheritance

In large Grace programs, it is common to define collections of classes that share many common features. The graphics classes of our library are a good example of such a collection. There are clearly many similarities between the `filledRect` class and the `framedRect` class, for example.

When two classes share many common features and behaviors, it is desirable to let them share the code that implements these features. Doing so could certainly save a programmer considerable time. It is also likely to improve the reliability of the code. If a method is only written once, it only needs to be debugged once.

Grace supports a mechanism called *inheritance* that provides the means to share code easily when implementing such related classes. We have actually been using this mechanism throughout this text. In almost every program, there has been one object with a header that states that it **inherit** `graphicApplicationSize(.....)`. You may have noticed that these objects share several common features. For example, within any object that inherits `graphicApplication`, you have been able to access the name `canvas` without declaring it. This is because the variable `canvas` is defined in the class `graphicApplication`. This one declaration (which you have never actually even seen) is shared by all classes that inherit `graphicApplication`. These classes are said to *inherit* this variable from `graphicApplication`. They also inherit other features from `graphicApplication` such as the `startGraphics` method that is invoked at the end of these objects.

While we have been using the **inherit** keyword throughout the text, we have only inherited `graphicApplicationSize`. In this chapter, we will provide a more general explanation of the use of the word **inherit**. You will learn that it is not limited to just this class. The word **inherit** can be used to inform Grace that we want to define a new class or object in terms of any previously written class. We will see how this mechanism can provide the means to logically organize the definitions of related classes and to share common code. This will also enable you to better understand how classes that inherit `graphicApplication` get access to those extra features.

16.1 Inheritance as Specialization

Before we delve into the details of Grace's interpretation of **inherit**, it is useful to discuss the nature of the relationship between two objects, one of which inherits from the other. In Grace, we say that a class or object that inherits another is a *subclass* or *subobject*. A class

that is inherited in this way is said to be a *superclass*. The term inheritance describes the relationship between a subclass or subobject and its superclass. A subclass or subobject is said to *inherit* features from its superclass. Inheritance in Grace, however, is different from the notions of inheritance common in human affairs. A human child inherits some features from one parent and some from the other but never actually inherits all the attributes of just one parent. On the other hand, when a new class or object inherits from an existing class in Grace, the subclass or subobject always inherits *all* of the features of the superclass. By features in this context we mean methods, definitions, and instance variable declarations. As a result, if one class inherits from another, the subclass will include every method and every instance variable found in the superclass plus whatever new features are described within the body of the subclass.

The consequence of the fact that inheritance in Grace is so complete and involves only a single “parent” is that any object that belongs to a subclass can do everything that an object of the superclass could do. A good analogy for this relationship is the inclusion of “option packages” in the marketing of automobiles. Whether you order a base model Toyota Prius or the fully loaded Toyota Prius v, you still have a Toyota Prius. The Toyota Prius v can do anything a base model Prius can do. There are some things you can do with the high end v models that you could not do with the base model, like turning on the lane departure system, but they are both still Priuses.

The idea here is that you don’t change what something is by adding features. You just make it more special or specialized. Thus, if you take people and train them to develop special abilities, they are still people. A doctor is still a person, a pilot is still a person and so on. In Grace, an object created from a subclass has all the features of an object of its superclass. However, some of those may perform in a slightly different way, and there may be extra features as well.

16.2 Using inherit

To illustrate the use of **inherit** in Grace, consider the user interface presented by the computer solitaire card game shown in Figure 16.1. The relevant feature of this game’s user interface is the fact that the author of the program apparently liked to use little, framed subregions of the screen to display various pieces of information. For example, near the upper right corner of the screen, the program displays the current score in a small frame.

score 

Above the score display, the program indicates how many “lives” the player has left by drawing the appropriate number of hearts within a similar frame.



Toward the bottom of the screen the program uses an animated framed graphic to display the amount of time remaining in a round. It does this by drawing a silver bar that shrinks as time passes within a matching frame.

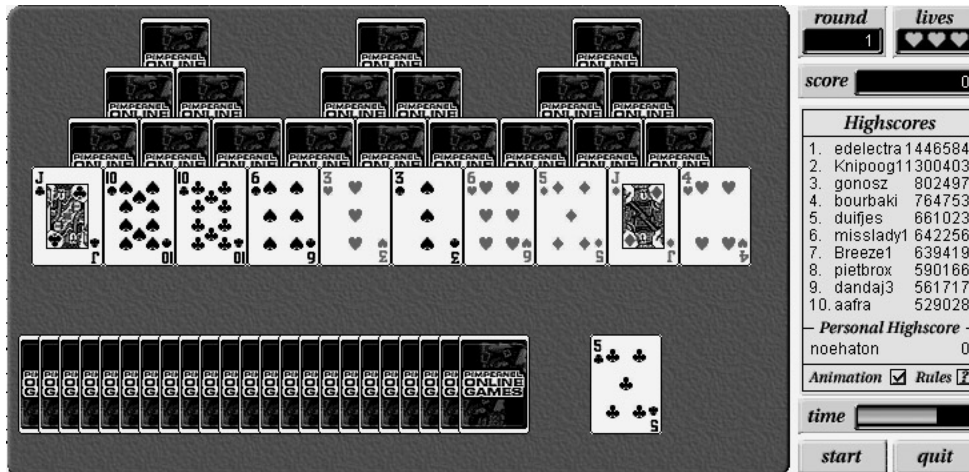


Figure 16.1: The user interface of an interactive card game.



Figure 16.2: An empty frame.



The framed displays seen in this card game provide a good example of a simple collection of classes that share important similarities. If we wanted to implement such game programs, we could define an independent class for each category of framed display used. It would be better, however, to exploit the relationships between these classes by defining a class describing an empty frame and then inheriting the features of that class to define classes of frames specialized to display information of various forms. In this way, we will only have to specify the details required to create and manipulate the frame itself once.

To begin, we will design and implement a class that implements an empty frame. We will not try to make our frames exhibit the fancy, three dimensional look of the frames used in the card game we showed to motivate this example. Instead, to keep things simple, we will form frames by overlapping a rectangle with a slightly smaller rectangle of a different color. The result will look like the images shown in Figure 16.2.

We will also keep the interface for this class quite simple. The only method we include will be named `highlight:=`. The `highlight:=` method will change the color of the frame from its initial shade of gray to bright red if it is provided with the value `true`, or change it back to cyan if the argument is `false`. This is done to draw attention to the frame. There are many other useful methods that might be included in this class, such as `moveTo` or `width:=`. Later in the chapter we will add some of these methods. For now, however, the two methods we have chosen are sufficient for our main purpose, explaining how `inherit` works.

The definition of a `framedDisplay` class that implements such frames is shown in Figure 16.3. The constructor is defined to expect parameters similar to those used to create a rectangle

```

dialect "objectdraw"

type FramedDisplay = {
  highlight:= (makeRed: Boolean) -> Done
}

// A FramedDisplay is a composite graphic intended to serve
// as a frame in which a program can display various forms
// of information.
class framedDisplayAt (pt: Point) size (dimensions: Point)
  on (canvas: DrawingCanvas) -> FramedDisplay {

  // Colors used for the frame
  def borderColor: Color = colorGen.gray
  def highlightColor: Color = colorGen.red
  def borderSize: Number = 7

  def border: Graphic2D = filledRectAt (pt) size (dimensions) on (canvas) // The border

  def body: Graphic2D = filledRectAt (pt + (borderSize @ borderSize)) // The background
    size ((dimensions.x - 2 * borderSize) @
      (dimensions.y - 2 * borderSize)) on (canvas)

  border.color := borderColor

  // Change the border's color to make it stand out
  method highlight:= (makeRed: Boolean) -> Done {
    if (makeRed) then {
      border.color := highlightColor
    } else {
      border.color := borderColor
    }
  }
}
}

```

Figure 16.3: Implementation of an empty framed display class

on the screen. For example, a construction of the form

```
framedDisplayAt (20 @ 20) size (100 @ 30) on (canvas)
```

would place a frame like that shown in Figure 16.2 at coordinates (20, 20) in the canvas. The frame would be 100 pixels wide and 30 pixels high.

An empty frame isn't very interesting. What we really want is to place some text or graphics within the frame. We could accomplish this by carefully positioning a separate graphical object over the framedDisplay. For example, the pair of constructions:

```
framedDisplayAt (20 @ 20) size (100 @ 30) on (canvas)
```

```
textAt (55 @ 40) with ("HELLO") on (canvas)
```

could be used to display the message



Of course, it would be better to define a class specialized for the task of displaying framed text so that we could use a single construction of the form

```
framedTextAt (20 @ 20) size (100 @ 30) with ( "HELLO" ) on (canvas)
```

If we wrote a definition for a `framedText` class from scratch, we would have to repeat much of the code from the `framedDisplay` class. We would still need to define all the instance variables found in `framedDisplay`. We would just add another instance variable for the `Text` object. We would also still want to define the `highlight:=` method. In addition, we might want to add methods to modify the appearance of the text. In our example, we will include just one such method, `textSize:=`.

We can easily avoid repeating the code that would be shared between these two class definitions by defining `framedText` as *inheriting* from `framedDisplay`. That is, at the start of the definition of `framedText` we will say

```
inherit framedDisplayAt (pt) size (dimensions) on (canvas)
```

Including the phrase **inherit** `framedDisplay...` tells Grace to include all the instance variables and methods found in `framedDisplay` in the new `framedText` class. This means we can omit the definitions for `highlight:=` from the body of the new class. They are inherited from the superclass `framedDisplay`. The only methods we need to define in the body of the new class will be methods to manipulate the `Text` object displayed within the frame. These methods can be written just as they would have been if `framedText` had been defined without using inheritance.¹

The complete specification of `framedText` as a subclass of `framedDisplay` is shown in Figure 16.4.

The best way to understand how **inherit** works in this example is to imagine starting out with the class defined by `framedDisplay` and then adding the extra code found in `framedText` to the end of the class definition. Now we have a class with all of the definitions, variable declarations and methods from both.

When an object is created using `framedText`, the initialization takes place in a straightforward way. Because the **inherit** clause is the first statement in the subclass, it causes the initialization to take place in the superclass using the parameters provided.

Suppose, for example, we evaluate

¹The code in the figure prefixes "fd." to `framedDisplay` because class `framedDisplay` is imported from the file "framedDisplay.grace" **as** `fd`.

```

dialect "objectdraw"
import "framedDisplay" as fd
// A FramedText object displays a specified text message on a
// background framed by a distinct border.

type TextDisplay = fd.FramedDisplay & type {
  textSize:= (newFontSize: Number) -> Done
}

class framedTextAt (pt: Point) size (dimensions: Point) with (contents: String)
  on (canvas: DrawingCanvas) -> TextDisplay {
  inherit fd.framedDisplayAt (pt) size (dimensions) on (canvas)

  def frameCenter: Point = pt + ((dimensions.x / 2) @ (dimensions.y / 2))

  def textColor: Color = colorGen.white
  def message: Text = textAt (frameCenter) with (contents) on (canvas)
  message.color := textColor
  message.fontSize := dimensions.y / 2

  positionContents

  method positionContents -> Done is confidential {
    message.moveTo (frameCenter)
    message.moveBy (-message.width / 2, message.fontSize / 2)
  }

  method textSize:= (newFontSize: Number) -> Done {
    message.fontSize := newFontSize
  }
}

```

Figure 16.4: Definition of the framedText subclass.

```
framedTextAt (200 @ 150) size (150 @ 25) with ("hello") on (canvas)
```

Looking at the class definition, this means parameter `pt` corresponds to `200 @ 150`, width is 150, height is 25, `contents` is `"hello"` and the `canvas` parameter is just the current value of `canvas`. Because the first line in the class definition is

```
inherit fd.framedDisplayAt (pt) size (dimensions) on (canvas)
```

the code in class `framedDisplay` will be executed with those values of the parameters. For example, `border` will be initialized to be a filled rectangle at `(250, 15)` with dimensions 150 by 25, while `body` will start at `(257, 22)` and will be 143 pixels in width and 18 pixels in height.

Once that code has been executed, the code in the new class (including initialization of the new definitions and variables – if there were any) in `framedText` will be performed. For example, the text item will be created on the canvas and method `positionContents` will be executed. The object is now fully initialized and ready to be used in the program.

The important point here is that both the inherited code and the new code need to have the appropriate initialization code run, and that the initialization of the inherited code is performed before that of the new code. In this example, that order makes no difference, but it may in other cases where the new code updates values defined in the inherited code. The code in method bodies is of course not executed until the methods are requested.

An **inherit** statement should only appear as the first command within a class body. If you do not include such a statement, then Grace assumes the class inherits from the default superclass `graceObject`, which contains default implementations of `==`, `!=`, and `asString`.

The default implementation of `==` is object identity. That is, if `o` and `o'` denote values then `o == o'` will return true if and only if they are the same object. Later we will discuss how we can change that definition.

The class definition shown in Figure 16.4 includes one public method that can be used to change the appearance of the text, `textSize:=`. When this method is invoked, the text should also be repositioned so that it will still be centered within the frame. A confidential method, `positionContents` is included to perform this task. This confidential method depends on an instance variable, `frameCenter`, which is initialized in the class initialization code to refer to the Point corresponding to the center of the frame.

The public method `textSize:=` defined in `framedText` can be applied to any object of the class in the usual way. Thus, if we define

```
def welcome: framedText = framedTextAt (20 @ 20) size (100 @ 30) with ("Hello") on (canvas)
```

we can then legally execute instructions such as

```
welcome.textSize:= 24
```

In addition, we can invoke public methods defined within `framedDisplay` on a `framedText` object. For example, we can change the color of the border around the text by saying

```
welcome.highlight := true
```

This is the advantage of using inheritance when appropriate. We get to use the code included in the superclass definition without having to rewrite it or even copy and paste it.

Exercise 16.2.1 *It would be helpful if the `framedText` class included a method named `text:=` that could be used to change the text displayed. The method would take a single String as a parameter and replace the text displayed in the frame with the contents of this parameter.*

Provide a definition for such a `text:=` method that could be added to the definition of the `framedText` class shown in Figure 16.4.

Exercise 16.2.2 We have only introduced a few of the mechanisms Grace provides to support the use of inheritance to define new classes. If we limit ourselves to these mechanisms, then when a new class inherits an existing class, it inherits the existing methods of the superclass “as is.” This is not always appropriate. To understand why, suppose we had added the following definition for a `moveBy` method in the `framedDisplay` class shown in Figure 16.3.

```
// move the parts of the framed display by offsets
method moveBy (xoff: Number, yoff: Number) -> Done {
    border.moveBy (xoff, yoff)
    body.moveBy (xoff, yoff)
    positionContents
}
```

Explain what would happen if we then declared a variable

```
def stopSign: framedText = framedTextAt (100 @ 100) size (80 @ 25) with ("STOP") on (canvas)
```

and then executed the statement

```
stopSign.moveBy (0, 50)
```

16.3 Confidential features are accessible in subclasses

In Section 8.1 we noted that features that are declared to be confidential are only accessible in the object in which it is defined or in any other object that inherits it. To illustrate this, suppose we wanted to create a class that would inherit from `FramedDisplay` to implement a display like the one used to show the time remaining as in Figure 16.1. It will draw a bar within the frame in such a way that the fraction of the frame filled is equal to the fraction of the time remaining. The picture below shows what our display would look like when set to indicate that about 3/4 of the time had expired.



We will design a version of this class with a very minimal interface. We will name the class `framedBarMeter`. The constructor for the class will expect all the parameters required when constructing a `framedDisplay` plus a value between 0 and 1 for the initial fraction to be displayed. Thus, execution of

```
framedBarMeterAt (350 @ 20) size (80 @ 20) fraction (0.75) on (canvas)
```

would create an 80 pixel by 20 pixel frame positioned at (350,20) on the canvas with a bar that filled three quarters of the frame.

The only method we will include in `framedBarMeter` is `fraction:=`, which will resize the bar based on the parameter value passed to the method (which should again fall between 0 and 1). We will define the class by inheriting `framedDisplay`, so it will also be possible to use the `highlight` method with an object of this class.

One aspect of implementing this class that is a bit tricky is positioning the bar within the frame and determining the appropriate size for the bar. The bar will simply be created from class `filledRect`. We would like to place this filled rectangle just to the right of the left side of


```

dialect "objectdraw"
import "framedDisplay" as fd

type FramedBarMeter = fd.FramedDisplay & type {
  fraction:= (newFraction: Number) -> Done
}

// A framedBarMeter displays a variable length rectangle on a
// background framed by a distinct border.
class framedBarMeterAt (pt: Point) size (dimensions: Point) fraction (frac: Number)
  on (canvas:DrawingCanvas) -> FramedBarMeter {
  inherit fd.framedDisplayAt (pt) size (dimensions) on (canvas)

  // how far inside frame to start bar
  def inset: Number = borderSize + 2
  def barOffset: Point = inset @ inset

  // maximum length of bar
  def maxWidth: Number = dimensions.x - 2 * inset

  // create the blue bar
  def bar: Graphic2D = filledRectAt (pt + barOffset)
    size ((maxWidth * frac) @ (dimensions.y - 2 * inset)) on (canvas)
  bar.color := colorGen.blue

  // reset bar as a fraction of its maximum length.
  method fraction:= (newFraction: Number) -> Done {
    bar.width := maxWidth * newFraction
  }
}
}

```

Figure 16.5: A framedBarMeter class that references a confidential superclass member

the frame's border and set its width and height so that it always almost fills the frame from top to bottom and fills the frame from side to side when the fraction has been set to 1.

To place the filled rectangle for the bar in the correct position initially, we will need to determine the point just inside the upper left corner of the frame. We would like to leave 2 pixels between the edge of the body and the bar. If we write

```

def inset: Number = borderSize + 2
def barOffset: Point = inset @ inset

```

then it should be clear that the point corresponding to the upper left corner of the bar can be computed by adding barOffset to point.

Recall that borderSize was defined in framedDisplay, and is by default confidential. However, confidential features are accessible both in their own class or object definition as well as in any classes or objects inheriting from that class. Thus we can access borderSize in class framedBarMeter.

A complete definition of the framedBarMeter class is provided in Figure 16.5.

Exercise 16.3.1 The program whose interface is shown in Figure 16.1 included a numeric display that presented the current score within a simple frame:

score 0

Define a class named `framedCounter` to implement such a display by inheriting from the `framedDisplay` class presented in Figure 16.3. The main differences between the behavior of the class you define and the `framedText` class we have considered will be that the class should include an `increment` method to increase the score by an integer amount provided as a parameter to the method. The initial value displayed should be 0 (i.e. the only parameters the constructor should expect are those passed to the class `framedDisplay`).

16.4 Inheritance Hierarchies

In Exercise 16.3.1, we suggested implementing a `framedCounter` class by inheriting from `framedDisplay`. The function of the `framedCounter` class is to display the value of a counter within a `framedDisplay`. As suggested in the exercise, it should include an `increment` method that increases the value of the counter displayed by an amount provided as a parameter.

To define this class as an extension of `framedDisplay`, you would define a `Text` instance variable and construct a `Text` object positioned within the frame to display the value of the counter. While this `Text` object would be used to display numeric data, it would still be a `Text` object just like the one placed within a frame by the `framedText` class. As a result, an alternative to the approach suggested in the exercise is to define the `framedCounter` class as an extension of `framedText`. To show how this would be done, we will define such a `framedCounter` class.

Our definition of the `framedCounter` class is shown in Figure 16.6. The class is really very simple. We inherit from the superclass and then define one new instance variable, `counter`, and one new method, `increment`. What makes this class interesting is that it is defined by extending a class that was itself defined as an extension of a third class.

When one class is defined by extending another, we know that the new class inherits all the instance variables and methods of the class it extends. To understand the definition of `framedCounter`, it is important to recognize that this applies both to features explicitly defined in the class being extended and to features that the extended class inherits from its superclass. For example, the `highlight:=` methods that `framedText` inherits from `framedDisplay` are inherited by `framedCounter` from `framedText`. In addition, of course, if `framedDisplay` had been defined by extending yet another class, `framedCounter` would inherit all the features of this fourth class.

To make sense of such complex inheritance relationships it is often helpful to draw a diagram depicting the connections between groups of classes. We now have a total of four classes in our collection of “framed displays.” The diagram in Figure 16.7 captures the inheritance relationships among these four classes. It shows that we have defined two classes that directly extend `framedDisplay` and that we have now defined a fourth class, `framedCounter` that extends `framedText`. The structure represented by this diagram is called the *inheritance hierarchy* for our collection of framed display classes.

Recognizing that it is possible to extend a class that was itself defined as an extension, we must reconsider the meaning of the terms subclass and superclass. Class `framedCounter` is

```

dialect "objectdraw"
import "framedDisplay" as fd

type TextDisplay = fd.FramedDisplay & type {
  increment(amount: Number) -> Done
}

class framedCounterAt (pt: Point) size (dimensions: Point) on (canvas: DrawingCanvas) -> TextDisplay {
  inherit fd.framedDisplayAt (pt) size (dimensions) on (canvas)
  def shift: Point = 120 @ 10

  var counter: Number := 0
  def frameCenter: Point = pt + (dimensions / 2)

  def textColor: Color = colorGen.white
  var contents: String := "0"
  def message: Text = textAt (frameCenter) with (contents) on (canvas)
  message.color := textColor
  message.fontSize := dimensions.y / 2
  positionContents

  def scoreLoc: Point = pt + (dimensions / 2) - shift

  method positionContents -> Done is confidential {
    message.moveTo (frameCenter)
    message.moveBy (- message.width / 2, message.fontSize / 2)
  }

  method increment (amount: Number) -> Done {
    counter := counter + amount
    message.contents := counter.asString
    positionContents
  }
}

object {
  inherit graphicApplicationSize (400 @ 400)
  def counter: TextDisplay = framedCounterAt (200 @ 50) size (100 @ 40) on (canvas)
  method onMouseClick (pt: Point) -> Done {
    counter.increment (3)
  }
  startGraphics
}

```

Figure 16.6: Definition of framedCounter as an extension of framedText

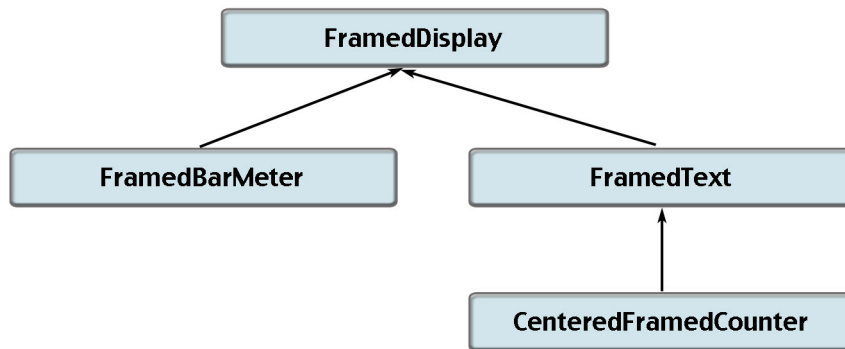


Figure 16.7: Inheritance hierarchy diagrammed for subclasses of `framedDisplay`

clearly a subclass of `framedText` because it directly extends `framedText`. As we discussed in the first section of this chapter, however, the more fundamental reason to describe `framedCounter` as a subclass of `framedText` is that a `framedCounter` object can do anything a `framedText` object can do (and more) because it possesses all the features that a `framedText` object possesses. In this sense, of course, it is fair to say that `framedCounter` is also a subclass of `framedDisplay` since a `framedCounter` possesses all the features associated with a `framedDisplay`.

In Grace, therefore, we say that a class `b` is a subclass of another class `a` if `b` inherits from `a` or if `b` inherits from some other class `c` and `c` is a subclass of `a`. A class `a` is a superclass of class `b` if `b` is a subclass of `a`. In terms of the diagram shown in Figure 16.7, one class is a subclass of another if there is a path in the diagram from the first class up to the second.

16.5 Overriding Method Definitions

In some situations, when we extend a class, we need to do more than add new instance variables and methods. We also need to change the behavior of some of the existing methods of the superclass so that they function appropriately for the new class. Grace makes this possible by allowing us to define a method in the body of a subclass with the same name and parameters as a method in the superclass. In this case, the new method is said to *override* the original method. As this name suggests, this means that the new method in some sense replaces the old method. When one uses the name shared by the new and inherited method in a method invocation, Grace will execute the code found in the new method body rather than the inherited code. This makes it possible to define a subclass in which we both add methods and instance variables and specialize some of the existing methods to better fit the purpose of the new class.

As a simple example, recall that when we first defined the `framedDisplay` class, we included a method named `highlight:=`. As defined, this method changes the color of the border of the frame. As the name `highlight:=` suggests, however, their logical function is not to change the border's color but rather to simply make some change in the appearance of the display that

will draw the user's attention.

If we want to draw attention to a `framedText` display, it might be more appropriate to change the color of the text than the color of the border. We can specify that we would prefer this behavior by overriding `highlight:=` in the definition of `framedText`. In particular, if we add a definition for `highlight:=` of the form:

```
method highlight := (makeHighlight: Boolean) -> Done {  
  if (makeHighlight) then {  
    message.color := highlightColor  
  } else {  
    message.color := textColor  
  }  
}
```

to the definition of the `framedText` class it will override the definition of `highlight:=` included in `framedDisplay`. A complete, revised specification of the `framedText` class including this definition of `highlight:=` is shown in Figure 16.8. Given this version of the class, if we define and associate a variable with a `framedText` object by saying

```
def gloat: FramedText = framedTextAt (100 @ 200) size (200 @ 50) with ("You lost!") on (canvas)
```

executing the invocation:

```
gloat.highlight := true
```

will change the color of the text "You lost!", but the color of the surrounding frame will be left unchanged. At the same time, if we were to invoke the `highlight` method on a object belonging to a different subclass of `framedDisplay` such as `framedBarMeter`, the original version of `highlight` would be accessed and the color of the border would change.

Exercise 16.5.1 *Define a class named `framedTextLeft` which behaves just like the version of `framedText` shown in Figure 16.8 except that the text displayed is left justified within the frame rather than centered. Do this by extending `framedText` and overriding `positionContents`. Note that the declaration of `message` is visible in the new subclass, making this change possible.*

16.5.1 Subtyping and Inheritance

Recall that in Section 5.5 we defined what it means for one type to extend another. Basically, one type extends another if the first has all of the methods of the second (and possibly some more). Thus we saw that both `Graphic2D` and `Text` extended `Graphic`.

If type B extends type A then an element with type B also has type A because it has all the features required to be an A (with possibly others as well).

When we define a subclass `b` of class `a`, then we begin with all the features of `a` and then potentially add more **defs**, **vars**, or **methods**, or override existing methods.

If class `a` generates objects of type A, and `b` is defined as a subclass of `a`, then its type B will extend A. Clearly adding new public features results in extending the type. Grace does not allow you to drop any methods in creating a subclass, so that is compatible with a type extension.

What about overriding methods in a subclass? If the types of the parameters and return type are the same in the original and overriding methods, then all is well. However, if they change, we might run into serious issues as a method in the subclass might not be able to

```

dialect "objectdraw"
import "framedDisplay" as fd
// A FramedText object displays a specified text message on a
// background framed by a distinct border.

type TextDisplay = fd.FramedDisplay & type {
  textSize:= (newFontSize: Number) -> Done
}

class framedTextAt (pt: Point) size (width: Number, height: Number) with (contents: String)
  on (canvas: DrawingCanvas) -> TextDisplay {
  inherit fd.framedDisplayAt (pt) size (width, height) on (canvas)

  def frameCenter: Point = pt + ((width / 2) @ (height / 2))

  def textColor: Color = colorGen.white
  def highlightColor: Color = colorGen.red

  def message: Text = textAt (frameCenter) with (contents) on (canvas)
  message.color := textColor
  message.fontSize := height / 2

  positionContents

  method positionContents -> Done is confidential {
    message.moveTo (frameCenter)
    message.moveBy (-message.width / 2, message.fontSize / 2)
  }

  method textSize:= (newFontSize: Number) -> Done {
    message.fontSize := newFontSize
    positionContents
  }

  method highlight := (makeHighlight: Boolean) -> Done {
    if (makeHighlight) then {
      message.color := highlightColor
    } else {
      message.color := textColor
    }
  }
}

```

Figure 16.8: Definition of framedText with specialized highlight:= method.

function with the parameters expected by the superclass. Thus the type of object generated by the subclass might not be an extension of that of the superclass. As a result we will insist that when a method is overridden in a subclass the types of the parameters and return type are the same in the subclass and superclass.²

Because of the restrictions on overriding methods, subclasses will always result in objects whose type extends that of the objects generated by the superclass.

For example, if we have defined the following variables

```
var someFrame: FramedDisplay
var someText: TextDisplay
var someCounter: CounterDisplay
```

all of the following forms of assignment statement would be considered legal:

```
someFrame := framedDisplayAt (...) ...
someFrame := framedTextAt (...) ...
someFrame := framedCounterAt (...) ...
someText := framedTextAt (...) ...
someText := framedCounterAt (...) ...
someCounter := framedCounterAt (...) ...
```

In addition, the assignments

```
someFrame := someText
someFrame := someCounter
someText := someCounter
```

are also legal.

Grace provides the same flexibility when determining whether the actual parameters provided in a method invocation are acceptable. An actual parameter expression is considered compatible as long as its type is the same as or an extension of the type specified in the declaration of the corresponding formal parameter. For example, suppose we defined a new class to keep track of a collection of text frames used in some program. To make it possible to add frames to such a collection, we might define a method with the header

```
method addFrame (someText: FramedText) -> Done { ... }
```

If `scoreDisplays` was the name of an object of this collection class, the invocations

```
scoreDisplays.addFrame (someText)
scoreDisplays.addFrame (someCounter)
```

would both be considered valid.

On the other hand, the assignments

```
someCounter := someFrame
someText := someFrame
```

would be illegal because, for example, a counter needs to provide an increment method and a general `FramedDisplay` cannot do that. Similarly the invocation

```
scoreDisplays.addFrame (someFrame)
```

would all be considered illegal. We note all these statements are illegal even if they occur in a context like

```
someFrame = framedCounterAt (...) ...
```

² Grace actually allows you to specialize the return type of a method – that is, replace the return type by a type that extends it – in a subclass definition. However, we will not need that capability here. Also, we note that changing the types of parameters can also cause problems with method requests of **self**, even ignoring issues of whether the resulting type extends the other. However, these are outside of the scope of this text.

```

someCounter = someFrame
someText = someFrame
scoreDisplays.addFrame (someFrame)

```

because the Grace safety checks are not powerful enough to understand that this would work at run time.. When determining the validity of an invocation or assignment statement, Grace only considers the types of the expressions and variables used in the statement or invocation, not the code that precedes the assignment.

In situations like this where Grace will reject a statement even though it is obvious to the programmer that the objects being used will be of an appropriate class, it is possible to tell Grace to accept the statement by using a match statement which ensures that the value really does have the correct type.

For example, if we were confident that `someFrame` was currently associated with a `CounterDisplay` we could say

```

match (someFrame)
  case {fc: CounterDisplay -> someCounter := fc}
  case {_ ->
    print "This isn't supposed to happen! someFrame is not a CounterDisplay
  }

```

When the program is run, Grace will check that the object associated with the name `someFrame` has type `CounterDisplay` before it makes the assignment. If it fails, the program will print an error message. (*Later we will see how to use exceptions to make the program terminate or handle the error if the match fails.*)

Exercise 16.5.2 *Suppose that classes named `a`, `b`, `c`, `d`, and `e` have been defined with the following class headers:*

```

class a -> A {...}

class b -> B {inherit d ...}

class c -> C {inherit a ...}

class d -> D {inherit a ...}

class e -> E {inherit d ...}

```

We will also assume that each inheriting class adds new methods and that their types are non-trivial extensions. For example, `B` is a non-trivial extension of `D`. Assume as well that the following variables have been defined:

```

var someA: A
var someB: B
var someC: C
var someD: D
var someE: E

```

- a. *Draw a diagram of the inheritance hierarchy relating these five classes similar to the diagram shown in Figure 16.7.*
- b. *Indicate which of the following assignment statements would be identified as errors during compilation.*

- `someD := someB`
- `someE := someA`
- `someA := someE`
- `someA := a (...)`
- `someD := someC`

c. Indicate whether each of the following code fragments would always, sometimes, or never result in the assignment statements in the `match`-`case` being executed when the program is run (assume that all variables have non-null values):

- `someA := d (...)`
`match (someA)`
`case{b: B -> someB := b}`
- `someA = b (...)`
`match (someA)`
`case{d: D -> someD := d}`
- `someA = c (...)`
`match (someA)`
`case{d: D -> someD := d}`
- `someA = someD`
`match (someA)`
`case{b: B -> someB := b}`
- `someA = someD`
`match (someA)`
`case{c: C -> someC := c}`

16.5.2 Overriding the default `==`, `!=`, and `asString` methods

We mentioned earlier that all classes implicitly inherit from the class `graceObject`. In particular if you define a class or object that does not have an **inherit** clause, Grace implicitly assumes that the new class inherits from `graceObject`. While it is rarely useful to actually construct an object from this class, we will see that it provides useful default implementations of some important methods that can be inherited and/or overridden.

The type of objects generated from class `graceObject` is `Object`:

```
type Object = {
  // true if other is same as self
  == (other: Object) -> Boolean
  // true if other is not same as self
  != (other: Object) -> Boolean
  // return string representation of object
  asString -> String
  ...
}
```

Object has other methods not shown above because they will not be relevant for us in this text. We will see below that it can be useful to declare identifiers (especially formal parameters) of type Object when we wish to accommodate any kind of object in a method.

Because the class `graceObject` implements `Object`, it has implementations of methods `==`, `!=`, and `asString`. The version of `==` defined in the `graceObject` class returns true if and only if the two expressions refer to the same object. Having the same structure and values of `defs` and `vars` is not sufficient. Thus if we have

```
def o = object{}
def o' = object{}
```

Then `o == o'` will return false, while `o == o` and `o' == o'` will both return true. The definition of `!=` simply returns the opposite of `==`. As we will see later, that definition will be very handy when we redefine `==`.

As you can imagine, the default version of `asString` couldn't possibly provide much information as an empty object has no `defs` or `vars` to provide information. As a result, it simply returns the string "an object".

It is very common to redefine these methods when we define new classes. You should always make it a habit to redefine `asString` to provide some information about the object.

It is also often useful to redefine `==`. This is especially true when we have immutable classes like `Point`, `String`, and `Number`, as we want them to be treated more as *values*, rather than objects with separate identities. For example, for the `Point` objects constructed using the `@` operator, points are equal if and only if they have the same x and y coordinates. Thus `(2 @ 3) == ((1+1) @ (4 - 1))` returns true.

As an example to see how we can override these methods, we define a `colorPoint` class in Figure 16.9. In this class we override both the `==` and `asString` methods.

The overridden definition of `asString` returns a string identifying the object as a `colorPoint` and provides information on its location and color.

The new version of `==` is more complex. You will notice that the definition of `==` in type `Object` takes a parameter of type `Object`. Thus we need to provide code that will determine whether any object at all is equal to the `colorPoint` object. If the other object is of type `ColorPoint` then we will say it is equal to the receiver if their colors and locations are the same. Otherwise, we will say they are not equal. In particular, if the other object does not have type `ColorPoint` then we will always answer false.

The code provided supports this specification. However, it involves a new kind of use of the `match` expression, introduced in Section 7.7.

When the `match` expression was introduced earlier, it was a compact way of using a single value to determine which code to be evaluated. The example provided there was to randomly choose a color depending on a numeric code.

In order to write our new `==` we need to use the `match` statement and the type of a value to determine which code to evaluate. Consider the following example:

```
match (e)
  case {n: Number ->
    print (n+1)
  }
  case {s: String ->
    print ("Got " ++ s)
```

```

dialect "objectdraw"

type ColorPoint = {
  color -> Color
  point -> Point
}

class colorPointAt (p: Point) color (c: Color) -> ColorPoint {
  method color -> Color {c}
  method point -> Point {p}

  // two color points are equal if they have the same point and color
  // if other is not a colorPoint then the answer is false
  method == (other: Object) -> Boolean {
    match (other)
    case {o: ColorPoint ->
      (o.color == c) && (o.point == p)
    }
    case {_ ->
      false
    }
  }

  method asString -> String {
    "colorPoint with point = {p} and color = {c}"
  }
}

```

Figure 16.9: Class illustrating redefinition of == method

```

}
case {u ->
    print "who knows!?! "
    print "I don't!! "
}

```

In the example, the print statement to be executed will depend on the type of the current value of `e`. For example, if the value of `e` is the number 7, then it will print 8. If the value of `e` is “milk” then it will print Got milk. If the value of `e` is `true` then it will print who knows!?! because Boolean extends Object.

The match statement will go through the associated case clauses one at a time until it succeeds in finding a match. A case statement has two pieces: the part before the `->`, and the part after. The part before the `->` is a pattern-matching expression. If we have a pattern of the form `p: Tp`, then it will succeed if the value in the match expression has type `Tp`. If it does succeed then the identifier `p` in the pattern will be associated with the value of the matched expression and the action to the right of the `->` will be executed. The cases are examined from top to bottom until one succeeds. When the first case succeeds, the corresponding action will be evaluated and control will pass to the statement after the match statement. If none succeed, a run-time error will occur, so you should be careful to make sure there is always a case that will succeed.

If there is a case with an identifier but no type, then that case will always succeed. That is the situation with the last case in the example above. If the action portion of the case does not use the value of the matched expression, the variable in the pattern can be replaced by an underscore: `_`. Thus the match statement above could have been written as:

```

match (e)
  case {n: Number ->
    print (n+1)
  }
  case {s: String ->
    print ("Got " ++ s)
  }
  case {_ ->
    print "who knows!?! "
    print "I don't!! "
  }
}

```

Let’s go through this example one more time: The system first checks to see if the value of `e` has type `Number`. If so, it associates the value with the identifier `n` of type `Number`, and prints the result of adding 1 to `n`. If the value of `e` is not a number, it continues to the next case and checks to see if the value of `e` has type `String`. If so, the value is associated with `s` and its value is added to the end of string “Got ” and then printed. If it is neither a number nor a string, it goes to the last case clause. The last case statement has an identifier without a type as its guard, and so always succeeds. Thus if the value of `e` is neither a number nor a string, the match statement will print “who knows!?!” and then “I don’t!!”.

Let’s go back to look at the overriding definition of method `==` in Figure 16.9. A match expression is used to determine whether or not the parameter is of type `ColorPoint`. If so, we give it the name `o` (which has static type `ColorPoint`) and compare its color and location with those of the receiver. If the parameter does not have type `ColorPoint` then we cannot ask it for

its color or point, so we simply return **false**. Thus an object generated by `colorPoint` is `==` to an object `o` iff that object also has type `ColorPoint` and has the same color and location.

What about `!=='`? Why didn't we need to override its definition as well? We'll come back to see that nothing needs to be changed when we talk about dynamic method dispatch in Section 16.5.4.

16.5.3 Using aliases to access overridden methods

In many situations, we do not really want to totally change the steps performed by a method when we override it. Instead, we simply want to extend the steps performed to account for the new instance variables added in the subclass. For example, suppose that we had included a `moveBy` method in our original definition of `framedDisplay`. This method would move the rectangles used to construct the frame and its background. A subclass like `framedText` would need to override this method because if it simply inherited the method, telling a `framedText` object to move would move the frame but leave the text behind. Thus the `framedText` class needs to override the `moveBy` method so that it moves both the frame and the text.

The task of overriding a method like `moveBy` is a bit like defining a subclass. In a subclass we still need to make sure that the actions specified in the superclass constructor are performed (via the **inherit** statement), but we want to add additional steps to the initialization process. Grace uses an **alias** clause to provide a means of accessing an overridden method from within a subclass.

To see how this mechanism is used, let us consider the details of how we would override a `moveBy` method in `framedText`. First, let us assume that we have added to `framedDisplay` a definition of `moveBy` as follows:

```
// move the parts of the framed display by offsets
method moveBy (xoff: Number, yoff: Number) -> Done {
  border.moveBy (xoff, yoff)
  body.moveBy (xoff, yoff)
}
```

Now, we need to add a method definition to the `framedText` class that will look something like this:

```
// move the display by specified offsets
method moveBy (xoff: Number, yoff: Number) -> Done {
  ... // code to move the body and border  ...
  positionContents
}
```

That is, we first need to write instructions to move the frame. Then, we can simply call `positionContents` to ensure that the text is repositioned appropriately.

The question is what to insert to move the body and border. While we could just copy the code from `framedDisplay` we can instead invoke the original `moveBy` method from the superclass. We do this by giving the method in the superclass a new name when we inherit it in the subclass. We can do this by adding an **alias** clause to the **inherit** statement:

```
inherit framedDisplayAt (pt) size (dimensions) on (canvas)
  alias fdMoveBy(,_) = moveBy(,_)
```

This provides a new name for the `moveBy` method provided in the framed display. The parentheses and underscores separated by commas show that the method takes two parameters.

You could put in names for the parameters, but that is not necessary, as the wild card character “_” shows the presence of arguments and the names wouldn’t be used anywhere. Both the new name and the original must have the same number of arguments.

Introducing this new name is important, because we are about to override the definition of `positionContents`. In that redefined method, we can just write

```
fdMoveBy (xoff, yoff)
```

when we want to refer to the definition from the framed display.

In general, we can refer to any method of a superclass by adding an **alias** clause as part of the **inherit** statement for that superclass.

Thus, the complete definition we should include in `framedText` to override `moveBy` appropriately is:

```
// move the display by specified offsets
method moveBy (xoff: Number, yoff: Number) -> Done {
  fdMoveBy (xoff, yoff)    // move the body and border
  positionContents
}
```

Of course, if we add a `moveBy` method to `framedDisplay`, we should modify all the subclasses that extend `framedDisplay` so that `moveBy` works appropriately for all subclasses.

At this point, the only other subclass of `framedDisplay` that we have defined is `framedBarMeter`. Just as we did with `framedText`, we will add a method to `framedBarMeter` that will override the `moveBy` found in `framedDisplay`. If we add an **alias** clause as above to the **inherit** statement of `framedBarMeter`, then the code that should be added to `framedBarMeter` is:

```
method moveBy (xoff: Number, yoff: Number) -> Done {
  fdMoveBy (xoff, yoff)
  bar.moveBy (xoff, yoff)
}
```

We only need to use an **alias** clause to add a new name for a method from the superclass if

- we are planning on overriding that method, and
- we need to invoke the old method in the overriding method body (or indeed anywhere else in that subclass definition).

Thus there is never a reason to provide an **alias** if a method is not being overridden, and even when it is overridden, it only needs an **alias** when the old definition is still needed in defining the new class.

Exercise 16.5.3 *In the preceding section we defined versions of `highlight:=` in `framedText` that highlighted the display by changing the color of the text rather than the color of the frame’s border. If we wanted to highlight a text display more boldly (even garishly) we could change the colors of both the text and the frame’s border. Please show the code that would be used to define such a version of `highlight:=`. The code you provide should work if we used it in place of the copies of `highlight:=` included in the definition of `framedText` found in Figure 16.8.*

16.5.4 Dynamic method invocation

When we invoke a method that has been overridden, Grace executes the code for the version of the method associated with the class of the object on which the method is invoked.

Sometimes, it is not possible to determine which version of a method should be used until the program is executing. For example, suppose that the `moveBy` method has been overridden as described in the preceding sections. Now, assume we declare a variable

```
var someFrame: FramedDisplay
```

and later execute the invocation

```
someFrame.moveBy (100, 50)
```

Which version of the `moveBy` method will be executed in this case?

At first, it may seem obvious that the version of `moveBy` included in the definition of `framedDisplay` would be used. Recall, however, that in Grace objects that are generated by a subclass of a class have all the methods of the superclass, and hence have the same type. Thus, while `framedText` generates objects of type `FramedText`, because `FramedText` extends `FramedDisplay` (i.e., it has all the same methods as well as possibly some extras), any object generated by `framedText` has type `FramedDisplay` as well as type `TextDisplay`.

Thus, it is legal to assign a value to the `someFrame` variable by executing the assignment `someFrame := framedTextAt (50 @ 50) size (80 @ 25) with ("Follow me") on (canvas)`

If the version of `moveBy` defined in `framedDisplay` was used to execute the invocation

```
someFrame.moveBy (100, 50)
```

after `someFrame` was assigned the `FrameText` object described by the construction, then the frame would be moved leaving the text “Follow me” behind. Fortunately, when the value associated with `someFrame` is a member of a `framedText`, Grace will execute the version of `moveBy` associated with `framedText`.

In general, Grace invokes the version of a method associated with the object (e.g., as defined in the object or class that defined it) regardless of the type of the name through which the object is accessed. Recall that the process of identifying the correct method to use is called *dynamic method invocation*. In the case of classes and subclasses, it ensures that invocations of overridden methods behave as expected.

Dynamic method invocation can be very important when one method in a class is defined in terms of another method of the class which is itself later overridden. For example, consider how we would add a `moveTo` method to `framedDisplay`. One might expect that the process of adding this method to `framedDisplay` would be similar to that of adding `moveBy`. First, we would add the method to `framedDisplay` and then override it in each subclass. In fact, the way Grace handles overridden methods makes implementing `moveTo` much simpler.

Let’s start by thinking about what the definition of `moveTo` would look like in the `framedDisplay` class. Recall that we can move an object to a specified location by moving it by the correct offsets from its current location. That is, we can implement a `moveTo` by using a `moveBy` method. A definition of `moveTo` for `framedDisplay` based on this observation is shown below.

```
method moveTo (newPt: Point) -> Done {  
  moveBy (newPt.x - border.x, newPt.y - border.y)  
}
```

It simply moves the entire `framedDisplay` by the offsets needed to move the border from its current position to its desired final position.

The invocation

```
moveBy (newPt.x - border.x, newPt.y - border.y)
```

is just an abbreviation for

```
self.moveBy (newPt.x - border.x, newPt.y - border.y)
```

It is important to remember this because it reinforces the fact that the invocation refers to the `moveBy` method of `self`, the object whose `moveTo` method was invoked. Grace interprets `self` dynamically. That is, even though the reference to `self` appears in the text of the `framedDisplay` class, if the object to which `moveTo` is being applied is actually a member of a class that extends `framedDisplay` and overrides `moveBy` then Grace will invoke the version of `moveBy` associated with the subclass. So, if we have executed the statement

```
def welcomeMessage: TextDisplay = new framedText("Click to begin")
```

and we later say

```
welcomeMessage.moveTo (center)
```

Grace will execute the version of `moveTo` found in `framedDisplay` (since we have not overridden this method in `framedText`). When it executes the invocation of `moveBy` within `moveTo` it will select the specialized version of `moveBy` defined in the `framedText` class. The version of `moveBy` defined in `framedText` first invokes `fdMoveBy`, which is the version of `moveBy` defined in `framedDisplay`. This moves the border and background. Then, `framedText`'s version of `moveBy` calls `positionContents` which moves the text itself.

Accordingly, adding a new `moveTo` method to `framedDisplay` does not require further modification to any of its subclasses. All it depends on is that `moveBy` works appropriately for all these subclasses.

This is exactly the same reason why we will never need to override the method `!=",` It is defined to be the opposite of `==,` so when `==` is updated, the evaluation of `!=",` will use the new value of `==.` This is a very important feature of object-oriented languages. Dynamic method invocation means that updating a method means that all other methods depending on it are implicitly updated.

16.5.5 Planned obsolescence

In certain situations, it is useful to define a method in a superclass with the expectation that it will be overridden by every class that extends the superclass. Such an arrangement can make it possible to define extensions without needing to override many methods of the superclass.

Consider the difference between the process used in the previous section to define the `moveBy` and `moveTo` methods for our framed display classes. The addition of a `moveBy` method to `framedDisplay` adds new functionality to the class but it also increases the work required to define an extension of the class. Almost every class that extends `framedDisplay` will have to override `moveBy` or it will not work correctly. This is not an uncommon situation. Suppose we added a `width:=` or `height:=` method to `framedDisplay`. The `framedText` class would have to override these methods to re-center the text. The `framedBarMeter` class would also have to override each of these methods to resize the bar it displays.

Adding the `moveTo` method to `framedDisplay`, however, did not make it any more complicated to extend the class. It was not necessary to override `moveTo` in either of the subclasses we have written. Adding `moveTo` is simpler because its implementation relied on `moveBy`, a method we assumed that all subclasses of `framedDisplay` would override appropriately.

Suppose that we assumed that all subclasses of `framedDisplay` would define a method similar to the `positionContents` method defined in `framedText`. If this method were also defined in `framedDisplay` itself, then we could invoke it as the last step within the definitions of `moveBy`, `width:=` and `height:=` defined within `framedDisplay`. If each subclass defined a version of `positionContents` that did whatever was appropriate to reposition or resize its contents to match the size and position of the frame, it would no longer be necessary for subclasses to override `moveBy`, `width:=` or `height:=`.

The idea of `positionContents` will be to position all pieces of the object relative to the border of the display. Thus the definition of `positionContents` included within `framedDisplay` will look like:

```
method positionContents -> Done is confidential {
  body.moveTo (border.location + borderOffset)
  body.size := (border.width - 2 * borderSize) @ (border.height - 2* borderSize)
}
```

The body of this method contains code to adjust the framed rectangle corresponding to the body of the display. It adjusts both its location and its size so that it fits nicely within the border. Thus if we move or change the size of the border, a request to `positionContents` will reset body accordingly.

We will place method requests to `positionContents` in `moveBy`, `width:=` and any other method that changes the location or dimensions of the display area. Thus, the revised version of the `moveBy` method for `framedDisplay` will look like:

```
// move the parts of the framed display by offsets
method moveBy (dx: Number, dy: Number) -> Done {
  border.moveBy (dx, dy)
  positionContents
}
```

A complete version of `framedDisplay` revised to use `positionContents` is presented in Figure 16.10.

A version of `framedText` revised to be compatible with our new version of `framedDisplay` is shown in Figure 16.11. The biggest change is that it is no longer necessary to override `moveBy` in this class. The version defined in the superclass works correctly because it relies on `positionContents`, which we have rewritten so that it first calls the superclass version to adjust body and also has code to adjust the location of `message`.

In recognition of this change, we have also made a slight change in the constructor. It no longer invokes `positionContents`. Instead, it places the text in the correct position explicitly. We make this change because it is, in general, dangerous to invoke any method that might be overridden from within a class's constructor. In this case, `positionContents` might try to move `message` before it has been created.

Although `framedText` is a subclass of `framedDisplay`, the `framedText` class itself can be extended so that it serves as the superclass for some other class. If this is done, `positionContents` might be overridden by the new subclass. The creation of an object from the subclass will result in execution of the initialization code in `framedDisplay` because it inherits from it. If the constructor for `framedDisplay` were invoked in this way, any invocation of `positionContents` in the `framedDisplay` constructor would actually invoke the version of `positionContents` defined within the subclass. This new version of `positionContents` might reference `defs` or instance variables defined in the subclass that were initialized by code appearing after the first line of its constructor. In this case, they would not yet be initialized when `positionContents` was invoked.

```

dialect "objectdraw"

type FramedDisplay = {
  highlight:=(makeRed: Boolean) -> Done
}

// A FramedDisplay is a composite graphic intended to serve
// as a frame in which a program can display various forms
// of information.
class framedDisplayAt (pt: Point) size(dimensions: Point)
  on (canvas: DrawingCanvas) -> FramedDisplay {

  // Colors used for the frame
  def borderColor: Color = colorGen.gray
  def highlightColor: Color = colorGen.red
  def borderSize: Number = 7
  def borderOffset: Point = borderSize @ borderSize

  def border: Graphic2D = filledRectAt (pt) size (dimensions) on (canvas) // The border

  def body: Graphic2D = filledRectAt (pt + borderOffset)
    size (dimensions - (borderSize @ borderSize) * 2) on (canvas) // The background

  border.color := borderColor

  method positionContents -> Done is confidential {
    body.moveTo (border.location + borderOffset)
    body.size := (border.width - 2 * borderSize) @ (border.height - 2 * borderSize)
  }

  // Change the border's color to make it stand out
  method highlight := (makeRed: Boolean) -> Done {
    if (makeRed) then {
      border.color := highlightColor
    } else {
      border.color := borderColor
    }
  }

  method moveBy (dx: Number, dy: Number) -> Done {
    border.moveBy (dx, dy)
    positionContents
  }

  method moveTo (newPt: Point) -> Done {
    moveBy(newPt.x - border.x, newPt.y - border.y)
  }
}

```

Figure 16.10: Version of framedDisplay based on positionContents.

```

dialect "objectdraw"
import "framedDisplay2" as fd
// A FramedText object displays a specified text message on a
// background framed by a distinct border.

type TextDisplay = fd.FramedDisplay & type {
  textSize:= (newFontSize: Number) -> Done
}

class framedTextAt (pt: Point) size (dimensions: Point) with (contents: String)
  on (canvas: DrawingCanvas) -> TextDisplay {
  inherit fd.framedDisplayAt (pt) size (dimensions) on (canvas)
  alias fdPositionContents = positionContents

  var frameCenter: Point := pt + dimensions / 2

  def textColor: Color = colorGen.white

  def message: Text = textAt (frameCenter) with (contents) on (canvas)
  message.color := textColor
  message.fontSize := dimensions.y / 2
  message.moveBy (-message.width / 2, message.fontSize / 2)

  method positionContents -> Done is confidential {
    fdPositionContents
    frameCenter := border.location + dimensions / 2
    message.moveTo (frameCenter)
    message.moveBy (-message.width / 2, message.fontSize / 2)
  }

  method textSize:= (newFontSize: Number) -> Done {
    message.fontSize := newFontSize
    positionContents
  }

  method highlight := (makeHighlight: Boolean) -> Done {
    if (makeHighlight) then {
      message.color := highlightColor
    } else {
      message.color := textColor
    }
  }
}

```

Figure 16.11: framedText with revised positionContents.

To complete the revision, we also include a version of `framedBarMeter` that is compatible with the version of `framedDisplay` shown in Figure 16.10. It can be found in Figure 16.12. In the original version of this class, there was no instance variable to keep track of the fraction currently displayed by the bar. We need to add such a variable in this version because its value is needed within `positionContents`.

Exercise 16.5.4 *In this section, we suggested that revising our collection of framed display classes to depend upon the existence of the `positionContents` method would simplify the definition of methods such as `width:=` and `height:=`. We did not, however, actually discuss how such methods would be implemented. For this exercise, we would like you to implement one of these methods, `width:=`. To help you appreciate the significance of the changes we made to the design of our classes in this section, we ask that you consider how to implement `width:=` in both the versions of the display classes presented in this section and the earlier versions that did not assume `positionContents` was defined by all classes in the collection.*

- a. *Show how to modify the definitions of `framedDisplay`, `framedText`, and `framedBarMeter` found in Figures 16.3, 16.4, and 16.5 to include a `width:=` method. When invoked, this method should both change the width of the frame and reposition its contents as necessary. In particular, these versions of the classes will not use a `positionContents` method.*
- b. *Next, show how to modify the definitions of `framedDisplay`, `framedText`, and `framedBarMeter` found in Figures 16.10, 16.11, and 16.12 to include a `width:=` method.*

16.6 Summary

Inheritance is an important tool for advanced programming in Grace. Its use can save work by facilitating the sharing of common code. More importantly, it can provide the means to explicitly identify important relationships among related classes in a program. Structuring code in a way that makes it easy to grasp the relationships among the parts is critical to producing programs that are understandable and therefore maintainable.

In Grace, a new class can be defined to inherit features from an existing class by including an **inherit** clause in the header of the new class. In this case, the new class is treated as if it contained declarations of all the methods and instance variables of the class it extends.

We say that a class defined using **inherit** is a subclass of the class it extends and that the class it extends is a superclass of the new class. The subclass/superclass relationship is transitive. That is, if class `c` extends class `b` and class `b` extends class `a`, `c` is a subclass of both `b` and `a`. Similarly, `a` is a superclass of both `B` and `C` in this situation.

Because a class that extends another inherits *all* the features of the superclass, Grace allows us to use an object of a given class as if it were an object of any of its superclasses. Suppose that class `A` is a superclass of class `B`. Grace will allow us to assign a value known to be of type `B` to a variable declared to refer to objects of type `A`. We can also pass a value known to be of type `B` to a method expecting a parameter of type `A` or return a value of type `B` from a method declared to return a value of type `A`.

Grace provides several mechanisms to provide the programmer with the flexibility to allow a class to use the valuable features of its superclass while replacing or modifying features that

```

dialect "objectdraw"
import "framedDisplay2" as fd

type FramedBarMeter = fd.FramedDisplay & type {
  fraction := (newFraction: Number) -> Done
}

// A framedBarMeter displays a variable length rectangle on a
// background framed by a distinct border.
class framedBarMeterAt (pt: Point) size (dimensions: Point) fraction (frac: Number)
  on (canvas: DrawingCanvas) -> FramedBarMeter {
  inherit fd.framedDisplayAt (pt) size (dimensions) on (canvas)
  alias fdPositionContents = positionContents

  // how far inside frame to start bar
  def inset: Number = borderSize + 2
  def barOffset: Point = inset @ inset

  // maximum length of bar
  def maxWidth: Number = dimensions.x - 2 * inset
  var theFraction: Number := 1

  // create the blue bar
  def bar: Graphic2D = filledRectAt (pt + barOffset)
    size ((maxWidth * frac) @ (dimensions.y - 2 * inset)) on (canvas)
  bar.color := colorGen.blue

  // reset bar as a fraction of its maximum length.
  method fraction := (newFraction: Number) -> Done {
    theFraction := newFraction
    positionContents
  }

  method positionContents -> Done {
    fdPositionContents
    bar.moveTo (border.location + barOffset)
    bar.width := (border.width - 2 * inset) * theFraction
  }
}

```

Figure 16.12: Definition of the framedBarMeter class.

are not quite suited to the purpose of the new class. The `protected` access modifier enables a programmer to give subclasses access to instance variables that are hidden from other classes. If the definition of a given method associated with a superclass is not quite appropriate for use with a subclass, the subclass can override the original definition by including its own definition. At the same time, Grace provides the means to access an overridden method within the new class definition. Add an **alias** clause to the **inherit** statement that provides a new name for the inherited method. You can then invoke it as usual to access the original inherited method.

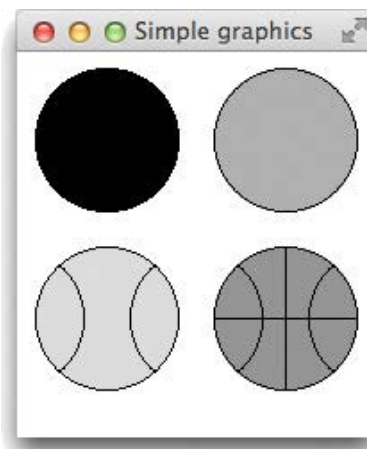
The opportunity to extend a class imposes additional concerns on the programmer when defining a class. Without inheritance, the issues to be addressed when designing a new class focused on the interface the new class provided to other classes that invoked its methods. With inheritance, we also have to consider whether the mechanisms we include in a class will meet the needs of other classes defined by extending it. In some cases, in fact, classes are designed with the assumption that they will only be used as the basis for designing other classes through inheritance.

Inheritance is an important tool to avoid copying working code to multiple places, making it easier to maintain your code. However, you should be careful not to over-use inheritance as it can also make it harder to figure out exactly what your code is doing.

16.7 Chapter Review Problems

Exercises 16.7.1 through 16.7.3 are related to the following scenario.

Suppose that we want to define classes to draw balls used in various games. As suggested by the drawing below, if we start with the ball consisting of a simple `FilledOval`. We can make it look like a rubber ball by painting it red (use your imagination!) and adding a border. Thus, we might define a class named `rubberBall` by creating a class with these three components. Similarly, we can make a `rubberBall` look like a tennis ball by changing the color to a shade of yellow and adding appropriate curved lines. Then, given a tennis ball, if we change its color to orange and add a vertical and horizontal line we get a picture that looks like a basketball.



All of the balls will implement the same interface:

```

type BallType = {
  moveBy (dx: Number, dy: Number) -> Done
  moveTo (newLocn: Point) -> Done
  contains (pt: Point) -> Boolean
  visible:= (bValue: Boolean) -> Done
  color:= (newColor: Color) -> Done
}

```

In the questions below, we ask you to consider the relationships between the three classes: `rubberBall`, `tennisBall`, and `basketBall` corresponding to the three shapes shown in the picture above. Figure 16.13, shows possible implementations of `rubberBall` and `tennisBall`. Exercise 16.7.3 asks you to provide an implementation for `basketBall`.

Exercise 16.7.1 *Draw an inheritance hierarchy diagram for the three classes `rubberBall`, `tennisBall`, and `basketBall` described above.*

Exercise 16.7.2 *In Figure 16.13 we show possible implementations of two of the three ball classes discussed above. Using these definitions and assuming that we have defined the following:*

```
def ball1: BallType = tennisBallAt (110 @ 10) diameter (80) on (canvas)
```

```
def ball2: BallType = basketBallAt (10 @ 110) diameter (80) on (canvas)
```

Indicate what each of the following invocations would do. It is possible that the invocations would be identified as errors either before the program was run or during execution. For each example you should either indicate how the invocation would affect the drawing shown above or explain what type of error would be detected.

a. `ball2.moveBy (50, 50)`

b. `ball1.color := colorGen.blue`

c. `ball2.moveTo (0 @ 0)`

d. `ball2.visible := false`

Exercise 16.7.3 *Provide a definition for a class `basketBall` that could be used to draw basketballs. Your class should extend the `tennisBall` class shown in Figure 16.13.*

Exercises 16.7.4 through 16.7.5 are related to the following scenario.

We can define classes to draw on a canvas objects resembling those ubiquitous sticky notes. The simplest sticky class, `SimpleSticky`, creates objects consisting of a yellow sticky outlined in black. A slightly more refined sticky class, `MessageSticky`, creates objects that also contain a line of text in the sticky. Finally the `ErasableMessageSticky` class generates objects that contain a message, but also contains a small “go away” box in the upper left corner that can be clicked on to make the Sticky disappear.

For simplicity we will only provide each class with only two methods, `contains`, and `react`. The first is self-explanatory. The `react` method will take a `Location` as a parameter,

```

class rubberBallAt (pt: Point) diameter (d: Number) on (aCanvas: DrawingCanvas) -> BallType {
  def ball: Graphic2D = filledOvalAt (pt) size (d @ d) on (aCanvas)
  ball.color := colorGen.red
  def border: Graphic2D = framedOvalAt (pt) size (d @ d) on (aCanvas)

  method moveBy (dx: Number, dy: Number) -> Done {
    ball.moveBy (dx, dy)
    border.moveBy (dx, dy)
  }

  method moveTo (newLocation: Point) -> Done {
    moveBy (newLocation.x - ball.x, newLocation.y - ball.y)
  }

  method contains (locn: Point) -> Boolean {
    ball.contains (locn)
  }

  method visible:=(bValue: Boolean) -> Done {
    ball.visible := bValue
    border.visible := bValue
  }

  method color:=(newColor: Color) -> Done {
    ball.color := newColor
  }
}

class tennisBallAt (pt: Point) diameter (d: Number) on (aCanvas: DrawingCanvas) -> BallType {
  inherit rubberBallAt (pt) diameter (d) on (aCanvas)
  alias rbMoveBy(_, _) = moveBy(_, _)
  alias rbVisible:=(_) = visible:=(_)
  def cutSize: Number = 100
  def rightCutStart: Number = 90 + (180 - cutSize) / 2
  def leftCutStart: Number = 270 + (180 - cutSize) / 2
  def leftCut: Graphic2D = framedArcAt (pt + ((d * 2 / 3) @ 0)) size (d @ d)
    from (rightCutStart) to (rightCutStart + cutSize) on (aCanvas)
  def rightCut: Graphic2D = framedArcAt (pt - ((d * 2 / 3) @ 0)) size (d @ d)
    from (leftCutStart) to (leftCutStart + cutSize) on (aCanvas)
  ball.color := colorGen.yellow

  method moveBy (dx: Number, dy: Number) -> Done {
    rbMoveBy (dx, dy)
    leftCut.moveBy (dx, dy)
    rightCut.moveBy (dx, dy)
  }

  method visible:= (bValue: Boolean) -> Done {
    rbVisible:=(bValue)
    leftCut.visible:= bValue
    rightCut.visible:= bValue
  }
}

```


will draw a small red dot at the `Location`, and will change the outline of the sticky to red. If it has a message, it will also change the message to red. However, if the location is in the “go away” box of the `ErasableMessageSticky` then the entire sticky will be hidden (though the red dots will remain on the canvas).

For the following problems, assume that we plan to define these three classes using inheritance.

Exercise 16.7.4 Draw an inheritance hierarchy diagram for the three classes `SimpleSticky`, `MessageSticky`, and `ErasableMessageSticky` using inheritance as suggested in the description above.

Exercise 16.7.5 In Figure 16.14 we show possible implementations of two of the three sticky classes discussed above. Using these definitions, provide a definition for a class `ErasableMessageSticky` that could be used to draw stickies with a message and “go away” box. Your class should extend the `MessageSticky` class shown in Figure 16.14.

Exercise 16.7.6 What would be output by a program composed of the following code?

```

type MailItem = {
  open -> Done
  examine -> Done
  file -> Done
  process -> Done
}

type EnvelopedItem = MailItem & type {
  fileContents -> Done
}

class mailItemFrom (descriptor: String) -> MailItem {
  method description -> String is confidential {
    descriptor
  }

  method open -> Done {}

  method examine -> Done {
    print "Read {description}"
  }

  method file -> Done {
    print "Recycled {description}"
  }

  method process -> Done {
    open
    examine
    file
  }
}

class catalogFrom (descriptor: String) -> MailItem {

```

```

dialect "objectdraw"

type Sticky = {
  contains (pt: Point) -> Boolean
  react (pt: Point) -> Done
}

class stickyAt (locn: Point) size (dimensions: Point) on (canvas: DrawingCanvas) -> Sticky {
  def interior: Graphic2D = filledRectAt (locn) size (dimensions) on (canvas)
  interior.color := colorGen.yellow
  def outline: Graphic2D = framedRectAt (locn) size (dimensions) on (canvas)

  method contains (pt: Point) -> Boolean {
    outline.contains (pt)
  }

  method react (pt: Point) -> Done {
    def dot: Graphic2D = filledRectAt (pt) size (2 @ 2) on (canvas)
    dot.color := colorGen.red
  }
}

class messageStickyAt (locn: Point) size (dimensions: Point) with (mess: String)
  on (canvas: DrawingCanvas) -> Sticky {
  inherit stickyAt (locn) size (dimensions) on (canvas)
  alias stickyReact(_) = react(_)
  def gap: Number = 2
  def message: Text = textAt ((locn.x + dimensions.x / 2 - 20) @ (locn.y + dimensions.y / 2 - 10))
    with (mess) on (canvas)

  method react (pt: Point) -> Done {
    stickyReact(pt)
    message.color := colorGen.red
  }
}

```

Figure 16.14: The SimpleSticky and MessageSticky classes for Exercise 16.7.5

```

inherit mailltemFrom (descriptor)

method description -> String is confidential {
  "catalog from {descriptor}"
}

method examine -> Done {
  print "Order goodies from {description}"
}
}

class magazineFrom (descriptor: String) -> Mailltem {
  inherit mailltemFrom (descriptor)

  method description -> String is confidential {
    "{descriptor} magazine"
  }

  method file -> Done {
    print "Place {description} on coffee table"
  }
}

class envelopedFrom (descriptor: String) -> EnvelopedItem {
  inherit mailltemFrom (descriptor)

  method open -> Done {
    print "Rip open the envelope"
  }

  method file -> Done {
    fileContents
    print "Throw out the envelope"
  }

  method fileContents -> Done {
  }
}

class billFrom (descriptor: String) -> EnvelopedItem {
  inherit envelopedFrom (descriptor)

  method description -> String is confidential {
    "bill from {descriptor}"
  }

  method examine -> Done {
    print "Pay {description}"
  }

  method fileContents -> Done {

```

```

    print "Save {description} in filing cabinet"
  }
}

class birthdayCardFrom (descriptor: String) -> EnvelopedItem {
  inherit envelopedFrom (descriptor)

  method description -> String is confidential {
    "birthday card from {descriptor}"
  }

  method fileContents -> Done {
    print "Stick {description} on the fridge"
  }
}

def delivery: List[[MailItem]] = list[[MailItem]] [catalogFrom ("L. L. Bean"), birthdayCardFrom ("from Mom"),
  magazineFrom ("Time"), billFrom ("Phone company")]

for (delivery) do {item: MailItem ->
  item.process
}

```

16.8 Programming Problems

Exercise 16.8.1 *In most of the examples in this text the “main” program object extends `graphicApplication`. The class `graphicApplication` is not a standard Grace class. It is part of our `objectdraw` library. `graphicApplication` is defined as a subclass of another `objectdraw` class named `application`. The main difference between `graphicApplication` and `application` is that `graphicApplication` is designed to make it easy for students to handle mouse events and place objects on a canvas in their very first days as Grace programmers.*

You may have noticed that to handle events associated with buttons, scrollbars, other GUI components and event keystrokes you have to approach the handler in a different way. You have to use a method from the button with a name like `onMousePressDo` and associate an action with the handler. Thus if we write

```

myButton.onMousePressDo {mouseEvt: MouseEvent ->
  print "pressed mouse button down"
}

```

then every time the mouse button down, the action specified by the anonymous function

```
{mouseEvt: MouseEvent -> print "pressed mouse button down"}
```

will be executed – in this case just printing out the simple message. Within an object that extends `graphicApplication`, however, all you have to do is define a method like `onMousePress`.

That is because the `graphicApplication` method includes the following code:

```

method onMousePress(mouse : Point) -> Done {}
  canvas.onMousePressDo { event!: MouseEvent ->
    onMousePress(event!.at)
  }
}

```

That is, when a new `graphicApplication` is created, it associates the action
`{event: MouseEvent -> onMousePress(event!.at)}`

with a mouse press on the canvas. Thus when the user presses the mouse button on the canvas, Grace executes the action with the actual event as the parameter. That in turn makes a method request of `onMousePress` with the parameter `event!.at`, which is the location of the mouse press.

Because the definition of `onMousePress` in `graphicApplication` doesn't do anything, nothing will happen in an object inheriting from `graphicApplication` when the user presses the mouse button unless the programmer has overridden `onMousePress` with a new definition. For example if an object inheriting from `graphicApplication` overrides the method with

```
method onMousePress (pt: Point) -> Done {  
  print "Mouse was pressed at {pt}"  
}
```

then the system will print out the location of the mouse whenever the user presses the mouse button.

We would like you to define two classes named `simpleButton` and `buttonAndGraphicApplication` that will make it just as easy to handle events associated with clicking on a button as it is to handle a mouse event in an object inheriting `graphicApplication`.

Chapter 17

Exceptions

It is not at all uncommon for a program to do unexpected things when run. In many cases a program behaves so badly that all the computer can do is display an error message. A sample of the type of error message many Grace development systems produce in such situations is shown below. Does it look familiar?

```
BoundsError at line 622 of collectionsPrelude: index 0 out of bounds 1..12
  called from Exception.raise at line 622 of collectionsPrelude
  called from object.at at line 38 of YearlyCalendar
  called from object.insertEventOn()with at line 39 of calendarGUI
  called from block.apply at line 508 of calendarGUI
  called from execution environment
```

Many such errors are the results of mistakes made by the programmer. One might forget to initialize a variable or might specify an incorrect condition in a loop that allows an index to exceed an array's range. In such situations, all you can do is try to understand the error messages the system produces and examine your code carefully to identify the source of the problem.

Sometimes, however, an error occurs in a “correct” program because a situation arises that the program was not designed to handle. For example, if a program uses a value entered in a `NumberField` to determine which element in an array to access, the program might encounter an indexing error if the user enters an inappropriate value in the `NumberField`.

In some sense, such problems are the fault of the program's user rather than its author. After all, the problem can be avoided by correcting the user's input without changing the program's text. However, if the only explanation presented to the user is an error message like the one shown above, the user is very likely to blame the programmer. At the very least, the user will have more trouble than necessary determining how to correct the input. Accordingly, a well written program should anticipate such user errors and include code to validate all input before using it. If user input is invalid, the program itself should display a message that the user will be able to comprehend.

In this chapter, we will learn about *exceptions*, a feature of the Grace language that supports the construction of programs that are robust in the face of errors in user input and other unexpected events. An exception is an object that a method can use to signal that it is unable to complete its intended function. Code can be written to respond appropriately

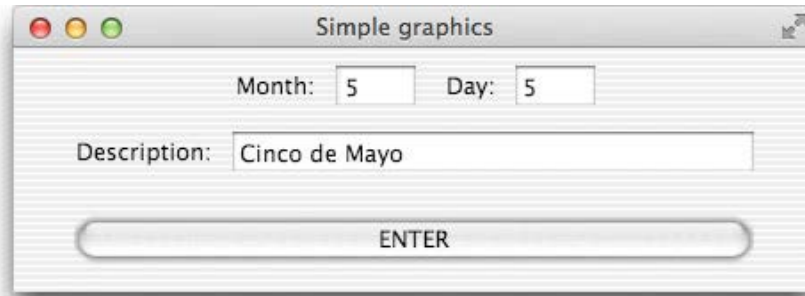


Figure 17.1: Interface used to enter new calendar events

when a method generates an exception rather than performing its expected function. Such code is placed in a new control structure presented in this chapter, the `try-catch` statement.

17.1 Exception Handlers

To provide a context that illustrates the use of Grace’s exception mechanism, we will consider the implementation of a calendar management program. Such a program would include a data structure in which events scheduled for various days could be recorded and would provide an interface through which a user could enter a description of an event to be scheduled on a specific day. In Section 14.1 we showed how a two dimensional array of `String`s could be used to represent such a calendar of events. We will assume that the same type of array is used to represent the calendar in this chapter.

The array we have in mind would be declared as

```
def dailyEvent: List[[List[[String]]]] = emptyList[[List[[String]] ]]
```

with the intent that

```
dailyEvent.at (month).at (day)
```

refers to a `String` describing events scheduled on the month and day specified. `dailyEvent.at (12).at (25)` might refer to the `String` “New Year’s Day” while `dailyEvent.at (12).at (25)` might refer to the `String` “Christmas Day.”

This array would be declared as an instance variable of a `yearlyCalendar` class that maintains all events scheduled for a given year. This class would provide methods to manipulate a calendar, including a method to associate an event with a date and a method to get the current event description for a date. We will assume the first method is declared with the header:

```
method insertEventOn (month: Number, day: Number) with (description: String)  
    -> Done
```

To enable a user to enter new events, the program might pop up a simple window like that shown in Figure 17.1.

Two `NumberFields` and a `TextField` could be used for the user to provide values for the calendar entry. When the “Enter Data” button is pressed, the information could be inserted into the actual calendar data structure.

What happens if the user accidentally enters a word like “October” rather than a number like 10 for the month? Different systems behave in different ways, but in Grace the system returns a 0 if a non-numeric entry is obtained in a `NumberField`. When the program attempts to access a month via 0, it discovers that that is an illegal subscript and raises a `BoundsError` exception just as was shown earlier in this chapter. At this point the program crashes – not a happy circumstance!

As we all know, users (even ourselves sometimes!) do make mistakes. In this case, putting in the name of the month rather than its number seems quite natural. We want to make sure that our programs do not crash when users make plausible errors.

In this case we could just check the input to make sure the value returned from the month and day fields are numbers within appropriate bounds. Certainly it is easy to check to see if the value returned is greater than 0. However, it is much trickier to check whether the upper bound on the date is legal. That would require us to use the method `getDays` of `yearlyCalendar` to return the number of days in a given month. Unfortunately, however this method that was declared to be confidential when `yearlyCalendar` was written. Thus we do not have easy access to it in the program that sets up the GUI interface for adding and displaying items from the calendar.

Exercise 17.1.1 *Assume for the moment that there is a public method `getDays(monthNumber: Number)` in the class `yearlyCalendar` that returns the number of days in the given month. Please write the code that tests to see if the month and day are legal before calling `insertEventOn`. You may assume that the original code to insert the new information into the calendar is `cal.insertEventOn (monthField.number, dayField.number) with (dataField.text)`*

Thus `monthField` and `dayField` are `NumberFields` containing the values of the month and day, while `dataField` is a `TextField` holding the information about the appointment.

Print a helpful error message if the entries are not legal numbers.

Grace makes it possible to perform error checking and recovery using a construct called a `try-catch` statement. As its name suggests, this construct has two main parts. The `try` clause specifies the action the programmer would like Grace to perform. This is then followed by one or more `catch` clauses that specify various actions to perform in the event that an error occurs while executing the code in the `try` clause. In this and the following sections we will show a series of examples illustrating the use of the `try-catch` to provide error handling for our calendar example.

Suppose our program puts up a window with `NumberFields` `monthField` and `dayField`, `TextField` `dataField`, and a `Button`, `entry`. Figure 17.2 shows how the `try-catch` construct can be used to add an entry to the calendar. The block associated with the mouse press of `button entry` begins with the word `try` followed by a sequence of statements surrounded by curly braces. This is the `try` part of the construct. Grace will attempt to execute these statements, but if an error occurs during the process, Grace will examine the `catch` that follows to find instructions on how to handle the error.

The line ending with

```

enter.onMousePressDo { mouseEvt: MouseEvent ->
  try {
    cal.insertEventOn (monthField.number, dayField.number) with (dataField.text)
    cal.printYear
  } catch {e: BoundsError ->
    print (e.message)
    print "Please enter legal numeric values for the month and year"
  }
}

```

Figure 17.2: Exception handling with a try–catch clause

```

catch {e: BoundsError ->

```

is the beginning of the `catch` part of the construct. It is followed by a list of instructions to be executed if a bounds error occurs while attempting to follow the instructions in the `try` part. In this case, we say that the `insertEventOn()` `with()` method raises or throws an exception and that the `catch` phrase handles or catches the exception. If no errors occur while executing the `try` part, then Grace will skip the instructions in the `catch` part.

17.2 Exceptions are Objects

If you look carefully at the first line of the `catch` clause shown in the last section:

```

catch {e: BoundsError ->

```

you will notice that we have not completely explained its syntax. What is the role of the text `e: BoundsError` in this construct?

The name `BoundsError` is used to identify the kind of error that this `catch` will handle. `BoundsError` is the name of a class of objects used in Grace to represent information about errors related to accessing and updating values in lists. Grace includes many other exception classes, such as `NoSuchMethod` and `NoSuchObject`, that are used to represent other types of errors.

When a list method detects an error with an illegal subscript, it constructs a `BoundsError` object. This object contains information describing the error that can be useful within a `catch` clause. To make it possible to access this object within the code of the `catch` clause, Grace requires the programmer to declare a name that will refer to this object, just as with a `catch` associated with a `match` statement. As long as the exception raised is of kind `BoundsError`, the `catch` clause will associate the exception object with the identifier `e` and execute the body of the `catch` clause using that value of `e`.

Objects of the `BoundsError` class and of Grace’s other exception kinds include a method named `message` that returns a `String` that provides information about the cause of the exception. The `String` returned by `message`, when invoked on an exception object raised by `BoundsError` consists of the phrase “index *n* out of bounds *low..high*” where *n* is the number being used as a subscript, and *low* and *high* are the lowest and highest values that can be legally used. Thus in the example earlier in the chapter, the value 0 was used as the subscript but only

```

type Exception = type {
  // the exceptionKind of this exception.
  exception -> exceptionKind

  // the message that was provided when this exaction was raised.
  message -> String

  // the data object that was associated with this exception
  // when it was raised, if there was one. Otherwise, the string "no data".
  data -> Object

  // the source code line number of the raise request that created this exception.
  lineNumber -> Number

  // the name of the module containing the raise request that created this exception.
  moduleName -> String

  // a description of the call stack at the time that this exception was raised.
  // backtrace.first is the initial execution environment; backtrace.last is the
  // context that raised the exception.
  backtrace -> List[[String]]
}

```

Figure 17.3: The type of exceptions

values in the range from 1 to 12 could be used.

The other operations supported by exceptions are given in Figure 17.3

17.3 Exception Propagation

The try-catch clause in Figure 17.2 catches an exception that was actually raised in the execution of the method `insertEventOn` in the class `yearlyCalendar`. The problem was the evaluation of `dailyEvent.at(0)` where `dailyEvent` is a list of list of `String`. To make sense of this, you will need to understand how exceptions are propagated.

When an exception occurs, Grace looks for a try-catch to handle the exception in each method along the sequence of methods that were invoked to lead to the point where the error was encountered. If there is no appropriate try-catch in the method where the error is detected, Grace looks for an appropriate try-catch in the method that invoked the one where the error occurred. If no try-catch is found in the method's invoker it looks at the invoker's invoker and so on until a try-catch is found or the sequence of invocations is exhausted. If no try-catch can be found, then Grace displays one of the complex error messages of which we have shown examples in this chapter. We say that an exception propagates up the chain of invocations until it finds a handler or the end of the chain of invocations.

Exercise 17.3.1 *Figure 17.4 shows the code for three methods containing a number of try/catch statements. The methods are not intended to perform any useful function. They are merely designed to provide some examples of exception propagation.*

```

def names: List[[String]] = emptyList[[String]]
names. add ("Alessandro")
names. add ("Beau")
names. add ("Za'eem")
var s: String

method justTesting (part1: String, part2: String) -> Done {
  print "starting with {part1} and {part2}"
  try {
    firstStep(part1.asNumber, part2)
    if(part1.asNumber < 3) then {
      print (names.at(4))
    }
  } catch {e: ProgrammingError ->
    print "Error caught by 1st catch: {e.message}"
  }
}

method firstStep (part1: Number, part2: String) {
  try {
    def partTwo: Number = part2.asNumber
    if(partTwo == 2) then{
      print (s)
    }
    try {
      secondStep (part1, partTwo)
    } catch {e: BoundsError ->
      print "error caught by 2nd catch: {e.message}"
    }
  } catch {e: ProgrammingError ->
    print "error caught by 3rd catch: {e.message}"
  }
}

method secondStep (part1: Number, part2: Number) {
  print (names.at (part1+part2))
}

```

Figure 17.4: Methods for Exercise 17.3.1

To answer this question you will need to know that accessing an uninitialized variable results in an exception of kind `ProgrammingError`.

What would be the result of each of the following method invocations?

- a. `justTesting ("1", "2")`
- b. `justTesting ("Four", "3")`
- c. `justTesting ("7", "7")`

17.4 Raising Exceptions

The code shown in Figure 17.2 exhibits a troubling design flaw. Classes should hide the details of the implementation of the objects they represent. Following this principle, the code that uses a `Calendar` object should not need to know whether the internal implementation of the `Calendar` object depends on a list or some other data structure. Unfortunately, the fact that the code in Figure 17.2 handles `BoundsError` exceptions generated by `setEvent` makes this code dependent on the assumption that the `Calendar` implementation is based on an array.

To ensure that the `Calendar` class encapsulates the details of its implementation, we should redesign the `insertEventOn` method so that it does not produce an `BoundsError` exception when it is invoked with invalid parameter values. We can't just ignore such invalid parameters, but we can change the method so that it signals that an error has been detected by raising a different type of exception.

We will define a new exception class and use objects of this class to represent errors detected by `insertEventOn`. We will create an exception kind named `DateRangeException` which will indicate that a method has been passed an illegal or inappropriate date component.

We can define a new kind of exception from an existing type using the `refine` method.

```
def DateRangeException = EnvironmentException.refine("Date range exception")
```

This defines `DateRangeException` to be an exception that “refines” or specializes `EnvironmentException`. In particular, this means that any `catch` clause that handles an `EnvironmentException` will also `catch` and handle a `DateRangeException`. If the exception is printed, it will always begin with the argument to `refine`.

Having defined this new kind of exception, we can modify the method `insertEventOn()` with() to raise the `DateRangeException` rather than `BoundsError`:

```
method insertEventOn (month: Number, day: Number) with (description: String)
  -> Done {
  try {
    dailyEvent.at (month).at (day) put (description)
  } catch {e: BoundsError ->
    DateRangeException.raise("The date entered was month: {month}, day: {day}\n" ++
      "but the month should be between 1 and 12 and the day should be legal")
  }
}
```

As you can see, the new method catches the `BoundsError` and then instead raises `DateRangeException`. The argument to the `raise` method includes whatever string you wish to have printed if the exception is not caught.

For example, if the user invokes `insertEventOn (12, 35)` with ("Christmas"), and the exception is not caught the system will print something like

```
Date range exception at line ...: The date entered was month: 12, day: 35
but the month should be between 1 and 12 and the day should be legal
```

Of course, a program can raise an existing exception at any point in your code. For example, your program could test an input value to see if it is in a given range, and if not write

```
BoundsError.raise("the value ...")
```

Executing this will result in the run-time looking for a try-catch that will catch a `BoundsError`, and if none is in the current method, search along the call path to find the most recent call that has an appropriate catch clause. If none are encountered then the program will halt and print out an error message with the values associated with the exception that has been raised as well as print out a trace of where the program was in its execution when the exception was raised.

17.5 Summary

In this chapter, we have examined Grace's mechanisms for handling exceptional conditions that occur during program execution. When a method detects an error condition that it wishes to report using the exception mechanism, it raises an exception that can be caught by the program. If it is not caught then the program will terminate, printing an error message whose value depends on the exception raised as well as a trace of where the program was in its execution when the exception was raised.

It is good practice to wrap all code that might raise an exception with a try-catch statement. This construct is used to describe how to respond if exceptions are raised while executing a sequence of statements. The try-catch construct has two components. The first, the try part, is just the set of instructions whose exceptional conditions are to be handled by the construct. When asked to execute a try-catch statement, the computer begins by executing the statements in the try part. If these statements execute without raising any exceptions, then the execution of the try-catch is considered complete.

The second component of a try-catch is a list of clauses each identifying an exception kind and a sequence of statements that should be executed in the event that one of the statements in the try part raises an exception that belongs to the identified exception kind. When an exception is raised, the computer abandons the execution of the statements in the try part and examines the list of catch clauses sequentially searching for the first clause that specifies how to handle the kind of exception raised. If such a catch clause is found, its statements are executed. If an appropriate catch clause cannot be found in the current method, the exception propagates backward along the chain of method invocations searching for an appropriate catch clause.

A new exception kind may be declared in Grace to be a refinement of an existing exception kind. A catch clause whose parameter is associated with a particular exception kind will also handle any other exception kind that refines it.

In many cases, exceptional conditions are the result of user error. Robust programs should be designed to respond to user errors in appropriate ways that assist users to recognize and correct their mistakes. Grace's exception mechanisms are a valuable tool for the construction of such programs. In addition, Grace's exception mechanisms are essential to handling unexpected events that occur when a program depends on external sources such as files or network connections for input processing as we will see in the next chapter.

Programmers can explicitly raise exceptions by making a `raise` method request on an exception kind. The `String` parameter to such a request constitutes the message associated with the exception. It can be accessed by making a `message` method request to the exception.

17.6 Chapter Review Problems

Exercise 17.6.1 *Suppose we wanted to write code to keep track of appointments for a single day. We might define an `appointment` class generating objects of type `Appointment` to represent information about a single item on the schedule. This class would provide methods like `description`, `location`, `asString`, etc. to access various properties of an appointment.*

We could also define an object `freeTime` with type `Appointment` that represents a slot without an appointment. The methods `description` and `location` would be defined to raise a new exception `NoAppointment` when they are requested on `freeTime`. `NoAppointment` could be a refinement of `ProgrammingError`. We could then define and initialize a list

```
schedule: List[[Appointment]] = emptyList[[Appointment]]  
for (1 .. 24) do {list.add(freeTime)}
```

with one element of the list associated with each hour of the day. Thus the list would be initialized with free time at each hour.

- a. *Define the type `Appointment`, the object `freeTime`, and a class `appointment` that can generate new appointments given a location and description (but no time).*
- b. *Define a method `printSchedule` to print out items on the schedule in a readable fashion. Include only the non-`freeTime` items.*
- c. *Define a method `addApptAt` that takes a time, location and description, each of type `String`. If the time parameter fails to be a number, it should raise a `ProgrammingError` exception with a helpful error message. If it represents a number, but it outside of the range from 1 to 24, it should raise a `BoundsError` exception. Otherwise it should create an appropriate appointment and insert it into the schedule in place of whatever used to be there.*

Exercise 17.6.2 *We have seen that a collection of objects can be either associated with the elements of a built-in list or organized into a recursive list. When objects are placed in a list, we can access selected element by providing the index. It is possible to include a method to provide similar access to the elements of a recursive list.*

Recall the definition of the type `URLList` to represent recursive lists of URLs presented in Section 11.1.2. Suppose that we add an entry of the form:

```
// extract a URL from the list by specifying its position  
method selectByIndex (index: Number) -> String
```

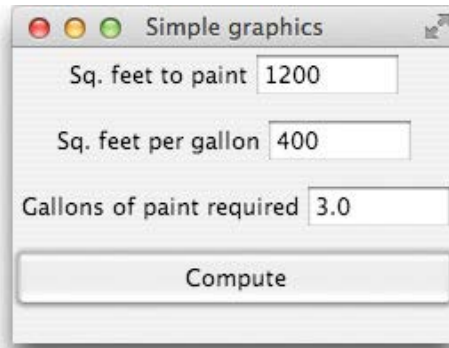


Figure 17.5: Interface for program described in Exercise 17.7.1

to the `URLList` definition with the intent that `selectByIndex(1)` would return the first URL in the list, `selectByIndex(2)` would return the second URL and so forth. Furthermore, we would like the method to throw an `BoundsError` exception if the value provided for `index` is negative or too large. `BoundsError` is a pre-defined Grace exception type.

We would have to add definitions of `selectByIndex` to both `emptyURLList` and the `nonEmptyURLList` classes.

- a. Please provide definitions for these methods compatible with the definitions of these classes found in Section 11.1.2. It will be sufficient if the `String` returned by applying `message` to the exceptions raised simply returns `"Invalid index"`. (As a challenge, consider how to include the `index` provided to the initial invocation in the message associated with the exception.)
- b. It becomes much easier to provide a clearer error message if a check was made that the `index` is valid before requesting `selectByIndex`. However, to do that requires that you know the length of the list. Add a method `size` to both `emptyURLList` and the `nonEmptyURLList` class.

17.7 Programming Problems

Exercise 17.7.1 Write a program that can be used to calculate information about paint coverage. The program should display a window with a “Calculate” button and three text fields for the number of gallons of paint available or used, the number of square feet covered by a gallon of paint and the number of square feet painted. A sample of this interface is shown in Figure 17.5. The user is expected to fill in any two of the three fields. Then, when the calculate button is pressed the program should place the correct value in whichever field was left empty. Pressing calculate should have no effect unless exactly two of the three fields contains numeric values.

The built-in class `numberField` will not allow the user to type in a non-numeric value, which keeps the user out of trouble. However, there is an alternative implementation that will allow

the user to put in a non-numeric value, but will raise an exception if the program requests the numeric value using the method `number`. Thus this field will take in any text, but will blow up if the program requests a number and the value doesn't correspond to a numeric value.

We would like you to build this implementation using inheritance and exception handling and then to write the paint program with these number fields. Thus, rather than using the class `numberField` directly to build the user interface, define a new class `maybeNumberField` that will implement both of the types `TextField` and `NumberField`. It will have the following structure:

```
type MaybeNumberField = TextField & NumberField
class maybeNumberFieldLabeled(theLabel: String) -> MaybeNumberField {
  inherit textFieldLabeled(theLabel)
  ...
}
```

As you can see, it will be defined to inherit from `textField`. However, we also want it to have the behavior of a `NumberField`. Thus it must implement the methods `number` and `number:=()`. As we remarked above, the built-in `numberField` class does not allow the user to fill in a non-numeric value. However, in `maybeNumberField` your program should raise a `BadNumberError` exception if the user asks for the numeric value, but the contents of the field don't correspond to a number.

With this class you should be able to write the program described earlier in this exercise. Note that the results of `"".asNumber` is 0. Thus you can detect an empty field by either using method `text` and comparing with `""` or using `number` and comparing with 0.

Exercise 17.7.2 Define a new classes named `myDate` and `DateFormatError`. Define the `DateFormatError` exception as a refinement of the `EnvironmentException` exception kind. It will be similar to the `DateRangeError` defined in the text. The `myDate` class should be designed to represent a date. It should provide accessor methods named `month`, `day`, and `year` which each return an `Number` value. It should also implement an `asString` method that will return the date's representation in the "mm/dd/yy" format.

The `myDate` class should take a string as a parameter and interpret this string as a date in the format "mm/dd/yy". If the constructor finds that its parameter value is not in the correct format, it should raise a `DateFormatError` exception including a message indicating that an improperly formatted date was encountered and including the contents of the parameter string. The constructor should be forgiving about the use of blank spaces and leading zeroes. That is, both

```
7/1/04
and
7 / 1 / 4
```

should be considered valid representations for July 1, 2004.

Write a simple program to test your class. It should provide one `textField` in which a user can enter a date, a button to press to request that the date entered be used to create a `myDate`, and another `textBox` in which the program will either display the `asString` output of the object created or the message associated with the exception raised by the constructor.

Chapter 18

Streams

If you examine the menu bars provided by a variety of computer applications, you will find that they exhibit a bit of diversity. A word processor is likely to have a “Format” menu while a web browser provides a “History” or “Favorite” menu and a spreadsheet may offer a “Tools” menu. They all, however, are likely to have a “File” menu and they are all likely to give it a place of honor before most of the other menus in the menu bar. What makes the “File” menu so common and so important?

Files are a computer’s long term memory. You have seen that a program can make a computer “remember” things by associating pieces of information with variable names. Once a program stops executing, however, all such information is forgotten. People who use computers want to be able to work on a project one day and have the computer recall the work that was completed, even if they turn the computer off and don’t continue working on the project for several days. This requires the ability to remember things even when a program is stopped and later restarted. Files provide such memory.

Most programs include a “Save” or “Save as” item in their File menus to enable users to save a copy of their current work as a document in a file. There is also usually an “Open” or “Import” item in the File menu that can be used to bring information that was previously saved back into a program for further work. Of course, to make such menu items work as expected, someone has to write program statements that will instruct the computer to copy information from program variables to a file or vice versa when one of these items is selected. In this chapter, we will study the mechanisms Grace provides to enable a programmer to write such code.

Grace provides relatively limited facilities for reading from and writing to streams. To a large extent this has to do with the limitations of using a web browser as our programming environment. As you might imagine, you might be rather unhappy if you went to a web page and then a program on that web page gained access to all of the files in your system, doing something nasty like copying all of your mail messages, erasing all of your files, or writing a virus program on your computer so that it could not only infect your files, but also those of anyone you communicate with.

Thus you won’t be surprised to find that web browsers limit access to the users file system, aside from when the user explicitly requests that access be allowed. This latter exception is why you can upload a photograph or other file to a web site like facebook or instagram.

Programs running in web browsers have similar limitations. A good example is Java applets, which can be embedded in web pages, which have very limited access to user files.¹

The net result of this is that there is a relatively small amount to learn about Grace's file-handling facilities. In the next few sections, we will show you the basics of how they are used.

The files stored on your computer contain data encoded in many different formats. You may have pictures stored in JPEG files, sounds encoded as MP3 and MP4 files, Quicktime movies, MPEG movies, sound files and files of many other varieties. Among these there are also many files that contain nothing more exotic than simple text. We will focus our examination of Grace's file mechanism on learning how to work with text files.

18.1 Bookmark Files: An Example

As discussed in the introduction, most programs can be explicitly instructed to access files by selecting an item in the program's File menu. Many programs also use files in ways users are not explicitly aware of. If you play a game program that tells you the ten highest scores as you start or finish each game, that program is using a file to remember these scores. When you top one of the existing high scores, it saves new information in the file even though you might not explicitly select a "Save" item from the File menu. When you start the program, it extracts all the previous high scores from this file even though you never select "Open."

Another, more practical example of a file that is accessed implicitly is the file used to keep track of bookmarks created by the user of a web browser. Like a high score list, you expect your web browser to remember your bookmarks even if you quit and later restart your browser. To do this, the program must place a copy of your bookmarks in a file rather than merely keeping track of the bookmarks with program variables. If you search your disk hard enough, you will find this file hiding somewhere. It is probably named favorites.htm or bookmarks.html. Whenever you add a bookmark, your browser sends data describing the new bookmark to this file. Whenever you restart your browser, it must examine the contents of the file to build the bookmark list or menu you expect to see.

As our first example of the construction of Grace code that uses files, we will consider the manipulation of a bookmarks file. For each bookmark you create, your browser stores a pair of strings: a web address or URL and a few words that describe the site at the address.² Accordingly, storing bookmarks will require nothing more than a simple text file. Of course, we don't want to implement a complete web browser. Instead, we will focus on the design of a class to hold a list of bookmarks including methods to retrieve old bookmarks from a file and a method to place the complete list in a file.

First we will need a class to keep track of a single bookmark. A simple version of such a class is shown in Figure 18.1.

The class `bookmark` takes two strings as parameters and returns a `Bookmark` object. The first parameter should be a short description of a web site that could be used as an entry in a

¹By way of contrast, Java applications, which run outside of a browser, have much greater access to files.

²Actually it stores a whole lot more information as well, but we'll ignore that for now. Feel free to look up how to access the bookmarks file for your browser and operating system combination so you can see what the file actually looks like.

```

type Bookmark = {
  description → String
  address → String
}

class bookmark(theDescription: String, theAddress: String) {
  method description → String {theDescription}
  method address → String {theAddress}
  method asString → String {description++" ---> "++address}
}

```

Figure 18.1: A class of objects designed to represent single bookmarks

```

def gbm: Bookmark = bookmark("Google","www.google.com")
def abm: Bookmark = bookmark("Amazon", "www.amazon.com")
def pbm: Bookmark = bookmark("Pomona College","www.pomona.edu")
def bookmarks: List[[Bookmark]] = list[gbm,abm,pbm]

method printBookmarks(marks: List[[Bookmark]]) {
  for (marks) do {bm: Bookmark →
    print(bm)
  }
}

printBookmarks(bookmarks)

```

Figure 18.2: Creating and printing a list of bookmarks

bookmarks menu. The other should be the URL (Universal Resource Locator) or address for the site. The class provides two methods to access these strings — `description` and `address`. It also includes a `asString` method that returns a single string showing the complete contents of the bookmark by concatenating the short description with the address while sticking an arrow between them. For example, `asString` might return

```
Google home page ---> http://www.google.com
```

if applied to a `Bookmark` object representing a bookmark for the Google web site. Notice that the object is immutable. That is, there is no way to change the contents of a bookmark once it has been created.

As you might guess, we will need to keep track of an entire list of bookmarks. This can be done easily by creating a list of bookmarks as in Figure 18.2.

You will notice that we added a method `printBookmarks` at the end of the figure that prints out all of the `bookmarks`. There is nothing particularly new or interesting about this. However, it will serve as a model for how we write a list of bookmarks to a file.

18.2 Opening a file for writing

The module `io` provides access to the Grace facilities for manipulating files. You will need to import it for any program that reads or writes files. Let's look at some simple code for a

```

import "io" as inout

method saveBookmarksFile(marks: List[[Bookmark]]) {
  try {
    def myFile: inout.FileStream = inout.open ("bookmarks.txt", "w")
    for (marks) do {bm: Bookmark →
      myFile.write(bm.asString++"\n")
    }
    myFile.close
  } catch {ex:EnvironmentException →
    print "can't access bookmarks file"}
}

saveBookmarksFile(bookmarks)

```

Figure 18.3: Opening a file and writing

method printing the contents of our list of `Bookmarks` to a file.

At its core, the method `saveBookmarksFile` is similar to the earlier method `printBookmarks`, with a fairly simple `for` loop at its core, but there are a few complications due to the fact that we are writing to a file.

First, while Grace does not require it, it is always a good idea to wrap all code accessing a file in a `try()catch()` statement. In case we cannot open the file (e.g., because another application has opened it) or cannot read from the file, we want to be able to at least print out a message to tell the user what happened. As you can see from the code, file errors raise an `EnvironmentException`, so that is what must be caught and handled. In this particular case it simply prints out a message that it can't access the bookmarks file.

In order to read from a file, we must first open it. The `open` method obtained from the `io` module allows the programmer to create or access a file in the system. We open a file by providing the name of the file (as a `String`) and the *mode* in which the file will be opened. For writing a file, the mode to be used is "w", for writing.

In Figure 18.3, we ask the imported module, named `inout` in the `import` statement, to open a file `bookmarks.txt` for writing. If a file with this name has not been loaded into the left pane of the browser (like your active program files), then an empty file with that name will be created there. If it already exists, its contents will be erased.³

As we can see in the figure, the internal name of this file is defined to be `myFile`, and has type `FileStream`. Thus until the file is closed, the *internal* name `myFile` will be associated with the file with *external* name "bookmarks.txt". If you want to refer to the file in your program, you must refer to it as `myFile`, while if you want to save the file to your hard drive from the browser window, you must refer to it as `bookmarks.txt`.

Objects with type `FileStream` have a large collection of methods. The only two that are relevant to us now are `write` and `close`. The `write` method takes an argument of type `String` and writes it to the file at the current position of the file's read-write pointer. For a file that is opened for writing (i.e., in mode "w"), that is always at the current end of the file.

³In the current implementation, the file will only be created if a file with that name was created earlier and then deleted.

Thus if the program executes

```
myFile.write("he")
myFile.write("llo")
```

the file will contain the string “hello” (without the quotes).

In Figure 18.3, the `for` loop writes the contents of the three bookmarks into the file. After executing the loop, the contents of the file `bookmarks.txt` will be:

```
Google ---> www.google.com
Amazon ---> www.amazon.com
Pomona College ---> www.pomona.edu
```

You will notice that we appended the string `"\n"` to the end of each of the bookmarks. That was necessary because, as we saw above, `write` does not automatically add a “new line” character at the end of each `write` the way that `print` does. While this is an extra bit of work for the programmer, it allows the programmer finer control over the layout of the final output.

Finally, when you are finished reading or writing a file, always close it by sending it a `close` method request. This terminates the connection between the internal name and the external file. If you do not close the file, the new contents may not be written, so don’t forget!

As you already know, files in the “Files” pane on the left side of the browser window are retained between accesses to the Grace compiler web page. However, if the browser cache is cleared or you open a different browser, the file is likely not going to be there. Thus we recommend that if you want to save the file, you select the file and then click “Download” from the menu bar above the editing pane. If you need it later, you can always reload it, just as you reload program files, by clicking the up-arrow icon.

18.3 Writing web pages

The output file produced above was pretty readable, but to be really usable our bookmarks page would be loadable into a web browser and have clickable links that would take us to the sites without having to type in the actual address. That is our goal in this section.

If we simply load the file generated by the `saveBookmarksFile` method in a browser, we would see a web page that looks like Figure 18.4. This is not at all what we were looking for as the user would have to copy and paste the URL’s into the browser to get to the given web pages.

Instead we would like a web page that displays more like Figure 18.5.

While web pages can get very complicated and are typically built with powerful tools, we will focus on a very simple kind of web page that we can easily generate with a Grace program and that displays as above.

The difference between a page like that displayed above and what we have produced so far is a matter of encoding. The real bookmarks file contains text that uses a language called HTML (or Hypertext Markup Language) to describe what we should see when we view the bookmarks. This is the standard language used to describe the layout and behavior of web pages. Fortunately, we don’t have to fully understand this language to make our code produce a file that imitates the behavior of a real bookmarks file.

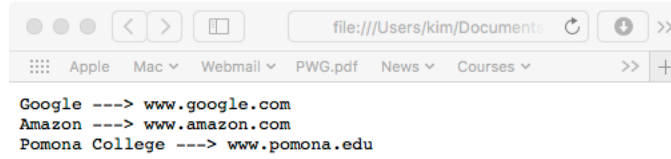


Figure 18.4: Browser display of file produced by `saveBookmarksFile`

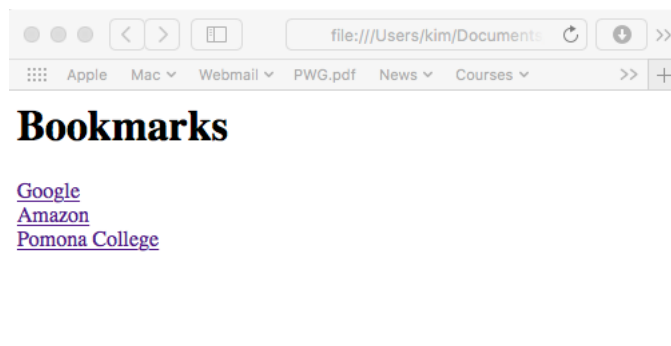


Figure 18.5: Browser display of a usable bookmarks file


```

<html>
<body>
<H1>Bookmarks</H1>
<p><DL>
  <DT><A HREF="http://www.google.com">Google</A></DT>
  <DT><A HREF="http://www.amazon.com">Amazon</A></DT>
  <DT><A HREF="http://www.pomona.edu">Pomona College</A></DT>
</DL></p>
</body>
</html>

```

Figure 18.6: Source view of HTML for web page shown in Figure 18.5

Web pages are formatted using tags in a language called html (for hypertext markup language). Each opening tag has a matching closing tag. For example, the opening tag `<html>` has a closing tag `</html>`. In each case, the closing tag is constructed from the opening tag by replacing the initial `<` by `</`. Thus `<p>` is closed with `</p>`. The file produced by our program has a name that ends in “.html”, one of the standard suffixes used for text files that contain web pages.

The HTML text that produces the browser display shown in Figure 18.5 is shown in Figure 18.6.⁴

When we create a file of bookmarks like this, only the lines starting with `<DT>` vary. Everything in the first four lines can stay the same, while the last three lines can also stay the same. Only the middle three lines (or however many bookmarks you have) will vary depending on the bookmarks.

We note there that the actual bookmarks files in modern browsers are quite a bit more complicated than what we are showing here, but the ideas are similar. We are actually generating a fairly simple web page that will display a list of clickable links.

Let’s focus our attention on the three lines that start with `<DT>` and actually encode the bookmarks that we want displayed. If we look at these lines carefully we can see that each line contains five parts:

1. the text

```
<DT><A HREF="http://
```

2. the address portion of a bookmark

3. the text

```
" >
```

4. the short description of the bookmarked site

5. the text

⁴ In most browsers, there is a menu item that allows one to view the actual HTML text describing the file being viewed.

```

// return a string encoding the bookmark in a standard format
// expected for a clickable entry in a web page}
method asBookmarkEntry → String {
  " <DT><A HREF=\"http://\" ++ address++ "\">" ++ description ++ "</A></DT>\n"
}

```

Figure 18.7: Definition of the `asBookmarkFileEntry` method

```

method saveBookmarksAsWebFile(marks: List[Bookmark]) {
  try {
    def myFile: inout.FileStream = inout.open ("bookmarks.html", "w")
    myFile.write("<html>\n<body>\n<H1>Bookmarks</H1>\n<p><DL>\n")

    for (marks) do {bm: Bookmark →
      myFile.write(bm.asBookmarkFileEntry++ "\n")
    }

    myFile.write("</DL></p>\n</body>\n</html>\n")

    myFile.close
  } catch {ex: EnvironmentException →
    print "can't access bookmarks file"}
}

```

Figure 18.8: Method to create file allowing selection of bookmarks in web page

`</DT>`

While these lines may look complicated, all one has to do to produce them is concatenate the appropriate substrings together. It really is no more complicated than the `asString` method included in the `Bookmark` class shown in Figure 18.1. A method named `asBookmarkFileEntry` is shown in Figure 18.7.

This method, which would be included in the definition of the `bookmark` class, will return a `String` describing a single bookmark using the HTML notation expected in a standard bookmarks file. If we simply change the line

```
myFile.write (bm.asString ++ "\n");
```

from the `saveBookmarksFile` method shown in Figure 18.3 to

```
myFile.write (bm.asBookmarkFileEntry++ "\n")
```

then invoking the `saveBookmarksFile` method will produce lines describing all the bookmarks in a `BookmarkList` in the appropriate format. Thus we can see that, in many cases, placing the data we want in a text file has more to do with manipulating `Strings` than with the details of reading files. The final version of the method to save the bookmarks in html is given in Figure 18.8.

Exercise 18.3.1 Write a method named *saveHighScores* that could be used in a game program to write the latest list of high scores to a file. Assume that the method will be passed a list of `ScoresDescs` as a parameter where a `ScoreDesc` is an object that implements at least the two methods:

```
// Returns the numerical score  
method score -> Number
```

```
// Returns name of player who earned the score  
method playerName -> String
```

Each line of the file should consist of the value of a single score followed by a space and the player's name. The file created should be named `highscore.txt`.

18.4 Readers

When a browser first starts executing, it typically reads the contents of a bookmarks file and uses those contents to create a bookmarks menu. In this section we will learn to write the code that would enable a browser to access a bookmarks file in this way.

In the preceding section, we encountered a `Grace` class that enables us to access an external file and to write text into that file. It should come as no surprise that `Grace` also includes objects that can be used to extract text from files. These objects are created by the same `open` method used to create writers. The only differences are that we open them in mode “r” (for read) rather than “w”, and that they must exist in the browser before being opened. This last bit means that before you can open a file for reading in `Grace`, you must first upload it into the browser by clicking on the upload icon (the arrow pointing upward) in the upper left corner of the window when on the `Grace Editor` web page.

A simple example of reading a file can be found in Figure 18.9. Let's see how this works.

If you attempt to open a file for reading that does not already exist in the Files pane of the `Grace Editor` window, the system will throw an `EnvironmentException`. This will need to be handled or the program will crash.

Once a file has been successfully opened, a variety of methods are available for getting the contents from the file. If you want to get all of the contents of the file at once, requesting a `read` method on it will return the complete contents of the file. Thus if the file contains three lines (like our `bookmarks.txt` example earlier, the results returned will be a string containing the full contents, with different lines joined by newline characters (i.e., “\n”). Thus printing the string will reproduce the contents of the file in the program output pane below the `run` button.

Of course, getting the file contents as a single giant string is often not the most convenient way of obtaining information from a file. More often, we want to read the contents either a line at a time or a character at a time.

In Figure 18.9, we have followed the `print(contents)` with a while loop that reads the file contents in a line at a time using the `getline` method and then prints each of them separately, with the line number prepended. Executing the `getline` method also moves the file read-write pointer to the end of that line. On the provided code, the loop continues as long as we are not at the end of the file (checked with method `eof`). Note that if you don't check for the end of file and just keep reading, you will get empty strings as the result of the read method request.

The output is as given below if the program above is run with the same file contents as earlier in the chapter.

```

// Open file for reading
def myReadFile = inout.open ("bookmarks.txt","r")

// Read the entire contents of the file into contents
var contents: String := myReadFile.read

// close the file
myReadFile.close

// print the contents of the file
print(contents)

// print file contents out a line at a time
// This time with line numbers
var index: Number := contents.indexOf("\n")
var start: Number := 1
var lineNumber: Number := 1
while {index > 0} do {
  print (lineNumber.asString ++ ": " ++ contents.substringFrom (start) to (index-1))
  lineNumber := lineNumber + 1
  start := index + 1
  index := contents.indexOf("\n") startingAt(start)
}

```

Figure 18.9: Reading from a text file

```

1: Google ---> www.google.com
2: Amazon ---> www.amazon.com
3: Pomona College ---> www.pomona.edu

```

Notice that the `getLine` method takes all characters of the line up to and including the newline character. That is why we used the `substringFrom()to()` method to grab all of the characters aside from the newline character from each line.

If desired we could also read a character at a time using the `next` method. Since we won't need that method here, we won't discuss it further.

Let's look at another example of reading in text from a file. In the last section we showed how to write all the entries from a list of `Bookmark` into a file that could be used as a web page. Let's go the other direction and start with a web page of the form we generated earlier and extract the list of `Bookmarks` from it. See the code in Figure 18.10.

We begin by creating an empty list `newMarks` to hold the book marks and then open the file `bookmarks.html` for reading. We now process each line of the input until we reach the end of the file.

We read each line using the `getline` method and then check to see if the line contains the substring "`<A`". If not, then it doesn't contain a link and can be ignored. If so, we want to create a `Bookmark` object from the string and add it to `newMarks`. We use a helper method, `extractBookmark` to extract the address and description from the line and then package that information up into a bookmark. We leave it as an exercise to read the string manipulation code in that method.

```

method retrieveBookmarks → List[[Bookmark]] {
  def newMarks: List[[Bookmark]] = list.empty
  try {
    def bookmarkFile: inout.FileStream = inout.open("bookmarks.html", "r")
    while {!bookmarkFile.eof} do {
      def line: String = bookmarkFile.getline
      if (line.indexOf("<A ") > 0) then {
        newMarks.add(extractBookmark(line))
      }
    }
    bookmarkFile.close
  } catch {ex: EnvironmentException →>
    print "can't open bookmarks file"}
  newMarks
}

method extractBookmark(line: String) → Bookmark {
  def start: Number = line.indexOf("://") + 3
  def end: Number = line.indexOf(">") startingAt (start + 1) - 1
  def address: String = line.substringFrom (start) to (end)
  def endDescription: Number = line.indexOf("</A>") - 1
  def description: String = line.substringFrom (end + 2) to (endDescription)
  bookmark (description, address)
}

```

Figure 18.10: The `retrieveBookmarks` method

Exercise 18.4.1 *Sometimes, it is useful to know just how large a file is. With this in mind, write a method named `countLines` that takes the name of a file as a `String` parameter and returns the number of lines in the file as a result. Your method should return 0 if any sort of `EnvironmentException` occurs while you are trying to access the file.*

18.5 Summary

In this chapter we have seen how Grace programs can read and write data from files. To get access to a file, we need to open it in either read or write mode. If you are opening it for reading, you will need to make sure it is showing in the file pane, while if you are opening it for writing, it will be created there if it does not already exist – and the contents will be erased if it is there.

If you look through all our file examples you may be surprised to discover that we only used four new methods in this chapter: `read`, `getline`, `write`, `eof`, and `close`. While there are other methods available for items of type `FileStream`, the methods presented are almost the only ones needed to write programs that manipulate files and perform network communication. Do make sure that whenever you are done with a file you remember to close it. This is especially important when writing to a file as otherwise the contents may not be updated after the writes.

The mechanisms presented in this chapter are of great practical significance since few useful programs can be written without the ability to save and retrieve data stored in files.

18.6 Chapter Review Problems

Exercise 18.6.1

- a. *Explain what happens if you attempt to open a file for reading using an expression such as*

```
inout.open ("myBookmarks", "r")
```

but the file named does not exist in the file pane.

- b. *Explain what happens if you attempt to open a file for writing using a construction such as*

```
inout.open ("myBookmarks", "w")
```

but the file named does not exist in the file pane.

Exercise 18.6.2 Below you will find several code segments accompanied by brief descriptions of their intended purposes. Each segment contains one or more mistakes. Identify the mistake(s) and explain how to correct the code.

- a. The following method is intended to behave somewhat like the `indexOf` method for `Strings`. Given the name of a file and a `String` to look for, it should return the number of the first line that contains the search string or 0 if the string cannot be found. As shown, the code contains two important, but subtle logical errors that might not even be detected after extensive testing. Explain what is wrong and show how to fix the problems.

```
method file (fileName: String) indexOf (toFind: String) → Number {
  def source: inout.FileStream = inout.open (fileName, "r")

  var lineNum: Number := 0
  while { !source.eof } do {
    def curLine: String = source.getline
    lineNum := lineNum + 1
    if (curLine.indexOf( toFind ) != 0) then {
      return lineNum
    }
  }

  source.close
  return 0
}
```

- b. The following method is supposed to store all the `Strings` in the list passed to it as its first parameter into the file whose name is provided as its second parameter. Each item in the list should be printed on a separate line. It contains an error that will affect the output. Identify the error and explain how to fix the program.

```
method saveStringArray(lines: List[[String]], fileName: String) → Done {
  try {
    def destination: inout.FileStream = inout.open (fileName, "w")

    for (lines) do {line: String →
      destination.write (line)
    }

    destination.close
  } catch {ex:EnvironmentException → print "error writing {fileName}"}
}
```

- c. The method `fileConcat` is intended to concatenate two files to produce a third file. The names of the three files are provided as `String` parameters to the method. It uses a second method shown below named `copyToDest` to copy the contents of each of the two source files to the destination. `fileConcat` is supposed to work by first using `copyToDest` to copy the first source file to the destination and then using it again to copy the contents of the second source file to the destination. Unfortunately, when the program is run the final contents of the destination file are not as expected. What is wrong with the program as shown? What is left in the destination file? How could it be fixed?

```

method fileConcatFrom(source1name: String, source2name: String) to (destName: String) {
  copyFrom (source1) to (dest)
  copyFrom (source2) to (dest)
  source1.close
  source2.close
  dest.close
}

```

```

method copyFrom (sourceName: String) to (destName: String) → Done {
  def source: inout.FileStream = inout.open (sourceName "r")
  def dest: inout.FileStream = inout.open (destName, "w")
  while {!source.eof} do {
    def curLine: String = source.getline
    dest.write (curLine ++ "\n")
  }
  source.close
  dest.close
}

```

Exercise 18.6.3 Each of the following methods is designed to perform an operation on one or more text files containing words stored one per line in alphabetical order. For each method, briefly explain what operation it performs.

a.

```

method performOP1(source1: inout.FileStream, source2: inout.FileStream,
  destination: inout.FileStream) → Done {

  var isDone: Boolean := s1.eof || s2.eof

  var line1: String
  var line2: String
  if (!isDone) then {

```



```

line1 := source1.getline
line2 := source2.getline

do {
  if ( line1 < line2) then {
    if (source1.eof) then {
      isDone := true
    } else {
      line1 := source1.getline
    }
  } else {
    if (line1 == line2) then {
      destination.write(line1 ++ "\n")
      if (source1.eof) then {
        isDone := true
      } else {
        line1 := source1.getline
      }
    }
  }

  if (source2.eof) then {
    isDone := true
  } else {
    line2 := source2.getline
  }
} while {!isDone}
}
}

```

b.

```

method performOP2(source: inout.FileStream,
  destination: inout.FileStream) {

  var line1: String
  var line2: String

  if (source.eof) then {return}
  line1:= source.getline
  destination.write (line1 ++ "\n")

  while {!source.eof} do {
    line2 := line1
    line1 := source.getline
    if (line1 ≠ line2) then {
      destination.write(line1 ++ "\n")
    }
  }
}
}

```

c.

```
method performOP3(source1: inout.FileStream,
    source2: inout.FileStream) → Boolean {

    while {!source1.eof && !source2.eof} do {
        var line1: String := source1.getline
        var line2: String := source2.getline
        if (line1 ≠ line2) then {return false}
    }

    return source1.eof && source2.eof
}
```

d.

```
method performOP4(source1: inout.FileStream, source2: inout.FileStream) → Boolean {

    while {!source1.eof && !source2.eof} do {
        var line1: String := source1.getline
        var line2: String := source2.getline
        while {(line1 > line2) && !source2.eof} do {
            line2 := source2.getline
        }
        if (line2 ≠ line1) then {
            return false
        }
    }

    source1.eof
}
```

18.7 Programming Problems

Exercise 18.7.1 *In Exercise 18.4.1 we asked you to write a method to count the number of lines in a text file. Please write a complete program that counts not only the number of lines in a file, but also the total number of characters and the total number of words. For our purposes, a word is any sequence of non-blank characters that fall together on a single line.*

When the program begins execution it should display a blank JTextArea in its window and then use a JFileChooser to present a dialog box through which the user can select the file to be examined. The program should then count the lines, words and characters in the file. When it is done, the results should be displayed in the JTextArea in a format similar to that shown in Figure 18.11.

Exercise 18.7.2 *No one likes to receive junk mail or spam, but someone has to send it! With this in mind, please write the following program which would be very handy for compiling large mailing lists of addresses.*

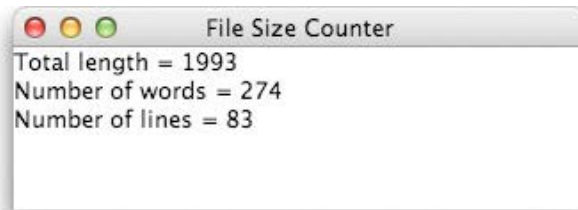


Figure 18.11: Output format desired for Exercise 18.7.1

The program we have in mind will read in two text files containing lists of addresses. To keep things simple, you may assume the contents of each line is a single email address. Also, assume that the addresses are sorted in alphabetical order. The goal of the program will be to create a new file containing every address that appeared in either of the input files but no duplicates. You may assume that there are no duplicates in either input file.

Chapter 19

Searching and Sorting

We all spend too much time searching for things and trying to arrange or sort our possessions so that it is easier to find them when they are needed. Computers spend lots of time searching and sorting as well. For example, consider a search engine like Google. First it must traverse all of the web pages it can find, keeping track of a variety of information. Then it must sort it into some kind of order so that it becomes easier to search that massive amount of information when a user types in a query.

In this chapter we will look at the problems of searching and sorting. These operations are so important that efficient versions of these algorithms are typically supplied in software libraries. One important reason for examining these algorithms is to illustrate that very clever algorithms can be extremely efficient compared to naive algorithms. Another reason for looking at them is to provide an example of how we can analyze the complexity of an algorithm, which will help provide estimates on the comparative running time of different algorithms.

We begin by looking at linear and binary searches and their complexity. We then look at three sorting algorithms, the selection, insertion, and merge sorts. While there are many other useful sorting algorithms, these three are sufficiently different to suggest the variety of algorithms possible.

19.1 Searching

There are many of kinds of searches that can be done on a computer. For example, one can look for the highest value in a list of numbers, or the most frequently occurring value in a list. We have seen examples of searches based on such global criteria in Chapter 13.

Other kinds of searches involve matching some specified external criterion. Examples include:

- Looking in a collection for a specific value (e.g., looking for your name in a list of prize winners).
- Looking for a value with a specific property (e.g., finding the object on a computer screen containing the mouse location).
- Looking for a record in a database (e.g., finding all CDs by a given artist).

```

method search (numList: List[[Number]]) for (key: Number) -> Number {
  for (1 .. numList.size) do {i: Number ->
    if (key == numList.at (i)) then {
      return i // immediately return if find key
    }
  }
  0 // didn't find key
}

```

Figure 19.1: Iterative version of linear search of a Number list.

- Searching for all occurrences of a value in a collection (e.g., finding all occurrences of the word “search” in this text).

In this section, we will focus on the first kind of search in this list, but exercises will investigate some of the other kinds of searches listed above.

19.1.1 Linear Searches

We begin with a very simple linear search of a list. For simplicity we will consider a list of Numbers, though it is easy to modify the code for lists of values of other types. One place where such a search would be useful is in an example described in Section 13.5. There a list held the bib numbers of racers in the order in which they finished. By searching for a given bib number we could determine the place in which that racer finished.

We will use the variable `key` for the element to be found. For this search we will simply start looking through the list from the beginning, returning the index in the list of the key if it is found and 0 if it is not. As in Section 13.5, we select 0 because that is not a legal subscript in the list, and hence will be clearly recognizable as an indication that `key` is not in the list. The method is shown in Figure 19.1.

The code for the method is straightforward, using a `for` loop to iterate through all of the elements of the list. Notice that we return the value of the index `i` as soon as we find a match. If we complete the `for` loop without finding `key` then 0 is returned.

19.1.2 Recursive Processing of Lists

In Section 11.1.2 we wrote recursive classes representing lists of URLs. These classes included a recursive method `contains` that determined whether or not a given URL was in the list. We can also write recursive methods on built-in Grace lists, including one that will perform a linear search of a list. Because a list is not a recursively-defined structure, this recursive method is more like those given in Section 11.2.

A method, `rhelpSearch`, that performs a recursive version of linear search is shown in Figure 19.2. This recursive method takes three parameters rather than two. The extra parameter, `start`, indicates where in the list the method should begin the search.

The body of the method is quite simple. There are two base cases. The first is when `start` is beyond the end of the list. In that case, 0 is returned to indicate that the key is not in the list. The second base case is when `key` is at position `start`, in which case `start` is returned as

```

// return index of key in numList if search from start
method rhelpSearch (numList: List[[Number]]) for (key: Number) startingAt (start: Number) -> Number {
  if (start > numList.size) then {return 0}
  if (key == numList.at (start)) then {return start}
  rhelpSearch (numList) for (key) startingAt (start + 1)
}

```

Figure 19.2: Recursive version of linear search

the location of key. If both of those fail, then we recursively search starting from the next position in the list.

We can apply the rules listed in Section 11.2 in order to convince ourselves that this method works correctly. We've already seen that we have two base cases. We can measure the complexity of a recursive call by the number of elements between start and the end of the list. Each recursive call is simpler because we begin one place farther along the list, and hence the number of elements left to be searched is one smaller.

Finally, if we assume that `rhelpSearch(list) for (key) startingAt (start + 1)` works correctly, then it is easy to argue that the method works correctly.

- If start is beyond the last element of the list, then key is clearly not in the list between `list.at (start)` and the end of the list, so returning 0 is appropriate.
- If key is equal to `list.at (start)` then start is clearly the index of first element equal to key between `list.at (start)` and the end of the list, so returning it is correct.
- Otherwise the method invocation returns the results of `rhelpSearch (numList) for (key) startingAt (start + 1)`. If `rhelpSearch (numList) for (key) startingAt (start + 1)` returns 0 then, by our assumption on the correctness of that invocation, we know that key does not occur between `start + 1` and the end of the list, and hence does not occur between start and the end of the list. On the other hand, if it returns the non-negative Number `n` then we know that `n` is the first index of key between `start + 1` and the end of the list. Because `list.at (start)` is not key, `n` must be the first index of key between start and the end of the list.

This recursive version of search requires more parameters than non-recursive versions. Generally we prefer to have the same parameters for calls to the recursive and non-recursive versions of a method, as the user should not have to worry about whether the method body is recursive or iterative. We can accomplish this by providing an extra method whose only job is to make an initial call of the recursive method with the extra parameters. The method `rsearch` below is such a method.

```

method rsearch (numList: List[[Number]]) for (key: Number) {
  rhelpSearch(numList) for (key) startingAt (1)
}

```

All that it does is call `rhelpSearch` with the list and key parameters as well as with a 1 as the initial value of the start parameter.

It follows from our discussion above that the call to `rsearch` will return the index of the first occurrence of key between 1 and the end of the list. In other words, it will return the index of

the first occurrence of `key` in the entire list. It returns 0 if it is not found. Thus this method has exactly the same description (and set of parameters) as the `search` method in Figure 19.1.

Exercise 19.1.1 *Suppose the list passed to the search methods only has relevant entries in the first n entries (which may be smaller than `numList.size`). How would you change the iterative and recursive search methods to accommodate this more restricted search? Notice that you will need to add a new parameter for the size.*

Exercise 19.1.2 *Suppose you wish to find the index of the last entry of `key` in a list. How would you change the iterative and recursive search methods to accommodate this search?*

Exercise 19.1.3 *Suppose that instead of searching a list of Numbers, you are to search a list of elements of type `L` for a key of type `K`. Assume there is a parameterless method `key` of `L` that returns a value of type `K`. How must you modify the code of the iterative and recursive search methods to handle these types of data?*

19.1.3 Complexity of Linear Search

With operations like searching that may potentially be run on very large collections of data, it is often important to predict how much time the task may take. An obvious way to do this would be to time the operations on a wide variety of data. However, there are several problems with this.

Not surprisingly, different results will be obtained on different computers. A brand new top-end computer will run faster than an older computer. If the operation is running on a larger data set, it will usually take more time to finish. Moreover, running the same method on the same computer may not always result in the same time on data of the same size. Finding an element at the beginning of a long list will take a much shorter time than looking for an element that is near the end or that is not there. Also, different times will be obtained depending on how many other tasks the computer is performing at the time the method is run.

As a result we generally prefer measures that will help predict relative timing, rather than provide an exact time in milliseconds. For searches, we are most interested in counting the number of comparisons with the key that must be made in a list of size n in order to find the key (or determine that it is not there). Of course, we may be lucky and find the key as the first element of the list, but that kind of “best-case” information is less useful than two other measures: “worst-case” and “average-case” behavior.

For the iterative search in Figure 19.1, it is easy to see that the key is compared with every element in the list until a match with the key is found. The largest number of comparisons (“worst-case”) is obtained when the key either is not in the list or only occurs as the last element of the list. In that case n comparisons are made if the list has n elements. The same number of comparisons are made in the `rSearch` method, as it also compares the key with successive elements until it finds the key element.

Calculating the average number of comparisons made requires some assumptions on our part. First, we assume that the element being searched for occurs exactly once in the list. Second we assume that the key element is equally likely to be in any of the places in the list.

Under those assumptions, the key element is as likely to be in the first half as the second, so it is reasonable to assume that on average we will find it about halfway through the list, so roughly $n/2$ comparisons will be required on average.

19.1.4 Binary Searches

In this section we show how we can find elements with fewer comparisons if we assume that the list to be searched is in order. For simplicity, in the rest of this section we will assume that the list to be searched is in *non-decreasing order*. That is, successive elements either get larger or stay the same in moving from left to right through the list. Similar results can be obtained if the list is in non-increasing order.

If the list is ordered, the intuitive analysis above on the average number of comparisons for a linear search can be extended to cases where the key does not occur in the list, because if we reach an element in the list that is larger than the key, then we know the key cannot possibly occur in the rest of the list. For example, suppose the key is 10 and the list holds the integers:

2 4 8 11 15 22 24 25 34 42 47 53 54 72 93 97

Once we get to the 11 in the list, it is clear that 10 will not occur in the rest of the list. On the other hand, the knowledge that the list is in order does not really reduce the number of comparisons made with a linear search if the key is in the list. As a result, if the list is in order, the number of comparisons is roughly $n/2$ whether the key is in the list or not.

We can take advantage of the ordering of a list in a more clever way to greatly reduce the number of comparisons necessary to find a key. To illustrate, if you are interested in looking up someone's name in a large city's telephone directory, you would never consider using a linear search. Because we know the names in the directory are listed in alphabetical order, if we are looking for Sally Miller, we would open the phone book and start looking roughly in the middle of the book, rather than starting at the first listing on the first page. Of course, if the name was Joe Aardvark or Mary Zyzygy we would start our search closer to the beginning or the end of the book, but it turns out that we do not need even that amount of cleverness to search more efficiently in an ordered list.

To illustrate our strategy, let's examine the simple game of guessing a number in a given range. When a guess is made, the person who chose the number must respond with either yes (the guess is correct), too high (the actual number is smaller than the guess), or too low (the actual number is larger than the guess).

Suppose the game is being played by Alice and Bob, where Alice chooses a number between 1 and 7, inclusive, and Bob has to guess the number. In the absence of any other knowledge about how Alice chose the number, the optimal strategy is to always guess the middle of the range. Thus Bob's first guess should be 4, because there are three numbers above and three below 4 in the range of numbers. Suppose Alice responds that the guess is too low. Then Bob knows that her number is between 5 and 7. If Bob next guesses 6 then he either gets it right or reduces the range to only one remaining element. Thus in the worst case Bob can name the correct number by his third try. Notice that the same strategy works on all intervals of the same size. For example if the range is from 8 to 15 then Bob can guess the number in at most three tries again. (The first guess is 11.)

It is easy to see that at most three guesses are necessary, no matter what number Alice chooses in that range. Now suppose Alice and Bob play a new game where the range of numbers is from 1 to 15. As before, Bob's strategy is to pick the number, 8, in the middle of the range. If he gets it right then the game is over. If he gets it wrong, however, he will have reduced the range to 7 numbers. That is, if Alice says "too high" then he will know the number is in the range from 1 to 7 and if she says "too low" then he will know the number is in the range from 9 to 15. However, we saw above that if the range has size 7 then we can guess the number in at most 3 guesses. Thus after the first guess in this game, Bob will either have gotten it correct or will have reduced the range to one that he can solve with at most 3 guesses. Thus with a range of size 15 Bob can always win with at most 4 guesses.

It is easy to see that a range of size 31 (leaving ranges of 15 on either side after a first guess of 16) can be won with 5 guesses, a range of size 63 can be won in 6 guesses, and a range of size 127 can be won in 7 guesses. Jumping a bit farther ahead, a range of size 1023 can be won in 10 guesses. In general we can show that a number in a range of size at most $2^n - 1$ can always be found in at most n guesses.

We can write this in a slightly different way using logarithms base 2. Recall that $\log_2 m = n$ if and only if $m = 2^n$. Let $k = 2^n - 1$. Then $n = \log_2(k + 1)$. Substituting these into our answer above, we see that a number in a range of size at most k can always be found in at most $\log_2(k + 1)$ guesses.

We will use exactly this strategy of searching by always guessing the middle element to find a key in an ordered list. In this case, we choose the middle element by choosing the element whose index is in the middle of the range of elements to be searched. This search strategy is called a *binary search* because it involves reducing the search range roughly in half after each guess. The general technique of dividing a problem into smaller pieces so that only problems whose size is a fraction of the original have to be solved is called a *divide-and-conquer* strategy.

We will provide both iterative and recursive methods to perform binary searches. This time we will begin with the recursive solution as we believe it is easier to understand. The code for the recursive binary search is given in Figure 19.3. As before, we have a starter method that makes the initial call of the `rhelpBinarySearch` method. In explaining this code we informally use the shorthand `list[start .. end]` to stand for the range of values between `list.at(start)` and `list.at(end)`.

Let's ignore the first comparison between `start` and `end` for the moment and just focus on the rest of the method. We first calculate `midIndex` as the index of the element halfway between `start` and `end`. If there are an odd number of elements in the range, it will be exactly the middle element. However, if there are an even number of elements, there is no middle element, so we will return the index of the element just below the half-way point. We do this by truncating the average of `start` and `end`. Thus if `start` is 1 and `end` is 6, `midIndex` will be 3, rather than the problematic 3.5.

As in the number guessing game above, we then compare the key to the element in the middle of the list. If that element matches the key then we return the index. If key is smaller than that element, then the key cannot occur among the elements in `list[midIndex..end]`. As a result, we return whatever index is obtained by doing a recursive binary search of `list[start..midIndex-1]`. Similarly, if key is greater than the middle element then we know that key

```

method rBinarySearch (numList: List[[Number]]) for (key: Number) -> Number {
  rhelpBinarySearch (numList) for (key) from (1) to (numList.size)
}

method rhelpBinarySearch (numList: List[[Number]]) for (key: Number)
  from (start: Number) to (end: Number) -> Number {
  if (start > end) then {return 0}
  def midIndex: Number = ((start + end) / 2).truncated
  if (key == numList.at (midIndex)) then {return midIndex}
  if (key < numList.at (midIndex)) then {
    rhelpBinarySearch (numList) for (key) from (start) to (midIndex - 1)
  } else {
    rhelpBinarySearch (numList) for (key) from (midIndex + 1) to (end)
  }
}

```

Figure 19.3: Recursive binary search

cannot occur in the elements `list[start..midIndex]`, so we return the index obtained by doing a binary search of `list[(midIndex + 1) .. end]`.

For example, suppose as before that `key` is 22 and the list holds the integers shown below. To aid our explanation we place the indices of the list above the actual elements.

```

index:           1 2 3  4  5  6  7  8  9 10 11 12 13 14 15
list.at (index:  2 4 8 11 15 22 24 25 34 42 47 53 54 72 93

```

We begin with `start = 1` and `end = 15` and evaluate `rhelpBinarySearch(list)` for (22) from (1) to (15). Then `midIndex = 8`. Because `22 < list.at (8) = 25`, we evaluate `rhelpBinarySearch(list)` for (22) from (1) to (7). That is `start` remains at 1, while `end = 7`. Now `midIndex` is calculated as 4 and we find `22 > list.at (4) = 11`. Thus we evaluate `rhelpBinarySearch(list)` for (22) from (5) to (7). With `start = 5` and `end` remaining at 7, we calculate `midIndex = 6` and find `22 = list.at (6)`. Thus 6 is returned through all of the method calls as the final answer.

Now, what happens if we look for something that is not in the list? For example, suppose we look for `key 21` rather than 22. As above, we get to the call of `rhelpBinarySearch(list)` for (22) from (5) to (7), we calculate `midIndex = 6`, but now we find `21 < list.at (6)`. Thus we call `rhelpBinarySearch(list)` for (22) from (5) to (5) (searching a range including only one value). We calculate `midIndex = 5` and find that `21 > list.at (5)`. Thus we call `rhelpBinarySearch(list)` for (22) from (5) to (4). But now the value of `start` is greater than the value of `end`, indicating that there are no elements left to search. Now we can understand why we have the initial comparison, `start > end`, at the beginning of the method body. If that comparison is true then there is no hope of finding the key, and we return 0.

Because we are using exactly the same strategy as in the number guessing game, the binary search of a list with k elements takes at most $2 * \log_2(k + 1)$ comparisons, as one test for equality and one test for $<$ is made for each calculation of the middle index. In particular, a binary search of 1000 elements takes at most 20 comparisons ($2 * \log_2(1001) < 20$), while a binary search of one million elements takes at most 40 comparisons ($2 * \log_2(1,000,001) < 40$).

```

method iterativeBinarySearch (numList: List[[Number]]) for (key: Number) -> Number {
  var start: Number := 1
  var end: Number := numList.size
  while {start <= end} do {
    def midIndex: Number = ((start + end) / 2).truncated
    if (key == numList.at (midIndex)) then {return midIndex}
    if (key < numList.at (midIndex)) then {
      end := midIndex - 1
    } else {
      start := midIndex + 1
    }
  }
}
0
}

```

Figure 19.4: Iterative binary search

This is quite an improvement over the one million comparisons that would be needed with a linear search. When you consider that the world-wide web contains nearly 5 billion web pages, you can see that having a fast way of searching through a lot of data would be quite valuable.

We concluded that the largest number of comparisons that must be made in searching for a key in a list of size k is $2 * \log_2(k + 1)$. Interestingly, if the key is guaranteed to be in the list, the average case is just about two fewer comparisons, roughly $2 * \log_2(k + 1) - 2$. For example, if you label each item of the sample list of values given earlier with the number of comparisons necessary to find that element, you will find that the average number of comparisons necessary is 4.4, while the maximum is 6.

It is also possible to write an iterative binary search algorithm by imitating what happens with the recursive version. The iterative algorithm is shown in Figure 19.4. In comparing this method with the recursive version we see that the two extra parameters, `start` and `end` have been replaced by local variables with the same names. The outermost if-else statement testing if there are any elements left to be searched has been replaced with a while loop where the code to be executed when no elements are left is now placed just after the end of the loop. Inside the while loop the logic is the same as before, but rather than placing recursive calls in the branches of the last conditional statement, we simply update the variables corresponding to those that were changed for the recursive calls. For example, in the then clause, the call of `helpBinarySearch (numList) for (key) from (start) to (midIndex - 1)` is replaced by an assignment updating `end`, the only parameter that was changed in the recursive call. Similarly, the call in the final else clause is replaced by an assignment to `start`, the only parameter changed in the corresponding recursive call. If you trace through the execution with our previous example of a list where we are looking for a key value of 22, you will see that essentially the same computations and comparisons are made as in the recursive case.

In order to compare the relative efficiency of linear and binary search, we can use the calculations we made earlier of the number of comparisons necessary to search a list. Table 19.1 records the number of comparisons necessary in the worst case to do a linear search on an unsorted list and to do a binary search on a sorted list of size n . While the differences are

n	<i>linear(unsorted)</i>	<i>binary(sorted)</i>
7	7	6
15	15	8
31	31	10
127	127	14
1027	1027	20
1,056,783	1,056,783	40

Table 19.1: Worst case number of comparisons to search a list of size n

minor for small lists, when the list has size greater than 1000, the differences are significant. In general, doubling the size of the list only increases the number of comparisons by 2 for binary search versus doubling the number of comparisons for linear search.

Exercise 19.1.4 Show by example that the binary search algorithm will not return the correct answer if the list is not in order.

Exercise 19.1.5 Suppose that instead of searching a list of Numbers, you are to search a list of type T for a key of type K . Assume there is a method `key` of T that returns a value of type K . You may also assume that K has a method:

method `compareTo(K other)` \rightarrow Boolean

where `compareTo` returns a negative integer, zero, or a positive integer depending on whether the receiver is less than, equal to, or greater than `other`. How must you modify the code of the recursive binary search method to handle these types of data?

19.2 Using Preconditions and Postconditions

Throughout this text we have followed the practice of writing comments before methods in order to provide informal specifications of what the methods do. However, to make it clearer to users under what conditions a method may be invoked and what the results of executing the method will be, it is useful to provide more formal comments in terms of *preconditions* and *postconditions*.

A precondition specifies the minimum conditions that must be true to guarantee that an invocation of the method will work correctly. For example,

```
// PRE: The elements of list are in non-decreasing order
```

would be the precondition for `rBinarySearch`. As in this example, the precondition is written as a comment that begins with `PRE:`. It is the responsibility of the caller to ensure that the precondition is true before calling the method. The precondition of a method typically mentions all of the parameters of the method and may also include information about the state (instance variables) of the object containing the method. The precondition for `rBinarySearch` does not mention the parameter `key` because any integer value may be used for `key`.

A postcondition guarantees what will be true when a method has completed execution. The postcondition is only guaranteed to be true if the preconditions were satisfied before the method was called. A postcondition for the method `rBinarySearch` is:

```
// POST: Returns 0 if key does not occur in list. Otherwise returns  
//       the index of an occurrence of key in list.
```

The programmer of the method is responsible for guaranteeing that the postcondition holds after the method execution if the precondition was true before the method began execution.

We typically put these together as the comments before the method declaration as follows:

```
// PRE: The elements of list are in non-decreasing order  
// POST: Returns 0 if key does not occur in list. Otherwise returns  
//       the index of an occurrence of key in list.  
method rBinarySearch (numList: List[[Number]]) for (key: Number) -> Number {...}
```

Sometimes there is no precondition necessary, and in that case the PRE clause can be omitted entirely. Notice that the method `iterativeBinarySearch` has exactly the same pre- and postconditions. Because they both also have exactly the same parameters, one can be substituted for the other in a program without affecting the program results.

You should think of the combination of precondition and postcondition as a contract between the caller and the provider of a method. If the caller guarantees that the preconditions are true before the method body begins execution, then the provider of the method guarantees that the postcondition will be true after the method execution completes. If the precondition is false before the method is invoked, then the method body is under no obligation to ensure that the postcondition will be true afterward.

This contract allows a clear division of labor and allows a programmer who wishes to use a method to clearly understand what is required and what the results will be of a method without needing to look at the code in the method. This makes the job of the programmer easier because it is one less thing that she or he needs to worry about. For the rest of this chapter we will be careful to include preconditions and postconditions for all methods.

Exercise 19.2.1 *Write the precondition and postcondition for the method `rhlpBinarySearch` that is called by `rBinarySearch`.*

19.3 Sorting

In the last section we saw that the binary search algorithm was much more efficient than linear search if it was applied to an ordered list. In this section we examine three algorithms for sorting the elements in a list. The first two require approximately $n^2/2$ comparisons to sort the list, while the last requires significantly fewer. As before, we develop the algorithms for lists of `Number`, but it will be easy to change them to handle lists of any type as long as the elements can be compared to each other.

The algorithms presented in this chapter will sort the elements of an list to be in non-decreasing order. It would be easy to change each of these algorithms to instead sort in non-increasing order by replacing occurrences of `<` by `>` and vice-versa. As a result we won't bother to show these variants.

```

// POST: Elements of numList have been rearranged into non-decreasing order.}
method iterativeSelectionSort (numList: List[[Number]]) -> Done {
  for (1 .. (numList.size - 1)) do {i :Number ->
    // numList[1 .. i-1] contains the i-1 smallest elements
    // of the original list, sorted properly while the remaining elements of the original
    // list are in numList[i .. numList.size].

    def smallestIndex: Number = indexOfSmallestIn (numList) startingAt (i)
    swap (i, smallestIndex) in (numList)
  }
}

// PRE: start must be valid index for list
// POST: returns an index of the smallest value in list[start .. list.size]
method indexOfSmallestIn (numList: List[[Number]]) startingAt (start: Number) -> Number is confidential {
  var smallIndex: Number := start
  for ((start + 1) .. numList.size) do {i: Number ->
    // numList.at (smallIndex) is smallest in numList[start .. (i-1)]
    if (numList.at (i) < numList.at (smallIndex)) then {
      smallIndex := i
    }
  }
  smallIndex
}

// PRE: i and j must be valid indices for list
// POST: The values stored in list.at (i) and list.at (j) have been interchanged.
method swap(i: Number, j: Number) in (numList: List[[Number]]) -> Done is confidential {
  def temp: Number = numList.at (i)
  numList.at (i) put (numList.at (j))
  numList.at (j) put (temp)
}

```

Figure 19.5: Iterative selection sort and helping methods.

19.3.1 Selection Sort

Probably the simplest way to think about sorting a collection of objects is to first find the smallest element and put it in the first available slot, then find the second smallest element and put it in the next slot, and so on. The *selection sort* works in exactly this way, with only one minor change: we wish to perform the sort in place. That is, rather than copying all of the data to a new list, we want to simply move data around the same list, using as little extra space as possible.

We present the iterative version of the selection sort first. It is shown in Figure 19.5. The method `iterativeSelectionSort` itself is quite simple as we have pushed off most of the details into the helper methods. Let's talk about these first.

The method `swap(i,j) in (numList)` simply interchanges the value in `numList.at (i)` and `numList.at (j)`. It uses a local variable, `temp`, to keep track of the value of `numList.at (i)` temporarily so that it is not lost when that variable is updated in the following statement. It is then

associated with `numList.at (j)` in the last line of the method.

Exercise 19.3.1 *Why does the following code not work for the body of `swap`:*

```
numList.at (i) put (numList.at (j))
numList.at (j) put (numList.at (i))
```

Show what happens when you call `swap (1, 2)` in `(numList with this alternative code when numList.at (1) = 5 and numList.at (2) = 7)`.

The method `indexOfSmallestIn` iterates through the indices of `numList[start .. numList.size]`, keeping track of which one corresponds to the smallest element seen so far. The comment inside the `for` loop of `indexOfSmallest` states that `list.at (smallestIndex)` is the smallest element of `numList[start .. (i-1)]`. This statement indicates an invariant of the method. An invariant is a statement that is true every time execution passes through that point.

Let's convince ourselves that the invariant is true every time execution passes through that point. We do this by essentially presenting an argument via mathematical induction, though we hope you will find the argument compelling even if you haven't seen proofs by mathematical induction.

The first time the invariant is encountered during execution is immediately after `i` is initialized with `start`. Thus `list[start .. (i-1)]` only refers to the single element `list.at (start)`. Because `smallestIndex` was initialized with `start`, the invariant is clearly true.

Next we see that if the invariant is true before executing the body of the loop, then the invariant will remain true at the end of the loop body. So, assume that `list.at (smallestIndex)` is the smallest element of the list elements in `list[start..i-1]`. Because we are going to execute the body of the loop again, we also know that `i < list.length`, and hence `i` is a valid index of `list`.

The body of the loop checks to see if `list.at (i)` is smaller than `list[smallestIndex]`, and if so, updates `smallestIndex` with `i`. In this case, `i` was smaller than the smallest element of `list[start..i-1]`, so it is clearly the smallest element of `list[start..i]`. On the other hand if the conditional is false then `list[i]` is no smaller than `list[smallestIndex]`, so the latter element is the smallest in `list[start..i]`. Thus either way, `list[smallestIndex]` is the smallest in `list[start..i]`. Thus when execution goes to the top of the `for` loop and `i` is increased by one, it will again be the case that `list[smallestIndex]` is the smallest element in `list[start..i-1]`.

Thus every time execution returns to the top of the loop, the invariant will continue to be true. As a result, when the `for` loop finally terminates with `i` equal to `numList.size+1`, we know that `numList.at (smallestIndex)` will be the smallest in `numList[start..numList.size]`, as desired.

Now we are ready to see why `iterativeSelectionSort` correctly sorts the list. Each time through the `for` loop in `iterativeSelectionSort`, the index of the smallest remaining element in the unsorted part of the list – the elements in `list[i..numList.size]` – is assigned to `smallestIndex`, and the element associated with `smallestIndex` is swapped with the element currently in `list.at (i)`. This results in the smallest remaining element being placed in `list.at (i)`. Because this is done for each slot in the list, starting with 1, at the end all of the `listnumList` elements are placed in order.

Notice that the value of `i` during the last execution of the body of the `for` loop will be `list.length-2`. The reason for this is that after the second to last element is filled, there is only one element remaining, the largest element of the list, and it must be in the correct spot.

Exercise 19.3.2 *Give an argument like that used for the invariant in `indexOfSmallestIn` to show that the invariant in the method `iterativeSelectionSort` is always true.*

Let's look at the operation of selection sort on the following list, `elts`, of 5 integers:

```
index:      1  2  3  4  5
elts.at (index):  48 62 38 51 15
```

If we execute `iterativeSelectionSort(elts)` then `i` is initialized to 1 as we enter the for loop. We begin by finding the index, 5, of the smallest element, 15, in `elts[1..5]`. We then swap that element with the one in `elts.at(0)`, leaving us with:

```
index:      1  2  3  4  5
elts.at (index):  15 62 38 51 48
```

We next increment `i` so it has value 2. We find the index, 3, of the smallest element in `elts[2..5]`. Then `elts.at(3)` is swapped with `elts.at(2)`, leaving:

```
index:      1  2  3  4  5
elts.at (index):  15 38 62 51 48
```

We increment `i` again so it has value 3. We find the index, 5, of the smallest element in `elts[3..5]`. Then `elts[3]` is swapped with `elts[5]`, leaving:

```
index:      1  2  3  4  5
elts.at (index):  15 38 48 51 62
```

We increment `i` again so it has value 4. We find the index, 4, of the smallest element in `elts[4..5]`. Then `elts.at(4)` is swapped with `elts.at(4)`, which has no effect.

```
index:      1  2  3  4  5
elts.at (index):  15 38 48 51 62
```

When `i` is incremented again, it will be equal to 5, which is `elts.size`, so the for loop will terminate and `elts` is now in order.

```
index:      1  2  3  4  5
elts.at (index):  15 38 48 51 62
```

The development of an alternative, recursive method to perform a selection sort is straightforward. To sort the elements in the list, `numList`, starting at the element with index `start` proceed as follows:

- If there are less than two elements in `numList` starting with index `start` then `numList` is already sorted. You don't need to do anything.
- Otherwise, find the smallest element in the portion of `numList` beginning with index `start` and swap it with `numList.at(start)`.
- Selection sort the elements in the list starting with the element with index `start+1`.

The Grace code for two methods to accomplish the selection sort can be found in Figure 19.6. As with the searches, the first is a starter method that calls `recursiveSelectionSort` and specifies that the sort begins with the element at index 0. Notice that the code of `recursiveSelectionSort` follows the outline above and uses the methods `indexOfSmallest` and `swap` listed in Figure 19.5.

We can convince ourselves in the usual way that `recursiveSelectionHelper` is correct. If `start >= numList.size` then there is at most one element to be sorted and a list of size 1 (or 0) is already in order. For the recursive case, where the number of elements to be sorted is 2 or more, we assume that `recursiveSelectionHelper` works properly on lists smaller than the current one. The method moves the smallest element of `numList[start..list.length-1]` to `start` and then sorts `numList[start+1..numList.size]` through a recursive call. By assumption, that recursive call

```

method recursiveSelectionSort (numList: List[[Number]]) -> Done {
  recursiveSelectionHelper(numList) startingAt (1)
}

// PRE: start must be a valid index for list
// POST: list[start..list.length-1] has been rearranged and
//       is in non-decreasing order.
method recursiveSelectionHelper (numList: List[[Number]]) startingAt (start: Number) -> Done {
  if (start < numList.size) then {
    def smallestIndex: Number = indexOfSmallestIn(numList) startingAt(start)
    swap(start, smallestIndex) in (numList)
    recursiveSelectionHelper (numList) startingAt (start + 1)
  }
}

```

Figure 19.6: Recursive selection sort.

orders the elements of `list[start+1..numList.size]` properly. Moreover, the element at `start` is less than or equal to all elements in the rest of the list. Thus all elements in `list[start..numList.size]` are in order.

Exercise 19.3.3 *Suppose that instead of sorting a list of Numbers, you are to sort a list of type T for a key of type K . Assume there is a method `key` of T that returns a value of type K . You may also assume that K has a method:*

```

method compareTo(K other) -> Number

```

where `compareTo` returns a negative integer, zero, or a positive integer depending on whether the receiver is less than, equal to, or greater than `other`. How must you modify the code of the iterative or recursive selection sort method to handle these types of data?

19.3.2 Complexity of Selection Sort

Because we are often interested in sorting large collections of data, it would be useful to have some indication of how long it would take to sort a large list using the selection sort. As with our searching methods, we calculate the total number of comparisons necessary to sort a list. Suppose the original list has n elements, where $n > 1$. It takes $n - 1$ comparisons to find the smallest element of the list (compare the first with the second, compare the smaller of those with the third, etc.). Once the smallest element has been placed in the first slot we must find the smallest of the remaining $n - 1$ elements, which takes $n - 2$ comparisons. We continue with successively smaller lists, with the searches requiring $n - 3$ comparisons, $n - 4$ comparisons, \dots , all the way down to the last pass through the list when there are only two elements and it takes only a single comparison to determine the smallest element.

If we add up all of these comparisons, we get $S = (n - 1) + (n - 2) + \dots + 1$. We can calculate this sum by using the following clever observation. If we reverse the order of the

terms on the right side of the sum and then add the two equations, we get:

$$\begin{array}{r} S = (n-1) + (n-2) + \dots + 1 \\ S = 1 + 2 + \dots + (n-1) \\ \hline 2S = n + n + \dots + n \end{array}$$

The right side of the equation consists of $n - 1$ copies of n added together, equaling $n(n - 1)$. Thus we get $2S = n(n - 1)$ and hence the total number of comparisons of elements to do a selection sort on an list of size n is given by:

$$S = \frac{n(n-1)}{2}.$$

Notice that the number of comparisons of elements is the same regardless of the initial order of the elements. Thus the average case time is the same as the worst case. In the next section we introduce a different sorting algorithm whose worst case time is the same as selection sort, but whose average number of comparisons is half of that for selection sort.

19.3.3 Insertion Sort

Insertion sort is an algorithm that many people use when they need to sort a relatively small collection of cards or papers by hand. A very informal iterative explanation of the algorithm is as follows:

- Start with a single card in your hand.
- Take the second card and put it either before or after the first card so that the two cards are in order.
- Insert the third card into the ordered collection of the first two cards so that all three are now in order.
- Insert the fourth card in the appropriate place in the existing ordered collection of three cards so that all four are in order.
- Continue until all cards have been inserted properly.

The method `iterativeInsertionSort` in Figure 19.7 implements this version of insertion sort. It follows the outline above, inserting the element in `numList.at(2)` into the list consisting only of `numList.at(1)`, then inserting `numList.at(3)` into the list consisting of `numList.at[1..2]`, and so on until finally, `numList.at(lastIndex)` is inserted into `numList.at[1..lastIndex]`, where `lastIndex` is `numList.size`.

The precondition of the helper method `insertNext` states that `numList.at[1..lastIndex-1]` is already in order. The first four lines of the method (up to and including the `newPosn := newPosn + 1` statement) result in setting `newPosn` to be the index where `numList.at(lastIndex)` will be inserted. That is, `numList.at(newPosn-1) <= numList.at(lastIndex) < numList.at(newPosn)`. In order to insert the new element there, we must move all of the elements in `numList[newPosn..lastIndex-1]` up one slot in the list. That is the purpose of the `for` loop. Finally, the new element, which has been saved in `tempElt` is placed into `numList.at(newPosn)`. Because `numList[1..lastIndex-1]` was in order before the sort, `numList.at[1..lastIndex]` will be in order at the end.

```

//      POST: list is sorted into non-decreasing order
method iterativeInsertionSort (numList: List[[Number]]) -> Done {
  for (2 .. numList.size) do {i: Number ->
    insertNext (i) in (numList)
  }
}

// PRE: list[1..lastIndex-1] is in non-decreasing order and
//      2 <= lastIndex <= list.length,
// POST: list[1..lastindex] contains the same elements as it did before
//      the execution of the method, but now is in non-decreasing order.
method insertNext (lastIndex: Number) in (numList: List[[Number]]) -> Done {
  var newPosn: Number := lastIndex - 1

  // find where element at lastIndex goes
  while {(newPosn > 0).andAlso {numList.at (newPosn) > numList.at (lastIndex)}} do {
    newPosn := newPosn - 1
  }
  newPosn := newPosn + 1

  def tempElt: Number = numList.at (lastIndex)

  // make space for moving element at lastIndex into newPosn
  for ((newPosn .. (lastIndex - 1)).reversed) do {i: Number ->
    numList.at (i + 1) put (numList.at (i))
  }

  numList.at (newPosn) put (tempElt)
}

```

Figure 19.7: Iterative insertion sort

```

// POST: list is sorted into non-decreasing order
method recursiveInsertionSort (numList: List[[Number]]) -> Done {
  recursiveInsertionHelper (numList) endingAt (numList.size)
}

// PRE: lastIndex must be valid index for list
// POST: The elements originally in list[0..lastIndex] have been
//       rearranged to be in non-decreasing order.
method recursiveInsertionHelper (numList: List[[Number]]) endingAt (end: Number) -> Done {
  if (end > 1) then {
    recursiveInsertionHelper (numList) endingAt (end - 1)
    insertNext (end) in (numList)
  }
}

```

Figure 19.8: Recursive insertion sort.

Exercise 19.3.4 Trace the operation of insertion sort on the following list, *elts*, of 5 integers:

```

index:      1  2  3  4  5
elts.at (index): 48 62 38 51 15

```

Exercise 19.3.5 What will the value of `newPosn` be after the while loop if `numList.at (lastIndex)` is smaller than all elements of `numList[1..lastIndex-1]`. Explain why this results in `numList.at (lastIndex)` being inserted in the correct place in `numList`.

Exercise 19.3.6 Determine invariants to be placed at the top of the while and for loops in `insertNext`. Argue as we did with `indexOfSmallest` that each is always true when execution returns to the top of the loop.

A recursive description of insertion sort is even simpler. To do an insertion sort of a collection of n elements:

- If there are one or fewer items, stop as you are done.
- Otherwise perform an insertion sort of the first $n - 1$ elements; then insert the n th element in order.

The implementation of the recursive insertion sort is in Figure 19.8. The code for the method `recursiveInsertionSort` is just as described above. The method `recursiveInsertionSort` calls `recursiveInsertionHelper` with the correct initial value for the parameter `end`.

Exercise 19.3.7 Provide an argument that the method `recursiveInsertionSort` works correctly. You may assume that `insertNext` is correct. That is, if the preconditions are met before executing the method, then the postconditions will be true at the end.

19.3.4 Complexity of Insertion Sort

As with the searches and selection sort, we wish to compute the complexity of insertion sort. We will see that in the worst case, the number of comparisons necessary to perform an insertion sort on a list of size n is the same as for a selection sort. However, on average the insertion sort does only about half of the comparisons.

Most of the complexity in the algorithm comes from the `insertNext` method. When `insertNext` is called, there are `lastIndex-1` elements in the list before `lastIndex`. In the worst case, the while loop will have to compare `list[lastIndex]` with every one of those elements. On average, however, the new element will go in the middle of the list, so only half as many comparisons will be needed.

If we look at the iterative version of the algorithm, we see that `insertNext` is called with `i` starting out at 2 and increasing up to `numList.size`. Thus if `numList.size` is n , the total number of comparisons in the worst case is $1 + 2 + \dots + (n - 1) = \frac{n(n-1)}{2}$ as before. However, in the average case, the number of comparisons in each call of `insertNext` is half of the worst case time, so the sum is as well. Thus in the average case, the total number of comparisons is closer to $\frac{n(n-1)}{4}$.

Exercise 19.3.8 *We have been analyzing the worst case behavior of methods, but it can also be interesting to look at best case behavior.*

- a. *Suppose all of the elements in a list are already in order. How many comparisons are necessary to perform an insertion sort?*
- b. *Does the number of comparisons necessary to perform a selection sort depend the ordering of the list?*
- c. *Suppose a list is partially sorted. Which of the two sorts is likely to be much faster?*

Exercise 19.3.9 *Compare the number of assignment statements involving elements of the list that are executed in the selection sort with the number in the insertion sort. If we take those into consideration, how will that affect the comparison of the selection sort with the insertion sort? Run experiments on large randomly selected lists to check the time complexity of each.*

19.3.5 Merge Sort

We've now seen two sorts whose worst-case complexity is $\frac{n(n-1)}{2}$. We will see in this section that we can do significantly better if we try a divide-and-conquer strategy.

Before introducing this new sort, let's go back and think about how the recursive versions of the selection and insertion sorts worked on a list with n elements. With the selection sort we first did some work – found the smallest element, and then made a recursive call, performing a selection sort of the remaining $n - 1$ elements of the list. With the insertion sort we first made a recursive call, performing an insertion sort of the first $n - 1$ elements of the list, and then did some work – inserted the last element where it belonged in the already sorted elements. In each case the non-recursive work involved roughly n comparisons, and

roughly $n - 1$ recursive calls were made. Given this, it isn't surprising that the total number of comparisons for each sort was roughly a constant times n^2 .

When we presented the recursive version of the linear and binary searches, we discovered that making recursive calls on half as many elements was much more efficient than making recursive calls with only one less element. We can use this idea to create more efficient sorts as well. We will design an algorithm so that successive recursive calls will be on half as many elements as the previous call. Of course, unlike the case of binary search where we essentially threw away half the elements after each comparison, we must sort all of the elements of the list. However, a divide-and-conquer strategy like this will reduce the depth of the recursion from n layers of calls to $\log_2 n$.

The *merge sort* will be similar to the recursive insertion sort in that we make recursive calls and then do work to fix up things at the end. The general idea is as follows:

- If there are fewer than 2 elements to sort, you are done.
- Otherwise recursively (merge) sort the elements in the first half of the list and then sort the elements in the second half of the list.
- Merge the elements in the sorted first half with the elements in the sorted second half.

With the insertion sort, we were able to insert the last element after having sorted all but the last element. For the merge sort, because we have broken the list into two pieces, we must merge the two sorted sublists. For example, suppose we have two sorted sublists held in the same list:

```
index:          1 2 3 4 5      6 7 8 9 10
elts.at (index): 15 38 48 51 62    7 12 23 49 88
```

We will merge them into a new list, sorted. We start by comparing the first elements of each half of elts, elts.at (1) and elts.at (5). The smallest, elts.at (5), is assigned to sorted.at (0).

```
index:          1 2 3 4 5 6 7 8 9 10
sorted.at (index): 7 - - - - - - - - -
```

Below we indicate the elements of elts that have been moved to sorted by replacing them with dashes:

```
index:          1 2 3 4 5      6 7 8 9 10
elts.at (index): 15 38 48 51 62    - 12 23 49 88
```

We now compare the leftmost remaining elements of the halves of elts, elts.at (1) and elts.at (7), determining that elts.at (7) is the smallest. We move that into the next slot in sorted:

```
index:          1 2 3 4 5 6 7 8 9 10
sorted.at (index): 7 12 - - - - - - - -
```

The original list is now:

```
index:          1 2 3 4 5      6 7 8 9 10
elts.at (index): 15 38 48 51 62    - - 23 49 88
```

Comparing the remaining leftmost elements of elts, we determine that elts.at (1) should be moved next:

```
index:          1 2 3 4 5 6 7 8 9 10
sorted.at (index): 7 12 15 - - - - - - -
```

The original list is now:

```

// POST: elts of list are rearranged in non-decreasing order
method mergeSort (numList: List[[Number]]) -> Done {
  // create list full of 0's
  def copy: List[[Number]] = emptyList[[Number]]
  mSortHelper (numList) from (1) to (numList.size) using (copy)
}

// PRE: left and right are valid indices of list with left <= right + 1.
// POST: elts of list[left..right] are rearranged in non-decreasing order.
method mSortHelper (numList: List[[Number]]) from (left: Number) to (right: Number)
  using (copy: List[[Number]]) -> Done {
  if (left < right) then {
    def middle: Number = ((left + right) / 2).truncated
    mSortHelper (numList) from (left) to (middle) using (copy)
    mSortHelper (numList) from (middle + 1) to (right) using (copy)
    mergeInto (numList) from (left) dividingAt (middle) to (right) source (copy)
  }
}

```

Figure 19.9: Recursive merge sort

```

index:          1  2  3  4  5      6  7  8  9 10
elts.at (index): - 38 48 51 62    -  - 23 49 88

```

We leave it to the reader to finish this example of merging two lists. Notice that once one of the halves has been completed, the remaining elements of the other half must be moved into the remaining slots in sorted.

The Grace code for the recursive merge sort can be found in Figure 19.9, and is essentially the same as that explained above. As before, the initial call to `mSortHelper` will be made by the method `mergeSort`, which adds parameters to indicate the beginning and end of the range of elements to be sorted, as well as a “helper” list that will be used when merging the two sorted lists together.

The merge method is presented in Figure 19.10. First all of the elements from `numList` are moved into `copy` in preparation for merging them together and placing them in the correct order in `numList`. The indices `left` and `right` keep track of the subscripts of the smallest unmerged elements in the left and right halves of the list. At each step in the first while loop the smallest of the two elements at `left` and `right` is moved into `numList.at (target)`. After the element has been moved, the index of the element that was moved is increased by 1 and the `target` index is also increased by 1.

This continues until all of the elements in one of the two halves has been used up. At that point, all of the remaining elements in the other half must be copied from `copy` into `numList`. Because one of the halves has already been exhausted, only one of the last two while loops’ bodies is actually executed. Whichever is executed simply copies the remaining elements from that half to `numList`.

Exercise 19.3.10 *Assuming that `mergeInto` does what it is supposed to, apply the rules presented in Section 11.2 to show that when the merge sort terminates, all of the elements will be in order. Reason first about the correctness of `mSortHelper` and use that to show `mergeSort` is*


```

// PRE: 0 <= left <= middle <= right < numList.size, and numList[left..middle]
//       and numList[middle+1..right] are both in non-descending order.
// POST: The two halves (numList[left..middle] and numList[middle+1..right])
//       have been merged together so that numList[left..right] is now in
//       non-descending order.
method mergeInto (numList: List[[Number]]) from (left: Number) dividingAt (middle: Number)
    to (right: Number) source (copy: List[[Number]]) -> Done {
  var indexLeft: Number := left
  var indexRight: Number := middle + 1
  var target: Number := left

  for (left .. right) do {i: Number ->
    copy.at (i) put (numList.at (i))
  }

  while {(indexLeft <= middle) && (indexRight <= right)} do {
    if (copy.at (indexLeft) < copy.at (indexRight)) then {
      numList.at (target) put (copy.at (indexLeft))
      indexLeft := indexLeft + 1
    } else {
      numList.at (target) put (copy.at (indexRight))
      indexRight := indexRight + 1
    }
    target := target + 1
  }

  while {indexLeft <= middle} do {
    numList.at (target) put (copy.at (indexLeft))
    indexLeft := indexLeft + 1
    target := target + 1
  }

  while {indexRight <= right} do {
    numList.at (target) put (copy.at (indexRight))
    indexRight := indexRight + 1
    target := target + 1
  }
}

```

Figure 19.10: MergeInto method used in merge sort

correct.

Exercise 19.3.11 *As noted in the text, only one of the last two while loop bodies is actually executed. Why is there not a conditional statement that selects which of the two while loops should be executed?*

Exercise 19.3.12 *The mergeInto method is traditionally presented as an iterative method, as it was here. However, we could write it recursively. Write a recursive mergeInto method. As usual, the new mergeInto method should call the recursive method, which will need to be provided with extra parameters corresponding to indexLeft and indexRight.*

Hint: The initial for loop and the final two while loops should be done in the main mergeInto method, not the recursive one.

19.3.6 Complexity of Merge Sort

Let's analyze the complexity of the merge sort algorithm by counting the number of comparisons of elements in the list. Notice that no comparisons of elements are performed in either the sort or mergeSort method. All comparisons are performed in the merge method.

During the mergeInto method, every time we compare two elements of copy, we place one of them in the next slot in numList. When we have exhausted all of the elements in one half of the list, the remaining elements in the other half are simply moved into numList without any further comparisons being necessary. Thus merging the two halves of a list with a total of k elements takes at most $k - 1$ comparisons.

Let $T(n)$ be the function that will tell us how many comparisons must be made to perform a merge sort on a list of size n in the worst case. For example, $T(0) = T(1) = 0$, because no comparisons are necessary to sort a list of size ≤ 1 . $T(2) = 1$ because only one comparison is necessary to merge sort a list of size 2.

Because merge sort is so regular, it is possible to write down a formula relating the number of comparisons to merge sort a list of size n with the number of comparisons to merge sort smaller lists. Suppose n is even. Then to merge sort a list of size n , we must merge sort the two halves, which have size $n/2$, and then merge the two halves together. Thus,

$$T(n) \leq T(n/2) + T(n/2) + (n - 1) = 2 * T(n/2) + (n - 1)$$

The inequality arises because $n - 1$ is an upper bound on the number of comparisons to merge together the two halves of the list of size n . Because we are only computing upper bounds anyway, we can simplify the above formula to

$$T'(n) = T'(n/2) + T'(n/2) + n = 2 * T'(n/2) + n$$

by replacing the $n - 1$ by the even larger n and replacing the inequality by “=”.

$T'(n)$ thus represents an upper bound on the number of comparisons needed to merge sort a list of size n . As before, we will set $T'(0) = T'(1) = 0$.

Table 19.2 contains the values of $T'(n)$ for values of n that are powers of 2. Notice that the values calculated on the right hand side can be rewritten in the form $n * (\log_2 n)$, so it seems reasonable to hypothesize that

$$T'(n) = n * (\log_2 n)$$

n	$T'(n)$
$1 = 2^0$	$0 = 1 * 0$
$2 = 2^1$	$2 * T'(1) + 2 = 2 = 2 * 1$
$4 = 2^2$	$2 * T'(2) + 4 = 8 = 4 * 2$
$8 = 2^3$	$2 * T'(4) + 8 = 24 = 8 * 3$
$16 = 2^4$	$2 * T'(8) + 16 = 64 = 16 * 4$
$32 = 2^5$	$2 * T'(16) + 32 = 160 = 32 * 5$
...	...
$n = 2^k$	$2 * T'(n/2) + n = n * k = n * (\log_2 n)$

Table 19.2: Table of values of $T'(n)$: measuring the complexity of merge sort

for n a power of 2. We could prove this using mathematical induction, but we omit that proof here. Instead we simply verify that with this formula, $T'(0) = T'(1) = 0$, as required, and that, if n is even,

$$\begin{aligned}
 2 * T'(n/2) + n &= 2 * (n/2) * (\log_2 n/2) + n \\
 &= n * ((\log_2 n) - (\log_2 2)) + n \\
 &= n * ((\log_2 n) - 1) + n \\
 &= n * (\log_2 n) \\
 &= T'(n)
 \end{aligned}$$

With some extra work, we could also show that $n * (\log_2 n)$ is an upper bound on the number of comparisons in merge sorting an list of size n for all $n > 0$, including those that are not powers of 2.

Exercise 19.3.13 *In the binary search, the $\log_2 n$ in the count of the total number of comparisons made arose because that is the number of times that a list of size n can be split in half. Give an intuitive explanation of why the $\log_2 n$ factor appears in the complexity of merge sort.*

Exercise 19.3.14 *We can write a similar iterative merge sort program if we come at it in a very different way. Here is a sketch of how to proceed:*

- a. *Interchange the first and second elements if they are out of order, similarly with the third and fourth, fifth and sixth, etc.*
- b. *The first pair of elements are in order as are the second pair. Merge those together so the first four elements are now in order. Do the same with the third and fourth pairs of elements, etc.*
- c. *The first four elements are in order, as is the second quadruple. Merge those into an ordered list of eight elements. Continue merging other quadruples in pairs.*
- d. *Continue merging successively longer groups until all of the elements are in order.*

One challenge in writing a method to do this is that you cannot easily merge runs of elements in place. Instead, you must merge them into another list. Use a list like `copy` in the recursive merge sort to merge runs of elements. You can then interchange the roles of the original list and `copy` when you have finished a pass through the entire list.

n	<i>Selection</i>	<i>Insertion</i>	<i>Merge</i>
20	190	116	64
40	780	416	166
100	4,950	2,565	542
1000	499,500	245,132	8709

Table 19.3: Average number of comparisons to sort a list of size n

19.4 Impact of the Complexity of Searching and Sorting

Table 19.3 lists the average number of comparisons necessary to sort a list of size n with the selection, insertion, and merge sorts. The selection sort always take the same number of comparisons, no matter what the initial ordering is of the list. On the other hand, the insertion and merge sorts depend on that initial ordering. As a result, we averaged the number of comparisons for several randomly chosen initial orderings of lists to obtain the numbers for the insertion and merge sorts. The worst-case time for the insertion sort is the same as the selection sort, while the best case time (for a previously sorted list) is essentially the size of the list.

It is clear from the table that the average number of comparisons necessary using the insertion sort is roughly half that for the selection sort. Because selection sort is of roughly n^2 complexity, multiplying the size of the input by k results in multiplying the number of comparisons by k^2 . For example, doubling the size of the input (e.g., from 10 to 20) results in the number of comparisons quadrupling. Similarly a ten-fold increase in the size of the input (e.g., from 100 to 1000) results in a hundred-fold increase in the number of comparisons. Roughly the same increases hold for the insertion sort.

Because the merge sort is an $n * (\log_2 n)$ sort, the number of comparisons grows more slowly than selection and insertion sorts. Thus when the list size doubles from 20 to 40, the number of comparisons goes up by a factor of about 2.5. When the list size increases from 100 to 1000, the number of comparisons goes up by a factor of 15. As a result, the number of comparisons for merge sort grows much more slowly than that for selection and insertion sort. As a result, selection sort on a list of size 1000 takes 50 times as many comparisons as a merge sort on a list of the same size.

Exercise 19.4.1 *A binary search is much faster than linear search, but binary search requires a sorted list.*

- a. *Suppose you are given an unsorted list and an element you want to search for. Does it makes sense to merge sort a list and then do a binary search rather than doing a linear search? Provide the total number of comparisons to do each of these on a list of size n .*
- b. *How many searches must be made on the same list of size n to make the number of comparisons the same with either doing a series of linear searches or doing a single merge sort and then all binary searches?*
- c. *Given this information, why does the phone company sort all of the names in the phone book?*

19.5 Summary

In this chapter we examined linear and binary searches, as well as selection, insertion, and merge sorts. For all algorithms aside from the merge sort we introduced both iterative and recursive algorithms. We introduced the use of pre- and post-conditions to specify the behavior of methods, invariants to prove the correctness of algorithms, and we discussed the complexity of algorithms.

If a list has n elements then roughly n comparisons are necessary to perform a linear search in the worst case, while $n/2$ are necessary on average if the element to be found is actually in the list. Binary searches only work on sorted lists, but result in many fewer comparisons. Roughly $2 * \log_2 n$ comparisons are required on a sorted list of size n .

Both the selection and insertion sorts require roughly $n^2/2$ comparisons on a list of size n in the worst case, while the merge sort requires $n * (\log_2 n)$ comparisons on a list of size n .

19.6 Chapter Review Problems

Exercise 19.6.1 *In Section 19.1.3, we claimed that the average number of comparisons to find the key using linear search was roughly $n/2$ under the assumption that key occurs exactly once in list and that key is equally likely to be in any of the places in the list. A careful calculation of the average time complexity under the above assumptions would require us first to calculate the number of comparisons necessary for each of the n possible positions where key might occur, add these up, and then divide by n to compute the average. Use this to prove that the average number of comparisons to find the key using linear search is $(n + 1)/2$ if the list has n elements.*

Exercise 19.6.2 *Let $elts$ be a list of length 5 with values 89, 23, 15, 93, 7. Hand simulate each of the following sorts on the list $elts$*

- a. *Iterative selection sort.*
- b. *Recursive selection sort.*
- c. *Iterative insertion sort.*
- d. *Recursive insertion sort.*
- e. *Recursive merge sort.*

Exercise 19.6.3 *Suppose that it takes 1 second to perform a selection sort on an list of 1000 elements. Approximately how long will it take to perform a selection sort on a list of 2000 elements? 5000 elements? 1,000,000?*

Chapter 20

Introduction to Object-Oriented Design

By now you're probably fairly comfortable with object and class implementation. That is, given a description of an object or class that includes instance variables, constants, and method headers, you can fill in the necessary details (i.e., you can write the method bodies) to make the class usable. But how would you go about designing a class from scratch? If you were given the description of a problem, how would you determine what objects or classes should be defined? This chapter will provide you with a methodology for analyzing problem descriptions so that you can begin to design classes of your own.

In this chapter we will discuss *object-oriented design*. We begin with a discussion of objects and their properties and show how entities in the real world can be mapped to Grace objects. These objects provide *abstractions* of their real-world counterparts. We then introduce a simple programming problem and take you through the process of designing a program to solve it. Next we make that problem a bit more complex and discuss extensions to our original solution. Finally, we discuss the notion of *encapsulation* and illustrate its importance.

20.1 What is Object-Oriented Design?

The design methodology that we introduce here is called *object-oriented design*. An important first step in object-oriented design is to recognize that very often our software is mimicking the real world. We can take inspiration from the real world to help us decompose a problem into objects. We can analyze the real-world objects to identify their properties and behavior, and we can then model those in our software. For example, recall the Timer object from Section 6.4. The timer object, `stopWatch`, behaved very much like a (simple) real stopwatch. Similarly, the `FallingBall` objects from Chapter 9 behaved very much like falling balls.

An object is an entity that provides an *abstraction* in our program. We call it an abstraction because we can use an object without knowing how it is implemented. That is, we abstract away the details and understand the object at a higher level. This follows naturally from the way we use objects in the real world. For example, you can use a real stopwatch (or a telephone, computer, car, etc.) without understanding exactly how it works.

A goal in object-oriented design is to first define a specification of what an object is capable of – given by its type specification – and then provide object and class definitions that implement this specification. Importantly, a well-designed type specification and implementing

class will allow you (or another programmer) to construct and use objects without having to know precisely how they work.

20.2 Properties and Behaviors

As we just noted, entities in the real world often serve as the inspiration for objects in our programs. By analyzing real-world objects and carefully considering their properties and behaviors, we identify the specific aspects that we need to model in our programs. To illustrate this idea, we will take you through the process of designing the `Timer` class from Section 6.4.

20.2.1 Modeling Properties of Objects

We begin by considering the properties of the entity that we would like to model – in this case, a stopwatch. Typically a stopwatch has

- buttons that control various types of functionality, such as resetting, timing, stopping, and so on;
- a display area;
- internal memory to store the timing information to be displayed.

We can think of the properties of an object as the *nouns* and *adjectives* that describe it. That is, the properties include the various components of the object as well as its physical appearance.

Next we need to consider which of those properties really need to be modeled. We can go about this by considering the ways in which the object will be used. In the program from Section 6.4, a `Timer` is simply used to tell us whether we have clicked on a target object quickly enough. The stopwatch itself is never actually displayed. As a result, we need not worry about modeling a stopwatch’s physical appearance. Instead, there’s just one property of interest to us – the stopwatch’s internal memory.

While a program provides a model of the real world, we always need to make decisions about how realistic our model should be. While we typically strive for some degree of realism, there is no need to model properties that are irrelevant to our application.

Once we have decided which properties to model, we need to decide how to model them. We model properties with instance variables and constants. Since the only information that we need to remember in the stopwatch’s memory, elapsed time, is calculated by finding the difference between the current time (which is easily accessible) and some fixed start time, we only need to model the start time. So we declare a single instance variable, `startTime` to be of type `Number`.

20.2.2 Modeling Behavior

In addition to having various properties, objects also have various behaviors. That is, they have certain types of functionality. For example, stopwatches can

- reset to zero
- report elapsed time
- stop
- resume timing after being stopped

We can think of behaviors as the *verbs* that describe an entity's actions.

As with properties, we will want to model some of these behaviors, but not necessarily all of them. In our program, there is no need for the stopwatch to suspend timing or to resume from a suspended state. All we need to be able to do is report elapsed time and reset.

How do we model behavior? We define a method for each type of behavior that we want to allow. Therefore, we define one method to reset and another to report elapsed time. We call them `reset` and `elapsedMilliseconds`, respectively. We also define another method, `elapsedSeconds` to report elapsed time in seconds.

20.3 Design Fundamentals

Now that you have some basic understanding of the fundamental idea of modeling real-world objects by considering their properties and behaviors, let's step through a complete design process. In this section, we will aim to design and later implement a simple version of the shell game. The game works as follows: three cups are placed (upside-down) on a table. Player 1 places a marble under one of the cups, letting Player 2 see where it has been placed. The first player then shuffles the cups, while the second player watches. Player 2 then has to guess where the marble is.

Our simple program will work very much like this. We will assume there are two players. The game will begin with three cups drawn in a window, along with one marble, as shown in Figure 20.1. Player 1 will place the marble into one of the cups by dragging the marble on top of a cup. When the mouse is released, the marble has been dropped into the cup, so it is no longer visible. Player 1 then shuffles the cups by dragging them around the screen. When Player 1 stops moving the cups, Player 2 will guess which cup contains the marble. Clicking on a cup will result in the cup being raised and the marble being shown, if it's there.

20.3.1 Design Step 1: Identify the objects to be modeled

Given this specification of the desired program behavior, let's think about the classes that will be needed in our implementation. As discussed above, object-oriented design seeks to model the world. So the entities described in the problem definition help to give us clues about the types of objects needed in the program. The description above mentions three types of objects: players, cups, and a marble. We do not need to model the players, since those will be the users of our software (though we'll certainly want to provide them with an interface). However, we do need to model the cups and the marble.

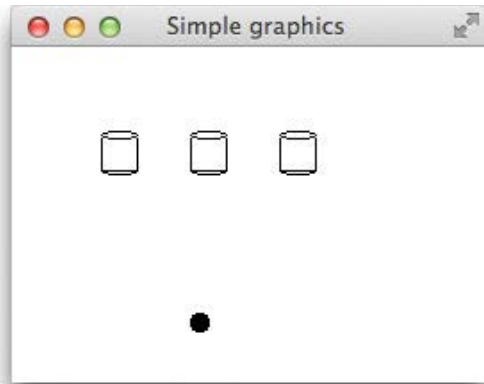


Figure 20.1: User interface for a simple Shell Game

20.3.2 Design Step 2: List the properties and behaviors of each type of object

Next we need to list the properties and behaviors of each type of object we identified. We'll begin by analyzing the properties and behaviors that our cups need to have. Each cup has two important properties:

- It has an image on the screen that represents it.
- It is either empty or it contains a marble.

Of course, cups in the real world have these properties and more, but we only need to model the properties that are necessary for our program.

Each cup also needs to behave in specific ways:

- A cup can be moved.
- A cup can have a marble placed in it.
- A cup can be raised to reveal its contents, if any.

An outline for the `cup` class is shown in Figure 20.2. The first block lists the properties to be modeled; the second block lists the behaviors.

We'll eventually define a `cup` class, modeling the cup properties with instance variables and the cup behaviors with methods. Before we do so, however, let's continue our analysis by considering the marble. The marble has the following properties:

- It has an image on the screen that represents it.
- It is either in a cup or it is not.

Let's think about the second of these properties. There is certainly a relationship between the cups and the marble. A cup can contain the marble and the marble can be contained in a cup. But do we really need to model this both from the cup's perspective and from

- List of cup properties
 - Graphical image of the cup
 - Whether the cup contains the marble
- List of cup behaviors
 - Move the cup
 - Place a marble in the cup
 - Reveal the contents of the cup; removes marble from cup, if there

Figure 20.2: First level of design for a cup class: properties and behaviors

- List of marble properties
 - Graphical image of the marble
- List of marble behaviors
 - Move the marble
 - Place the marble in a cup
 - Remove the marble from a cup

Figure 20.3: First level of design for a Marble class: properties and behaviors

the marble’s perspective? It turns out that we do not need to. We will choose to model containment from the cup’s perspective. That is, we will leave our design for the `cup` class as in Figure 20.2. The only property of the marble that we will model is the image of it.¹

Now let’s consider the ways in which our marble must behave:

- A marble can be moved.
- A marble can be dropped into a cup.
- A marble can be removed from a cup.

As with the cup, we’ll want a class that will allow us to model the marble properties and behaviors. An outline of the `Marble` class is given in Figure 20.3. As with the cup before, let’s specify the type for the marble. See Figure 20.4.

Before we go any further in filling in the details about cups and marbles, we need to consider one more object. We need to provide a user interface to allow our two players to play the shell game. We will define an object inheriting `graphicApplicationSize` that will set up the program and that will allow the players to interact with it. The object should allow a player to drag a cup or a marble. It should also allow a player to click on a cup to release

¹The reason for making this choice is that it seems more natural to us to think of cups as vessels that can either contain something or not.

```

type Marble = {
  // Move the marble and its contents by dx @ dy
  moveBy (dx: Number, dy: Number) → Done

  // Drap this marble in the cup, hiding it
  addTo (cp: Cup) → Done

  // If this marble is contained in a cup, remove it from the cup
  removeFromCup → Done
}

```

Figure 20.4: The first pass at a type for a marble

- Properties that describe the shell game
 - three cups
 - a marble
 - the current position of the mouse
 - whether a marble or cup has been selected
 - which object has been selected
- Allowable game behaviors
 - Place three cups and a marble on the canvas
 - Move a cup or marble
 - Show contents of a cup

Figure 20.5: First level of design for a shellGame object: properties and behaviors

its contents. Like cups and marbles, games have descriptive properties. If we think of the controller class as the game itself, we notice that it has an appearance – three cups and a marble on a canvas. In addition, the state of the game at any given time depends upon the position of the mouse and whether a player has currently selected a cup or the marble with the mouse.

The outline for this object, which needs to inherit `graphicApplicationSize` is given in Figure 20.5.

20.3.3 Design Step 3: Model properties with instance variables

Now we can begin to refine our design. We start by thinking about the instance variables that will allow us to model the properties of each type of object. For a cup, we decided above that we wanted to have an image that would represent it in a window. If we intend to make the cups look like those in Figure 20.1, then we need several variables to represent the various components of the image. We will represent the outline of a cup with a `FramedRect` and two `FramedOvals`, one for the top and one for the bottom. The white interior will be

```

// A class to represent a cup used in the Shell Game
class cup(...) -> Cup {

  // List of cup properties

  // Graphical image of the cup
  def cupBottom: Graphic2D = ...
  def bottomFrame: Graphic2D = ...
  def cupSide: Graphic2D = ...
  def sideFrame: Graphic2D = ...
  def cupTop: Graphic2D = ...
  def topFrame: Graphic2D = ...

  // Whether the cup holds the marble
  var containsMarble: Boolean := ...

  // List of cup behaviors
  // Move the cup
  // Place a marble in the cup
  // Reveal the contents of the cup; removes marble from cup, if there
}

```

Figure 20.6: Second level of design for a Cup class: modeling properties with instance variables

represented by a `FilledRect` and two `FilledOvals`. We will name the instance variables `sideFrame`, `topFrame`, `bottomFrame`, `cupSide`, `cupTop`, and `cupBottom`, respectively. In addition to providing the right appearance for a cup, each of these graphical objects can be moved, which will allow us to implement the necessary functionality for a cup.

Exercise 20.3.1 *Why do we need all six instance variables to draw a cup? In particular, why do we need the three filled shapes?*

Next, we want to model whether a cup is empty or not. We can do this by declaring a variable, `containsMarble`, of type `boolean`. If the cup contains the marble, `containsMarble` will be `true`; it will be `false` otherwise.

Through this process, we refine our initial outline of the `cup` class as in Figure 20.6. We don't yet need to worry about initializing the variables, though that will be important before we can start testing..

Now we go through the same process for the `Marble` class. The one property of a marble that we want to model is its appearance on the canvas. Since a marble is simply a filled circle, we will declare a variable, `theMarble`, to be of type `FilledOval`, as in Figure 20.7.

Next we turn to the `shellGame` object. As shown in Figure 20.8, we declare three variables of type `Cup` and one of type `Marble` to keep track of the objects on the canvas.

We know that mouse positions are described by `Locations`, but when we think about the mouse position a bit more, we realize that we don't actually need to explicitly keep track of the current mouse position. The event-handling methods such as `onMousePress` and `onMouseDrag`, which will be essential for manipulating the objects on the canvas, already give us the current mouse location. We recall, however, that in order to drag an object on the screen, it is useful

```

// A class to represent a marble
class marble (...) -> Marble {

//Marble properties
// Graphical image of the marble
def marbleImage: Graphic2D = ...

//Marble behaviors
// Move the marble
// Place the marble in a cup
// Remove the marble from a cup
}

```

Figure 20.7: Second level of design for a Marble class: modeling properties with instance variables

to know the previous mouse location. So we declare a variable, `lastMousePos` of type `Location` and change the comment to reflect our decision. This illustrates an important part of the design process. While there is a well-defined set of steps to guide the process of design so that the decisions made at each step are logical, we still need to be open to the idea of making modifications to earlier decisions.

Next, we need to know which object, if any, has been selected by the player. Note that we need to keep track of a few different pieces of information here. First, we need to know whether an object has been selected. Second, we need to know which one, if any. There are many possibilities here. In order to make our design as simple as possible, we will define three variables. The first, the boolean variable `marbleSelected` tells us whether the marble has been selected. Since there is only one marble, if the value of this variable is true, then we immediately know which object is selected. The second variable, `cupSelected`, indicates whether one of the cups has been selected. If so, the third variable, `selectedCup`, tells us which cup has been selected. If `cupSelected` is false, then the meaning of `selectedCup` is irrelevant.

20.3.4 Design Step 4: Model behaviors with methods

Next we turn to the behaviors that we need to model for each class of objects in our program. Remember that we will be modeling behaviors with methods. In this part of the design process, however, we won't concern ourselves with the details of those methods. Instead, we will focus on the method headers (or *signatures*), since they describe the interface between the various entities in our program.

Beginning with the `cup` class, we see that we need three methods. The first should move a cup. Since we want the caller of this method to be able to specify the direction and distance of the movement, this method will take two parameters, a displacement in the x direction and a displacement in the y direction.

We also need a method that will drop a marble into the cup. Since a marble is being dropped into the cup, we should take that as a parameter. The next method should remove the marble that is in the cup, if any. There is no need for any parameters to this method. However, this method does point out that we are missing an instance variable. If the cup contains a marble, then that marble needs to become visible to the players of the game. In

```

// Object definition for a simple Shell Game
def shellGame: GraphicApplication = object {
  inherit graphicApplicationSize(500 @ 500)

  // Properties that describe the shell game
  // Cups for hiding the marble
  def cup1: Cup = ...
  def cup2: Cup = ...
  def cup3: Cup = ...

  // the marble to be hidden
  def theMarble: Graphic2D = ...

  // Has the marble been selected for dragging
  var marbleSelected: Boolean := ...

  // Has the cup been selected for dragging
  var cupSelected: Boolean := ...

  // Cup selected. Meaningful only if cupSelected is true
  var selectedCup: Cup

  // the last position of the mouse}
  var lastMousePosition: Point

  // Allowable game behaviors
  // Place three cups and a marble on the canvas
  // Move a cup or marble
  // Show contents of a cup
}

```

Figure 20.8: Second level of design for a ShellGame class: modeling properties with instance variables

```

// A class to represent a cup used in the Shell Game}
class cupAt (upperLeft: Point) size (sz: Point) on (canvas: DrawingCanvas) → Cup {

  // List of cup properties

  // Graphical image of the cup
  def cupBottom: Graphic2D = ...
  def bottomFrame: Graphic2D = ...
  def cupSide: Graphic2D = ...
  def sideFrame: Graphic2D = ...
  def cupTop: Graphic2D = ...
  def topFrame: Graphic2D = ...

  // Whether the cup holds the marble
  var containsMarble: Boolean := ...

  // If containsMarble is true, holds that marble
  var theMarble: Marble

  // Move the cup and its contents by dx @ dy
  method moveBy (dx: Number, dy: Number) → Done

  // Drap the marble in the cup, hiding it
  method dropln (ball: Marble) → Done

  // Lift the cup up, and if it holds a marble, show it
  method showContents → Done
\}

```

Figure 20.9: Third level of design for a Cup class: modeling behaviors with methods

order to empty itself of a marble, the cup needs to keep track of the marble that was placed into it in the method `dropln`. So we add an instance variable, `theMarble` of type `Marble`. All of our modifications to the `cup` class are shown in Figure 20.9.

We still need to add a lot of details to the class; for example, we need to initialize instance variables and provide the method bodies. However, we do now have enough information to specify the public interface or type that corresponds to these objects. Our first cut at this is now shown in Figure 20.10. Because this is an iterative process, we may discover later that we need more methods in the type. However, any new methods are easily added when needed.

At this point we have just about completed our abstraction of a cup. Before going on, however, we need to consider what external information is needed in order to create the object. These will end up as the formal parameters for the class.

When a cup is created, it will appear on the canvas, so we have to think about the information that will be necessary for placing it on the screen. We will need to know a cup's size; we also need to know the canvas on which it will be placed, as well as the location on that canvas. The canvas and location need to be passed to the class as parameters. But what about the size? In the shell game, every cup should look the same. Otherwise, it's


```

type Cup = {
  // Move the cup and its contents by dx @ dy
  moveBy (dx: Number, dy: Number) → Done

  // Drap the marble in the cup, hiding it
  dropln (ball: Marble) → Done

  // Lift the cup up, and if it holds a marble, show it
  showContents → Done
}

```

Figure 20.10: The first pass at a type for a cup

too easy to guess which one contains the marble. So in this case, we have a choice. We can define constants in the `cup` class that specify the size of a cup. Or we can get this information through parameters to the constructor. If we do the latter, then the class that sets up the shell game will be able to control the visual appearance of the game by specifying not only where the cups should be, but how large they should be. There are reasonable arguments for making either choice, but let's pick the one that allows the width and height to be specified as parameters. The formal parameters for the class are included in Figure 20.9.

We now go through a similar exercise for the `marble` class. Earlier we determined that we wanted to model three behaviors for a marble, which means that we need three methods. The first needs to give the marble the ability to move. As with a cup (or any graphical object you've seen in this text), we will allow the caller to specify the x and y displacements for the move.

A second method needs to place the marble into a cup. This should cause us to think a bit more carefully, since we already determined that placing a marble into a cup should be a behavior provided by cups. Do we really want both marbles and cups to be responsible for the action of placing a marble in a cup? In this case, the answer is no. We'll leave this responsibility to the cup. In a sense, we will think of the marble as becoming part of the cup, once it's in there. (This makes sense, since interactions with a cup – for example, dragging it – affect the marble contained within it.)

Our modifications to the `Marble` class are shown in Figure 20.11. At this point the design is incomplete, as we have not yet provided formal parameters for the class, but it gives us enough information to see that a `Marble` is nothing more than a `FilledOval` that can move. As a result, we determine that this class really isn't necessary after all! We can simply use a filled oval throughout our program, rather than defining a specialized `marble` class. There is some risk in taking this approach. Defining a separate `marble` class would make it easier to add specialized behavior to our marble in some future, more sophisticated version of the program. In this case, however, giving up this potential benefit of abstraction seems justified to keep our design a bit simpler.

Next we consider the object controlling the game. We will initialize the setup in this object. We will control the rest of the game by handling mouse events. It is easy to see that the `onMouseClicked` method should cause the contents of a cup to be displayed, if the cup is not empty. Selecting and moving cups and marbles will be a little harder though, since it will

```

// A class to represent a marble
class marble (...) -> Marble {

    // Graphical image of the marble
    def theMarble: Graphic2D

    // Move the marble
    // double dx, dy - distance to move in the vertical and horizontal directions}
    method move(dx: Number, dy: Number) -> Done
}

```

Figure 20.11: Third level of design for a Marble class: modeling behaviors with methods

involve more than a single mouse event-handling method. In the `onMousePress` method, we will determine which object, if any, has been selected. In `onMouseDown`, we will move the selected object. If a marble is dragged to a cup, then in `onMouseRelease` we need to drop the marble into that cup. These new design details are given in Figure 20.12.

You may have noticed that our analysis of the controller class has pointed out a deficiency in the `cup` class. In the controller we need to determine which, if any, object has been selected during a mouse press. So a `cup` object needs to tell us whether it contains a given mouse location. Therefore, we add a `contains` method to that class and its type, which will have the following specification:

```

// Returns true if the cup contains the given mouse location, mouseLoc;
// returns false otherwise
method contains (pt: Point) -> Boolean

```

20.3.5 Design: the Process of Refinement

Before moving on to the implementation of our design, it is useful to reflect on the process of design. Designing software is a process of *refinement*. We never begin with implementation details. We always start with the highest level of abstraction. That is, we begin by considering the objects in the program in their most general sense, abstracting away the details of implementation. At each step in the design process, we *fill in more and more of the details*. That is, we make our plan less and less abstract.

There is a second sense in which refinement occurs during the process of design. In the last section you saw that in filling out the details, we decided to make changes to our design. In working out the details, you will often find yourself *refining and changing earlier design decisions*. As you gain practice with design, you will likely find that you can develop the design of a small program, like our shell game, without having to make significant changes along the way. But even the most experienced software designers refine their original ideas when working on large projects.

20.3.6 Filling in the Details

We can now begin to fill in the implementation details for our shell game program. Basically, this phase involves initializing instance variables and constants as well as filling in the bodies

```

// Game to display three cups, hide a marble under one, and drag the cups
def shellGame: GraphicApplication = object {
  inherit graphicApplicationSize(500 @ 500)

  // Cups for hiding the marble
  def cup1: Cup = ...
  def cup2: Cup = ...
  def cup3: Cup = ...

  // the marble to be hidden
  def theMarble: Graphic2D = ...

  // Has the marble been selected for dragging
  var marbleSelected: Boolean := false

  // Has the cup been selected for dragging
  var cupSelected: Boolean := false

  // Cup selected. Meaningful only if cupSelected is true
  var selectedCup: Cup

  var lastMousePosition: Point

  // Show contents of cup clicked on
  method onMouseClick (mousePos: Point) → Done {
    ...
  }

  // When mouse is pressed, determine which object is selected
  method onMousePress (mousePos: Point) → Done {
    ...
  }

  // Move the selected cup or marble by the amount the mouse has moved
  method onMouseDrag (mousePos: Point) → Done {
    ...
  }

  // Dragging marble when release mouse over a cup
  // put the marble in the cup
  method onMouseRelease (mousePos: Point) → Done {
    ...
  }

  startGraphics
}

```

Figure 20.12: Third level of design for a ShellGame class: modeling behaviors with methods

of the methods specified in our design. As we will see in this section, if a method is very simple, we can write the code for the method body right away. In some cases, however, it is useful to outline the method body before plunging into the process of writing actual Grace statements.

We'll begin, as before, with the `cup` class. First, we consider the initialization of the cup. Creating a cup that looks like the image in Figure 20.1 is not terribly difficult, but it does require a bit of thought. We need to construct the base first, then the side, and finally the top. We also need to determine the size of the top and bottom ovals, relative to the size of the cup itself. We'll say that the top and bottom will be one tenth the size of the cup and will define a constant `heightRatio` that represents the ratio of the full cup height to its top and bottom pieces, in this case, 10. You are welcome to think through the details of how all the pieces fit together, but this is not necessary for our purposes.

In addition to constructing the pieces of the cup, we need to remember to initialize `containsMarble` to `false`, to indicate that the cup is empty initially. All constants must be initialized, as well as most variables. Some variables can have values assigned later, as long as you are careful not to access them before they are initialized. The variable, `theMarble`, is an example of this as it only needs to have a value when `containsMarble` is true. The complete `Cup` initialization is shown in Figure 20.13.

Exercise 20.3.2 *Why do we need to initialize the components of the cup in the order specified? What happens if we construct the top first?*

In addition to initializing definitions and instance variables, there are four methods that we need to provide. The `moveBy` method is straightforward, as illustrated in Figure 20.14. Moving a cup simply involves moving the various graphical objects that together make up a cup. Recall from Chapter 6 that this notion of passing the work on to other objects is called *delegation*. In addition, we move the contents of the cup along with the cup. The `contains` method is similarly straightforward. We simply return whether the given location is contained in one of the graphical objects that represent the cup.

Exercise 20.3.3 *The contains method in the Cup class only checks for containment in the framed components. Why don't we need to check all the pieces of the cup?*

The other two methods require a bit more thought. In the `dropln` method there are several things we need to take care of. First, we need to note that the cup now contains a marble, i.e., that it is no longer empty. Visually, we also have some work to do. Once a marble is dropped into a cup, it should no longer be visible on the canvas as a separate entity. We also need to remember the marble for later, because it is, in essence, now part of the cup. Rather than immediately writing out the Grace code, we write these ideas as comments, as shown in Figure 20.14. Once we have done this, the process of translating into Grace should be fairly easy. The method becomes:

```
// Drop the marble in the cup, hiding it
// ball - marble to be dropped in}
method dropln (ball: Marble) → Done {
  // remember that the cup contains a marble
  containsMarble := true
```

```

class cupAt(upperLeft: Point) size (sz: Point) on (canvas: DrawingCanvas) → Cup {
  // constants used to display the cup
  def width: Number = sz.x
  def height: Number = sz.y
  def heightRatio: Number = 10
  def yOffset: Number = sz.y/heightRatio
  def otherYOffset: Number = height/heightRatio
  def openY: Number = height

  // complicated pieces of cup image
  def sideFrame: Graphic2D = framedRectAt (upperLeft) size (sz) on (canvas)

  def cupBottom: Graphic2D = filledOvalAt(upperLeft + (0 @ (height-otherYOffset)))
    size (sz.x@(2*otherYOffset)) on (canvas)
  cupBottom.color := colorGen.white

  def bottomFrame: Graphic2D = framedOvalAt (upperLeft + (0 @ (height - otherYOffset)))
    size (sz.x @ (2*otherYOffset)) on (canvas)

  def cupSide: Graphic2D = filledRectAt (upperLeft) size (sz) on (canvas)
  cupSide.color := colorGen.white

  def cupTop: Graphic2D = filledOvalAt(upperLeft - (0 @ yOffset)) size (sz) on (canvas)
  cupTop.color := colorGen.white

  def topFrame: Graphic2D = framedOvalAt (upperLeft - (0 @ otherYOffset))
    size (sz.x @ (2*otherYOffset)) on (canvas)

  // Whether the cup holds the marble
  var containsMarble: Boolean := false

  // If containsMarble is true, holds that marble
  var theMarble: Marble

```

Figure 20.13: Filling in the details for a Cup class: constants and instance variables

```

// Move the cup and its contents by dx @ dy
method moveBy (dx: Number, dy: Number) → Done {
  cupSide.moveBy (dx, dy)
  cupTop.moveBy (dx, dy)
  cupBottom.moveBy (dx, dy)
  sideFrame.moveBy (dx, dy)
  topFrame.moveBy (dx, dy)
  bottomFrame.moveBy (dx, dy)

  if (containsMarble) then {
    theMarble.moveBy (dx, dy)
  }
}

// Drop the marble in the cup, hiding it
method dropln (ball: Marble) → Done {
  containsMarble := true
  theMarble := ball
  theMarble.visible := false
}

// Lift the cup up, and if it holds a marble, show it
method showContents → Done {
  if (containsMarble) then {
    theMarble.visible := true
    containsMarble := false
  }
  moveBy(0, -openY)
}

// Returns true if the cup contains the given mouse location, pt;
// returns false otherwise
method contains (pt: Point) → Boolean {
  cupSide.contains (pt) || cupTop.contains (pt) || cupBottom.contains (pt)
}
}

```

Figure 20.14: Filling in the details for a Cup class: methods

```

    // save the marble for later
    theMarble := ball
    // make the marble disappear in the cup
    theMarble.visible := false
}

```

As you can see, this process of *iterative refinement* helps to make design and implementation simpler, by allowing us to look at a method in broad terms before worrying about details. But it also has a very convenient side effect. The ideas that we write as comments can serve as the documentation for our method. By the time we write our method, our comments are already there!

It is worthwhile to go through the same exercise for the `showContents` method. In this method we have to show the contents of the cup, if it is not empty. We also need to make note of the fact that the cup no longer contains anything. Once we fill in the details, the method looks like this:

```

// Lift the cup up, and if it holds a marble, show it
method showContents → Done {
  if (containsMarble) then { // if the cup actually holds the marble
    // show the marble
    theMarble.visible := true
    // note cup no longer contains a marble
    containsMarble := false
  }
  // lift the cup
  moveBy(0, -openY)
}

```

Note that we have added the constant `openY` that specifies the distance to lift the cup.

Exercise 20.3.4 *In our implementation of the method `showContents` we first removed the marble from the cup and then lifted the cup. Why didn't we lift the cup before removing its contents?*

Turning to the object controlling the game, `shellGame`, we proceed similarly by including initialization code and filling in the bodies of the methods. A first level of such details is shown in Figures 20.15 and 20.16. Notice that we have added some new constants. e.g., `marbleLoc` and `marbleSize` to be used in initializing the other definitions and variables. This is quite common.

Having created the three cups and the marble, it is useful to ask ourselves whether any other instance variables need initialization here. In looking over the instance variables, we determine that we should set the booleans `marbleSelected` and `cupSelected` to `false`.

With the game set up, we can now turn to the methods that control the game. In `onMousePress` we determine, which, if any object has been selected. We first consider the cups. If one of them contains the mouse location, we remember it; otherwise, we check whether the marble contains the mouse location. It is not an accident that we decided to check the cups first. Remember that at times the marble will be hidden inside a cup. We wouldn't want to pick up a hidden marble! Since we only want to pick up a marble that is visibly outside of a cup, we make that the last condition that we check. See the details in Figure 20.16.

```

def shellGame: GraphicApplication = object {
  inherit graphicApplicationSize(500 @ 500)

  // location and size of the marble
  def marbleLoc: Point = 250 @ 300
  def marbleSize: Point = 20 @ 20

  // Cups for hiding the marble
  def cup1: Cup = cupAt (100 @ 200) size (50 @ 50) on (canvas)
  def cup2: Cup = cupAt (200 @ 200) size (50 @ 50) on (canvas)
  def cup3: Cup = cupAt (300 @ 200) size (50 @ 50) on (canvas)

  // the marble to be hidden
  def theMarble: Graphic2D = filledOvalAt(marbleLoc) size (marbleSize) on (canvas)

  // Has the marble been selected for dragging
  var marbleSelected: Boolean := false

  // Has the cup been selected for dragging
  var cupSelected: Boolean := false

  // Cup selected. Meaningful only if cupSelected is true
  var selectedCup: Cup

  var lastMousePosition: Point

```

Figure 20.15: Filling in the details for a ShellGame class: initializing constants and instance variables.


```

// Show contents of cup clicked on
method onMouseClick (mousePos: Point) → Done {
  if (cupSelected) then {
    selectedCup.showContents
  }
}

// When mouse is pressed, determine which object is selected
method onMousePress (mousePos: Point) → Done {
  cupSelected := false
  marbleSelected := false
  if (cup1.contains (mousePos)) then {
    selectedCup := cup1
    cupSelected := true
  } elseif {cup2.contains (mousePos)} then {
    selectedCup := cup2
    cupSelected := true
  } elseif {cup3.contains (mousePos)} then {
    selectedCup := cup3
    cupSelected := true
  } else {
    marbleSelected := theMarble.contains (mousePos)
  }
  lastMousePosition := mousePos
}

// Move the selected cup or marble by the amount the mouse has moved
method onMouseDrag (mousePos: Point) → Done {
  def diff: Point = mousePos - lastMousePosition
  if (cupSelected) then {
    selectedCup.moveBy(diff.x, diff.y)
  } elseif {marbleSelected} then {
    theMarble.moveBy (diff.x, diff.y)
  }
  lastMousePosition := mousePos
}

// If dragging the marble when release mouse over a cup, put the marble in the cup
method onMouseRelease (mousePos: Point) → Done {
  if (marbleSelected) then {
    if (cup1.contains (mousePos)) then {
      cup1.dropln (theMarble)
    } elseif {cup2.contains (mousePos)} then {
      cup2.dropln (theMarble)
    } elseif {cup3.contains (mousePos)} then {
      cup3.dropln (theMarble)
    }
  }
  marbleSelected := false
}
startGraphics
}

```

Figure 20.16: Filling in the details for a ShellGame class: mouse event-handling methods

Note that this decision works nicely for us, but it does have one other effect we should mention. If the mouse is positioned over two overlapping objects, only one can be selected (and it need not be the top-most one). Normally, we would want to fix this, but we ignore that problem here.

The last statement in `onMousePress` remembers the location of the mouse. As you have seen in many examples throughout this text, when an item is to be dragged, we need to remember the initial location of the mouse in order to determine how far the item must be moved with each change in the mouse position.

If an object has been selected, then the `onMouseDrag` method causes that object to be dragged. First we check whether a cup has been selected. If so, we move it in the usual way. Otherwise we check the marble and move it, if it's appropriate to do so. We also remember the location of the mouse, so that we can continue to move the selected item properly.

If a marble is being dragged, then the `onMouseRelease` method determines if it is released over a cup, and, if so, drops the marble into the cup. Going the other direction, if the mouse is clicked over a cup, the `onMouseClicked` method shows the contents of that cup.

20.3.7 Summary of the Design Process

As you have seen, the process of design is one of iterative refinement. We begin by thinking about the problem at its highest level of abstraction. At each step, we provide more and more details. The process of object-oriented design can be summarized as follows:

1. Identify the objects to be modeled in your program.
2. For each type of object identified:
 - (a) List its properties.
 - (b) List its behaviors.
3. Model properties with instance variables.
4. Model behaviors with methods. *Don't forget that you will need initialization code for each class to set up objects of that type.* Focus on the method headers. What types of information need to be passed in as parameters? What will be the result of each method invocation?
5. Fill in implementation details.
 - If a method is very simple, write the Grace code.
 - If a method is more complex, write comments that describe the steps that need to be implemented. These will allow you to organize your thoughts without becoming bogged down in details. They will also serve as your method documentation later. If it becomes clear at this point that the desired method will be very complex, it may be appropriate to add confidential methods to the design of the class to simplify its eventual implementation.

As you fill in the details, you might find that you need to modify your design somewhat. It is not uncommon for a design to change as it develops. If you follow this set of steps, it will help you to organize your ideas so that any design changes along the way will probably be minimal.

20.4 Incremental Testing and Debugging

The process of design follows the idea that it is better to attack a large problem first by carefully planning and then by filling in the details a little at a time. The belief is that in doing this, the programmer will avoid getting confused by the details and will, therefore, be able to produce a more coherent and bug-free program.

We can also apply these ideas to the process of testing. Rather than testing a program only when it is completely written, we can test individual components as we develop them. This typically makes the process of debugging much easier for the following two reasons:

1. Testing smaller (and therefore simpler) pieces of code generally makes it easier to find bugs.
2. When we begin to test larger, more complex components, we can rely on the behaviors of the individual, smaller entities being correct.

In this section we revisit the shell game described in the previous section, discussing how we can test the code as we write it.

20.4.1 Developing and Testing Individual Classes

In writing our shell game program, we determined that we needed two classes, one to describe `Cups` and one to provide a user interface to the game, i.e., the object supporting user interaction. This program structure is typical. That is, we can generally expect to have an object that controls the overall behavior and user interaction with the program, as well as other classes that make up individual entities in the program. As described above, it makes sense to test individual behaviors of the objects in the program before we put everything together. For the shell game, this means testing the behavior of `Cups` before writing and testing the `shellGame` object that controls user interaction. In fact, when testing the `cup` class, we can focus first on the case where a cup is empty and then consider the case where a cup contains a marble.

Let's review the process of filling in the implementation details of the `cup` class, so that we can explore how you can interleave development and testing. First we wrote the constructor. Then we wrote methods for familiar behaviors (i.e., those that allow us to drag an object around on the canvas). Finally, we wrote methods that were conceptually a bit more complex.

We can follow this same plan to test and debug the `cup` class. As soon as we have written the initialization code, we can write a test object that simply constructs a cup. Because a cup is meant to appear in a certain way on the canvas, we can check that the initialization code works appropriately by making a cup (or, even better, several). If they appear as we expect them to, we can move on. If not, we can make corrections before attempting to do anything else with cups.

```

// Game to draw one cup and drag it
def shellGame: GraphicApplication = object {
  inherit graphicApplicationSize(500 @ 500)

  // constants omitted

  // Cups for hiding the marble
  def cup1: Cup = cupAt (100 @ 200) size (50 @ 50) on (canvas)

  // Has the cup been selected for dragging
  var cupSelected: Boolean := false

  // location of mouse after last mouse action
  var lastMousePosition: Point

  // When mouse is pressed, determine which object is selected
  method onMousePress (mousePos: Point) → Done {
    cupSelected := cup1.contains (mousePos)
    lastMousePosition := mousePos
  }

  // Move the selected cup or marble by the amount the mouse has moved
  method onMouseDrag (mousePos: Point) → Done {
    def diff: Point = mousePos - lastMousePosition
    if (cupSelected) then {
      cup1.moveBy(diff.x, diff.y)
    }
    lastMousePosition := mousePos
  }

  startGraphics
}

```

Figure 20.17: A simple controller to test that a cup can be dragged

Now we can go on to the task of implementing the methods that will allow a cup to be dragged around on the canvas, i.e., `contains` and `moveBy`. Once they are written, we can test that they work appropriately. Notice, however, that the `moveBy` method shown in Figure 20.14 refers to the marble. In particular, if the marble is contained in a cup, we want to move it; otherwise we don't. This is clearly important functionality that we need to test, but first we can check that cups can be dragged appropriately by themselves. To test this, we can write simple `onMousePress` and `onMouseDrag` methods in the graphic application as shown in Figure 20.17. As dragging a cup is really no different from dragging any other graphical object, the methods in Figure 20.17 should be quite familiar.

When testing that a cup can be dragged, try pressing the mouse in a variety of different places on the cup itself. Then try pressing the mouse outside the cup, to be sure that points outside it are not somehow considered to be contained in the cup.

If all is well when we test the dragging functionality, we can move on. Specifically, we test whether a cup will show its contents appropriately when we click on it. Remember that

we're testing an empty cup at this point, so nothing should actually be revealed by the cup. The `onMouseClicked` method that we add to our test controller looks like this:

```
// Show contents of cup clicked on
method onMouseClick (mousePos: Point) → Done {
  if (cupSelected) then {
    cup1.showContents
  }
}
```

Once we are convinced that an empty `Cup` will do the right thing, we can test whether a `Cup` with a marble in it behaves appropriately as well. To do this, we construct a marble (i.e., a `FilledOval`) in our test controller. At this point, we could add code to allow a user to select a marble and drag it around the canvas so that it can be dropped into a cup. However, it is better to simply place the marble in `cup1` and see whether it is dragged around with the cup and then dropped from it, as desired. We modify the test controller as in Figure 20.18. Notice that in the initialization code, we construct a cup and then a marble in essentially the same location. This simulates the action of the marble being dropped into the cup.

20.4.2 Putting the Pieces Together: Testing the Controller

As discussed above, the process of writing implementation details for a class can (and should be) interleaved with testing it. If there are multiple types of objects in a program, each individual class should be tested and debugged before all of the pieces are put together. At first this might seem to be too much extra work. However, if you follow this procedure, you will find that it is much easier to test and debug the controller class that coordinates all of the interactions, including those with the user. In the shell game, if the `cup` class has been thoroughly tested and debugged before we write the details of `ShellGame`, we can expect the testing and debugging of `ShellGame` to go much more smoothly. This is because we can assume that any incorrect behavior is due to the new code.

With the `shellGame` object, we begin in the same way that we did with the `cup` class. We first implement and test the initialization code that sets up everything. Then we move on to test other functionality.

The initialization code constructs three cups as well as a marble, this time in a location separate from any of the cups. It also sets two instance variables, the boolean `marbleSelected` and `selectedCup`, which is meant to refer to a `Cup`, if selected. Testing begin involves confirming that all is set up properly on the canvas.

Next we complete `onMousePress` and `onMouseDrag`. To test these, we can try dragging each of the three `Cups` around, as well as the marble. We should also test that dragging the mouse on the canvas (without selecting an object) does nothing to break the program.

The next method to be written is `onMouseRelease`, which is meant to drop the marble into a cup, if appropriate. Testing this method involves dragging the marble, moving it over a `Cup` and then releasing it. This should make the marble disappear. Now dragging the cup with the marble should move both objects together.

Finally, we write the method `onMouseClicked`, which reveals the contents of a cup. To test this method, try clicking in a variety of different places—on the canvas, in an empty cup, in a cup containing a marble.

```

// Game to display a cup, hide a marble under it, and drag the cup
def shellGame: GraphicApplication = object {
  inherit graphicApplicationSize(500 @ 500)

  // constants omitted

  // Cups for hiding the marble
  def cup1: Cup = cupAt (100 @ 200) size (50 @ 50) on (canvas)

  // the marble to be hidden
  def theMarble: Graphic2D = filledOvalAt(110 @ 225) size (marbleSize) on (canvas)
  cup1.dropIn(theMarble)

  // Has the cup been selected for dragging
  var cupSelected: Boolean := false

  // location of mouse after last mouse action
  var lastMousePosition: Point

  // Show contents of cup clicked on
  method onMouseClick (mousePos: Point) → Done {
    if (cupSelected) then {
      cup1.showContents
    }
  }

  // When mouse is pressed, determine which object is selected
  method onMousePress (mousePos: Point) → Done {
    cupSelected := cup1.contains (mousePos)
    lastMousePosition := mousePos
  }

  // Move the selected cup or marble by the amount the mouse has moved
  method onMouseDrag (mousePos: Point) → Done {
    def diff: Point = mousePos - lastMousePosition
    if (cupSelected) then {
      cup1.moveBy(diff.x, diff.y)
    }
    lastMousePosition := mousePos
  }

  startGraphics
}

```

Figure 20.18: A simple graphic application to test that a cup with a marble can be dragged and its contents revealed

Again, this process of incremental testing should make it relatively easy to localize the source of any bugs. If you are confident that you are testing thoroughly, any new bugs are likely to be the result of the most recent code that was written. This is not a guarantee, of course, but it is a reasonable assumption in many cases.

20.5 Classes That Don't Model the Real World

When we introduced object-oriented design in Section 20.1, we said that a major idea behind the approach was the notion of modeling the real world. In thinking about a problem, we would first identify the objects to be modeled. For example, in the shell game, we defined a `Cup` class to model the cups used in the game. But not all classes model entities in the real world. In this section we will design a more interesting shell game to help illustrate some of the ways in which classes can be defined that don't model the real world.

20.5.1 An Animated Shell Game

Our earlier shell game provided an interface that would allow two or more people to play a simple game. However, we must admit this version was too simple to be interesting. It was just too simple to watch the cup being dragged with the marble. At this point in a program's development, one of two things happens. One possibility is that everyone decides its not very interesting and it is simply discarded. However, the second is that the programmer modifies the program to better meet user needs and desires.

Let's consider the second option here and create a version of the program in which a person plays against the computer. The game begins with three cups and a marble as before. As in the earlier version, a player will place the marble into a cup, by dragging it to a cup and then releasing it. All of the shuffling will be done by the computer, however. When the player moves the mouse off the canvas and the marble is hidden in a cup, the shuffling begins. Once the animation ends, the player guesses which cup contains the marble by clicking on that cup. As before, if a cup is clicked, it raises up to show its contents. However, the cup remains raised only a few seconds, and then descends to its prior position, pushing the marble below it on the screen if the marble was contained in it.

The interface described in the animated version of the game is nearly identical to the first version. The only difference is that the player has no control over the movement of the cups. Since there are such strong similarities between the two versions of the shell game, we will see that we can reuse many of our earlier design ideas.

20.5.2 Identifying Classes that Don't Represent Entities in the Real World

Let's go through the entire process of design for this new version of the shell game. We can see from the program description that our design will need to model cups and marbles. As before, we will define a `cup` class; in fact, we can reuse the earlier `cup` class. A simple filled oval is all we need to represent the marble. We will surely need a graphical application object to provide the interface to the players. While we will need to make some modifications, we will see that the earlier object will provide a good starting point for the new one.

Are there any other entities that we need to model? The only real objects in the program description are the cups, marble, and player, as we have already discussed. But there are certainly some things we haven't yet considered. In particular, we haven't figured out how we will model the animation process. In Chapter 9 you learned that animations could be created by writing classes that use objects of type `Animator`. That is precisely how we will construct the animation here.

In identifying the need for this new class, which we will call `Shuffler`, we have done something that was not discussed before. We have introduced a class that does not model any kind of real world object. People can certainly shuffle cups, but we typically don't see machines that have cup shuffling as their purpose. While object-oriented design takes inspiration from the idea of modeling real-world objects and their relationships, there are times when this won't be quite enough, as in our animated shell game. In this case, it is not the process of modeling the real world but rather it is our knowledge of the techniques for creating animations in *Grace* motivates the inclusion of a class in our design.

20.5.3 Completing the Design for the Animated Shell Game

We have already identified the objects to be modeled in our program. As we saw earlier in this chapter, the next step in the design process is to identify the properties and behaviors of each type of object. The marble, as before, can be modeled simply with a filled oval. The cups in the new shell game have identical properties to the earlier cups, and they have nearly identical behaviors. As will become obvious soon, the cups need to be able to do a bit more here. The two additional behaviors that we need to model for a cup are:

- the ability to move to a specific location;
- the ability to report where the cup is currently located;

However, we shall also modify the behavior of method `showContents` so that it lifts, showing the marble if it is in the cup, and then returns to where it was.

The implemented methods that perform these are given in Figure 20.19.

The `showContents` method is the most interesting of the three. If the cup contains the marble then the marble is made visible and released from the cup, and then the cup is moved up. After a delay of two seconds, the cup is moved down to be back in place, and if the marble was originally part of the cup, then the marble is also moved down to be out of the way of the cup.

Next, we need to identify properties and behaviors of a cup shuffler. Identifying properties of a shuffler might seem tricky, but certainly the needed behavior is obvious. The shuffler needs to provide an animation that moves three cups on the canvas. As shown in Figure 20.20, this will be accomplished by the `start` method. The initialization code for this class is pretty trivial. The three cups will be passed in as parameters to the `shuffler` class, and we will initialize a slot for cups to move to in order to get out of the way of a moving cup.

Before we discuss the shuffling process in any more detail, let's think a bit more about the properties of a shuffler. Since there is no such thing as a cup shuffler in the real world, it's hard to imagine how we might describe one. But there is a way to handle this. Recall from Section 20.2.1 that we can think of properties as *nouns* and *adjectives*. We also know that


```

// move the cup so the upper-left-hand corner is at pt
method moveTo (pt: Point) → Done {
  def diff = pt - location
  moveBy (diff.x, diff.y)
}

// return location of cup
method location → Point {
  cupSide.location
}

// Lift the cup up, and if it holds a marble, show it. Then move the cup back.
method showContents → Done {
  def containedMarble: Boolean = containsMarble
  if (containsMarble) then {
    theMarble.visible := true
    containsMarble := false
  }
  moveBy(0, -openY)
  timer.after(2000) do {
    moveBy (0, openY)
    if (containedMarble) then {
      theMarble.moveBy (0, openY)
    }
  }
}
}

```

Figure 20.19: Extra methods required by the Cup class

```

A class to animate shuffling in a Shell Game}
class shuffler(cup1: Cup, cup2: Cup, cup3: Cup) → Startable {
  // missing constants

  // shuffle the three cups until the user is confused
  method start { ... }
}

```

Figure 20.20: High level design for a Shuffler class

these will be represented by instance variables. So what instance variables will be necessary for a shuffler? Our description of shuffling gives us some clues: The shuffler needs to provide an animation that moves three cups on the canvas. If we look at the nouns in this description, we immediately identify that the shuffler will need access to the cups to be shuffled. However, because they are already there as parameters to the class – and will not be reassigned during the life of the object, we don't need to give them new names as Grace **defs** or **vars**.

We can use the close relationship between the constructor and the `run` method to help guide us in the next phase of design, where we outline what needs to be done by each of these. Let's begin by sketching out what needs to be done in the `run` method. There we need to swap two cups, then swap two more, and so on. We need to do this often enough that the player of the game will lose track of which cup contains the marble. We'll do twenty swaps, and we'll define a constant, `numberOfMoves` with this value.

Now let's think a bit more about the actual animation of the swap. When we select the two cups to be exchanged, we can't simply exchange their locations because the swap won't be visible to the player. After all, the three cups look identical. So we'll make the shuffling look like the illustration in Figure 20.21. A cup is randomly selected and is moved aside, out of its current location. Another cup is then moved into its space. The first cup can now be moved into the space left free by the second cup. Then the process can begin all over again.

If shuffling is to proceed as just described, there needs to be a temporary location so that cups can be “moved out of the way.” We will define this as a constant, `holdingPosition`, in the shuffler class. The temporary location will need to be remembered in a constant, so we will need to add that to the high-level design that was shown in Figure 20.20. Finally, the `start` method will need something to help it randomly select cups for shuffling. As we've seen earlier, we can get access to this through the library `random`. Once the class has set up everything as described above, the animation can be started by invoking the `start` method. The design of the shuffler class at this point, illustrating the ideas we just summarized, is shown in Figure 20.22.

Now that we have detailed designs for the `cup` and `shuffler` classes, we can complete our design of the application object, `animatedShellGame`. The user interface for this version of the game is nearly identical to the earlier non-animated version, so we'll reuse ideas from the earlier object. First, we can remove some of the functionality of the earlier version. After all, the player no longer has any control over the movement of the cups. Therefore, we can eliminate the need for the instance variables `cupSelected` `selectedCup`, which were used for dragging a cup. We can also eliminate the statements in `onMousePress` and `onMouseDrag` that are concerned with selecting and dragging cups. These methods now only need to be concerned with the marble.

However, the previous version of `onMouseClicked` depended on the `onMousePress` method identifying which cup was selected. Thus some of the functionality originally need in `onMousePress` will need to be shifted to `onMouseClicked`. That method will now check to see which cup, if any, was clicked on, and will cause that cup to request its `showContents` method, resulting in the animated lifting and then replacing of the cup, displaying the marble if it was there.

Now we can consider the `onMouseEnter` method. When the mouse enters the canvas, this should start the shuffling, but it only makes sense to do any shuffling if the marble is actually hidden in a cup. So we'll need some way to keep track of whether the marble is hidden or

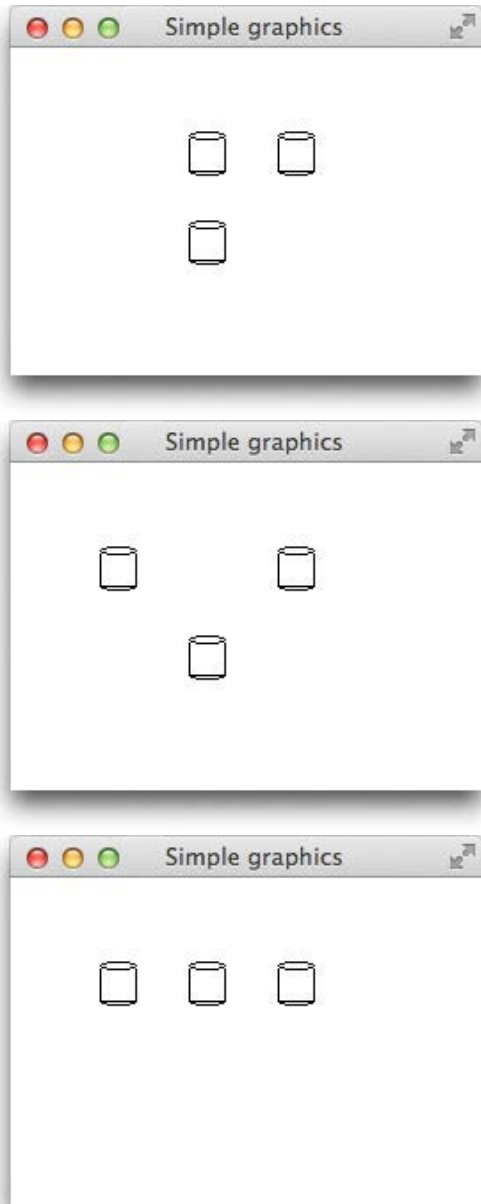


Figure 20.21: Animation of shuffling

```

// A class to animate shuffling in a Shell Game
class shuffler(cup1: Cup, cup2: Cup, cup3: Cup) → Startable {
  def holdingPosition: Point = 200 @ 350
  def numberOfMoves: Number = 20
  def delayTime: Number = 300

  // shuffle the three cups numberOfMoves times
  method start → Done {
    // Repeat the following numberOfMoves times
    // Randomly select two cups to move
    // Move the first cup out of the way to holdingPosition
    // Move the second cup to the location freed by the first
    // Move the first cup from the holdingPosition to the new free space
  }
}

```

Figure 20.22: Filling in the design details for the shuffler class

not. One obvious way to do so is to declare a boolean instance variable to keep track of this information. Initially the value of this variable should be false. In the case that the marble is dragged to a cup, we need to remember that the marble becomes hidden. Once the marble is revealed by a click of the mouse in a cup, we need to note that the marble is again visible. Figure 20.23 shows the new controller design just discussed.

20.5.4 Filling in the Details for the Animated Shell Game

Filling in the details of our new animated shell game program is now quite easy. We simply have to follow the comments that we wrote, outlining our ideas. Again, these comments serve not only to help guide the implementation but to document our ideas to future readers of our classes.

The few additions that we needed to make to the cup class were quite simple, and we listed the details of those methods earlier. Instead, we can devote our attention to the classes shuffler and animatedShellGame object. In fact, our design of animatedShellGame is so detailed that the implementation of that class is quite straightforward and is left as an exercise. The shuffler class is somewhat more interesting, so we'll devote the remainder of this section to that class.

We've already written the initialization code for the shuffler. The implementation of the start method is more interesting, as it involves doing a bit of work to select cups for swapping. We could accomplish cup selection by selecting a random integer in the range 1 to 3, and then using that random number as follows:

```

def cupChoice: Number = rand.integerIn 1 to 3
if (cupChoice == 1) then {
  selectedCup := cup1
} else if (cupChoice == 2) then {
  selectedCup := cup2
} else {
  selectedCup = cup3
}

```

```

def animatedShellGame: GraphicApplication = object {

  // Details from earlier controller omitted . . .

  // Is the marble in a cup?
  var marbleHidden: Boolean := false

  // When mouse is pressed, determine if marble is selected
  method onMousePress (mousePos: Point) → Done {
    // if the marble is not currently hidden and it contains the mouse location it is selected
    // remember the mouse location for any dragging that will be done
  }

  // If it is selected, move the marble by the amount the mouse has moved
  method onMouseDrag (mousePos: Point) → Done {
    // if the marble has been selected move the marble
    // remember the mouse location for any further dragging that will be done
  }

  // IF dragging marble and release mouse over a cup
  // put the marble in the cup
  method onMouseRelease (mousePos: Point) → Done {
    // if the marble was selected and is currently positioned over a cup
    // drop the marble into that cup
    // remember that the marble is hidden

    // the marble is no longer selected upon release of the mouse
  }

  // Show contents of cup clicked on
  method onMouseClick (mousePos: Point) → Done {
    // if a cup has been selected and it contains the marble
    // show the marble and remember that it is no longer hidden
  }

  method onMouseExit (mousePos: Point) → Done {
    // if the marble is hidden shuffle the cups
  }
}

```

Figure 20.23: Design of the application object for the animated shell game

where `selectedCup` is of type `Cup`. The problem, however, is that we have to choose two cups. Using the code above to select two cups might cause us to pick the same one twice.

There are a number of ways we could handle this problem. One clever idea is to use the random number generator to select the one cup that *won't be swapped*. If the selected random integer is 1, we will swap cups 2 and 3; if the random integer is 2, we will swap 1 and 3; if the random integer is 3, we will swap 1 and 2. The `start` method, which implements this idea, is shown in Figure 20.24.

As promised by the comments in the design, the `run` method selects two cups to be shuffled, moves the first out of the way, and then moves the second to its original place. The first is then moved to the original location of the second. There are a few interesting things to note in the implementation, however. First, before moving either randomly selected cup, we need to remember the old locations. If we don't do this, we won't know where to move the cups when we're ready to actually swap them. Second, we need to pause between each move; if we didn't, the moves would happen so quickly that they wouldn't be visible to the player.

Exercise 20.5.1 Complete the implementation of the `animatedShellGame` application from the design given in Figure 20.23.

20.6 Guidelines for Writing Comments

When developing the designs in the previous sections, we wrote our ideas in the form of comments. We retained the comments after developing our code, since they described the various aspects of the classes we had written. While the design process outlined above should result in an appropriate level of documentation, the following guidelines will help you determine whether you have included all the information you need to include. Note that these are general guidelines and that your instructor might ask you to follow slightly different conventions, such as GraceDoc conventions, for commenting.

- Each class and object expression should have a comment. This should be placed at the top of the class or object expression and should include a description of the class as well as the author's name and the date. If the class has parameters, their use should be described as well.

```
// A class to represent a cup for a shell game.  
// Written by . . . , 2/14/04.
```

```
// Construct a cup on the canvas.  
// upperLeft -- the upper left corner of the cup  
// sz -- the desired width and height of the cup}  
// canvas -- canvas on which cup is to be drawn}
```

```
class cupAt (upperLeft: Point) size (sz: Point) on (canvas: DrawingCanvas)
```

In the interest of saving space, we leave the author information out of all of the examples in the text.

- Each constant and variable should be commented. The comment associated with a constant or variable can be written either on the line before it, as follows:

```

// class to shuffle the three cup parameters
class shuffler(cup1: Cup, cup2: Cup, cup3: Cup) → Startable {
  // temporary position to move cup out of way
  def holdingPosition: Point = 200 @ 350
  // number of times to shuffle the parameters
  def numberOfMoves: Number = 20
  // delay between moves
  def delayTime: Number = 300

  // shuffle the three cups until the user is confused
  method start → Done {
    // two cups to be swapped
    var firstMover: Cup
    var secondMover: Cup

    // Swap two randomly chosen cups 20 times
    animator.for (1..20) pausing (3 * delayTime) do {count →
      // choose the cups to be moved
      match(rand.integerIn 1 to 3)
        case {1 →
          firstMover := cup2
          secondMover := cup3
        }
        case {2 →
          firstMover := cup3
          secondMover := cup1
        }
        case {3 →
          firstMover := cup1
          secondMover := cup2
        }
      // location of cups to be swapped
      def firstLocation: Point = firstMover.location
      def secondLocation: Point = secondMover.location

      // move the cups with delays between moves
      firstMover.moveTo (holdingPosition)
      timer.after (delayTime) do {
        secondMover.moveTo (firstLocation)
        timer.after (delayTime) do {
          firstMover.moveTo (secondLocation)
        }
      }
    }
  }
}

```

Figure 20.24: Implementation of shuffling

```
// whether the cup contains a marble
var containsMarble: Boolean
```

or can be written beside it:

```
var containsMarble: Boolean // whether the cup contains a marble
```

Note that the comment succinctly describes the purpose of the variable. If a group of variables or constants serve a related purpose, they can be summarized by a single comment:

```
// the three cups on the screen
def cup1: Cup = ...
def cup2: Cup = ...
def cup3: Cup = ...
```

- Each method should have a comment that describes its purpose. The method comment should describe the parameters as well as the return value, if any:

```
// Move the marble
// dx, dy – distance to move in the vertical and horizontal directions
method moveBy(dx: Number, dy: Number)
```

Note that the method comment should not describe how the method accomplishes its task. If it is important to describe the way a method works, such comments should be placed inside the method.

- Write comments describing how each method works. That is, write comments inside each method that explain what is happening at different parts of the method body. Short methods of 10 or fewer lines typically don't even require a comment, as they are generally simple enough to understand as they are. Comments should describe what is happening at each point, now how. Over-commenting (e.g., simply translating each of code to a comment) can be more distracting than helpful.

20.7 Encapsulation and Information Hiding

Grace is designed to make all instance variables and constants confidential by default. This protects those variables from being inappropriately modified by other objects. If you want to provide access to those values, you can do so through declaring them to be readable or writeable. This serves as a mechanism for providing exactly the type of access you want and no more. The idea of taking your variables and data structures and wrapping a shell around them in the form of a class definition is called *encapsulation*. The notion of hiding the details of your implementation as much as possible is called *information hiding*. These two related ideas are an important part of object-oriented design.

There are several reasons for designing and implementing objects and classes this way. First, as already mentioned, it protects variables from being modified inappropriately by other objects. If several people are involved in a project, this allows them to proceed with the implementations of their individual classes without the fear that their variables will be “corrupted” by others. Even in the case when a single person is implementing an entire program, this notion provides protection from inadvertent modification of variables.

Second, this mechanism makes it possible to present classes to others for their use by simply specifying an interface. That is, it allows us to specify the functionality of a class of objects in terms of its methods, without giving any additional details. We have seen how to do this by declaring the types of objects generated by a class or object expression. This is extremely beneficial to the user of that class, since s/he can create objects and invoke methods on them without needing to know anything about their implementation. The user can work with the *abstraction* in its simplest form.

It should be easy to convince you that this is useful. After all, this has allowed you to use the `objectdraw` library from the beginning. Once you were told the types of objects you could construct and the methods you could invoke, you could make use of those without needing to understand anything more about them. This is not just something that helps the novice programmer. The Grace language consists of many class definitions that make it easier for programmers to do their jobs. In addition, programmers can share libraries, such as `objectdraw`, with others. The ability to make use of pre-existing classes means that programmers can focus on writing new classes with new functionality, rather than having to do everything from scratch.

Third, encapsulation and information hiding make it possible to change the details of a class definition without affecting others who might be using that class. During the time that we wrote this text, we modified the `objectdraw` library several times. Virtually all of those changes were completely transparent to users of our library. That is, they were invisible. While the details changed, the interface was the same, so old code did not “break” when new versions of the library were introduced.

In the remainder of this section we will look more closely at some examples that illustrate the utility of encapsulation and information hiding.

20.7.1 Nested Rectangles: A Graphical Object Class: Nested Rectangles

Imagine for a moment that we have extended the `objectdraw` library to include a new type of graphical object called `NestedRects`. One of these is pictured in Figure 11.1. If you read Chapter 11 this will be familiar to you. A `NestedRects` object is constructed by specifying the `x` and `y` coordinates of the upper left corner, width and height of type `double`, and a `DrawingCanvas`. Once constructed, a `NestedRects` object can be moved to a particular location or it can be removed from the canvas. More specifically, the constructors and methods available for the `NestedRects` class are as given in Figure 20.25.

Given this new type of graphical object, let’s write a simple program that does the following. It begins with a `NestedRects` object on the canvas. The `NestedRects` should be drawn at location (50, 50); the width and height should be 19 and 21, respectively. When the user presses the mouse on the canvas, the `NestedRects` should move to the location of the press. When the mouse is released, the `NestedRects` that was just moved remains there, and a new one is constructed at (50, 50), having width and height twice the size of the last one moved.

The program in Figure 20.26 does exactly what we have described. If you run this simple program, you will find that you can produce some interesting pictures, such as the one in Figure 20.27. But that isn’t our reason for introducing it here.

Since we introduced the `nestedRects` class in Chapter 11, you probably imagine that the class we defined there was used together with the controller in Figure 20.26 to create the

```

// Create set of nested rectangles w/upper left corner at pt &
// dimensions given by dimension on canvas
class nestedRectsAt (pt: Point) size (dimension: Point)
    on (canvas: DrawingCanvas) -> Movable {

    // move full set of nested rectangles by (dx,dy)
    method moveBy (dx: Number, dy: Number) -> Done{ ... }

    // move full set of nested rectangles to newPt
    method moveTo (newPt: Point) -> Done { ... }

    // remove all nested rectangles from canvas
    method removeFromCanvas -> Done { ... }
}

```

Figure 20.25: Outline of nested rectangles class

image in Figure 20.27, but that isn't the case. Instead, we used the `nestedRects` class definition shown in Figure 20.28. This alternate implementation makes use of an array of `Graphic2Ds` to store the individual components of the `nestedRects` object. We are able to calculate exactly how big this array needs to be as long as we know the width and height of the object, as well as the size difference (in this case, eight) between successively smaller components of it.

The only way to detect which implementation we used was to look at the **import** statement in Figure 20.26. It gives away that we imported the definition of the nested rectangles class from a file named `NestedRectsIterative.grace`, as that file also had a definition of a class with that name. Similarly we could use the earlier representation by changing that one **import** statement to refer to the file with the recursive definition of nested rectangles.

The two implementations of the `nestedRects` class are equally valid. An array is a perfectly reasonable choice of data structure to represent the nested rectangles, as is a recursive structure. More importantly, the choice of one or the other is irrelevant to anyone making use of this class. There is no reason for a programmer to think of a `nestedRects` object as “an array of rectangles” or as a “recursive structure of rectangles.” A `nestedRects` object is simply a graphical object that can be created, moved, and removed from a canvas.

20.7.2 The Important Relationship between Encapsulation and Abstraction

The `nestedRects` example illustrated the important relationship between encapsulation and abstraction. Encapsulating the details of an implementation in a class allows us to present an abstraction to anyone using that class. As already mentioned, you have been taking advantage of these ideas from the beginning, when using the `objectdraw` library. Grace programmers make use of these important concepts every day.

To summarize, encapsulation and its counterpart, information hiding, are important because:

- they allow the programmer to present a class to others for their use by simply specifying an interface. Thus the user can work with the abstraction in its simplest form.
- they protect instance variables from being modified inappropriately by other classes.

```

dialect "robjectdraw"
import "NestedRectsIterative" as nr

// application to draw interesting pictures from nested rectangles
def rectangleDrawer: GraphicApplication = object {
  inherit graphicApplicationSize (800 @ 800)

  // location of new rectangles to draw
  def startingLocn: Point = 50 @ 50

  // Initial dimensions of nested rectangles
  def startDimensions: Point = 19 @ 21

  // Dimensions of last nested rectangles drawn
  var dimensions: Point := startDimensions

  // Most recently drawn nested rectangles
  var theRects: nr.Movable := nr.nestedRectsAt (startingLocn)
    size (dimensions) on (canvas)

  // Move the most recently drawn nested rects to pt
  method onMousePress (pt: Point) -> Done {
    theRects.moveTo (pt)
  }

  // Draw a new nested rects twice the size of the last,
  // but at the same initial location
  method onMouseRelease (pt: Point) -> Done {
    dimensions := dimensions * 2
    theRects := nr.nestedRectsAt (startingLocn)
      size (dimensions) on (canvas)
  }

  startGraphics
}
}

```

Figure 20.26: A simple program to draw and move nested rectangles

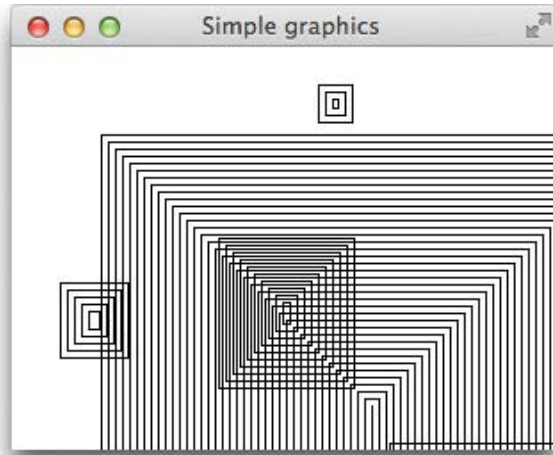


Figure 20.27: A sample drawing produced by running `RectangleDrawer`

- they make it possible to change the details of a class definition without affecting others who might be using that class.

20.8 Summary

This chapter provided an introduction to object-oriented design. It discussed the important ideas of abstraction, encapsulation and information-hiding. These concepts provide a foundation for object-oriented design techniques, but they are also of importance to many other aspects of computer science.

In this chapter we also took you through the process of design, which we can outline as follows:

1. Identify the types of objects in the problem description that you will need to model with classes.
2. For each class, list the properties and behaviors that you want to model. If the class has a real-world counterpart, that can serve as inspiration for your ideas here, though you likely will not model every aspect of a real-world object.
3. Model each property as an instance variable.
4. Model each behavior as a method. Specify only method headers, i.e., signatures, at this point in the design process.
5. Refine your design. If a method is very simple, you can refine the design by immediately writing the Grace statements to implement it. If a method is more complex, write comments describing the steps to be followed.

```

dialect "robjectdraw"

// Create set of nested rectangles w/upper left corner at pt @
// dimensions width x height
class nestedRectsAt (startPt: Point) size (startDimensions: Point)
  on (canvas: DrawingCanvas) -> Movable {

    def shift: Number = 4 // gap between adjacent rectangles

    // return smaller of a and b
    method min (a: Number, b: Number) -> Number {
      if (a < b) then {a} else {b}
    }

    // How many rectangles should be in the structure
    def nestingDepth: Number = ((min (startDimensions.x,startDimensions.y)) /
      (2 * shift)).truncated + 1

    // upper left corner of last set of nested rectangles drawn
    var upperLeft: Point := startPt

    // dimensions of last set of nested rectangles drawn
    var dimensions: Point := startDimensions

    // create and initialize set of nested rectangles
    def theRectangles: List[[Graphic2D]] = emptyList

    for (1 .. nestingDepth) do {depth: Number ->
      theRectangles.add(framedRectAt (upperLeft) size (dimensions) on (canvas))
      upperLeft := upperLeft + (shift @ shift)
      dimensions := dimensions - ((shift @ shift) * 2)
    }

    // move full set of nested rectangles by (dx,dy)
    method moveBy (dx: Number, dy: Number) -> Done{
      for (theRectangles) do {rect: Graphic2D ->
        rect.moveBy (dx, dy)
      }
    }

    // move full set of nested rectangles to newPt
    method moveTo (newPt: Point) -> Done {
      def diff:Point = newPt - theRectangles.at(1).location
      moveBy (diff.x, diff.y)
    }

    // remove all nested rectangles from canvas
    method removeFromCanvas -> Done {
      for (theRectangles) do {rect: Graphic2D ->
        rect.removeFromCanvas
      }
    }
  }
}

```

547
Figure 20.28: An array-based NestedRects class

Remember that design is a process of refinement. It involves refinement of high-level ideas into more and more concrete Grace statements. It also involves refinement of the design plan itself.

20.9 Chapter Review Problems

Exercise 20.9.1 *What do we mean when we say that an object provides an abstraction in a program?*

Exercise 20.9.2 *How do we model properties of objects? How do we model behaviors?*

Exercise 20.9.3 *List the steps that we follow when doing object-oriented design.*

Exercise 20.9.4 *We say that design is a process of refinement. Describe two ways in which refinement plays a role in the design process.*

Exercise 20.9.5 *Write appropriate method comments for the following method of a `Circle` class:*

```
method resize (sizeDiff: Number) → Done
```

This method resizes the circle by changing the size of the diameter by `sizeDiff`.

20.10 Programming Problems

Exercise 20.10.1 *In this exercise you will be asked to design and then implement an animation program. The program should draw a faucet and a glass on the canvas. The faucet should drip individual droplets of water into the glass until the glass is reasonably full. (It needn't be filled completely.) A picture of the canvas with the faucet and glass before any dripping has occurred is shown in Figure 20.29.*

- a. *The first step in the design process is to identify the classes that will be needed. Suggest four classes that would be appropriate to model the objects mentioned in the problem description.*
- b. *List the properties and behaviors that need to be modeled for each of the four classes included in your answer to part (a).*
- c. *For each class, model the properties with instance variables.*
- d. *For each class, model the behaviors with methods.*
- e. *Fill in the implementation details for each class.*

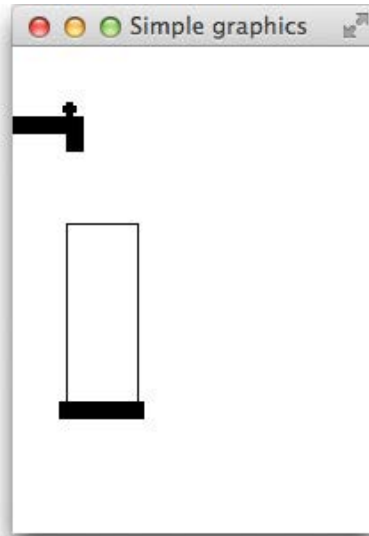


Figure 20.29: Leaky faucet and glass before the faucet drips

Exercise 20.10.2 *In this exercise you will be asked to design and implement a Hangman program. The program should draw a text message on the canvas that contains a hyphen in the place of each letter of the word to be guessed. A list of all the letters the player has guessed so far should also be displayed. The player should guess a letter by selecting from a JComboBox. If the player guesses a letter that is in the target word, the word should be updated on the canvas. If the player guesses a letter that is not in the target word, an additional body part should be drawn. If the player guesses a letter that has already been guessed, the program should tell the player by displaying a message. A snapshot of such a program after several guesses is shown in Figure 20.30.*

- a. *As always, the first step of the design process is to identify the classes that will be needed. List the classes you would include in this program's design. Don't place too much functionality in the application object. In particular, we would expect you to include two or more classes in your design.*
- b. *List the properties and behaviors that need to be modeled for each of the classes corresponding to the classes you included in your answer to part (a).*
- c. *For each class, model the properties with instance variables.*
- d. *For each class, model the behaviors with methods.*
- e. *Fill in the implementation details for each class.*

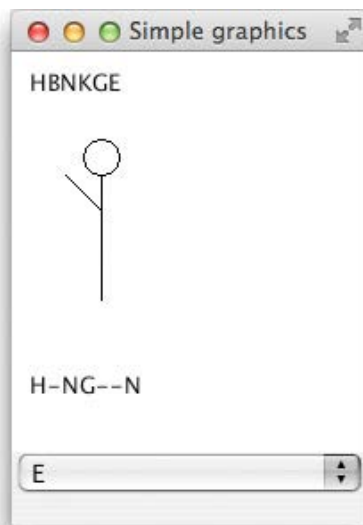


Figure 20.30: A Hangman game

Appendix A

Style Guidelines

The presentation of your program is just as important as its correctness. While good comments or good variable names will not affect the correctness of your program, they will make it easier to read your program.

Readability of code is important for several reasons. First, it is an important way for you to keep track of your thoughts as you design and write your program. The more careful you are about organizing your thoughts, the less likely it is for your code to be error-prone. In the event that there are problems with your code, you will be able to debug it more easily if you can read and follow the work you have done. Second, it will allow others to better understand what you have written. This is particularly important if you are working on a team project or if you are writing code that will be modified over time.

This appendix provides some general rules of good programming style. Your instructor might choose to supplement this with additional guidelines.

A.1 Commenting

Comments are extremely important. Take some care to write accurate yet concise comments.

- Be specific with your comments. What happens when the mouse is pressed? What is the `filledRect` class used for? Your comments should describe the intent of the code. What are you trying to accomplish with this piece of code?
- Do not make your comments too long. Depending on the program used for printing, a line of comment or code that is over 80 characters either will not print or will wrap around to a new line in a fairly unreadable fashion. If it does not print, the TA or instructor will not be able to read it. Either revise it or write it on multiple lines.
- Do not overcomment! Overcommenting can make the program hard to read just as much as under-commenting. Do not comment every line. If they are simple instructions, most people will understand them without your comments. If you think you need commenting, try commenting chunks of code under the same comment. Aside from comments on new identifiers introduced (defs, vars, and methods) there should be few comments in the body of your code. If you feel your program cannot be understood

without a lot of comments then it probably is too complex and you should rewrite it to be simpler.

- Delete any extraneous code that is not used. In particular, if you have written and then commented out code, remove it.

You should write comments for:

1. **Program** Each program should have a series of comments that specify your name, the date of creation (and revisions), and a brief description of what the program does.
2. **Class:** At the top of each class, you should write a brief description of what the class is for, and how the class parameters determine features of the objects generated.
3. **Methods:** Above each method heading there should be a brief description of what the method does. You should also describe what role each parameter plays and what the return result if any) means. If the method code is simple you do not need to comment in the method body. If the method body is complicated, you might want to add some comments to certain areas that will help the reader understand the intent of your code. Be careful not to overcomment!
4. **Variables and constants:** In general, variables and defs should all have comments as to what the identifier stands for. Occasionally several closely related variables or defs can be grouped together under the same comment if that comment is relevant to all. (A comment like: "Several defs" is **never** useful.)

A.2 Blank Lines

Blank lines are used to delineate different areas of the code. The instance variable declarations and defs at the top of your program should be separated from the header of the class and the header of the first method. There should always be space between methods. It is advisable to break up long method bodies and long declarations into logical pieces using blank lines. Always leave a blank line before a comment line.

A.3 Names

You should always choose identifier names that suggest the meanings of the things being named. If the purpose of a method is to draw a rectangle, then a good name for that method is `drawRect`. If there is a variable of type `Graphic2D` that is used as the stem of a flower, then a good name would be `stem` or `stemRect`.

- Names that have nothing to do with the program are very bad names! Ex. `var frodo: Graphic2D`
- Names that are not specific or descriptive enough are generally bad names. Ex. `line` or `l` are not as good as `foulLine` if the identifier represents a foul line.

- Try not to name objects sequentially. You should only do this when the objects are related in some way that is not otherwise expressible. For example, `box1`, `box2`, `box3` are not as good as `darksBasket`, `whitesBasket`, and `colorsBasket` if the rectangles represent laundry baskets for clothes of specific colors.

By convention, identifiers representing objects (e.g., vars, defs, and parameters) should begin with lower case letters, as should class and method names. Types names should start with capital letters. If multiple words are glued together to form an identifier, use a capital letter for the first letter of each new word. For example, `onMouseRelease`.

Only those features of objects which must be accessible to clients should be made public. Methods that are designed to help implement other operations should generally be made confidential. Publicly available methods should reflect the operations supported by an object rather than just advertise its implementation. Thus limit the use of readable, writable, and public definitions and variables.

Where possible, prefer local variables and definitions in methods over the use of instance variables. This makes your code easier to understand and results in less storage being needed for objects. Of course if values need to be saved between invocations of methods then instance variables will be necessary.

A.4 Format

Your program should be organized as neatly as possible. The structure of the text in the program should give the reader an overall sense of the program's logical organization. The Grace compiler requires that statements at the same level be aligned and that extra indentation be provided with statement that continue on a new line. Bodies of classes, methods, and blocks of all sorts must be indented. This indentation will make your program much easier to read.

Appendix B

Built-In Grace Types, Classes, and Libraries

B.1 Introduction

Grace has several built-in types and objects, and a growing selection of dialects and libraries that define other useful types and objects. Built-ins can be used as-is, dialects must be declared with the **dialect** statement, and libraries must be imported with the **import** statement.

B.2 Built-in Types

Grace supports built-in objects with types Object, Number, Boolean, and String.

B.2.1 Object

All Grace objects understand the methods in type Object. These methods will often be omitted when other types are described.

```
type Object = {  
  
  == (other: Object) -> Boolean  
  // true if other is equal to self  
  
  != (other: Object) -> Boolean  
  /= (other: Object) -> Boolean  
  ≠ (other: Object) -> Boolean  
  // the inverse of ==. All three variants have the same meaning.  
  
  hash -> Number  
  // the hash code of self, a Number in the range 0..232  
  
  match (other: Object) -> SuccessfulMatch | FailedMatch  
  // returns a SuccessfulMatch if self "matches" other  
  // returns FailedMatch otherwise.  
  // The exact meaning of "matches" depends on self.
```

```

asString -> String
// a string describing self

asDebugString -> String
// a string describing the internals of self

:: (other:Object) -> Binding
// a Binding object with self as key and other as value.
}

```

We will also not discuss or include the pattern combinators `|` (or) and `&` (and) used in generating compound patterns.

B.2.2 Number

Number describes all numeric values in *minigrace*, including integers and numbers with decimal fractions. (Thus, *minigrace* Numbers are what some other languages call floating point numbers, floats or double-precision). Numbers are represented with a precision of approximately 51 bits.

```

type Number = {

+ (other: Number) -> Number
// sum of self and other

- (other: Number) -> Number
// difference of self and other

* (other: Number) -> Number
// product of self and other

/ (other: Number) -> Number
// quotient of self and other

% (other: Number) -> Number
// modulus of self and other (remainder after division)

.. (last: Number) -> Sequence
// the Sequence of numbers from self to last

< (other: Number) -> Boolean
// true iff self is less than other

<= (other: Number) -> Boolean
≤(other: Number) -> Boolean
// true iff self is less than or equal to other

> (other: Number) -> Boolean
// true iff self is greater than other

>= (other: Number) -> Boolean

```

\geq (other: Number) \rightarrow Boolean
// true iff self is greater than or equal to other

prefix- \rightarrow Number
// negation of self

compare (other: Number) \rightarrow Number
// a three-way comparison: -1 if (self < other), 0 if (self == other), and +1 if (self > other).

inBase (base: Number) \rightarrow String
// a string representing self as a base number (e.g., base 2)

truncated \rightarrow Number
// number obtained by throwing away self's fractional part

rounded \rightarrow Number
// whole number closest to self

floor \rightarrow Number
// largest whole number less than or equal to self

ceiling \rightarrow Number
// smallest number greater than or equal to self

abs \rightarrow Number
// the absolute value of self

sqrt \rightarrow Number
// the square root of self

sgn \rightarrow Number
// the signum function: 0 when self == 0, // -1 when self < 0, and +1 when self > 0

isNaN \rightarrow Boolean
// true if this Number is a NaN

sin \rightarrow Number
// trigonometric sine (self in radians)

cos \rightarrow Number
// cosine (self in radians)

tan \rightarrow Number
// tangent (self in radians)

asin \rightarrow Number
// arcsine of self (result in radians)

acos \rightarrow Number
// arccosine of self (result in radians)

```

atan -> Number
// arctangent of self (result in radians)

lg -> Number
// log base 2 of self

ln -> Number
// the natural log of self

exp -> Number
// e raised to the power of self

log10 (n: Number) -> Number
// log base 10 of self
}

```

B.2.3 String

String constructors are written surrounded by double quote characters. There are three commonly-used escape characters:

- `\n` means the newline character
- `\\` means a single backslash character
- `\"` means a double quote character.

There are also escapes for a few other characters and for arbitrary Unicode codepoints; for more information, see the Grace language specification.

String constructors can also contain simple Grace expressions¹ enclosed in braces, like this: `"count = {count}."` These are called string interpolations. The value of the interpolated expression is calculated, converted to a string (by requesting its `asString` method), and concatenated between the surrounding fragments of literal string. Thus, if the value of `count` is 7, the above example will evaluate to the string `"count = 7."`

Strings are immutable. Methods like `replace()` with always return a new string; they never change the receiver.

```

type String = {
  * (n: Number) -> String
  // returns a string that contains n repetitions of self, so "abc"*3 = "abcabcabc"

  ++ (other: Object) -> String
  // returns a string that is the concatenation of self and other.asString

  < (other: String)
  // true if self precedes other lexicographically

  <= (other: String)
  ≤ (other: String)
}

```

¹ It is a limitation of *minigrace* that expressions containing `{braces}` and `"quotes"` cannot be interpolated into strings.


```

// (self == other) || (self < other)

== (other: Object)
// true if other is a String and is equal to self

!= (other: Object)
≠ (other: Object)
// (self == other).not

> (other: String)
// true if self follows other lexicographically

>= (other: String)
≥ (other: String)
// (self == other) || (self > other)

at (index: Number) -> String
// returns the character in position index (as a string of size 1); index must be in the range 1..size

first -> String
// returns the first character of the string, as a String of size 1. String must
//not be empty

asDebugString -> String
// returns self enclosed in quotes, and with embedded special characters quoted.
// See also quoted.

asLower -> String
// returns a string like self, except that all letters are in lower case

asNumber -> Number
// attempts to parse self as a number; returns that number, or NaN if it can't

asString -> String
// returns self, naturally

asUpper -> String
// returns a string like self, except that all letters are in upper case

capitalized -> String
// returns a string like self, except that the initial letters of all words are in upper case

compare (other: String) -> Number
// a three-way comparison: -1 if (self < other), 0 if (self == other), and +1 if (self > other)

contains (other: String) -> Boolean
// returns true if other is a substring of self

endsWith (possibleSuffix: String)
// true if self ends with possibleSuffix

```

```

filter (predicate: Block1[[String, Boolean]] ) -> String
// returns the String containing those characters of self for which predicate returns true

fold [[U]] (binaryFunction: Block2[[U, String, U]] ) startingWith(initial: U) -> U
// performs a left fold of binaryFunction over self, starting with initial.
// For example, fold {a, b -> a + b.ord} startingWith 0 will compute the sum
// of the ords of the characters in self

hash -> Number
// the hash of self

indexOf (pattern: String) -> Number
// returns the leftmost index at which pattern appears in self, or 0 if it is not there.

indexOf [[W]] (pattern: String) ifAbsent (absent: Block0[[W]]) -> Number | W
// returns the leftmost index at which pattern appears in self; applies absent if it is not there.

indexOf (pattern: String) startingAt(offset) -> Number
// like indexOf(pattern) except that it returns the first index  $\geq$  offset, or 0 if there is no such index.

indexOf [[W]] (pattern:String) startingAt(offset) ifAbsent(action:Block0[[W]]) -> Number | W
// like the above, except that it applies action if there is no such index.

indices -> Sequence [[T]]
// an object representing the range of indices of self (1..self.size)

isEmpty -> Boolean
// true if self is the empty string

iterator -> Iterator[[String]]
// an iterator over the characters of self

lastIndexOf(sub:String) -> Number
// returns the rightmost index at which sub appears in self, or 0 if it is not there.

lastIndexOf [[W]] (sub:String) ifAbsent (absent:Block0[[W]]) -> Number | W
// returns the rightmost index at which sub appears in self; applies absent if it is not there.

lastIndexOf [[W]] (pattern:String) startingAt (offset) ifAbsent (action:Block0[[W]]) -> Number | W
// like the above, except that it returns the last index  $\leq$  offset.

map[[U]] (function:Block[[String,U]]) -> Iterable[[U]]
// returns an Iterable object containing the results of successive applications of function to the
// individual characters of self. Note that the result is not a String, even if type U happens to be String.
// If a String is desired, use fold()startingWith "" with a function that concatenates.

match (other: Object) -> SuccessfulMatch | FailedMatch
// returns SuccessfulMatch match if self matches other, otherwise FailedMatch

ord -> Number
// a numeric representation of the first character of self, or NaN if self is empty.

```

```

replace (pattern: String) with (new: String) -> String
// a string like self, but with all occurrences of pattern replaced by new

size -> Number
// returns the size of self, i.e., the number of characters it contains.

startsWith (possiblePrefix: String) -> Boolean
// true when possiblePrefix is a prefix of self

startsWithDigit -> Boolean
// true if the first character of self is a (Unicode) digit.

startsWithLetter -> Boolean
// true if the first character of self is a (Unicode) letter

startsWithPeriod -> Boolean
// true if the first character of self is a period

startsWithSpace -> Boolean
// true if the first character of self is a (Unicode) space.

substringFrom (start: Number) size (max: Number) -> String
// returns the substring of self starting at index start and of length max characters,
// or extending to the end of self if that is less than max.  If start = self.size + 1, or
// stop < start, the empty string is returned.  If start is outside the range
// 1..self.size+1, BoundsError is raised.

substringFrom (start: Number) to (stop: Number) -> String
// returns the substring of self starting at index start and extending
// either to the end of self, or to stop.  If start = self.size + 1, or
// stop < start, the empty string is returned.  If start is outside the range
// 1..self.size+1, BoundsError is raised.

substringFrom (start: Number) -> String
// returns the substring of self starting at index start and extending
// to the end of self.  If start = self.size + 1, the empty string is returned.
// If start is outside the range 1..self.size+1, BoundsError is raised.

trim -> String
// a string like self except that leading and trailing spaces are omitted.

quoted -> String
// returns a quoted version of self, with internal characters like " and \ and newline escaped,
// but without surrounding quotes.
}

```

B.2.4 Boolean

The Boolean literals are **true** and **false**.

```
type Boolean = {  
    not -> Boolean  
    prefix ! -> Boolean  
    // the negation of self  
  
    && (other: Boolean) -> Boolean  
    // return true when self and other are both true  
  
    || (other: Boolean) -> Boolean  
    // return true when either self or other (or both) are true  
}
```

In conditions in ‘if’ statements, and in the operators **&&** and **||**, a Block returning a boolean may be used instead of a Boolean. This means that **&&** and **||** can be used as “short-circuit”, also known as “non-commutative”, operators: if the argument is a block, it will be evaluated only if necessary.

```
type BlockBoolean = { apply -> Boolean }  
type BlockOrBoolean = BlockBoolean | Boolean
```

B.2.5 Blocks

Blocks are anonymous functions that take zero or more arguments and may return a result. There is a family of ‘Block’ types that describe block objects.

```
type Block0[[R]] = {  
    // type of parameterless functions that return a value of type R  
    apply -> R  
}
```

```
type Block1[[T,R]] = {  
    // type of functions that take a single argument of type T and return a value of type R  
    apply(a:T) -> R  
}
```

```
type Block2[[S,T,R]] = {  
    // type of functions that takes two parameters of types S and T, and returns a value of type R  
    apply(a:S, b:T) -> R  
}
```

B.2.6 Point

Points can be thought of as locations in the cartesian plane, or as 2-dimensional vectors from the origin to a specified location. As a result they can also be used to represent the width and height of an object. Points are created from Numbers using the **@** infix operator. Thus, **3 @ 4** represents the point with coordinates (3,4).

```

type Point = {
  x -> Number
  // the x-coordinates of self

  y -> Number
  // the y-coordinate of self

  + (other:Point) -> Point
  // the Point that is the vector sum of self and other, i.e. (self.x+other.x) @ (self.y+other.y)

  - (other:Point) -> Point
  // the Point that is the vector difference of self and other, i.e. (self.x-other.x) @ (self.y-other.y)

  prefix - -> Point
  // the point that is the negation of self

  * (factor:Number) -> Point
  // this point scaled by factor, i.e. (self.x*factor) @ (self.y*factor)

  / (divisor:Number) -> Point
  // this point scaled by 1/factor, i.e. (self.x/divisor) @ (self.y/divisor)

  length -> Number
  // distance from self to the origin

  distanceTo(other:Point) -> Number
  // distance from self to other
}

```

B.2.7 Binding

A binding is an immutable pair comprising a key and a value. Bindings are created with the infix `::` operator, as in `k::v`, or by requesting `binding.key(k) value(v)`.

```

type Binding[[K, T]] = {
  key -> K
  // returns the key
  value -> T
  // returns the value
}

```

B.3 Collection objects

The objects described in this section are made available to all standard Grace programs. (This means that they are defined as part of the *standardGrace* dialect.) As is natural for collections, the types are parameterized by the types of the elements of the collection. Type arguments are enclosed in `[[` and `]]` used as brackets. This enables us to distinguish, for example, between `Set[[Number]]` and `Set[[String]]`. In Grace programs, type arguments and their

brackets can be omitted; this is equivalent to using `Unknown` as the argument, which says that the programmer either does not know, or does not care to state, the type.

B.3.1 Common Abstractions

The major kinds of collection are sequence, list, set, and dictionary. Although these objects differ in their details, they share many common methods, which are defined in a hierarchy of types, each extending the one above it in the hierarchy.

The simplest is the type `Iterable[[T]]`, which captures the idea of a (potentially unordered) collection of *elements*, each of type `T`, over which a client can iterate:

```
type Iterable[[T]] = {
  iterator -> Iterator[[T]]
  // Returns an iterator over my elements. It is an error to modify self while iterating over it.
  // Note: all other methods can be defined using iterator. Iterating over a dictionary
  // yields its values.

  isEmpty -> Boolean
  // True if self has no elements

  size -> Number
  // The number of elements in self; raises SizeUnknown if size is not known.

  sizeofUnknown (action: Block0[[Number]]) -> Number
  // The number of elements in self; if size is not known, then action is evaluated and its value returned.

  first -> T
  // The first element of self; raises BoundsError if there is none.
  // If self is unordered, then first answers an arbitrary element.

  do (action: Block1[[T,Unknown]]) -> Done
  // Applies action to each element of self.

  do (action:Block1[[T, Unknown]]) separatedBy (sep: Block0[[Unknown]]) -> Done
  // applies action to each element of self, and applies sep (to no arguments) in between.

  map[[R]] (unaryFunction: Block1[[T, R]]) -> Iterable[[T]]
  // returns a new collection whose elements are obtained by applying unaryFunction to
  // each element of self. If self is ordered, then the result is ordered.

  fold[[R]] (binaryFunction: Block2[[R, T, R]]) startingWith(initial:R) -> R
  // folds binaryFunction over self, starting with initial. If self is ordered, this is
  // the left fold. For example, fold {a, b -> a + b} startingWith 0
  // will compute the sum, and fold {a, b -> a * b} startingWith 1 the product.

  filter (condition: Block1[[T, Boolean]]) -> Iterable[[T]]
  // returns a new collection containing only those elements of self for which
  // condition holds. The result is ordered if self is ordered.

  ++ (other: Iterable[[T]]) -> Iterable[[T]]
  // returns a new object whose elements include those of self and those of other.
```

```
}
```

The type ‘Collection’ adds some conversion methods to ‘Iterable’:

```
type Collection[[T]] = Iterable[[T]] & type {  
  asList -> List[[T]]  
    // returns a (mutable) list containing my elements.  
  
  asSequence -> Sequence[[T]]  
    // returns a sequence containing my elements.  
  
  asSet -> Set[[T]]  
    // returns a (mutable) Set containing my elements, with duplicates eliminated.  
    // The == operation on my elements is used to identify duplicates.  
}
```

Additional methods are available in the type `Enumerable`; an `Enumerable` is like a `Sequence`, but where the elements must be *enumerated* one by one, in order, using a computational process, rather than being stored explicitly. For this reason, operations that require access to all of the elements at one time are not supported, except for conversion to other collections that store their elements. The key difference between an `Iterable` and an `Enumerable` is that `Enumerables` have a natural order, so lists are `Enumerable`, whereas sets are just `Iterable`.

```
type Enumerable[[T]] = Collection[[T]] & type {  
  
  values -> Enumerable[[T]]  
    // an enumeration of my values: the elements in the case of sequence or list,  
    // the values in the case of a dictionary.  
  
  asDictionary -> Dictionary[[Number, T]]  
    // returns a dictionary containing my indices as keys and my elements as values, so that  
    // my self.at(i) element is self.asDictionary.at(i)}.  
  
  keysAndValuesDo (action: Block2[[Number, T, Object]]) -> Done  
    // applies action, in sequence, to each of my keys and the corresponding element.  
  
  into(existing: Collection[[T]]) -> Collection[[T]]  
    // adds my elements to existing, and returns existing.  
  
  sorted -> List[[T]]  
    // returns a new List containing all of my elements, but sorted by their < and == operations.  
  
  sortBy (sortBlock: Block2[[T, T, Number]]) -> Sequence[[T]]  
    // returns a new List containing all of my elements, but sorted according to the ordering  
    // established by sortBlock, which should return -1 if its first argument is less than its second  
    // argument, 0 if they are equal, and +1 otherwise.  
}
```

B.3.2 Lineups

The Grace language uses brackets as a syntax for constructing lineup objects. For example, `[2,4,6,8]` is the line-up consisting of the first four even positive integers. `[]` constructs the empty lineup.

Lineup objects have type `Iterable[[T]]` for the appropriate `T`. They are not indexable, so can't be used like arrays or lists. They are primarily intended for initializing more capable collections, as in `list [2, 3, 4]`, which creates a list, or `set ["red", "green", "yellow"]`, which creates a set. Notice that a space must separate the name of the method from the lineup.

B.3.3 Sequence

The type `Sequence[[T]]` describes sequences of values of type `T`. Sequence objects are immutable; they can be constructed either explicitly, using `sequence [1, 3, 5, 7]`, or as ranges like `1 .. 10`.

type `Sequence[[T]] = Enumerable[[T]] & type {`

```
  at (ix: Number) -> T
  // returns my ixth element, provided ix is integral and 1 ≤ ix ≤ size

  first -> T
  // returns my first element

  second -> T
  // returns my second element

  third -> T
  // returns my third element

  fourth -> T
  // returns my fourth element

  fifth -> T
  // returns my fifth element

  last -> T
  // returns my last element

  indices -> Sequence[[Number]]
  // returns the sequence of my indices.

  keys -> Sequence[[Number]]
  // same as indices; the name keys is for compatibility with dictionaries.

  indexOf (sought: T) -> Number
  // returns the index of my first element v such that v == sought. Raises NoSuchElementException if there is none
  .

  indexOf[[W]] (sought: T) ifAbsent(action: Block0[[W]]) -> Number | W
  // returns the index of the first element v such that v == sought.
  // Performs action if there is no such element.

  reversed -> Sequence[[T]]
  // returns a Sequence containing my values, but in the reverse order.
```



```

contains (sought:T) -> Boolean
  // returns true if I contain an element v such that v == sought
}

```

B.3.4 Ranges

Ranges are sequences of consecutive integers. They behave exactly like other sequences, but are stored compactly. Ranges are created by two methods on the range class:

```

range.from (lower: Number) to (upper: Number)
  // the sequence of integers from lower to upper, inclusive. If lower = upper, the range contains a
  // single value.
  // if lower > upper, the range is empty. It is an error for lower or upper not to be an integer.

```

```

range.from (upper: Number) downTo (lower: Number)
  // the sequence from upper to lower, inclusive. If upper = lower, the range contains a single value.
  // if upper < lower, the range is empty. It is an error for lower or upper not to be an integer.

```

The .. operation on Numbers can also be conveniently used to create ranges. Thus, 3 .. 9 is the same as range.from (3) to (9), and (3 .. 9).reversed is the same as range.from (9) downTo (3).

B.3.5 List

The type List[[T]] describes objects that are mutable lists of elements that have type T. List objects can be constructed using the list request, as in list[[T]] [a, b, c, ...] or list (existingCollection). An empty list is created by emptyList.

```

type List[[T]] = Sequence[[T]] & type {

  at (n: Number) put(new:T) -> List[[T]]
  // updates self so that my nth element is new. Returns self.
  // Requires 1 ≤ n ≤ size+1; when n =size+1, equivalent to addLast(new).

  add (new: T) -> List[[T]]
  addLast(new:T) -> List[[T]]
  // adds new to end of self. (The first form can be also be applied to sets, which are not Indexable.)

  addFirst (new: T) -> List[[T]]
  // adds new as the first element of self. Changes the index of all of the existing elements.

  addAllFirst (news: Iterable[[T]]) -> List[[T]]
  // adds news as the first elements of self. Change the index of all of the existing elements.

  removeFirst -> T
  // removes and returns first element of self. Changes the index of the remaining elements.

  removeLast -> T
  // remove and return last element of self.

  removeAt (n: Number) -> T
  // removes and returns nth element of self

```

```

remove (element: T) -> List[[T]]
// removes element from self. Raises a NoSuchElementException if not.self.contains(element).
// Returns self

remove (element: T) ifAbsent (action: Block0[[Unknown]]) -> List[[T]]
// removes element from self; executes action if any of them is not contained in self. Returns self

removeAll (elements: Iterable[[T]]) -> List[[T]]
// removes elements from self. Raises a NoSuchElementException if any one of
// them is not contained in self. Returns self

removeAll (elements: Iterable[[T]]) ifAbsent(action:Block0[[Unknown]]) -> List[[T]]
// removes elements from self; executes action if any of them is not contained in self. Returns self

++ (other:List[[T]]) -> List[[T]]
// returns a new list formed by concatenating self and other

addAll (extension: Iterable[[T]]) -> List[[T]]
// extends self by appending extension; returns self.

contains (sought: T) -> Boolean
// returns true when sought is an element of self.

== (other: Object) -> Boolean
// returns true when other is a Sequence of the same size as self, containing the same elements
// in the same order.

sort -> List[[T]]
// sorts self, using the < and == operations on my elements. Returns self.
// Compare with sorted, which constructs a new list.

sortBy (sortBlock: Block2[[T, T, Number]]) -> List[[T]]
// sorts self according to the ordering determined by sortBlock, which should return -1 if its first
// argument is less than its second argument, 0 if they are equal, and +1 otherwise. Returns self.
// Compare with sortedBy, which constructs a new list.

copy -> List[[T]]
// returns a list that is a (shallow) copy of self

reverse -> List[[T]]
// mutates self in-place so that its elements are in the reverse order. Returns self.
// Compare with reversed, which creates a new collection.
}

```

B.3.6 Sets

Sets are unordered collections of elements without duplicates. The == method on the elements is used to detect and eliminate duplicates; it must be symmetric.

```

type Set[[T]] = Collection[[T]] & type {
  size -> Number
}

```

```

// the number of elements in self.

add (element:T) -> Set[[T]]
// adds element to self. Returns self.

addAll (elements: Iterable[[T]]) -> Set[[T]]
// adds elements to self. Returns self.

remove (element: T) -> Set[[T]]
// removes element(s) from self. It is an error if element is not present. Returns self.

remove (element: T) ifAbsent(block: Block0[[Done]]) -> Set[[T]]
// removes element from self. Executes action if element is not present. Returns self.

removeAll (elems: Iterable[[T]])
// removes elems from self. It is an error if any of the elems is not present. Returns self.

removeAll (elems: Iterable[[T]]) ifAbsent (action: Block1[[T,Done]]) -> Set[[T]]
// removes elems from self. Executes action apply(e) for each e in elems that is
// not present. Returns self.

contains (elem: T) -> Boolean
// true if self contains elem

includes (predicate: Block1[[T,Boolean]]) -> Boolean
// true if predicate holds for any of the elements of self

find (predicate: Block1[[T,Boolean]]) ifNone (notFoundBlock: Block0[[T]]) -> T
// returns an element of self for which predicate holds, or the result of applying notFoundBlock is there
// is none.

copy -> Set[[T]]
// returns a copy of self

** (other:Set[[T]]) -> Set[[T]]
// set intersection; returns a new set that is the intersection of self and other

-- (other:Set[[T]]) -> Set[[T]]
// set difference (relative complement); the result contains all of my elements that are not also in other.

++ (other:Set[[T]]) -> Set[[T]]
// set union; the result contains elements that were in self or in other (or in both).

isSubset (s2: Set[[T]]) -> Boolean
// true if I am a subset of s2

isSuperset (s2: Iterable[[T]]) -> Boolean
// true if I contain all the elements of s2

into (existing: Collection[[T]]) -> Collection[[T]]
// adds my elements to existing, and returns existing.

```

```
}
```

B.3.7 Dictionary

The type `Dictionary[[K, T]]` describes objects that are mappings from *keys* of type `K` to *values* of type `T`. Like sets and sequences, dictionary objects can be constructed using the class `dictionary`, but the argument to `dictionary` must be of type `Iterable[[Binding]]`. This means each element of the argument must have methods `key` and `value`. Bindings can be conveniently created using the infix `::` operator, as in `dictionary[[K, T]](k::v, m::w, n::x, ...)`. An empty dictionary can be created by evaluating `emptyDictionary`.

```
type Dictionary[[K, T]] = Collection[[T]] & type {  
  size -> Number  
  // the number of key::value bindings in self  
  
  at (key: K) put (value:T) -> Dictionary[[K, T]]  
  // puts value at key; returns self  
  
  at (k: K) -> T  
  // returns my value at key k; raises NoSuchElementException if there is none.  
  
  at (k: K) ifAbsent (action:Block0[[T]]) -> T  
  // returns my value at key k; returns the result of applying action if there is none.  
  
  containsKey (k: K) -> Boolean  
  // returns true if one of my keys == k  
  
  contains (v: T)  
  containsValue (v: T)  
  // returns true if one of my values == v  
  
  removeAllKeys (keys: Iterable[[K]]) -> Self  
  // removes all of the keys from self, along with the corresponding values. Returns self.  
  
  removeKey (key: K) -> Self  
  // removes key from self, along with the corresponding values. Returns self.  
  
  removeAllValues (removals: Iterable[[T]]) -> Self  
  // removes from self all of the values in removals, along with the corresponding keys. Returns self.  
  
  removeValue (removal: T) -> Self  
  // removes from self removal, along with the corresponding keys. Returns self.  
  
  keys -> Iterable[[K]]  
  // returns my keys as a lazy sequence  
  
  values -> Iterable[[K]]  
  // returns my values as a lazy sequence  
  
  bindings -> Iterable[[Binding[[K, T]]]]  
  // returns my bindings as a lazy sequence
```

```

keysAndValuesDo (action: Block2[[K, T, Object]]) -> Done
// applies action, in arbitrary order, to each of my keys and the corresponding value.

keysDo (action: Block2[[K, Object]]) -> Done
// applies action, in arbitrary order, to each of my keys.

valuesDo (action: Block2[[T, Object]]) -> Done
do (action:Block2[[T, Object]]) -> Done
// applies action, in arbitrary order, to each of my values.

copy -> Self
// returns a new dictionary that is a(shallow) copy of self

asDictionary -> Dictionary[[K,T]]
// returns self

++ (other:Dictionary[[K,T]]) -> Dictionary[[K,T]]
// returns a new dictionary that merges the entries from self and other.
// A value in other at key k overrides the value in self at key k.

-- (other:Dictionary[[K,T]]) -> Dictionary[[K,T]]
// returns a new dictionary that contains all of my entries except for those whose keys are in other
}

```

B.3.8 Iterators and for loops

Collection that implement the type `Iterable[[T]]` implement the internal and external iterator patterns, which provide for iteration through a collection of elements of type `T`, one element at a time. The method `do()` and its variant `do()separatedBy()` implement internal iterators, and method `iterator` returns an external iterator object, with the following interface:

```

type Iterator[[T]] = type {
  next -> T
  // returns the next element of the collection over which I am the iterator;
  // raises the Exhausted exception if there are no more elements.
  // Repeated request of this method will yield all of the elements of
  // the underlying collection, one at a time.

  hasNext -> Boolean
  // returns true if there is at least one more element,
  // i.e., if next will not raise the Exhausted exception.
}

```

Multiple iterators can exist on the same collection, for example, multiple iterator objects and multiple `dos` can be used to read through a file. Requesting `next` on one iterator advances its conceptual position, but does not affect other iterators over the same collection; nor does requesting `do` on a collection disturb any iterator objects. However, *it is an error to modify a collection object while iterating through it*. If you implement your own iterator, it is good

practice to detect this error and raise `ConcurrentModification`.

For-do loops on Iterable objects are provided by standard `Grace`. The method `for()do()` takes two arguments, an Iterable collection and a one-parameter block body. It repeatedly applies `body` to the elements of collection. For example:

```
def fruits = sequence ["orange", "apple", "mango", "guava"]
for (fruits) do { each ->
  print(each)
}
```

The elements of the sequence `fruits` are bound in turn to the parameter `each` of the block that follows `do`, and the block is then executed. This continues until all of the elements of `fruits` have been supplied to the block, or the block terminates the surrounding method by executing a `return`.

`for()do()` is precisely equivalent to requesting the `do` method of the Iterable, which is usually both faster and clearer:

```
fruits.do { each ->
  print(each)
}
```

A variant `for()and()do()` allows one to iterate through two collections in parallel, terminating when the smaller is exhausted:

```
def result = list [ ]
def xs = [1, 2, 3, 4, 5]
def ys = ["one", "two", "three"] for (xs) and (ys) do { x, y ->
  result.add(x::y) }
```

After executing this code, `result == [1::"one", 2::"two", 3::"three"]`.

The need for external iterators becomes apparent when it is necessary to iterate through two collections, but not precisely in parallel. For example, this method merges two sorted Iterables into a sorted list:

```
method merge [[T]](cs: List[[T]]) and (ds: List[[T]]) -> List[[T]] {
  def clter = cs.iterator
  def dlter = ds.iterator
  def result = emptyList

  if (clter.hasNext.not) then { return result.addAll(ds) }
  if (dlter.hasNext.not) then { return result.addAll(cs) }

  var c := clter.next
  var d := dlter.next

  while {clter.hasNext && dlter.hasNext} do {
    if (c <= d) then {
      result.addLast (c)
      c := clter.next
    } else {
      result.addLast (d)
      d := dlter.next
    }
  }
}
```

```

if (c <= d) then {
  result.addAll [c,d]
} else {
  result.addAll [d,c]
}
while {clter.hasNext} do { result.addLast (clter.next) }
while {dlter.hasNext} do { result.addLast (dlter.next) }
result
}

```

B.3.9 Primitive Array

Primitive arrays can be constructed using `primitiveArray.new(size)` where `size` is the number of slots in the array. Initially, the contents of the slots are undefined. Primitive arrays are indexed from 0 though `size - 1`. They are intended as building blocks for more user-friendly objects. Most programmers should use `list`, `set` or `dictionary` rather than `primitiveArray`.

```

type Array[[T]] = {
  size -> Number
  // return the number of elements in self

  at (index: Number) -> T
  // return the element of array at index

  at (index: Number) put (newValue: T) -> Done
  // update element of list at given index to newValue

  sortInitial (n: Number) by (sortBlock: Block2[[T, T, Number]]) -> Boolean
  // sorts elements 0..n. The ordering is determined by sortBlock, which should return -1
  // if its first argument is less than its second argument, 0 if they are equal, and +1 otherwise.

  iterator -> Iterator[[T]]
  // returns iterator through the elements of self. It is an error to modify the array while
  // iterating through it.
}

```

B.4 Built-In Libraries

B.4.1 Random

The random module object can be imported using `import "random" as rand`, for any identifier of your choice, e.g. `rand`. The object `rand` responds to the following methods.

```
between0And1 -> Number
// A pseudo-random number in the interval [0..1)
between (m: Number) and (n: Number) -> Number // A pseudo-random number in the interval [m..n)
integerIn (m: Number) to (n: Number) -> Number // A pseudo-random integer in the interval [m..n]
```

B.4.2 Option

The option module object can be imported using `import "option" as option`, for any identifier of your choice, e.g. `option`. The object `option` responds to the following methods.

```
type Option[[T]] = type {
  value -> T
  do (action:Block1 T, Done) -> Done isSome -> Boolean
  isNone -> Boolean
}
some T (contents:T) -> Option T
// creates an object s such that s.value is contents, s.do(action)
// applies action to contents, isSome answers true and isNone answers false

none T -> Option T
// creates an object s such that s.value raises a ProgrammingError,
// s.do(action) does nothing, isSome answers false and isNone answers true
```

B.4.3 Sys

The sys module object can be imported using `import "sys" as system`, for any identifier of your choice, e.g. `system`. The object `system` responds to the following methods.

```
type Environment = type {
  at (key: String) -> String
  at (key: String) put (value: String) -> Boolean
  contains (key: String) -> Boolean
}

argv -> Sequence String
// the command-line arguments to this program

elapsedTime -> Number
// the time in seconds, since an arbitrary epoch. Take the difference of two
// values to measure a duration.

exit (exitCode: Number) -> Done
// terminates the whole program, with exitCode.

execPath -> String
// the directory in which the currently-running executable was found.
```


`environ` → `Environment`
// the current environment.

Appendix C

Objectdraw API Summary

This appendix provides a brief outline of the public constructors and methods for the major classes in the objectdraw library for Grace.

C.1 Application

An object of type `Application` pops up an empty window. The user may add GUI components to this window. (See details in appendix D. An object of type `Application` is normally created by writing an object (or class) that inherits from the following class which pops up a window with dimensions given by `dimensions!` and has title given by `initialTitle`

```
class applicationTitle(initialTitle: String)
    size (dimensions!: Point)
```

This type contains the following simple methods. Other methods will be described in Appendix D

```
// A standalone window which contains other components.
```

```
type Application = Container & type {
```

```
    // The title of the application window.
```

```
    windowTitle -> String
```

```
    // Set the title of the application window.
```

```
    windowTitle:= (value: String) -> Done
```

```
    // width of the window
```

```
    width -> Number
```

```
    // height of the window
```

```
    height -> Number
```

```
    // Create window with listeners for mouse enter and exit actions
```

```
    startApplication -> Done
```

```
    // Close the window
```

```
    stopApplication -> Done
```

```
}
```

C.2 GraphicApplication

An object of type `GraphicApplication` pops up a window with a canvas installed for drawing on. It is also capable of responding to mouse events.

C.2.1 Creating a GraphicApplication

Objects that extend this class will pop up a window with a canvas for drawing when the `startGraphics` method is requested.

// Create a graphic application with a canvas that is has size given by dimensions

```
class graphicApplicationSize( dimensions: Point ) -> GraphicApplication
```

The class provides methods for accessing the canvas as well as providing methods for mouse-event handling.

```
type GraphicApplication = Application & type {
```

```
  // canvas holds graphic objects on screen
```

```
  canvas -> DrawingCanvas
```

```
  // Respond to a mouse click (press and release) in the canvas at the given
```

```
  // point.
```

```
  onMouseClick (mouse: Point) -> Done
```

```
  // Respond to a mouse press in the canvas at the given point.
```

```
  onMousePress (mouse: Point) -> Done
```

```
  // Respond to a mouse release in the canvas at the given point.
```

```
  onMouseRelease (mouse: Point) -> Done
```

```
  // Respond to a mouse move in the canvas at the given point.
```

```
  onMouseMove (mouse: Point) -> Done
```

```
  // Respond to a mouse drag (move during press) in the canvas at the given
```

```
  // point.
```

```
  onMouseDrag (mouse: Point) -> Done
```

```
  // Respond to a mouse entering the canvas at the given point.
```

```
  onMouseEnter (mouse: Point) -> Done
```

```
  // Respond to a mouse exiting the canvas at the given point.
```

```
  onMouseExit (mouse: Point) -> Done
```

```
  // must be invoked to create window and its contents as well as prepare the
```

```
  // window to handle mouse events
```

```
  startGraphics -> Done
```

```
}
```

The mouse event-handling methods (`onMouseClick`, `onMousePress`, etc. have a default behavior of doing nothing. The programmer should override any of these methods that are relevant, and provide them with the desired behavior in the subobject or subclass.

C.3 Animator

The animation library provides the following type definition and corresponding method implementations.

C.3.1 Creating an Animator object

```
// class to generate animator of objects with default pause of pauseTimeMS' milliseconds  
class animator.pausing(pauseTimeMS') -> Animator {
```

C.3.2 Types used in methods of type Animator

The following type definitions are used in the methods of Animator:

```
// type of a block that takes no parameters and completes an action  
type Procedure = {apply -> Done}
```

```
// type of a block that takes no parameters and returns a boolean  
type BoolBlock = {apply -> Boolean}
```

```
// type of objects that can be animated  
type NumberBlock = {apply -> Number}
```

C.3.3 Methods available in objects of type Animator

The animation library supplies the following methods. If the library is promoted using a statement of the form **import "animation" as animator** then the methods can be invoked by statements like

```
animator.while {...} pausing (...) do { ... }  
type Animator = {  
  // Repeatedly execute block while condition is true  
  while (condition: BoolBlock) pausing (pauseTime: Number) do (block:Block) -> Done  
  
  // Repeatedly execute block while condition is true, pausing pauseTime between iterations  
  // when condition fails, execute endBlock.  
  while (condition: BoolBlock) pausing (pauseTime: Number) do (block: Block)  
    finally(endBlock: Block) -> Done  
  
  // Repeatedly execute block while condition is true  
  // pausing variable amount of time (obtained by evaluating timeBlock) between iterations  
  // when condition fails, execute endBlock.  
  while (condition: BoolBlock) pauseVarying (timeBlock: NumberBlock) do (block: Block) -> Done  
  
  // Repeatedly execute block while condition is true  
  for[[T]] (range!: Sequence[[T]]) pausing (pauseTime: Number) do (block: Block[[T,Done]]) -> Done  
  
  // Repeatedly execute block while condition is true  
  // when condition fails, execute endBlock.  
  for[[T]] (range!: Sequence[[T]] pausing (pauseTime: Number) do (block:Block[[T,Done]])  
    finally (endBlock: Block) -> Done  
}
```

C.4 DrawingCanvas objects

Canvases are for drawing on. All graphic items are displayed on a canvas.

C.4.1 Constructing a DrawingCanvas object

Canvases are created when an object is created from class `graphicApplication` and may be requested from such an object via a request to `canvas`.

C.4.2 Methods available in objects of type DrawingCanvas

The following are the most useful methods available on `DrawingCanvas`

```
// clear the canvas  
clear -> Done  
  
// return the current dimensions of the canvas  
width -> Number  
height -> Number  
size -> Point
```

C.5 Graphic objects

C.5.1 Methods available for all Graphic objects

```
// location of object on screen  
x -> Number  
y -> Number  
location -> Point  
  
// Add this object to canvas c  
addToCanvas (c: DrawingCanvas) -> Done  
  
// Remove this object from its canvas  
removeFromCanvas -> Done  
  
// Is this object visible on the screen?  
isVisible -> Boolean  
  
// Determine if object is shown on screen  
visible:= (_: Boolean) -> Done  
  
// move this object to newLocn  
moveTo (newLocn: Point) -> Done  
  
// move this object dx to the right and dy down  
moveBy (dx: Number, dy: Number) -> Done  
  
// Does this object contain locn  
contains (locn: Point) -> Boolean
```

```

// Does other overlap with this object
overlaps (other: Graphic2D) -> Boolean

// set the color of this object to c
color:= (c: Color) -> Done

// return the color of this object
color -> Color

// Send this object up one layer on the screen
sendForward -> Done

// send this object down one layer on the screen
sendBackward -> Done

// send this object to the top layer on the screen
sendToFront -> Done

// send this object to the bottom layer on the screen
sendToBack -> Done

// Return a string representation of the object
asString -> String

```

C.6 Graphic2D Objects

C.6.1 Additional methods available for Graphic2D objects

Graphic2D objects have all of the methods of Graphic plus the following:

```

// dimensions of object
width -> Number
height -> Number
size -> Point

// Change dimensions of object
size:= (dimensions: Point) -> Done
width:= (width: Number) -> Done
height:= (height: Number) -> Done

```

C.6.2 Classes generating Graphic2D objects

```

// class to generate framed rectangle at (x',y') with size dimensions' created on canvas'
class framedRectAt (location!: Point) size (dimensions!: Point)
    on (canvas!: DrawingCanvas) -> Graphic2D

// class to generate filled rectangle at (x',y') with size dimensions' created on canvas'
class filledRectAt (location!: Point) size (dimensions!: Point).
    on (canvas!: DrawingCanvas) -> Graphic2D

// class to generate framed oval at (x',y') with size dimensions created on canvas'

```

```

class framedOvalAt (location!: Point) size (dimensions!: Point)
    on (canvas!: DrawingCanvas) -> Graphic2D

// class to generate filled oval at (x',y') with size dimensions; created on canvas'
class filledOvalAt (location!: Point) size (dimensions!: Point)
    on (canvas!: DrawingCanvas) -> Graphic2D

// class to generate framed arc at (x',y') with size dimensions'
// from startAngle radians to endAngle radians created on canvas'
// Note that angle 0 is in direction of positive x axis and increases in
// angles go clockwise.
class framedArcAt (location!: Point) size (dimensions!: Point)
    from (startAngle: Number) to (endAngle: Number) on (canvas!: DrawingCanvas) -> Graphic2D

// class to generate filled arc at (x',y') with size dimensions'
// from startAngle degrees to endAngle degrees created on canvas'
// Note that angle 0 is in direction of positive x axis and increases in
// angles go clockwise.
class filledArcAt (location!: Point) size (dimensions!: Point)
    from (startAngle: Number) to (endAngle: Number) on (canvas!: DrawingCanvas) -> Graphic2D

// class to generate an image on canvas' at (x',y') with size dimensions'
// The image is taken from the URL fileName and must be in "png" format.
class drawableImageAt (location!: Point) size (dimensions!: Point)
    url (imageUrl:String) on (canvas!: DrawingCanvas) -> Graphic2D

```

C.7 Line Objects

C.7.1 Additional methods available for Line objects

Line objects have all of the methods of Graphic plus the following:

```

// return the start and end of line
start -> Point
end -> Point

// set start and end of line
start:= (start!: Point) -> Done
end:= (end!: Point) -> Done
setEndpoints (start!: Point, end!: Point) -> Done

```

C.7.2 Class generating a Line object

```

// Create a line from start' to end' on canvas'
class lineFrom (start!: Point) to (end!: Point) on (canvas!: DrawingCanvas) -> Line

```

C.8 Text Objects

C.8.1 Additional methods available for Text items

Text objects have all of the methods of Graphic plus the following:

```

// return the contents displayed in the item
contents -> String

```



```

// reset the contents displayed to be s
contents:= (s: String) -> Done

// return width of text item
width -> Number

// return size of the font used to display the contents
fontSize -> Number

// Set the size of the font used to display the contents
fontSize:= (size: Number) -> Done

```

C.8.2 Class generating a Text object

```

// class to generate text at location' on canvas' initially showing contents'
class textAt (location!: Point) with (contents!: String) on (canvas!: DrawingCanvas)
    -> Text

```

C.9 Colors

C.9.1 Type Color

Elements of type `Color` represent (immutable) colors that can be associated with geometric objects.

```

type Color = {
  red -> Number    // The red component of the color.
  green -> Number // The green component of the color.
  blue -> Number   // The blue component of the color.
}

```

An object that can create Colors

The object `colorGen` contains a class for generating colors, some predefined colors, and a method for generating random colors.

Elements of type `Color` can be created using the following class, where the red, green, and blue components are numbers between 0 and 255

```

class colorGen.r(r')g(g')b(b') -> Color

```

A random color will be generated when `colorGen.random` is evaluated.

Predefined colors

The following colors are predefined in `objectdraw` and may be used in programs: `colorGen.white`, `colorGen.black`, `colorGen.green`, `colorGen.red`, `colorGen.gray`, `colorGen.blue`, `colorGen.cyan`, `colorGen.magenta`, `colorGen.yellow`, and `colorGen.neutral`.

C.10 Sounds

C.10.1 Type Audio

Elements of type Audio represent sounds that can be played.

```
type Audio = {  
  // An audio file that can be played  
  
  source -> String           // The source URL of the audio file.  
  source:= (value: String) -> Done  
  play -> Done              // Play the audio.  
  pause -> Done            // Pause playing the audio.  
  isLooping -> Boolean     // does the audio loop back to the start?  
  looping:= (value: Boolean) -> Done // determine whether the audio will loop  
  isEnded -> Boolean       // whether the audio has finished  
}
```

A class generating sounds

The class `audioUrl` generates objects of type Audio.

```
class audioUrl (source: String) -> Audio  
  // Creates an audio file from source, which is the web URL of a .wav file
```

Appendix D

Using GUI components in Grace

D.1 Introduction

Grace's objectdraw dialect supports GUI components modeled on those available in Java and Javascript.

Grace supports classes that generate GUI components of types `Button`, `TextBox` (for labels), `TextField` (for data entry), `NumberField` (for entering numbers), and `Choice` (for pop-up menus), all of which extend the type `Component`.

We begin by talking about these types and the classes that can be used to construct them. Later we will discuss how to place these components in a window.

D.2 GUI components

In this section we discuss the different kinds of GUI components supported by Grace and the methods that may be requested of them. However, first we take a brief detour to talk about events generated by components.

D.2.1 Events generated by GUI components

When the user interacts with a GUI event, the system generates an event that allows access to relevant information about the event, e.g., the component that was affected, the location of the mouse, etc. We list these types below:

```
type Event = {  
  // returns the component generating the event  
  source -> Component  
}  
  
type MouseEvent = Event & type {  
  // returns the mouse location when the event was generated  
  at -> Point  
}  
  
type KeyEvent = Event & type {  
  // returns the numeric code of the key pressed
```

```
code -> Number
}
```

These events are generated by the operating system in response to user actions. Thus we do not need to worry about how to create them.

The event handling methods all take as parameter actions which the programmer can use to respond to the events. They all take events as parameters and then execute code. Their types are given below:

```
// type of a block that takes an event as parameter
```

```
type Response = {apply: Event -> Done}
```

```
type MouseResponse = {apply: MouseEvent -> Done}
```

```
type KeyResponse = {apply: KeyEvent -> Done}
```

An element of type `MouseResponse` is a block that takes a `MouseEvent` as a parameter. For example,

```
{msEvent: MouseEvent -> print "the event was at {msEvent.at}"}
```

is a block that takes a `MouseEvent` as a parameter and then prints out the location associated with the event.

D.2.2 Component

Type `Component` contains the methods available on all GUI components. While no classes generate objects with type exactly matching `Component`, all classes generating GUI components support all the method of `Component`:

```
type Component = {
```

```
// The width of this component.
```

```
width -> Number
```

```
// The height of this component.
```

```
height -> Number
```

```
// The dimensions of this component
```

```
size -> Point
```

```
// Respond to a mouse click (press and release) in this component with the  
// given event.
```

```
onMouseClicked (f : MouseResponse) -> Done
```

```
// Respond to a mouse press in this component with the given event.
```

```
onMousePressDo (f : MouseResponse) -> Done
```

```
// Respond to a mouse release in this component with the given event.
```

```
onMouseReleaseDo (f : MouseResponse) -> Done
```

```
// Respond to a mouse move in this component with the given event.
```

```
onMouseMoveDo (f : MouseResponse) -> Done
```

```
// Respond to a mouse drag (move during press) in this component with the
```

```

// given event.
onMouseDownDo (f : MouseResponse) -> Done

// Respond to a mouse entering this component with the given event.
onMouseEnterDo(f : MouseResponse) -> Done

// Respond to a mouse exiting this component with the given event.
onMouseExitDo (f : MouseResponse) -> Done

// Respond to a key type (press and release) in this component with the given
// event.
onKeyTypeDo (f : KeyResponse) -> Done

// Respond to a key press in this component with the given event.
onKeyPressDo (f : KeyResponse) -> Done

// Respond to a key release in this component with the given event.
onKeyReleaseDo (f : KeyResponse) -> Done

// Whether this component will dynamically fill up any empty space in the
// direction of its parent container.
isFlexible -> Boolean

// Set whether this component will dynamically fill up any empty space in the
// direction of its parent container.
flexible := (value : Boolean) -> Done
}

```

The mouse event handlers all take parameters of type `MouseResponse`, which is a block that takes a parameter of type `MouseEvent` (see the previous section).

D.2.3 TextBox

An object of type `TextBox` is a GUI component that can serve as a label for another component.

```

type TextBox = Component & type {
  // The text contents of the box.
  contents -> String
  contents := (value : String) -> Done
}

// create a label show the string in contents'
class textBoxWith (contents' : String) -> TextBox

```

As shown above, the class `textBoxWith` is used to create objects of type `TextBox`. Thus `textBoxWith("Enter your name:")` will create a label that reads “Enter your name”.

D.2.4 Button

An object of type `Button` is a labeled button that can be pressed to generate an event. Type `Labeled` is a synonym of `Button` and is included because other types will extend it.

```

type Labeled = Component & type {
  // The label on the button.
  label -> String
  label := (value : String) -> Done
}

```

```

type Button = Labeled

```

```

class buttonLabeled (label' : String) -> Button

```

A button saying “Press Me” can be generated by `buttonLabeled ("Press Me")`.

D.2.5 TextField and NumberField

The types `TextField` and `NumberField` generate components where users can add strings and numbers. They both extend the type `Input`.

```

type Input = labeled & type {

```

```

  // Respond to this input gaining focus with the given action.
  onFocusDo (f : Response) -> Done

```

```

  // Respond to this input losing focus with the given action.
  onBlurDo (f : Response) -> Done

```

```

  // Respond to this input having its value changed (requires typing return).
  onChangeDo (f : Response) -> Done

```

```

}

```

```

type TextField = Input & type {

```

```

  // The contents of the text field.
  text -> String
  text := (value : String) -> Done

```

```

}

```

```

class textFieldLabeled (label' : String) -> TextField

```

```

type NumberField = Input & type {

```

```

  // The contents of the number field.
  number -> Number
  number := (value : Number) -> Done

```

```

}

```

```

class numberFieldLabeled (label : String) -> NumberField

```

```

type Choice = Input & type {

```

```

  // The currently selected option.
  selected -> String
  selected := (value : String) -> Done

```

```
}
```

```
class menuWithOptions (Iterable[[String]]) -> Choice
```

For example, evaluating `menuWithOptions ["one","two","three"]` creates a pop-up menu with three choices.

D.2.6 DrawingCanvas

An object of type `DrawingCanvas` is a rectangular portion of a window in which the programmer can draw graphic items of type `Graphic`, including filled and framed rectangles and ovals, as well as lines, text items, and images. These items are described more completely in other documents.

```
type DrawingCanvas = {  
  // Start drawing on the canvas. Will continue until the canvas is destroyed.  
  startDrawing -> Done  
  
  // add d to canvas  
  add(d: Graphic)->Done  
  
  // remove d from window  
  remove(d: Graphic)->Done  
  
  // tell the system to redraw the screen  
  notifyRedraw -> Done  
  
  // clear the canvas  
  clear -> Done  
  
  // Send d to top layer of graphics  
  sendToFront (d: Graphic)->Done  
  
  // send d to bottom layer of graphics  
  sendToBack (d: Graphic)->Done  
  
  // send d up one layer in graphics  
  sendForward (d: Graphic)->Done  
  
  // send d down one layer in graphics  
  sendBackward (d: Graphic)->Done  
  
  // return the current dimensions of the canvas  
  width -> Number  
  height -> Number  
  size -> Point  
}
```

```
class drawingCanvasSize (dimensions!: Point) -> DrawingCanvas
```

D.3 Adding components to a window

Most programs using the `objectdraw` library will require a canvas in the middle of the window. Objects and classes that **inherit** `graphicApplicationSize(dim)` will have a canvas preinstalled, while other GUI components can be added above or below it. On the other hand, objects or classes that **inherit** `applicationTitle (windowTitle) size (dim)` will start with no GUI components in the window.

D.3.1 Application and GraphicApplication

Types `Application` and `GraphicApplication` both extend type `Container`.

// The type of components that contain other components.

```
type Container = Component & type {  
  
    // The number of components inside this container.  
    numComponents -> Number  
  
    // Retrieve the component at the given index.  
    at (index: Number) -> Component  
  
    // Put the given component at the given index.  
    at (index: Number) put(component: Component) -> Done  
  
    // Add a component to the end of the container.  
    append (component: Component) -> Done  
  
    // Add a component to the start of the container.  
    prepend (component: Component) -> Done  
  
    // Perform an action for every component inside this container.  
    do (f: Procedure[[Component]]) -> Done  
  
    // Arrange the contents of this container along the horizontal axis.  
    // Components which exceed the width of the container will wrap around.  
    arrangeHorizontal -> Done  
  
    // Arrange the contents of this container along the vertical axis.  
    // Components which exceed the height of the container will wrap around.  
    arrangeVertical -> Done  
}  
  
// A standalone window which contains other components.  
type Application = Container & type {  
  
    // The title of the application window.  
    windowTitle -> String  
    windowTitle := (value : String) -> Done  
  
    // must be requested in order to pop up window when initialization complete  
    startApplication -> Done
```



```

    // Close the window
    stopApplication -> Done
}

class applicationTitle(initialTitle : String)
    size (initialDimensions: Point) -> Application

// Type of object that runs a graphical application that draws
// objects on a canvas and responds to mouse actions on the canvas
type GraphicApplication = Application & type {
    // canvas holds graphic objects on screen
    canvas -> DrawingCanvas

    // Respond to a mouse click (press and release) in the canvas at the given
    // point.
    onMouseClick (mouse : Point) -> Done

    // Respond to a mouse press in the canvas at the given point.
    onMousePress (mouse : Point) -> Done

    // Respond to a mouse release in the canvas at the given point.
    onMouseRelease (mouse : Point) -> Done

    // Respond to a mouse move in the canvas at the given point.
    onMouseMove (mouse : Point) -> Done

    // Respond to a mouse drag (move during press) in the canvas at the given
    // point.
    onMouseDrag (mouse : Point) -> Done

    // Respond to a mouse entering the canvas at the given point.
    onMouseEnter (mouse : Point) -> Done

    // Respond to a mouse exiting the canvas at the given point.
    onMouseExit (mouse : Point) -> Done

    // must be invoked to create window and its contents as well as prepare the
    // window to handle mouse events
    startGraphics -> Done
}

```

D.3.2 Adding Components

You can add components to an object of type `Application` or `GraphicApplication` by using the commands `prepend` and `append`. These methods add components either at the very beginning or at the very end of the list of components.

For example, suppose `drawingProgram` is an object of type `GraphicApplication`. Thus it starts out as having a canvas in the window. If `button1`, `button2`, and `menu` are GUI components then the results of executing:

```

prepend (button2)
prepend (button1)
append (menu)

```

will result in `button1` and `button2` being next to each other (in that order) above the canvas, while `menu` is centered below the canvas in the window.

You can have more control over the arrangement of the components by inserting them in containers and placing those containers in the window or in other containers. You can create a container by evaluating `emptyContainer`.

Here is an example of using containers to arrange the GUI components more carefully:

```

def southBox = emptyContainer
// items are laid out in southBox from top to bottom
southBox.arrangeVertical

// container row1 will hold boxLabel and sizeField
def row1 = emptyContainer
row1.append (boxLabel)
row1.append (sizeField)

// container row2 will hold colorLabel and colorMenu
def row2: Container = emptyContainer
row2.append (colorLabel)
row2.append (colorMenu)

// container row3 will hold three buttons
def row3 = emptyContainer
row3.append (slowButton)
row3.append (mediumButton)
row3.append (fastButton)

// Add the rows from top to bottom in southBox
southBox.append (row1)
southBox.append (row2)
southBox.append (row3)

// add the whole southBox to the bottom of the window, below the canvas
append (southBox)

```

The container `southBox` contains three other containers: `row1`, `row2`, and `row3`, which will be stacked one on top of the other (that is determined by `southBox.arrangeVertical`). Each of the row containers contains several GUI components, which are laid out next to each other, as shown in Figure D.1.

The same idea works with objects inheriting from `application` except there will be no canvas in the window.

D.4 Responding to events involving GUI components

As can be seen in the type `Component`, GUI components can respond to user actions. For example, components can respond to mouse presses or clicks, selecting an item in a pop-up menu, or entering a new item in a text or number field.

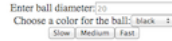
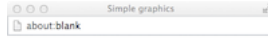


Figure D.1: Layout of GUI components in a window

D.4.1 Button events

An object of type `Button`, can respond to various mouse events. Normally, however, we will only have them respond to a press or click by a mouse. Suppose `tickleButton` has type `Button`. If we wish for it to respond to a mouse press by printing a message then we can write:

```
tickleButton.onMousePress {mEvt: MouseEvent -> print "Stop tickling me!"}
```

Now when the user presses the mouse on the button, it will print “Stop tickling me!”. The code after the arrow can use the formal parameter `mEvt`, for example by sending it method requests for `source` or `at`. However, it will be unusual to need those pieces of information (especially the location of the mouse in the button) in the code reacting to the press.

If you prefer to react to a mouse click or release instead, you can use the corresponding method. You can add as many actions as you like to respond to mouse events. They will all be executed when the corresponding event occurs.

D.4.2 Input events

An object of a type extending `Input`, e.g., `Choice`, `TextField`, or `NumberField` will generate a `Change` event when a new item is selected from a pop-up menu or a new entry is made to a field (terminated by hitting the **enter** key).

These are handled using the `onChangeDo` method of the component.

```
menu.onChangeDo {evt: Event -> ... menu.selected ...}  
myTextField.onChangeDo {evt: Event -> ... myTextField.text ...}  
myNumberField.onChangeDo {evt: Event -> ... myNumberField.number ...}
```

As you can see, once the event is triggered we can query the item to see what value has been provided.

Index

- !, 115
- !=, 105
- ÷, 77
- *, 77
- +, 77
- , 77
- /, 77
- //, 39
- <, 105
- <=, 105
- ==, 105
- >, 105
- >=, 105
- &&, 114
- ||, 114
- \n, 281

- absolute value, 136–139
- abstraction, 509
- addListener, 256
- algorithm, 4
- alias, 435–436
- and operator, 114
- Animated Object
 - with while loop, 223
- array, 355

- behavior, 159
- block, 102, 183
- Boolean, 145
- boolean, 106–108
- boolean expression, 112
- boolean operator, 114
- button, 248
- buttonPressed, 249

- canvas, 18
- catch, 455

- Choice, 252
- class, 14, 149, 159
- class instance, 159
- close, 476
- Color, 47
- color, 70
- comments, 39–491
- comparison operators, 105
- compiler, 7
- concatenation operator (++), 390
- condition, 102
- conditional statement, 99
- confidential, 212
- confidentialconfidential, 212
- console window, 87
- construction, 23
- coordinate system
 - Cartesian, 16
 - graphics, 16–20

- def**, 37
- definitions, 36
- DeMorgan’s Laws, 197
- divide-and-conquer, 488
- dynamic method invocation, 436–438

- elseif, 109
- empty string, 389
- encapsulation, 542
- eof, 476
- error
 - logic, 29
 - syntax, 28
- event, 8, 249
 - keystroke, 256
- event-driven programming, 8, 249
- exception, 453–463

- expression, 74
 - arithmetic, 76–79
 - boolean, 105, 112
- file, 465
 - reader, 473
 - writer, 467
- fontSize, 87
- for, 304
- getline, 476
- Grace, 7
- graphical user interface, 245
- graphics, 16, 17, 23, 33
- GUI, 245
- GUI components, 245
- height, 70, 73
- visible=, 43
- IDE, 9
- identifier, 36
- if statement, 99
- if-then statement, 103
- if-then-else statement, 102
- index, 317
- information hiding, 542
- inherit, 416
- inherit, 416–421
- inheritance, 415–444
- input, 245
 - keyboard, 256
- instance, 159
- instance variable, 159, 215–217
- integrated development environment, 9
- iterate, 484
- iteration, 270
- iterative, 271, 292, 488
- JButton, 248
- JPanel, 254
- keystroke events, 256
- language
 - programming, 6
- list, 317
 - creation, 319
 - declaration, 318
 - parallel, 337
- lists of lists, 365
 - constructing, 366
 - declaring, 366
 - traversing, 368
- literal
 - numeric, 74
- local variable, 215–217
- location, 70
- logarithms, 136
- logical operator, 114
- loop
 - counting, 187, 304
 - for, 304
 - nested, 190
 - while, 180
- loops
 - debugging, 309
- matchmatch, 200
- math functions, 136–139
- matrix, 374
 - column traversal, 379
 - diagonal traversal, 380
 - row traversal, 377
 - traversing, 376
- menu, 252
- method, 11
 - accessor, 69–70
 - event handling, 26
 - mutator, 33–36
 - request, 36
- method request, 159
- MouseEventMouseEvent, 249
- moveBy, 34
- moveTo, 34
- nested if statement, 116
- new line, 281
- object-oriented design, 509
 - modeling behaviors with methods, 516
 - modeling properties with instance variables, 514

- summary of design process, 528
- onMouseClicked, 27
- onMouseDown, 27
- onMouseEnter, 27
- onMouseExit, 27
- onMouseMove, 27
- onMousePress, 27
- onMouseRelease, 27
- operand, 77
- operator
 - arithmetic, 77
- or operator, 114
- output, 87
- override, 426

- panel, 254
- parameter, 11, 215–217
 - formal, 58
- pixel, 17
- Point, 51–53
- postcondition, 491–492
- precedence, 81
- precondition, 491–492
- print, 87
- program, 1
- programming, 30, 33
 - object-oriented, 149

- random number, 90
- read, 476
- Reader, 473
- recursion, 269–299
 - base case, 282, 292, 299
 - recursive case, 282, 292, 299
 - structural, 270–290
- recursive, 484, 488
- recursive methods, 290–298
- refinement in design process, 520
- removeFromCanvas, 45
- request
 - method, 13

- scope of identifier, 217
- search, 483–491
 - binary, 487–491
 - complexity of, 489
 - linear, 484–487
 - complexity of, 486
- self**, 164
 - as parameter, 164
- sendBackward, 57
- sendForward, 57
- sendToBack, 57
- sendToFront, 57
- short-circuit boolean operators, 120
- sizeof, 145
- sort, 492–505
 - insertion, 497–500
 - complexity of, 500
 - merge, 500–505
 - complexity of, 504
 - selection, 493–497
 - complexity of, 496
- state, 159
- statement, 75
- stream, 465–476
- String, 142–145
- String methods
 - asLower, 397
 - asUpper, 397
 - at, 402
 - size, 393
 - substring, 399
- style, programming, 117
- subclass, 415
- superclass, 416

- text, 247
- text field, 245
- text=, 247
- TextField, 247
- textField, 245
- throw, 459
- time, 140
- trigonometric functions, 136
- try, 455
- try-catch, 455
- two-dimensional lists, *see* lists of lists
- type, 129

URL, 279

variable, 38

 instance, 38, 159

 local, 167

while, 180

width, 70, 73

write, 476

x, 70

y, 70