Tutorial: Grace To Java

KIm B. Bruce

Contents

1 Hello World!

I have now added lots of material on Java to the documentation and handouts page on the course web pages. In particular you can find a link to a free Java text as well as complete documentation on the Java version of the objectdraw library.

We will be running programs inside the (free) Eclipse development environment, which is installed on all of our lab computers. On the documentation page you can find instructions on how to install eclipse on your own computer if you would like. We demonstrate there how to load a program into Eclipse.

The first program in any new language is by tradition one that prints "Hello World".

In Grace we wrote this as

```
print "Hello World!"
```

There is more overhead in Java, as all Java code must be enclosed in a class:

```
public class HelloJava {

   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

The name of the class is `HelloJava`. It has a single method `main`, which is run automatically when the program is started with this class. Inside the method is a single print statement that prints out "Hello World!".

Let's take this a line at a time. The first line of your program (after any imports) is generally a class header. Classes in Java, unlike those in Grace, do not have parameters. Initialization is generally done through separate "constructor" code, or, if we are writing code using the objectdraw library, using a special `begin` method. In this case we have no instance variables, and hence, nothing to initialize, so we get to skip that.

All identifiers introduced in Java need to have a declared visibility. Classes are virtually always declared to be public, as in this case. The body of the class is contained inside "{" and "}". This trivial class has only the `main` method. If you start executing a class, the system looks for a `main` method with exactly the same declaration as shown. All the magic words are required. It is public, of course. A method is static if it is associated with the class rather than instances of the class. That means you can execute it without having created an object from the class. The `String[] args` parameter is something that we will not generally use, but it allows us to get values from the environment if we desire.

The `println` method is in the library named `System`. It has a (public) object named `out` that can print strings in the output area at the bottom of the Eclipse window. `System.out.println (p)` will print the parameter p and then move to the beginning of the next line, so that any succeeding prints will show up on that next line. If we just write `System.out.println()` with no argument, the system will just move to the beginning of the next line. On the other hand, `System.out.print` will display the value of the parameter, but not move to a new line. Hence the next print will display on the same line.

2 Using objectdraw

Java also supports an objectdraw library — in fact, it predated the Grace objectdraw library by 15 years. The ideas are similar, but the syntax is different. This next program repeats the `Scribbler` program that we wrote early in the semester. Because I am preparing you more for more advanced courses, we are going to use a slightly different approach than regular CS 51. They use something called "Applet"s, which can be embedded in web pages, while I will use applications. Thus if you compare these programs with those written for CS 51 you will see some minor differences involving the use of the `main` method (they don't use one, while we will). Our approach will make it easier for you to see the similarities with Grace programs.

Click on the following link to see a running version of the Scribble program and its text: scribbler demo.

Before we get into details, let's talk about comments. Java has line-oriented comments starting with `//`, just like Grace. However, Java also has multi-line comments that begin with /* and end with */. Anything between that start and end symbol is ignored by the compiler. For example

```
/*  This
      is a
      multi-line
      comment
*/
```

Java also includes special comments that can be picked up automatically to generate documentation about a program (using a tool called Javadoc). These follow the same rules as for multi-line comments but have a few extra embellishments that are used to help the tool. Here is an example:

```
   /**
    * save the current mouse position when mouse pressed as first coordinate of
    * the line
    *
    * @param point -- location of mouse during press
    */
    public void onMousePress(Location point) {
```

JavaDoc comments start with `/**`; successive lines start with a `*`; the last line ends with the usual `*/`. Inside the comment, words that start with `@` have special meaning. An example is `@param` above. The rest of that line gives a description of the method's parameter `point`. Another common keyword used in comments is `@return`, which is generally followed by a description of the value returned by a method.

Programs that use the objectdraw library need two special features:

1. They must import the objectdraw library.

2. The main class must be declared as `extends WindowController`.

Ignoring for the moment the comments at the top of the file, this is done by writing:

```
import objectdraw.*;

public class Scribble extends WindowController {
```

Importing `objectdraw` is similar to declaring a Grace module to be written in the objectdraw dialect. (The `*` means import everything in that library.) The declaration that `Scribble extends WindowController` is similar to Grace's declaration that an object will `inherit graphicApplicationSize...`.

The class `Scribble` has one instance variable, `nextLineStarts`. The declaration is written as:

```
   private Location nextLineStarts;
```

This declares a variable called `nextLineStarts` and says that it has type `Location`, which is similar to Grace's type `Point`. The keyword `private` is a visibility modifier, which in Java precede declarations. For instance variables this modifier is generally `private`, which means that the identifier is visible to all objects of the same *class*, (but *not* to objects of its subclasses). Thus, `private` is incomparable to Grace's `confidential`, which makes a variable visible within a single *object*, and also to objects that inherit from it. Java's `protected` (described below) is a bit closer to Grace's `confidential`.

In general, instance variable declarations consist of a visibility modifier (generally `private`), followed by the type of the variable, followed by the name of the variable and then a `;`. There is no keyword like `var` in Grace that informs the reader that it is a variable declaration. The declaration can also include code to initialize the variable, e.g.,

```
private Location nextLineStarts = someLocn;
```

Notice that an `=` is used to assign values to variables rather than `:=`, as we are used to in Grace.

When writing Java you should use Java's standard naming conventions. Classes and types start with capital letters, like `Location` and `Scribble`. Variables and methods start with lower case letters. We will see later that constants are written in all caps, with _ used to separate words in the name, e.g., `PAUSE_TIME`.

There are two methods in the class `Scribble`: `onMousePress` and `onMouseDrag`. They serve the same purpose as the corresponding methods in Grace. As with variables, there is no Java keyword to indicate that something is a method. Instead you recognize this by the form of the declaration.

```
public void onMousePress(Location point) {
   nextLineStarts = point;
}
```

As with instance variables, methods start with a visibility annotation. There are three options for methods:

- public: Accessible both inside and outside of the class
- protected: Accessible in the class, the package, and in subclasses
- private: Accessible only inside of the class

Public means more or less the same as in Grace, but Java's protected and private annotations aren't comparable to any of Grace's annotations, because they allow access to other objects of the same *class*, whereas Grace's visibility annotations are concerned with access from other *objects*.

Methods declared `private` are typically helper methods used to simplify other code within the class. Protected methods are similar, but are often designed to be overridden in subclasses.

Method `onMousePress` takes a single parameter `point` with type `Location`. Notice that type annotations must appear before the identifier in both variable and parameter declaration. Similarly, the return type of a method always appears immediately before the method name. The type `void` in Java corresponds to `Done` in Grace; it is the return type of a method that performs an action but does not return a value.

Method names in Java are simpler than in Grace in that they are not allowed to have multiple parts. Thus in Java, the complete name of the method occurs before a single list of all the parameters. Methods that have no parameters are required to be terminated by () in Java. Thus one would request the width of a rectangle by writing `myRect.getWidth()`, while in Grace we would have written the somewhat simpler `myRect.width`.

There is no equivalent in Java to Grace's methods whose names terminate with ":=". Instead of having a method `width:=`, you must write a method like `setWidth` that can be used as in `myRect.setWidth(newWidth);`.[1]

Arguments can also be provided when we construct new objects from classes. Classes have special *constructors* that are separate from the class header. We will see these in more detail soon, but for now you should know that new objects are created by applying the `new` operator to a class name and list of arguments. For example, our `onMouseDrag` method for this class is:

```
public void onMouseDrag(Location point) {
   new Line(nextLineStarts, point, canvas);
   nextLineStarts = point;
}
```

The first line of the method constructs a new object from class `Line` with the parameters `nextLineStarts`, `point`, and `canvas`. In Grace we would have constructed this by writing `line.from (nextLineStarts) to (point) on (canvas)`. Notice the same information is provided to the class to initialize the new object, but in Grace, the multiple parts of the method name make it easier for the programmer to remember which argument is required in which position.

[1]Technically, in Java one could just create a public instance variable in Java and then assign to it from where ever you like, however, we consider this *evil* and urge you to avoid doing this and instead write a method.

The second line of the method body updates the value of the instance variable `nextLineStarts` with the value of the parameter `point`. Recall that this update uses `=` rather than `:=`.

Sadly (at least from my point of view), Java does not impose indenting restrictions on the programmer. A statement may be on one line or may drip over to as many lines as necessary without requiring continuation lines to be more indented than the first. So how does Java know when a statement is over? The key is the placement of a `;` after statements. Thus you'll see a `;` after both lines of `onMouseDrag`. However, you do not terminate a statement with a `;` if the semicolon would appear just before or after an opening { or just after a }. Thus the method header above is not terminated with a `;`, nor is the last line, which ends with }. In practice this means that statement that include blocks never have the blocks terminated with a `;`.

I should note that while Java doesn't care if you don't indent your program properly, the TA's and I do! Generally, we won't even look at a program that is not indented consistently, whether to help you or to grade it. I urge you to take advantage of Eclipse's "Format" tool, which is available under the "Source" menu. It doesn't always format your code as you might like it, but it generally does a good job.

Main programs like the one above that use the objectdraw library generally have a `begin` method that is used to initialize variables, but this program is too simple to require one. We'll seen an example later.

The only thing we have not yet talked about is the `main` method at the bottom of the class:

```
public static void main(String[] args) {
   new Scribble().startController(400,400);
}
```

As with the hello world program, this is the method that starts the program. When you tell Java to execute your program as an application, it looks for a `main` method with exactly this signature and then begins executing it. In this case it creates an object from our Scribble class and then immediately sends it a `startController` method request. This is like a combination of our `startGraphics` method that we place at the end of the main program and the `graphicApplicationSize (400 @ 400)` that we inherit from in Grace. In the example above, it creates an object with a 400 by 400 pixel window and the instance variables and methods specified in the class. In this case it will react to mouse presses and drags by drawing straight lines.

3 Pre-defined Data types in Java

The built-in data types in Java are more complex than those in Grace. For example, whereas Grace has only one type to represent numbers (`Number`), Java has at least four. Worse luck, Java has two different kinds of data types: primitive types and object types, which behave differently. The primitive types are included in the language for efficiency. Most of the time the differences won't make much of a difference, but sometimes they are critical. The main difference is that you cannot make method requests on primitive types; they have no methods associated with them. All primitive types are also immutable, but a lot of the object types are immutable as well, so that is not a clear distinguishing factor.

Numeric types. Let's begin with the numeric types. We will focus on just the most common two, `int` and `double`. The type `int` includes all integers, positive, negative, and 0. Their range extends from -2^{31} to $2^{31} - 1$. (If you need to represent larger numbers, use the type `long`, which has the same operations, but holds more values.) Elements of type `int` are written as in Grace and use the same arithmetic operations: +, -, *, /, and % (for the"mod" or remainder operator). The only significant difference from Grace is the meaning of the divide operator `/`. It gives you an *integer* result, the same as you would get by doing long division. Thus 17 / 3 gives an answer of 5. The remainder is what you get from %, so 17 % 3 is 2. Integer arithmetic operations always return integer values!

There are four other operations on integer variables `n` that are useful: `n++`, `++n`, `n--`, and `--n`. The first two are used to increment the value of n by 1. Thus, if `n` has the value 4 before the operation, then it has the value 5 after executing the operation. The second two are used to decrease the value of `n` by 1. If that is all that is going on, then why have four operations rather than 2?

It turns out that these operations also return a value as well as update `n`. The operations `++n` and `--n` return the value of `n` *after* the update has completed, while `n++` and `n--` return the value of n *before the update*. Thus if `n` has value 4, then `++n` updates `n` to 5 and returns 5. On the other hand, `n++` updates the value of `n` to 5, but returns the value 4. Thus if `n` starts out with the value of 4, then `(n++) + n` updates `n` to 5, while returning the value 9, while `++n + n` also updates `n` to 5, but returns the value 10.

To avoid this confusion we strongly recommend that you do *not* embed statements with `++` in the middle of arithmetic expressions. Use them only as stand-alone expressions to update `n`, or in `for` loops (see below).

If you want fractional values, then you should use the type `double`, which includes values with decimal points like 34.125 and -17.0. (The name `double` comes from the fact that it takes twice as much space as the other type that holds numbers with decimal point, `float` — which we won't be using.) Numbers of type `double` are essentially held in scientific notation internally, e.g. 0.34125×10^2. However, you write them, and they are generally printed out, in the standard way without powers of 10. The arithmetic operations work in the usual way for `double`, with one exception: there is no operator % — that is available only on `int`. The results of arithmetic operations on elements of type `double` are generally always elements of type `double`. Thus. the result of evaluating 1.5/0.5 is the double 3.0, not the `int` 3. When you write a constant that you want to be a `double`, you must include the decimal point. If you do include a decimal point, then you must have a digit on both sides. Thus write 3.0 not 3. and write 0.34 rather then .34.

Both `int` and `double` values can be compared with operators `==`, `!=`, `<`, `<=`, `>`, and `>=`.

As a convenience, Java will automatically convert `int`s to `double`s if you are performing "mixed mode" arithmetic, i.e., numeric operations that include both `int`s and `double`s. Thus, to perform 3+2.5, Java will first convert 3 to 3.0 and then do the addition. However, you must be careful, as it will only make the conversion when it needs to. Thus if you evaluate, 75.5*(1/2), the result will be the `double` 0.0. This is because the system first calculates 1/2, which is 0 since Java does integer division. It then multiplies the result by 75.5, giving the final result of 0.0. To force it to do the division as decimal or "floating point" division, you could write it as any one of 1.0/2, or 1/2.0, or 1.0/2.0.

Java will *not* convert automatically in the other direction. You may not assign a `double` value to an `int` variable, nor use a `double` in any other context that expects an `int`. If you want to convert a `double` to an `int`, then you must tell Java by writing `(int)` before the expression. Doing so will truncate the `double`, returning the integer part of the value. Thus `(int)(2.7/.7)` will return the `int` 3. If you wish to round to the nearest `int` instead, use the static `round` method from the Java `Math` library. Thus `Math.round(2.7/.7)` will return the `int` 4.

Boolean type. Java has boolean values like Grace: `true` and `false`. Because they are primitive values rather than objects, you can not make method requests of them, but they do have the usual pre-defined operations !, &&, and ||. The big difference from Grace is that the &&, and || operations are short-circuit: if their final value is determined by the value of the first argument, then they do not bother to evaluate the second argument. Thus, if `bval1` evaluates to false, then the condition `bval2` will not be evaluated in `bval1 && bval2`.

Primitive data types in Java are written with their initial letters in lower case. Thus, we write `int`, `double`, and `boolean`. There are corresponding object types `Integer`, `Double`, and `Boolean`, but we will not be using them.

String type. The type `String` is built-in for Java and is an object type, so the type name starts with a capital letter and you can request methods on Strings. As in Grace, strings are immutable, so these methods return new strings rather than modifying the receiver. It has many of the methods of `String` in Grace, as well as many more. A complete listing of the methods is available in Java API's. Strings may be specified by surrounding them with double quotes as in Grace. However there is no string interpolation to insert other values into strings. Instead the concatenation operator, + (note this is a single +, not Grace's ++), can be used to glue together strings. For example, if `x` is an `int` variable then the string `"x = {x}"` in Grace would be written in Java as `"x = " + x.toString` or just `"x = " + x`. As you can see in this example, `toString` (rather than `asString`) is used to convert a value to a string. As in Grace, if you concatenate a string with a non-string then the non-string will be converted to a string before concatenating it with the string.

Strings can be compared with the methods `equals` and `compareTo`. The method `equals` returns true if the expressions represent the same sequence of characters. The result of `s1.compareTo(s2)` returns an `int` that is negative if s1 comes before s2 in lexicographic (dictionary) order, 0 if they represent the same string, and a positive number if `s2` comes after `s1` is lexicographic order.

You must be careful about comparing strings. The next section explains why you should never use `==` to compare two strings. You must also be aware that all of the capital letters come before all of the lower case letters (thus `"Z"` comes before `"a"`). If you are interested in ordering, you therefore usually convert strings either to all caps (using method `toUpperCase()` or all lower case (using `toLowerCase` before making the comparison.

Java has a separate character type, written `char`. A character is written with single quotes. Thus `'a'` is a character, while `"a"` is a string of length 1. You can extract a character from a string using []. Thus "`hello[1]`" is the character `'e'`. (Java starts counting at 0 rather than 1!) You can read more about the `char` type by doing a Google search or looking up the information in any Java text.

4 n in Java

Java has two main equality operations, the operator `==` and the method `equals`. Because you cannot send method requests to primitive values, you must use `==` to compare them. It is more complicated with object types, however. The operator `==` on object types has a fixed meaning corresponding to object identity, while the `equals` method can be overridden in a class and is typically defined to mean the two objects have the same value. This is a bit subtle, so let's look at an example.

Suppose `p` and `q` are variables holding locations, constructed as follows:

```
Location p = new Location(1,3)
Location q = new Location(1,3)
```

then `p == q` is false, because they are different objects, even though they have the same x and y components. However, `p.equals(q)` will be true, because we defined the `equals` method in class `Location` to return true whenever the two objects have the same x and y values.

On the other hand, if we define another variable `r` and initialize it with `p`:

```
Location r = p
```

then `r == p`, because `r` and `p` both refer to the same object. Of course, we also have `r.equals(p)` because logically they are the same. In general if `x == y`, where `x` and `y` refer to objects, then we will also have `x.equals(y)`. However the reverse is NOT true.

For example, suppose we have the following Java code:

```
String a = "hello";
String b = "he";
String c = "llo";
System.out.println (a==(b+c));
System.out.println (a.equals(b+c));
```

When executed the first print will show `false` and the second `true`. Confusingly, however, the Java compiler tries very hard to statically determine the values of strings, and rather than creating multiple copies, shares them. This is generally fine because strings are immutable. However it means that the results of `==` are sometimes surprising. For example, if we execute the following:

```
String a = "hello";
String b = "he"+"llo";
System.out.println (a==b);
System.out.println (a.equals(b));
```

then the system will print out `true` for each. That is because the compiler is smart enough to realize that the two initial values are the same, but in the previous example, it was not quite smart enough to realize that `b+c` was the same as `a`.

The bottom line is that almost always the correct comparison operator for objects is the method `equals`, and when you define a new class, always define `equals` to mean the "right" thing (and when you define `equals`, remember to also define `hashcode` – at least once you learn enough to know what that is – e.g., in our CS 62). It is especially important that when you compare strings you NEVER use `==`, but instead use `equals`.

5 Types and Type-checking in Java

Java is a statically typed programming language. This means that when identifiers are introduced they must be declared with an associated type and the compiler will check over the program to make sure the type declarations are consistent with the code. Thus, it will catch at compile time an attempt to associate a string with an identifier declared to be `int`.

When you used the dialect `rtobjectdraw` in Grace, it complained if you did not associate a type with each identifier introduced. However, it did not check for consistency at compile time. It eventually will, but that has not yet been implemented. That will hopefully be in place in the next few months.

You will find that this early detection of errors will help you locate and fix those errors much faster. Java also has a much faster compiler, and will report multiple errors each time you compile your program.

Java differs from Grace in that class names can be used as types. Thus if `Line` is a class, we can declare a variable to have type `Line`:

```
    private Line diagonal = new Line(pt1,pt2,canvas)
```

Java also supports `interfaces`. Interfaces are like Grace's types. They include the specification of public methods (and constants), but no implementation information. We will see them in examples later.

6 Regular classes

Let's take a look at a more complex program, one that involves regular classes rather than just those corresponding to main programs. The program we will look at is a program to drag two Tshirts, again one that we wrote for Grace earlier.

Tshirt Let's start by looking at the Tshirt class.

The class starts with two import statements:

```
import objectdraw.*;
import java.awt.*;
```

Virtually all of our programs will import `objectdraw`, just as in Grace we used the corresponding dialect. We will also need to import the built-in Java libraries `java.awt.*` because class `Color` is defined in that library.

This time our class header does not extend `WindowController`, similarly to how in Grace, classes that were not main programs did not inherit `graphicApplication`.

The program has many constants. In Java, Grace's `def` is spelled `private static final`. You already know what `private` means. The keyword `final` means that the value may not be updated – for variables, you may not assign to it. The `static` is a little funnier in that it means that there is only one copy of the constant made, and it is shared with all instances of the class. It is a way of saving space when your program is running. Generally, that is the right choice when you are creating instance variables and want a constant. However, if you are making a constant inside a method, then you should just use `final` and omit the `static`.

Technically, `static` means that the identifier is associated with the class rather than the objects generated from the class. Grace does not have a corresponding notion. In Grace, by contrast, if we want just a single copy of something we put it in a separate object and share it — kind of like the "empty" objects we used to put in variables that didn't have anything useful in them yet.

The *x* and *y*-coordinates used in the objectdraw library have type `double`. It is OK to use `int`s, as they will be converted to type `double` when necessary, but when you ask an object for a coordinate, it will always return a double.

The instance variables declared in Tshirt are essentially the same as in the Grace Tshirt example. Their types are given by the classes they implement, e.g., `FramedRect`, `FilledRect`, `FramedOval`, etc. Variables that are of object types that are not initialized in their declarations are assigned value `null` by Java. (Identifiers of primitive types are initialized to 0, 0.0, or false, depending on their type.) You can think of the value `null` as being the same as being undefined in Grace. However, you can test the value of a variable to see if it is equal to `null` in Java, while that is not possible in Grace. If you get a "null pointer" error in Java, it usually means that you have forgotten to initialize a variable that you are using.

By the way, variables that are declared in methods are NOT initialized automatically; the program must assign them a value before they are used or Java will complain.

After the variable declarations, you will find the Tshirt constructor:

```
public Tshirt(double x, double y, DrawingCanvas canvas) {
   // create boundary rectangles
   sleeveTrim = new FramedRect(x, y + NECK_HEIGHT / 2,
                                                   SLEEVE_WIDTH, SLEEVE_HEIGHT, canvas);
 ...
```

The constructor's job main job is to initialize the objects created from the class. The parameters that would be included in the class header in Grace are included in the class constructor in Java. The constructor in Java must have the same name as the class (in this case `Tshirt`). Parameters are declared with their types as usual, but constructors are not provided with a return type (because they always return an object of the same type as the class). Constructors are invoked with the keyword `new`:

```
new Tshirt(50,50,canvas)
```

While variables can be initialized in their declarations, many are initialized in the constructor. Any other executable code that would have been included in the Grace class definition (outside of methods) will be found in the constructor in Java.

Methods are defined in ways similar to Grace, but with no keyword `method`. They will generally include the visibility annotation, return type, name, and parameters with their types. Remember there are no multipart method names in Java. Most of the names of methods in the objectdraw library in Java are similar to those in Grace. The most obvious differences are that `moveBy` is replaced by `move` and methods of the form attrib :=(...) are instead written in the form setAttrib. For example, rather than writing box.color := colorGen.red, in Java we write box.setColor(Color.RED). Similarly, rather than writing `point.x` as in Grace, we write `point.getX()` in Java. This brings up the important point that requests to parameterless methods in Java must be terminated with (). Finally, the type `Point` in Grace is written as `Location` in the Java objectdraw library.

Most of the methods in `Tshirt` return `void` (like `Done` in Grace, but `contains` returns a boolean. In Grace, a method that returns a value must always include a `return` before the value. Thus `contains` has

```
return body.contains(pt) || sleeves.contains(pt) || neck.contains(pt);
```

where Grace would simply write

```
body.contains(pt) || sleeves.contains(pt) || neck.contains(pt);
```

as the last line of the method.

Finally, you will notice that the `move` method includes the code

```
   this.move(x - sleeves.getX(), y - neck.getY());
```

The keyword `this` is used in Java in place of Grace's `self`. However it functions in the same way as Grace's `self`, and, like `self`, can be dropped as the receiver of a method request.

Drag2Shirts Let's now take a look at the main program. As usual, it is declared as a class that extends `WindowController`, and it has a `main` method that creates a new object from the class and sends it the `startController` method request.

As usual the class begins with a number of constant (`static final`) and variable definitions. This time, however, we need to do some initialization. While in most classes (like `Tshirt`) the initialization code goes in the constructor, classes extending `WindowController` do not have a constructor. Instead they have a special `begin` method:

```
    public void begin() {...}
```

As with the mouse event handling methods, this method must be declared to be public and must have exactly this header for the system to recognize it when the programs starts up.

With this program, the `begin` method creates new `Tshirt`s that are associated with variables `otherShirt` and `selectedShirt`. It also colors one red and the other green. These last statements are a bit more complex than for Grace. In Java, color names, must be prefixed with the name of the class that created them. (They are public constants of the class, hence their names are in all caps.) Second, rather than being able to write `otherShirt.color := red` as in Grace, Java does not allow programmers to write methods whose name include `:=`. As a result, methods in the Java version of the objectdraw library usually use a prefix of `set` rather than tacking `:=` on the end. Thus we can set the color of our shirt by writing `otherShirt.setColor(Color.RED)`. Other substitutions include `setX` rather than `x:=`, `setWidth` rather than `width:=`, etc.

Finally, recall that assignment statements in Java use `=` rather than `:=` and all statements are terminated with ";".

The objectdraw names for methods obtaining values from an object are also a bit different from Grace. Rather than just listing the name of a component, the style is to write `get` before the name of the component. Thus we write `getX`, `getColor`, `getWidth`, etc. A full listing of the Java objectdraw library methods can be found on the course webpages.

Let's now take a look at the mouse handling methods in Java. These should all look very familiar: `onMousePress`, `onMouseDrag`, etc. The only differences are those impacting all methods in Java – declaring the visibility (public), the return type (void) is written before the method name, the parameter type comes before the parameter name, and the type `Point` is replaced by `Location`.

However, there are also some subtle differences in the way that control constructs are written in Java. For example, keywords after conditions are omitted. Thus we leave out `then` in `if` statements and `do` in `while` and `for` loops. Also, while Grace treated `elseif` as a single word, they must be written as two separate words in Java. Here are examples of `if` and `while` statements:

```
 if (cond1) {
    ...
} else if (cond2) {
   ...
} else {
   ...
}

while (cond3) {
   ...
}
```

While it is legal to drop the { } in an `if` statement or `while` loop if the block includes only a single statement, it can make your program harder to understand so we strongly recommend it. (Recall that Java does not require proper indenting – though I do – so programs laid out poorly on the page generally can't be easily understood without curly braces.) For loops are more complex in Java, so we will discuss those later.

One last thing before we finish our discussion of this program. The method `onMousePress` includes the declaration of a local variable, `tempShirt`. Local variables must be associated with a type, like all other identifiers that are introduced, but they do not have a visibility annotation like `private` or `public`. Because the scope of the local variables (like parameters) is limited to the method they are defined in, they are even more limited than `private` or `public`, so you may not associate one of those other annotations with them.

7 Overloading methods and constructors

Java allows you to have several methods with the same names, but that take different kinds of parameters. A good example is the method name `moveTo` on graphic objects in the objectdraw library. One version takes a `Location`, while the other takes two `doubles`. The `Tshirt` class we just looked at had only one version of `moveTo`, one that took two doubles as parameters. However, we could add a second version that takes a location. Typically when we do this, we make the new versions call the original one like the following:

```
public void moveTo(Location pt) {
   moveTo(pt.getX(),pt.getY());
}
```

You can see that this version takes a location `pt` as parameter and then extracts the `x` and `y` components before calling the original version of `moveTo`.

There is no real advantage to overloading methods like this, but I wanted to warn you about them in case you saw them in code. Notice that I get exactly the same effect if I use a slightly different name for the method

```
public void moveToLocn(Location pt) {
   moveTo(pt.getX(),pt.getY());
}
```

Grace does not support this kind of overloading, as we would rather change the name a bit as above rather than take the chance of confusing programmers. (There are some complications in how the compiler makes a decision on which version of an overloaded method to call.) We urge you not to use overloading, but be prepared to recognize code that uses it.

Finally, we can also override constructors for a class. For example we could include a second constructor for `Tshirt` that takes a location and a canvas as parameters rather than two doubles and a canvas. We could simple add the following code after the first constructor body:

```
public Tshirt (Location pt, DrawingCanvas canvas) {
   this(pt.getX(), pt.getY(), canvas);
}
```

The body of this constructor uses the keyword `this` to call the original constructor of the `Tshirt` class. (I realize that it is confusing that `this` has two different meanings in Java, but at least they both have to do with accessing things in the same class.) If we wanted to do this in Grace, we would have to work a bit harder, but not too much.

In Grace, we defined:

```
class tShirtAt (startPosn: Point) size (size': Number)
                 on (canvas: DrawingCanvas) -> MoreDraggable {...}
```

To get another constructor that uses the same instance variables and methods, but requires different parameters, we could use inheritance:

```
class tShirtAtX (x: Number) y (y: Number) size (size': Number)
                 on (canvas: DrawingCanvas) -> MoreDraggable {
   inherit tShirtAt (x @ y) size (size') on (canvas)
}
```

Because we don't add or override anything from the superclass, we get something with exactly the same instance variables, defs, and methods as the original. Note that we used a slightly different name for the class as Grace will not allow us to write a second class with the same class name as another.

8 For loops

Java has two types of for loops. We will talk about the old style of `for` loop here and then talk about the newer form when we discuss arrays. As we noted earlier in the term, there are a number of `while` loops in Grace that have the same structure:

```
var index: Number := initValue
while (endTest) do {
   // do stuff
   index := index + 1
}
```

In the above, `endTest` is a boolean-valued expression.

Java (and before that C++, and C) introduced a `for` loop that allows a programmer to combine the first two lines and the last in the body into a single looping statement:

```
for (int index = initValue; endTest; index++) {
   // do stuff
}
```

While this looks pretty different from the Grace `for` loop, it serves the same purpose of iterating through a series of elements (indices in this case) and performing the same operations on each.

A simple example is the following loop that prints out the squares of all numbers from 0 to 9:

```
for (int index = 0; index < 10; index++) {
   System.out.println{"The value of "+index+" squared is "+(index * index)
}
```

The body of this loop is executed for the numbers 0, 1, 2, ..., 9, and then the loop terminates.

The print statement in the loop has a couple of interesting points. It concatenates a string, the value of index, another string, and the square of the index, with the result of the concatenation printed. When a value of another type is concatenated with a string, the system attempts to convert that value to a string and then concatenate it. Primitive types like `int`, `double`, and `boolean` have a built-in operation that does this conversion. Object types invoke their method named `toString`. As a result, you should implement a `toString` method in classes that you write. If you don't include it, a default (and very unhelpful) version will be inherited from the top of the class hierarchy, `Object`. Java does not have String interpolation like Grace. That is, I could *not* write the above as you would in Grace: `"The value of {index} squared is {index * index}"`.

Another point that will come up later is that in Java, like C and C++, we typically start counting at 0 rather than 1. Thus you will often see for loops where the index starts out as 0 rather than 1. If that is the case and you want it to go throughout the loop exactly `n` times, then you must have the termination condition by `index < n` rather than using `<= n`.

The indices in the for loops can start at any value you like as well as go up by any number each time through the loop. They can also have a more complex termination condition. Thus the following loop counts down from n to 1:

```
for (int index = n; index >=1; index--) {
   ...
}
```

while the following loop increases through the even numbers, terminating when i exceeds 50 or variable `gameOver` becomes true.

```
for (int index = 2; index <= 50 && !gameOver; index:= index + 2) {
   ...
}
```

9 Interfaces in Java

Most of the time programmers in Java use class names as types. However there are circumstances where it is very useful to have a type that is not associated with implementations. For example, we wrote a variation of the laundry program that held both Tshirts and Pants. If we had used either Tshirt or Pants as the type of the laundry object, it could not have been associated with both Tshirts and Pants. Instead we define an interface in Java:

```
import objectdraw.*;
import java.awt.*;

/**
 * Interface for a piece of laundry of an arbitrary shape.
 * @author kim
 *
 */
public interface Laundry {

    /**
     * Move the laundry relative to its current position
     * @param xOffset - distance to move horizontally
     * @param yOffset - distance to move vertically
     */
    void move(double xOffset, double yOffset);

    /**
     *  Move the laundry to a specific position:
     * @param x -- coordinates to move laundry to
     * @param y
     */
    public void moveTo(double x, double y);

    /**
     * Determine if pt is in laundry item
     * @param pt -- location wish to determine if in laundry
     * @return true if pt is in laundry
     */
    public boolean contains(Location pt);

    /**
     *  Set the color of the item:
```

```
     * @param newHue -- new color for item
     */
    public void setColor(Color newHue);

    /**
     * return color of the item
     * @return the color of the item.
     */
    public Color getColor();

    /**
     *  Remove the item from the canvas
     */
    public void removeFromCanvas();
}
```

Note that everything listed in an interface MUST be public. As a result, you can drop the use of public.

If we want a class to be used in a context expecting an element of the interface, we must declare that the class implements the interface in the class header:

```
public class Pants implements Laundry {
```

The statement that the class implements an interface is a promise that the class has everything promised by the interface. Java uses what is called a *nominal* type system, meaning that if you don't declare a class implements the interface, then you can't use it as an object of the interface even if it has all the features required. Grace, on the other hand, uses a *structural* type system, meaning that if an object has all the features required by the type then you can use it as an element of the type.

I like to use interfaces because they allow me to change the definitions of classes without tying me to a particular implementation. As we'll see, they are required in certain frameworks for GUI components.

One last thing about interfaces. A class can implement as many interfaces as it likes. Just separate them with commas in the class declaration:

```
    public class C implements A, B, C {
       ...
    }
```

We'll see later though that classes can only extend one other class.

10 Arrays

Note that the chapter on arrays from the Java version of our text is available on-line at *http://eventfuljava.cs.williams.edu/chapters/Bruce_chapter14.pdf. It goes into much more detail than we can here.*

Java has a class `ArrayLIst<T>` that is similar to Grace's lists. However, it also has a more primitive data structure, arrays, that form the building blocks of `ArrayList.` Here we are going to focus on arrays as it will provide you good intuition on the complexity of list operations. You can look up information about `ArrayList` is the documentation on Java libraries.

Arrays differ from Grace's list in several important ways. The most important is that arrays have a fixed size that is determined when the array is created. Another is that the operations on an array are limited. You can ask for an element at a given location, update an element at a location, or ask the array for the number of slots it has. However, more complex operations must be hand-coded by the programmer. Finally, the elements of an array are indexed starting at 0 rather than 1 like Grace.

Array types and expressions in Java use a special syntax. If `T` is a type then `T[]` is the type of arrays holding elements of type T. Similarly, a new array of `T` of size n is constructed by writing new T[n]. Thus we can declare and initialize an array `a` of `double` with 20 slots by writing:

```
double[] a = new double[20]
```

We access the elements of an array using a similar notation to refer to individual elements. Thus we can obtain the values of the first three elements of the array `a` by writing `a[0]`, `a[1]`, and `a[2]`. (Remember that we start numbering the array elements at 0, not 1.)

We can update the elements using a similar notation:

```
a[0] = 7.47
```

Because you must allocate all the space for an array when you create it, very often the number of active elements in an array is smaller than the total number of slots. If `a` is an array, then `a.length` will return the total number of slots in the array. Thus if `a.length` returns 20, then the array has elements `a[0]` through `a[19]` (because we start counting at 0, the last slot is always at one less than the length). By the way, notice that there are no open and closed parentheses after the method request of `length`. That is because `length` is a public instance variable of the array class. We think that is terrible style (if you want an instance variable to be public then you should write a method that returns its value – in Grace when you declare a variable to be `readable`, the system automatically generates such a method for you – and if you declare it to be public the system generates both "getter" and "setter" methods.

The fact that `length` returns the number of slots as opposed to the number in use makes the programmer's task more complicated. For example, think back to the `song` class in our Simon program in Grace. Here is a copy of the first part of my `song` class (omitting some of the methods, for example the `play` method).

```
class songFrom(buttonList: List[[NoisyButton]]) -> Song {

  // list of notes (buttons) in song
  var noteList: List[[NoisyButton]]

  // which note the user is about to play
  var currentNote: Number

  // create a new song
  newSong

  // randomly select one of the buttons
  method randomNote -> NoisyButton is confidential {
    // randomly pick a button from buttonList -- code elided
  }

  // create a new song with one note
  method newSong -> Done {
    noteList := list[[NoisyButton]] [randomNote]
    currentNote := 1
  }

  // Add a note to the song
  method addNote -> Done {
    noteList.add(randomNote)
    currentNote := 1
  }
```

```
   // what button should user play next?
  method expectedButton -> NoisyButton {
    noteList.at(currentNote)
  }

  // Move on to next note in song
  method advanceToNext -> Done {
    currentNote := currentNote + 1
  }

  // Has last note been played?
  method atEnd -> Boolean {
    currentNote == noteList.size
  }
```

Let's see how we would have to write this in Java:

```
public class Song {
   // the maximum possible length of a song
   private static final int MAX_SONG = 100;

   // array of notes (buttons) in song
   private NoisyButton[] theSong;

   // which note the user is about to play
   private int currentNote;

   // the length of the current song
   private int length;

   private ButtonPanel theButtons; // "notes" that can be selected for song

   // creates an initial song of one note.
   public Song(ButtonPanel someButtons) {
      // save the parameter in an instance variable
      theButtons = someButtons;

      // create a new short song
      theSong = new NoisyButton[MAX_SONG];
      makeNewSong();
   }

   // creates a new song of one note.
   public void makeNewSong() {
      length = 0;
      current = 0;
      addNote();
   }

   // adds a note to the end of the song.
   public void addNote() {
      if (length < MAX_SONG) {
         theSong[length] = theButtons.getRandomButton();
         length++;
         current = 0;
      }
   }

      // returns the current note in the song.
   public NoisyButton getExpected() {
      return theSong[current];
   }

   // moves to the next note of the song.
   public void next() {
      if (current < MAX_SONG - 1)
         current++;
   }

   // returns whether currently at the end of the existing song.
   public boolean atEnd() {
      return (current == length - 1);
   }
}
```

Let's make some comparisons between the code. We start with those directly relevant to arrays and lists.

In the Java version, the array `theSong` was declared as an array of `NoisyButton`. The program also includes the constant `MAX_SONG` which will be the size of the array. It also includes the declarations of the two instance variables `currentNote` and `length`, referring to where we are in the song and how many notes are currently in the song.

In the Grace version, we declare `theSong` as a list of `NoisyButton`, but don't need a constant for the maximum size of the list or a variable `length` for the current length of the song. We can always as `theSong` for its `size` if we want to know how many elements are currently in the song. Similarly, we do not have to specify a maximum size of a song as the list will expand to be as large as is needed.

In Java the array is created in the constructor. It has size `MAX_SONG`, then `makeNewSong` is requested to initialize the song to have exactly one note. Looking at the method `makeNewSong`, we see that it sets length and current to 0 (remember the 0th element is the first in an array) and then calls `addNote` to add the first note and bump up the `length` by one.

In Grace, the class initialization code in the class body calls `newSong`, which creates a new array with one button in it and then sets `currrentNote` to 1 (the first element in a list).

The Java constructor takes `someButtons` as a parameter. This can be used to select random buttons for the song, just as the class parameter `buttonList` permits the selection of random buttons (the code doing the selection is omitted in both cases). A major difference between Java and Grace is that the parameter to the constructor in Java is only visible in the constructor code, just as parameters are for methods. As a result if a constructor parameter is needed in other methods, then it will need to be stored in an instance variable. Thus we see the extra code in the constructor to save `someButtons` in the instance variable `theButtons`. This is not necessary in Grace, because the parameter to the class is visible throughout the entire class. Thus there is no instance variable needed to store the parameter `buttonList` in Grace. All class parameters are visible throughout their class definition.

Notice that we did not need to store the constructor parameters in our `Tshirt` example because they were only used in the body of the constructor. Only declare instance variables to store constructor parameters if they are needed in other methods of the class.

As an aside, some programmers like to use the same name for the constructor parameter as for the instance variable. Java allows you to do that if you prefix the name of the instance variable by `this`. Thus if we had used `theButtons` as the name of the parameter to the constructor then we could initialize the instance variable by writing `this.theButtons = theButtons`. The occurrence of `theButtons` on the right side of the assignment corresponds to the parameter because it is the nearest declaration, while writing `this.theButton` makes it clear that the programmer is referring to the `theButton` the is part of the object, and parameters are not part of an object in Java.

Let's now look at `addNote`. In Java, it must first check that there is space for the new element, add it in the first empty slot (do you see why theSong[length] is the first empty slot?), increase length, and then reset current to 0. In Grace, we can use the `add` method to add the new note to the end of the list, and reset `currentNote` to 1 (the beginning slot in the list). We do not have to worry about whether their is space for it (there always will be), or change `length` (as `noteList.size` will always keep track of it).

The other corresponding methods are pretty similar to each other. However, note that `advanceToNext` in Java requires a test to ensure that we are not off the end of the array, while `atEnd`'s test for the end is a bit more intuitive in Grace because the last element is at `noteList.size`.

Hopefully this comparison helped you understand the differences between lists in Grace and arrays in Java. But let's now take a look at how we write some of the methods for lists in Grace that are not available in Java for arrays.

Let's look first at adding a new element to an array. For simplicity, let's assume that we are working with an array `seq` of type `SomeThing` and instance variable `size` keeps track of the number of elements in `seq` that are in use. We assume that `size <= seq.length`.

Adding a new element to the end of `seq` is pretty straightforward:

```
public void addLast(SomeThing newElt) {
   if (size < seq. length) {
      seq[size] = newElt;
      size++;
   }
```

However, adding an element to the beginning of a list is much trickier. To add the new element we need to make a space by shifting all of the other elements over by 1.

```
public void addFirst(SomeThing newElt) {
   if (size < seq. length) {
      // shift over existing elements
      for (int index = size-1; index >= 0; index--) {
         seq[index+1] = seq[index]
      }
      seq[0] = newElt;
      size++;
   }
```

Exercise Why do we shift the last element to the right first, rather than shifting the element at 0 first?

Exercise Explain in detail what would happen if you made the horrible mistake of replacing the `+1` in the assignment statement in the `for`loop by.

```
         seq[index++] = seq[index]
```

Explain why these horrible things happened and write on the board 100 times "I will never use `++` in the middle of an expression!".

To polish these skills let's look back at the `DrawingList` program in Grace and the corresponding program `DrawingArray` in Java. Here is a link to the Grace program. Take a quick look over it to refresh your memory and now we'll take a look at the equivalent Java program.

```
import java.awt.*;
import javax.swing.*;

import objectdraw.*;

public class Drawing extends WindowController {

   // max number of objects to be displayed
   private static final int SIZE = 100;
   private static final int MAX_OBJECTS = 20;

   // menus for shape, color, and command
   private JComboBox<String> shapeChoice;
   private JComboBox<String> colorChoice;
   private JComboBox<String> commandChoice;

   // Array of objects on screen.
   private DrawableInterface[] shapes = new DrawableInterface[MAX_OBJECTS];

   // number of objects on screen
   private int numShapes = 0;

   // item currently selected for dragging
   private DrawableInterface selected;

   // mouse loc'n when last handled mouse
   private Location lastPoint;

   /**
    *  Set up GUI components for program
    */
   public void begin() {
      // create panel to hold choice buttons
      JPanel menuPanel = new JPanel();

      // menu for selecting or adding
      commandChoice = new JComboBox<String>();
      commandChoice.addItem("Add new item");
      commandChoice.addItem ("Recolor item");
      commandChoice.addItem("Move item");
      commandChoice.addItem("Delete item");
      menuPanel.add(commandChoice);

      // Set up menu for shapes
      shapeChoice = new JComboBox<String>();
      shapeChoice.addItem("Circle");
      shapeChoice.addItem("Square");
      menuPanel.add(shapeChoice);

      // Set up menu for colors
      colorChoice = new JComboBox<String>();
      colorChoice.addItem("Red");
      colorChoice.addItem("Green");
      colorChoice.addItem("Blue");
      menuPanel.add(colorChoice);

      // Add the panel to screen
      getContentPane().add(menuPanel, BorderLayout.SOUTH);
      validate();
   }

   /**
    *  When the user clicks in the canvas, check the settings of the command
    *  menu to determine what action to take.
    */
   public void onMousePress(Location point) {
      selected = null; // indicate nothing currently selected
      Object buttonLabel = commandChoice.getSelectedItem();
      if (buttonLabel.equals("Add new item")) {
         addNew(point);
      } else if (buttonLabel.equals("Recolor item")) {
         recolorShapeAt (point);
      } else if (buttonLabel.equals("Move item")) {
         selectShapeAt(point);
      } else {
         deleteShapeAt(point);
      }
   }

   /**
    * This method implements the "Add new item" command. Add new geometric
    * shape where clicked. Type and color of object is determined by the
    * settings of the color and shape menus.
    */
   private void addNew(Location point) {
      // only add if still room for more objects
      if (numShapes < MAX_OBJECTS) {
         Location centeredLocation = new Location (point.getX() - SIZE/2,
                                 point.getY() - SIZE/2);
         Object shapeString = shapeChoice.getSelectedItem();
         DrawableInterface newShape;

         // create new object to be shape chosen
         if (shapeString.equals("Square")) {
            newShape =
               new FilledRect(centeredLocation, SIZE, SIZE, canvas);
         } else {
            newShape =
               new FilledOval(centeredLocation, SIZE, SIZE, canvas);
         }

         newShape.setColor(getSelectedColor());
         shapes[numShapes] = newShape;
         numShapes++;
      }
   }

   /**
    * @return the color corresponding to the string selected in the color menu.
    */
   private Color getSelectedColor() {
      Color objectColor; // local variable - color of object

      // get color showing in the color menu
      Object colorString = colorChoice.getSelectedItem();

      // set objectColor to chosen color
      if (colorString.equals("Red")) {
         objectColor = Color.RED;
      } else if (colorString.equals("Green")) {
         objectColor = Color.GREEN;
      } else {
         objectColor = Color.BLUE;
      }
      return objectColor;
   }

   /**
    * Change the color of the item the user clicks on.
    * This implements the "Recolor Item" command.
    */
   private void recolorShapeAt(Location point) {
      int selectIndex = getIndexOf(point);

      if (selectIndex != -1) {
         shapes[selectIndex].setColor(getSelectedColor());
      }
   }

   /**
    * @param point location of interest
    * @return the index of the last element of shapes containing point
    *         return -1 if no element of shapes contains point
    */
   private int getIndexOf(Location point) {
      // Walk the array until we find the selected shape
      for (int selectIndex = numShapes - 1; selectIndex >= 0; selectIndex--) {
         if (shapes[selectIndex].contains(point)) {
            return selectIndex;
         }
      }
      return -1;
   }
```

```
/**
 *  Remove top-most geometric item clicked in.  If didn't click in any
 *  then don't do anything.
 * @param point location where user clicked.
 */
private void deleteShapeAt(Location point) {
   int selectIndex = getIndexOf(point);

   // if point is in one of the objects, delete it
   if (selectIndex != -1) {
      shapes[selectIndex].removeFromCanvas();
      removeEltWithIndex(selectIndex);
      shapes[numShapes] = null;
   }
}

/**
 *  remove shapes[index] by moving later elements back.
 * @param index: element to be removed
 */
private void removeEltWithIndex(int index) {
   for (int objNum = index; objNum < numShapes - 1; objNum++) {
      shapes[objNum] = shapes[objNum + 1];
   }
   numShapes--;
}

/**
 *  Set select to indicate top-most geometric item clicked in, and
 *  send it to front of screen.  If didn't click in any then
 *  set select to null.
 *  Update lastPoint to last place clicked so can drag it
 * @param point: where user clicked
 */
private void selectShapeAt(Location point) {
   int selectIndex = getIndexOf(point);

   // Remember which shape is selected so onMouseDrag can move it.
   if (selectIndex != -1) {
      selected = shapes[selectIndex];
      lastPoint = point;

      // Move the selected object to the front of the display
      // and to the end of the array if it is not already there
      if (selectIndex != numShapes-1) {
         selected.sendToFront();
         removeEltWithIndex(selectIndex);
         shapes[numShapes] = selected;
         numShapes++;
      }
      }
   }

   /**
    *  If something was selected then drag it and remember where left off
    */
   public void onMouseDrag(Location point) {
      if (selected != null) {
         selected.move(
            point.getX() - lastPoint.getX(),
            point.getY() - lastPoint.getY());
         lastPoint = point;
      }
   }

   public static void main(String[] args) {
      new Drawing().startController(400,400);
   }

}
```

Let's talk through this program. For now, I'm going to skip the GUI component set up, which is handled in the begin method. Instead just trust that it also creates and installs three pop-up menus (called `JComboBoxes` in Java) at the bottom of the window, below the canvas.

The identifier `shapes` is declared as an instance variable that holds values of type `DrawableInterface`, an interface that is satisfied by all of the graphical objects in the objectdraw library. It is initialized with capacity to hold 20 objects. When the array is created the 20 slots all hold the value `null`, which represents an uninitialized value. You can test a value to see if it is null (e.g., `shapes[4] == null`), but if you send a method request to `null`, you will get a `null pointer error`, which generally happens when you forget to initialize a value.

Other instance variables keep track of the number of shapes on the screen (`numShapes`), recall the item currently selected for dragging (if any), and the location where the mouse was when the last event was triggered (`lastPoint`.

The `onMousePress` method determines what action is to be taken by checking the item selected in the commandChoice test field. The actual actions taken are the responsibility of the methods `addNew`. `recolorShapeAt`, `selectShapeAt`, and `deleteShapeAt`.

The method `addNew` creates a new geometric object on the screen, guided by the values selected in the pop-up menus. Based on what is showing on the `shapeChoice` menu, it will create a square or circle. It will then set its color according to the value returned by method `getSelectedColor()`, which returns the color corresponding to the string showing on the `colorChoice` menu. Once the object has been created and its color set, it is added to the `shapes` array at the end, and `numShapes` is incremented.

We now have several `private` methods to help us write the methods to change color, delete, and move objects. Method `getSelectedColor` just grabs the text showing on the color menu and using an `if-else-if-else` returns the corresponding color. We'll talk about getting strings from menus later.

The method `getIndexOf` is a helper method (and is hence declared to be private). It determines the index of the top-most element in `shapes` that contains a point. It is very similar to the corresponding Grace method `indexOf`. They differ only in the syntax of the for loop.

Method `recolorShapeAt` uses `getIndexOf` to determine what object was clicked on. If there was on object clicked on then its color is changed. Method `deleteShapeAt` is similar, but with one twist. The selected element is removed from the canvas, but now we must remove it from the array. Unfortunately, there is no remove method for arrays (in fact, only `length` is defined). Thus we have to write our won. This is done in method `removeEltWithIndex`. The idea here is that we start with the element one to the right of the one we wish to remove and shove the elements to the left by one slot. Hence the assignment `shapes[objNum] = shapes[objNum + 1];` that goes through all the elements from index to the end. After all the elements have been shifted, we cut back the number of elements by 1.

Method `selectShapeAt` finds the object selected, moves it to the front of the screen and then moves it in the array to the last element (which makes it the top element in terms of searches). The element is moved to the last element of the array by first removing it from the array (leaving the array one element shorter) and then adding it as the new last element (which is easy) and bumping up `numShapes` back to the original value.

Method `onMouseDrag` is as before, except that the name of the method to shift an element is `move` rather than `moveBy`.

11 Multi-dimensional arrays

Multi-dimensional arrays in Java are a simple extension of one-dimensional arrays, formed by just adding another subscript slot. For example if we were to rewrite the TicTacToe program in Java, we would declare the board as:

```
// The grid contents
private int[][] marks = new int[NUM_ROWS][NUM_ROWS];
```

The type `int[][]` tell us that it is a two dimensional array holding `int`s. The construction expression `new int[NUM_ROWS][NUM_ROWS];` actually creates the array with `NUM_ROWS` rows and columns.

The check to see if a particular row is a win for one of the players could be written:

```
for (int col = 0; col < NUM_COLS; col++) {
     if (marks[row][col] != matchMark) {
          return false;
      }
  }
```

12 GUI components

Java has GUI components like those in Grace, though associating actions with components requires a few more steps.

Creating and adding GUI components to the screen To install and use a GUI component you must:

1. Declare a variable of the appropriate type and initialize it with the component.
2. Install it somewhere in the window using a layout manager
3. If you want your program to react to a use of the component, then you must associate a “listener” with the GUI component and write an appropriate method to perform the action.

Adding graphic user interface (GUI) items to the screen is pretty easy. Here are the steps:

1. Create the item and initialize it if necessary. E.g.,

   ```
   figureMenu = new JComboBox<String>();
   figureMenu.addItem("FramedSquare");
   figureMenu.addItem("FramedCircle");
   figureMenu.addItem("FilledSquare");
   ```

2. Add the items to the window, and validate the pane. E.g.,

```
      add(figureMenu, BorderLayout.SOUTH);
      add(colorMenu, BorderLayout.NORTH);
      validate();
```

Let's look at a simplified drawing program to see a few of those steps. DrawingProgram.
First we declare the `JComboBox` (pop-up menu):

```
// The menu determining which kind of object is drawn
private JComboBox<String> figureMenu;
```

Then initialize it in the begin method:

```
public void begin() {
   // Create menu for selecting figures
   figureMenu = new JComboBox<String>();
   figureMenu.addItem("FramedSquare");
   figureMenu.addItem("FramedCircle");
   figureMenu.addItem("FilledSquare");

   add(figureMenu, BorderLayout.SOUTH);
   validate();
   ...
}
```

The first statement creates the menu. The next three add entries to the menu. The `add` statement adds the `figureMenu` to the bottom of the screen. The last statement tells Java to arrange all the pieces nicely in the window.

Let's talk briefly about layout managers. The main window generated by WindowController uses `BorderLayout`. A window with `BorderLayout` has 5 slots that can hold items. They are the NORTH, SOUTH, EAST, WEST, and CENTER. There are public constants from the BorderLayout class referring to each of those. You can refer to them by prefixing the name with `BorderLayout`, e.g., `BorderLayout.SOUTH`.

The add method when you have border layout takes two parameters, one with the component, while the second has where to put the item. (See the code above.) When your class extends `WindowController` the CENTER portion of the screen will be filled with the drawing canvas.

Each of the slots in the window can hold only one item. Thus if you put the `figureMenu` is the south slot, then other GUI items must be put elsewhere. Luckily, Java has containers that can hold multiple items and they can be put in one of those slots. The containers in Java are generated by class `JPanel`. Adding a bit to the complexity, `JPanel`s use a different layout manager than our `WindowController`. It uses `FlowLayout` rather than `BorderLayout`. When you add items to a container using `FlowLayout` it simply adds them to the container from left to right and centers them. Thus, to add an item to a `JPanel`, we use an `add` method that only needs a single parameter, the item to be added to the panel.

The first drawing program that we described when explaining arrays used a `JPanel` to place three combo boxes in the south part of the window.

```
// menus for shape, color, and command
private JComboBox<String> shapeChoice;
private JComboBox<String> colorChoice;
private JComboBox<String> commandChoice;
```

The begin method then created the menus, put them in the `JPanel`, and then added the `JPanel` to the window in the south:

```
public void begin() {
   // create panel to hold choice buttons
```

```
        JPanel menuPanel = new JPanel();

        // menu for selecting or adding
        commandChoice = new JComboBox<String>();
        commandChoice.addItem("Add new item");
        commandChoice.addItem ("Recolor item");
        commandChoice.addItem("Move item");
        commandChoice.addItem("Delete item");
        menuPanel.add(commandChoice);     // Note only a single parameter -- no direction!

        // Set up menu for shapes
        shapeChoice = new JComboBox<String>();
        shapeChoice.addItem("Circle");
        shapeChoice.addItem("Square");
        menuPanel.add(shapeChoice);         // Note only a single parameter -- no direction!

        // Set up menu for colors
        colorChoice = new JComboBox<String>();
        colorChoice.addItem("Red");
        colorChoice.addItem("Green");
        colorChoice.addItem("Blue");
        menuPanel.add(colorChoice);        // Note only a single parameter -- no direction!

        // Add the panel to screen
        add(menuPanel, BorderLayout.SOUTH);  // Two parameters because adding to WindowController
        validate();
    }
```

Java supports more kinds of GUI components than just `JComboBox`. A relatively compact introduction to the most popular components and how to use them can be found at http://www.cs.pomona.edu/classes/cs051/handouts/SwingGUICheatSheet.html. These include `JButton` (corresponding to `Button` in Grace), JLabel (corresponding to `TextBox` in Grace), `JSlider`, `JTextField` (corresponding to `TextField` in Grace), and `JTextArea`.

We will use `JButton` in examples below. We construct a button via `new JButton(s)` where `s` is the label on the string. Methods `getText` and `setText(newS)` are available to retrieve and update the label.

Adding Listeners to GUI components As in Grace, if we wish to have a GUI component react to a user selection then we must associate an action with it. Here is the handler for Grace that we wrote for a clear button:

```
    clearButton.onMousePressDo {mevt:MouseEvent ->
       canvas.clear
    }
```

We used a method like `onMousePressDo` to associate an action with the `clearButton`. That action took a `MouseEvent` parameter `mevt` and then cleared the canvas.

For pop-up menus (items of type `Choice`), we used the method `onChangeDo`:

```
 // when user select new color, change color of newShape
  colorMenu.onChangeDo{ evt: Event ->
    match(colorMenu.selected)
      case {"red" -> newShape.color := red}
      case {"blue" -> newShape.color := blue}
      case {"green" -> newShape.color := green}
      case {"yellow" -> newShape.color := yellow}
  }
```

Until very recently, Java did not allow us to pass blocks like these to methods, so it developed a different mechanism for associating actions with GUI components. It associated `Listener` objects with GUI components. These listener objects had methods that could respond to user actions. For example, `JButton` objects are prepared to add `ActionListener`s. The interface ActionListener has a single method with the following header:

```
void actionPerformed (ActionEvent e)
```

If `lisOb` is an object of type `ActionListener` then we associate it with `clearButton` by writing `clearButton.addActionListener(lisOb)`. Once this has been done, any time the user clicks on `clearButton`, the system will make a method request of `actionPerformed` on `lisObj` – resulting in execution of that object's code for the method.

Different GUI items have different kinds of listeners. For now we will only focus on responses to buttons, pop-up menus, and text fields, and they all are associated with the same kind of listener, `ActionListener`. Read the GUI cheat sheet documentation for more information on the different kinds of listeners associated with different GUI items. More information is available in the documentation for the standard Java libraries.

For simplicity here, we will always have the main program (the class extending `WindowController`) act as the listener for all components. Thus to set up the main program to handle events generated by the user pressing a button, we must do the following:

1. Add `this` as a listener to the GUI item. E.g.,

   ```
   colorMenu.addActionListener(this);
   ```

2. Add a declaration that the class implements the appropriate listener interface. E.g.,

   ```
   public class DrawingProgram extends WindowController
                               implements ActionListener { ... }
   ```

3. Add the method promised by the listener interface:

   ```
   public void actionPerformed(ActionEvent event) {
      ...
   }
   ```

Let's see what this looks like in practice. We are going to write a Java program that is equivalent to our drawingProgram in Grace. The complete Java program can be found here.

This program has two pop-up menus, only one of which, the color menu, responds to user selections. The other one is only consulted when the user clicks on the screen. The color menu is declared with the other instance variables at the top of the screen:

```
// The menu determining which color the object should be
private JComboBox<String> colorMenu;
```

The `begin` method contains the code to initialize it and put it on the screen

```
public void begin() {
   ...
   // create menu for selecting colors
   colorMenu = new JComboBox<String>();
   colorMenu.addItem("Red");
```

```
      colorMenu.addItem("Blue");
      colorMenu.addItem("Green");
      colorMenu.addItem("Yellow");

      colorMenu.addActionListener(this);   // this object now notified of any selections on menu
       ...
      add(colorMenu, BorderLayout.NORTH);  // add menu to north side of window.
       ...
```

For it to be legal to use `this` as the listener, we need it to implement `ActionListener`. This requires us to implement the method `actionPerformed`, but also requires us to tell Java that we want this class to be an ActionListener:

```
public class DrawingProgram extends WindowController
                            implements ActionListener {...}
```

Remember Java will not believe a `DrawingProgram` is of type `ActionListener` unless we tell it so in the class header.

In the following we have the implementation of method `actionPerformed`. Because of this code, when the user makes a selection from the color menu, the method will first make sure `newShape` (the last object put on the screen) is different from `null` (i.e., has a value associated with it) and then calls the `private` helper method `setColorFromMenu` to change the color of `newShape`.

```
   /**
    * Select a new color for the last object drawn
    * @param event - contains information on object generating the event
    */
   public void actionPerformed(ActionEvent event) {
      if (newShape != null) {
         setColorFromMenu();
      }
   }

   /**
    * Change the color of the newest shape on canvas to new value of
    * the color menu.
    */
   private void setColorFromMenu() {
      Object colorChoiceString = colorMenu.getSelectedItem();
      if (colorChoiceString.equals("Red")) {
         newShape.setColor(Color.RED);
      } else if (colorChoiceString.equals("Blue")) {
         newShape.setColor(Color.BLUE);
      } else if (colorChoiceString.equals("Green")) {
         newShape.setColor(Color.GREEN);
      } else if (colorChoiceString.equals("Yellow")) {
         newShape.setColor(Color.YELLOW);
      }
   }
```

We didn't use the `event` parameter to `actionPerformed`, but in general it can be used to determine, for example, the source of the event. This is necessary in Java because one listener can be listening to many different GUI components, so we sometimes need to figure out which one. See the description of `ActionEvent` in the Java documentation for a listing of all methods.

`JTextBox` objects also use `ActionListeners`.

Thus the main differences between handling user events on GUI items in Grace and Java are:

1. It is a bit more complex to put an item in the window because the layout is more elaborate in Java. It also offers more fine control of the layout than Grace does.
2. Rather than associating an action directly with a GUI item, Java associates a listener with the GUI item. That listener must then have a method of the right form (e.g., `actionPerformed`) to handle the event.
3. Because Java has a nominal type system, you must declare in the header of the listener class that it implements the appropriate kind of listener, e.g. `ActionListener`. (In our case, we will always use the main program as the listener, so the class extending `WindowController` will declare that it implements the appropriate listeners.)

13 Simple Java I/O

Java has a fairly sophisticated I/O system that allows you to read and write across the internet as easily as reading and writing to the console or a local file. However, there is also a simple implementation of I/O that will serve for most purposes.

At the beginning of this tutorial we discussed `System.out.print` and `System.out.println`. They will output their parameter (and if using `println`, start a new line afterward). In the rest of this section we discuss the `Scanner` class.

To hook up console (keyboard) input to your program, begin by creating a scanner:

```
Scanner scan = new Scanner(System.in);
```

`System.in` is the Java name for keyboard input. If instead you want to read from a file on your hard drive (in the same folder as your program), you can create a scanner for the file `myText.txt` with

```
Scanner scanfile = new Scanner(new File("myText.txt"));
```

Scanner objects have methods to read `int`s, `double`s, `boolean`s, `String`s and more. Here are corresponding method specifications:

```
int nextInt()
double nextDouble()
boolean nextBoolean()
String next()
String nextLine()
```

For example, `scan.nextInt()` will return the next int typed into the console. Generally the scanner breaks the input into *tokens*, which are strings separated by white space (e.g., spaces, tabs, or returns) in the input. Thus we can read (and then print) an int followed by a String with

```
System.out.println(scan.nextInt());
System.out.println(scan.next());
```

You can test to see if there is another token in the input with `hasNext()`. You can test if the next token is an `int` using `hasNextInt()` and similarly for other built-in types. Each returns a boolean.

The method `nextLine()` differs from the others in that it reads everything on the input line until the return and then returns everything before the return. Thus it differs from `next()` in that it doesn't stop when it encounters white space, it just gobbles everything until the end of the line.

If you wish to write to a text file, you create a `PrintWriter` and send it `print` or `println` method requests.

```
PrintWriter myWriter = new PrintWriter("myText.txt")
```

and write using `myWriter.print("Some text to print")` or using `println` if you want it to terminate with a return.

14 Switch statement

Java does not have an equivalent of the match statement. The closest to it is the `switch` statement, but it is much more restricted and is very error prone. The switch is similar to the `match` statement in Grace in that an expression is evaluated and a different branch is executed depending on the value. However the expression must evaluate to an `int` or `char`. Here is a simple example of its use:

```
String dayToString(int day) {
   String dayName
   switch(day) {
      case 1: dayName = "Sunday";
                 break;
      case 2: dayName = "Monday";
                 break;
      case 3: dayName = "Tuesday";
                 break;
      case 4: dayName = "Sunday";
                 break;
      case 5: dayName = "Sunday";
                 break;
      case 6: dayName = "Sunday";
                 break;
      case 7: dayName = "Sunday";
                 break;
      default: dayName = "Error";
                 break;
   }
```

If you accidentally forget to put in the `break`, execution will continue to the next statement in the switch statement even if there is another `case:` there.

15 Performing Animations in Java

Java includes a construct called threads that allow several different activities to occur concurrently on your computer. While Grace will eventually support threads, our current web implementation does not support threads. (The reason is that we compile to the language javascript – essentially the native language of the web – and it does not support threads.). In this section I will explain how to create threads in Java and use them like we used the `Animation` library in Grace. You will be happy to know that threads are actually a bit easier to control in Java than they were in Grace, though of course the set up is longer, as always.

Note: In CS 51, we teach students how to use a class `ActiveObject` of the objectdraw library to avoid some complexities that arise in using the `Thread` class. We are going to introduce the standard Java `Thread` class.

When your Java program is started by the `startController(...,...)` method request, a special thread called the *event* thread pays begins execution. It executes the code in your `begin` method and then waits for events to happen in the window. If the user creates a mouse event, interacts with a GUI pattern, or even obscures or uncovers part of the window, the event thread does whatever is appropriate to respond to that event, e.g., repaint the screen, execute the code to respond to the mouse or other event generated, etc.

Typically we want the event thread to only do things that it can accomplish very quickly because when it is busy, your program cannot respond to other events or redraw the screen.

As a result if there are long-running tasks that we need the computer to accomplish (like performing animations), we put them in separate threads. To define a class that allows you to run animations you must

1. Define a class that `extends Thread`
2. Write a constructor to initialize any instance variables
3. Write a run method with signature `public void run()`. It is very important that it not take any parameters. Any parameters needed for the animation should be passed in to the constructor rather than the `run` method.
4. Include as `sleep` statement wherever you need a delay for the animation (usually inside a loop of some kind). Because the `sleep` may throw an `InterruptedException` you must wrap it in a `try-catch` construct that will catch that exception.
5. In the main program, create an object from the class and send it a `start()` method request. This will set up a new thread to execute the run method and then start the run method.

Let's go through each of these with an example. In Grace we created a program to play a rather pathetic game of pong (the ball just fell straight down and it did not interact with the paddle). Here is the code for animating the ball/

```
type FallingBall = {
   start -> Done
}

class ballSize (bSize) inside (boundary) moveRange (min, max)
                       hitBy (paddle) on (canvas)  -> FallingBall {

    def xShift = 0
    def yShift = 8

    def pauseTime = 30  // time gap between moves of ball

    method start {
        def theBall = filledOvalAt (boundary.location + (1 @ 1))
                               size (bSize @ bSize) on (canvas)

      //  print ("ball: {theBall.y}, canvas: {canvas.y}")
        animator.while {theBall.y < canvas.height} pausing (pauseTime) do {
            theBall.moveBy (xShift, yShift)
        } finally {
            theBall.removeFromCanvas
        }

    }
}
```

We can write the same program in Java by defining a class that extends `Thread`.

```
import objectdraw.*;

public class FallingBall extends Thread {
    // ball size and starting location
    private static final int BALLSIZE = 30;
    private static final int TOP = 50;
    private static final int BOTTOM = 600;

    // width of the screen
    private static final int SCREENWIDTH = 400;

    // how far the ball should fall in each iteration in loop
    private static final double Y_DISTANCE = 8;

    // delay between moves of the ball
    private static final int DELAY_TIME = 33;

    // image of the ball that will fall
    private FilledOval ball;

    /**
     * Create ball image and start falling
     * @param canvas - canvas where ball image is placed
     */
    public FallingBall(DrawingCanvas canvas) {
        ball = new FilledOval(SCREENWIDTH / 2, TOP, BALLSIZE, BALLSIZE, canvas);
    }

    /**
     *  Slowly move ball down until it is off the screen
     */
    public void run() {
        while (ball.getY() < BOTTOM) {
            ball.move(0, Y_DISTANCE);
            try {
                sleep(DELAY_TIME);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        ball.removeFromCanvas();
    }
}
```

Let's see how we match up with the list of instructions.

1. Class `FallingBall` is declared to extend `Thread`.
2. The constructor initialized the only instance variable `ball`.
3. The `run` method contains a `while` loop that drops the ball by `Y_DISTANCE` and then sleeps for `DELAY_TIME`. The `sleep` method does have the (annoying) `try-catch` wrapped around it to catch the possible `InterruptedException`. If it does get interrupted we just print out a stack trace that shows where the error occurred.

Because we are just using a regular `while` loop, there is not `finally` clause needed. Anything placed after the `while` loop will be executed when the while loop is done.

Thus we can see that we replaced an `animator.whilepausing()do` loop in a `start` method with a regular `while` loop in a `run` method, where we get to put the `sleep` wherever we like in the loop. *Don't forget to put a `sleep` command inside each loop if your want them to slowly move. Otherwise the loop will be complete before the screen gets a chance to refresh!*

Finally we need to create and start up the thread. This is essentially the same as in Grace. In Grace we wrote:

```
def pongBall: pb.FallingBall = pb.ballAt (point) size (ballSize) inside (boundary)
                   hitBy (paddle) on (canvas)
pongBall.start
```

In Java we write

```
FallingBall pongBall = new FallingBall(canvas);
pongBall.start();
```

or even just:

```
 new FallingBall(canvas).start();
```

if we don't need a name for the ball.

But where did the `start()` method come from? The only method in the class we defined was `run`. This is a bit of Java magic created by inheritance. The built-in class `Thread` has a method `start()` in it. When your program requests that method, it builds a new executable thread that is different from all of those currently executing in your computer. When it is ready, it will call the `run` method that you wrote.

This is a mechanism similar to that used by your main program. When you construct an object from a class that extends `WindowController`, you send it a `startController` method request. Method `startController` is defined in `WindowController`[2] and inherited by your program. The method `startController` sets up the window, places the `canvas` in the middle, and then calls your `begin` method.

Aside from these few changes in syntax, the threads in Java are used in ways similar to the `animator.while` loops in Grace. Because threads are simpler than using timers, we have no need for the `finally` clause with threads. Anything code after the loop is delayed until after the loop completes, because these are just normal while loops.

16 Inheritance in Java

Inheritance in Java is very similar to that in Grace, and in fact we have been using it from the beginning with our main programs (classes that extend `WindowController` and in the last section with classes that extend `Thread`).

Before we get into the nitty gritty of inheritance, let me remind you about the visibility declarations in Java. If a variable or method is declared to be `public`, then it is accessible to anyone who has a reference to the object. If it is declared to be `private` then it is only accessible inside the class that defines it. If it is declared to be `protected`, then it is accessible inside the class that defines it or in any subclass.[3] Thus `protected` is very similar to Grace's `confidential`.

To this point we have insisted that you declare all of the instance variables in Java to be private. However, you should declare them to be protected if you wish them to be accessible to subclasses. Never make an instance variable public. Instead if you want read or write access to a variable to be public, declare a getter or setter method to access them. (This is what Grace does for you automatically when you declare a variable to be readable or writable.)

In Java we write `extends` rather than `inherit` and the initialization code for the superclass is invoked in the constructor of the subclass. Here is a simple example in Java. First the superclass:

[2]Technically, that is a lie, as `startController` is actually defined in a superclass of `WindowController` named `Controller`.

[3]Technically, it is also accessible to any object generated by a class in the same package, but we are not discussing packages in this tutorial.

```
// A FramedDisplay is a composite graphic intended to serve
// as a frame in which a program can display various forms
// of information.
public class FramedDisplay {
   private static final int BORDER = 3; // Border thickness
   private static final int ROUNDNESS = 5; // Corner roundess

   // Colors used for the frame
   private static final Color BORDERCOLOR = Color.GRAY;
   private static final Color HIGHLIGHTCOLOR = Color.RED;

   private FilledRoundedRect body; // The background
   private FilledRoundedRect border; // The border

   // Create the display areas background and border and set
   // their colors appropriately
   public FramedDisplay( double x, double y,
                                 double width, double height, DrawingCanvas canvas ) {
        border = new FilledRoundedRect( x, y,
                                        width, height,
                                        ROUNDNESS, ROUNDNESS,
                                        canvas );

        body = new FilledRoundedRect( x + BORDER, y + BORDER,
                                      width - 2*BORDER, height - 2*BORDER,
                                      ROUNDNESS, ROUNDNESS,
                                      canvas );

        border.setColor( BORDERCOLOR );
    }

    // Change the border's color to make it stand out
    public void highlight( ) {
        border.setColor( HIGHLIGHTCOLOR );
    }

    // Restore the standard border color
   public void unHighlight( ) {
       border.setColor( BORDERCOLOR );
   }
}
```

Here is the code for the subclass:

```
// A FramedText object displays a specified text message on a
// background framed by a distinct border.
public class FramedText extends FramedDisplay {

   // Color used for the text
   private static final Color TEXTCOLOR = Color.WHITE;

   private Text message; // The message displayed
```

```
    private Location frameCenter; // Where message belongs

    // Create a FramedText object displaying the text contents
    // at the position and with the dimensions specified.
    public FramedText( String contents, double x, double y,
                             double width, double height, DrawingCanvas canvas ) {
         // construct the frame
         super( x, y, width, height, canvas );

         // Construct and appropriately position the message
         frameCenter = new Location( x + width/2, y + height/2 );
         message = new Text ( contents, x, y, canvas );
         message.setColor( TEXTCOLOR );
         positionContents();
    }

     // Position the message to center it in the frame
     private void positionContents() {
         message.moveTo( frameCenter );
         message.move( -message.getWidth()/2, -message.getHeight()/2 );
     }

     // Change the font size used
     public void setTextSize( int size ) {
         message.setFontSize( size );
         positionContents();
     }
}
```

In Grace we would have started the body of the method with `inherit framedDisplay....`. However in Java we add `extends FrameDisplay` to the class header. We invoke the initialization code in Java by inserting a `super` constructor as the first statement in the constructor of the subclass. The statement `super( x, y, width, height, canvas );` invokes the superclass (`FramedDisplay`) constructor. The rest of the body of the constructor initializes the new instance variables introduced in the subclass, though it could have also modified the initialization of instance variables of the superclass, if they were declared to be `protected`. In this case we did not need to access them, so all the variables in the superclass were declared to be private.

Java requires that the first statement of every constructor be a call to a super constructor, aside from one very important exception. If a class has a parameterless constructor, and the first line of its subclass constructor does not contain a call to a super constructor, then Java will automatically insert a call to the parameterless super constructor. This exception explains why we did not need to insert a call to a super constructor in classes extending `Thread` or `WindowController`. In the case of `WindowController` we do not even need to write the constructor at all. The system automatically runs the parameterless constructor for `WindowController`, which initializes the `canvas` and other instance variables from the superclass.

I you wish to request a method definition of some `m` from the superclass, you can just write `super.m(...)`.

As mentioned earlier, a Java class can only extend a single class, but it may implement as many interfaces as desired.

Just as in Grace, an interface may extend one or more existing interfaces. Where in Grace we wrote

```
type T = U & type {
     n(..) -> V
}
```

In Java we would write the same expression as

```
public interface T extends U {
     V n(...)
}
```

Examples in the objectdraw library in Java include `Drawable2DInterface` and `Drawable1DInterface` both extending `DrawableInterface`. Also `Resizable2DInterface` extends `Drawable2DInterface`. As in Grace, if an expression has an interface type then it also has all of its super interface types. Thus if an object has type `Drawable2DInterface` then it also has type `DrawableInterface`. Like classes, one interface is only a subinterface of another if it declared to be an extension of it.

That is we use the same `extends` clause to extend both classes and interfaces. However we use `implements` to indicate that a class provides elements of an interface.

Because classes can also be used as types in Java, if an expression has a type corresponding to a class then it also has the type of any superclass. Thus in our example above, if `obj` represents a `FramedText` then it also has type `FramedDisplay`

17 Summary

Here is a brief summary of what must be done to convert a Grace program to Java:

1. Grace has classes that take parameters. These parameters are visible thought the class definition. Java classes do not take parameters. Instead it includes a separate constructor that takes parameters and looks like a method declaration except that it has the same name as the class, and has no return type.

   E.g., in Grace we might write:

   ```
   class CWith(someParams) -> CType {...}
   ```

   while in Java we write

   ```
   public class C {
        ...
        public C(someParams) {...}
   ```

   Because Java has a separate constructor, the parameters are only visible in the constructor itself. If you want to access the parameters outside of the constructor you will have to save them as instance variables. E.g. if you need `canvas` outside of the constructor you might write:

   ```
   class C {
      ...
      DrawingCanvas myCanvas;
      ...
      public C(..., DrawingCanvas canvas) {
         myCanvas = canvas;
         ...
      }
   ```

2. You must put a semicolon at the end of lines in Java. However, never put a semicolon before a { or after a }. Proper indenting is optional (but *required* by your instructors). Use the `Format` command under the `Source` menu in Eclipse.

3. Parameterless method declarations and requests must include () even though there are no parameters. This is how Java tells the difference between a method and an instance variable.

4. Every (public) Java class must be in a separate file.

5. Every instance variable, constant, and method must have a visibility annotation. For now, use one of `private` and `protected`. Do *not* put these in front of local variables.

6. Types are written before the identifiers in declarations of variables and methods. E.g., `int x = 5;` and `public void onMouseClick(Location point)`.

7. Rather than a single `Number` type, Java has many. The type `int` is a primitive type holding integers, while `double` refers to numbers with decimal points. An `int` can be easily used as a `double`, but to convert a `double` to an `int` you must use a type cast: `(int)myDouble` or `Math.round(myDouble`, depending on whether you want to truncate the value or round it.

8. The Java types `int`, `double`, and `boolean` are primitive types, which mean that you may not make method requests of them. Object types should always be written with initial letter capitalized (e.g., `FramedRect`).

9. Java is statically typed, so every identifier introduced must be given a type: either a class or an interface. Interfaces are like Grace types, with no implementation information.

10. Assignments to variables are written with `=` instead of `:=`.

11. Constants are written in all caps and are declared as `static final` if they are the same for all objects of the class. If they depend on parameters to the class or method, just declare them as `final`.

12. Uninitialized instance variables are given the value 0 (if a primitive type) or null (if an object type). If you forget to initialize an instance variable you will get a "null pointer error" when you attempt to access it. Local variables are not automatically initialized. You must make sure to initialize them.

13. In Java we use keyword `this` in place of `self`. If `this` is the receiver of a method request you may leave out `this`. If a method parameter and an instance variable of the same class have the same name, placing a "`this.`" before the variable makes it refer to the instance variable. A common idiom is using the same name for a constructor parameter and instance variable and then using `this` in front of the instance variable when initializing it.

```
  class C {
     ...
     private String strVal;
     ...
     public C(String strVal) {
         this.strVal = strVal;  // right side is the parameter, left side is instance variable
     }
     ...
 }
```

14. You may overload methods and class constructors in Java (i.e., have more than one constructor or method with the same name). To be legal you must be able to distinguish between them with the number and types of parameters.

15. Arrays in Java are very primitive compared to Grace lists. Arrays cannot grow after they have been created. You can access the total number of slots in the array using `arr.length` if `arr` is the name of

the array. You can access and update slots using []: `arr[4] = val`. While Grace lists keep track of the number of active items in the array, Java does not, so you typically have to keep track of that value with an instance variable.

Most importantly remember that Java arrays of length n have slots labeled 0 to n-1. Thus `arr[0]` is the first element and `arr[arr.length-1]` is the last element. As a result "off-by-one" errors are very common in Java.

16. There are minor differences in the names of methods in the Java objectdraw library compared to Grace. Keep the objectdraw API documentation open at http://www.cs.pomona.edu/classes/cs051/handouts/objectdraw-api.html.