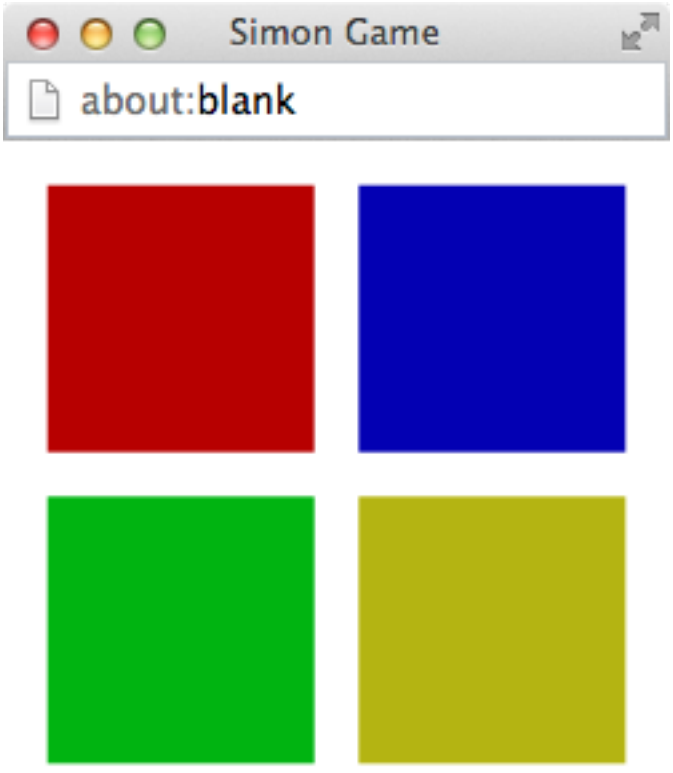
CS 51 Homework Laboratory # 8
Simon

Objective: To gain experience working with lists.

Many of you are probably familiar with the electronic toy named "Simon". Simon is a simple solitaire memory game. The toy is composed of a circular housing with four colored plastic buttons on top. A different musical note is associated with each button. The toy "prompts" the player by playing a sequence of randomly chosen notes. As each note is played, the corresponding button is illuminated. The player must then try to play the same "tune" by depressing the appropriate buttons in the correct order. If the player succeeds, the game plays a new sequence identical to the preceding sequence except that one additional note is added to the end. As long as the player can correctly reproduce the sequence played by the machine, the sequences keep getting longer. Once the player makes a mistake, the machine makes an unpleasant noise and restarts the game with a short sequence. [*Those of you who are fans of "Dance, Dance, Revolution" may recognize "Simon" as an early inspiration. See if you can figure out why it is similar and what extra would be necessary to program DDR.*]

For this laboratory exercise, we would like you to write a Grace program to allow one to play a simple game like "Simon". Like the original, our game will involve four buttons which the player will have to press in an order determined by the computer. We will keep the graphics simple by simply placing the four buttons in a 2 by 2 grid as shown below.



As soon as the buttons are displayed, your program should generate a list that initially consists of a single note/button. It should "play" a list by briefly highlighting the buttons that belong to the list in order. After a list is played, your program should wait while the player tries to repeat the sequence by clicking on the buttons in the appropriate order. If the player repeats the list correctly, the program should randomly pick a button to add to the end of the list and "test" the player on this new sequence. If the user makes a mistake, the program makes a "razzing" sound and then starts over with a one note sequence. We will explain all you need to know about making sounds in Grace below.

Your program will consist of two classes as well as the main program:

NoisyButton will describe buttons that act like those found on a Simon game. (We will provide this class.)

SimonGame will be your main program, an object of type `GraphicApplication`. We will provide the part of this that sets up the buttons.

Song will manage the sequence of buttons/tones corresponding to the "song" played by the game which the player needs to repeat. You are entirely responsible for this.

Audio clips, ButtonPanels and NoisyButtons To complete this lab, you will need to work with our `NoisyButton` class and one new feature of Grace, support for manipulating audio files.

Working with audio in Grace is quite simple. There is a class named `audio` that generates objects of type `Audio`. To create an object of type `Audio`, evaluate

```
audio.url(webAddress)
```

where `webAddress` is a string representing the location of a `.wav` file on the web. (Grace can only play `.wav` files. Others will need to be converted to this format.)

We will include five audio files in the starter folder for this lab. The files "tone.1.wav", "tone.2.wav", "tone.3.wav", and "tone.4.wav" describe the sounds the NoisyButtons should make. The file "razz.wav" contains the unpleasant noise your program should make when the user goofs.

There is only one method of the `Audio` type you will use in this lab. The method is named `play`. It expects no parameters and simply plays a sound. So, if you declare a variable as:

```
def nastyNoise: Audio = audio.url(
      "http://www.cs.pomona.edu/~kim/CSC051GF14/demos/SimonNotes/razz.wav")
```

then you can say:

```
nastyNoise.play
```

when you want to make a funny sound.

Our class `NoisyButton` produces "buttons" that look and act like those found on a Simon game. Unlike real buttons, our noisy buttons will live on the canvas. We provide you with a method `setUpButtons` in your main program that will lay these out nicely on the canvas to look like the picture shown earlier.

In order to determine which button was pressed, each button has a `contains` method. You will find it useful to write a helper method `getButtonPressed` that takes a `Point` and returns the button that was pressed. If the user didn't click on any of the buttons, it should return the default object `noButton`, defined in `NoisyButton.grace`. The method `getButtonPressed` should be used in `onMousePress` to retrieve the button that the user pressed when trying to reproduce the song.

The `NoisyButton` class provides one other method that you will use in your program. The method is named "`flash`". It makes the button flash and plays the sound associated with the button.

The `Song` class and `simonGame` object To complete this program, you will need to construct your "main program", `simonGame`, and a class that will manipulate the "song" played by the game.

Your `Song` class will manage the sequence of tones corresponding to the "song" played by the game. Internally, this class will represent the song using a list of `NoisyButton`s. You will need to use a variable to keep track of which note the user is expected to play next. For example, suppose there are currently 8 notes in the song, but the user has not yet guessed any notes. The class `song` will need to keep track that it is waiting for the user to play the first note. If the user gets the first note right, then it will need to remember that it is now waiting for the second note, etc.

The `song` class (and its type `Song`) must provide methods to play the song, to determine the next button the player is expected to click, to add a note to the song and several others. Determining exactly what methods are appropriate to include in `Song` and what parameters they should expect will be an important part of the work you should do to prepare for this lab. The key here is to consider what methods your `simonGame` program needs in order to interact with the song. The method names should reflect how they affect the song, not the underlying list. Think carefully what parameters the song class might need (e.g., what are the pieces from which it will construct a song).

The method that plays the song will need to use `animator` in order to provide delays between the notes as otherwise the notes will play too quickly and overlap each other. The song is represented by a list of `NoisyButton`s. Each `NoisyButton` knows how to play itself (i.e. each will respond to the invocation of its `play` method).

Finally, you will need to define a `simonGame` object. The initialization code for this class should create and place the noisy buttons on the canvas and create a new `Song` with a single loop. In addition, the class should include a `onMousePress` method so that react when the player clicks on a button. We provide the code to create and place the noisy buttons in a method named `setUpMethods`. It should look appropriate if your application creates a 250 by 250 pixel window, and should not need to be modified in any way.

The first thing to do in the `onMousePress` method is to determine what noisy button was clicked on. What happens next depends on whether or not the user clicked on the button corresponding to the next note in the song. If not, the program should make a nasty noise and start a new game (by creating the first note and playing it).

If the user got it right, there are two possibilities. The first is that it was the last note of the song. If so, add a new note and play the entire song to the user so they can start over with emulating the notes. If instead there are more notes to play, the program should keep track that the user is ready to play the next note, but then do nothing more. Of course the `onMousePress` method will be executed again when the user clicks the next button.

In summary, the `onMousePress` method begins execution when the user clicks on a button, and terminates when it needs to wait for the user to click a button again. The work it does in that method depends on whether or not the user's guess was correct, and, if so, whether the user still has more notes to repeat or whether she has finished all the notes in the song so far.

Design. This week we will again require that you prepare a written "design" for your program before lab. At the beginning of the lab, we will briefly examine each of your designs to make sure you are on the right track. Since you now should have enough experience to prepare a good design, this week's design will count as 20% of your total grade for this lab.

For each of the classes and objects you must write, the design should include:

- A list of the instance variables and defs you expect to include in the class or object definition,
- the header of the constructor for the class (including all parameter declarations),

- the headers of the methods you expect to define (including all parameters) in the class and a brief description of the function of the method (similar to the comment you would include to describe the method in the final program).
- a sketch of the code used in the body of each method and constructor, especially control structures like while, for, and if constructs.

Implementation. As usual, we suggest a staged approach to the implementation of this program. This allows you to identify and deal with logical errors quickly. As stated earlier, the size of your applet should be 250 pixels wide by 250 pixels tall.

- Make sure the `setUpButtons` method works properly and the screen looks right.
- Once the buttons are displayed, you should make sure they can flash. Test this by adding an `onMousePress` method to your `simonGame` that simply flashes the button that the user clicks on. The getButtonPressed method described above should be helpful for this. Test and see if it works.
- Now, you can start working on the `song` class. Write any definitions and variables needed for the class, and write a method to create a new Song with exactly one element (a noisy button). You will need to run this when the song is created, but also every time the user makes an error in trying to duplicate the song. In order to write this you will find it useful to have a helper method the randomly picks one of the four noisy buttons to add to the next position in the song.

  Test to make sure you can create a new song (your test should print out the contents of the song so that you make sure you are randomly selecting elements).
- You should now implement the `play` method for the song. To test if this all works, modify the `noisyButtonClicked` method in your controller so that when any button is clicked it adds a note to the song and then plays the entire song.
- Finally, add a variable to the `Song` class to keep track of the note the player should click next. For example, if the user has just clicked on the first two notes in the song then this variable should remember that the next note to be played is the third one. Then add methods to the class so that the `onMouseClicked` method of your main program can ask the `song` class which button it expects next and tell the `Song` when the correct button has been clicked, etc. Once these methods are available, modify the `onMousePress` method so that it reacts as the rules of Simon dictate when the player clicks on the buttons.

Warning: Do not use `button` *as an identifier in your program. It is the name of a GUI class and will give strange error messages if you use it as an identifier standing for a noisy button.*

Submitting Your Work The lab is due Tuesday or Wednesday at 11 PM as usual. When your work is complete you should deposit it in the appropriate dropbox. Make sure the folder name includes your name and the phrase "Lab 8". Also make sure that your name is included in the comment at the top of Song.grace and SimonGame.grace. Please do not modify the file NoisyButton.grace.

Before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations.

Sketch of classes provided for this program This file, `NoisyButton.grace`, need not be touched. Only the relevant parts are shown here.

```
type NoisyButton = {
  flash -> Done
  contains(pt: Point) -> Boolean
}

// default if user doesn't select a button
def noButton: NoisyButton = ...

class noisyButton.at(locn: Point) color (buttonColor: Color)
      sound (sound: Audio)
      on (canvas: DrawingCanvas) -> NoisyButton {...}
```

Startup file for Song Many pieces are missing.

```
dialect "rtobjectdraw"
import "timer" as timer
import "animation" as animator
import "NoisyButton" as nb

type NoisyButton = nb.NoisyButton

type Song = {
  newSong -> Done
  addNote -> Done
  play -> Done
  // what else?
}

// replace "new" with better name and any parameters needed
class song.new -> Song {

  def pauseBeforeSong: Number = 900

  def pauseBetweenNotes: Number = 400

  // add def and var declarations as needed

  // create a new song with one note
  method newSong -> Done {
    // ???
  }

  // Add a note to the song
  method addNote -> Done {
    // ???
  }
```

```
  // play the song after a suitable delay
  method play -> Done {
    timer.after(pauseBeforeSong) do {
      // use animator to play song with pauses
    }
  }

  // More methods???

}
```

Startup file for SimonGame We will provide you with the following start-up code to help you get going with the main program:

```
dialect "rtobjectdraw"
import "Song" as sg
import "NoisyButton" as nb

type NoisyButton = nb.NoisyButton
type NoisyButtonFactory = {
  at(locn: Point) color (buttonColor: Color)
      sound (soundfile: Audio) on (canvas: DrawingCanvas) -> NoisyButton
}

def noisyButton: NoisyButtonFactory = nb.noisyButton
def noButton: NoisyButton = nb.noButton

// type of song and its constructor
type Song = sg.Song
type SongFactory = {
  // ??? -> Song
}
def song: SongFactory = sg.song

def simonGame: GraphicApplication = object {
  inherits graphicApplication.size(250,250)

  def buttonList: List<NoisyButton> = list.empty<NoisyButton>

  // Initialize buttonList with four buttons and put on canvas
  setUpButtons

  // noise to play when player is wrong
  def nastyNoise: Audio = audio.url(
      "http://www.cs.pomona.edu/~kim/CSC051GF14/demos/SimonNotes/razz.wav")

  // declare any other values needed and finish initialization.
```

```
  // ????
  method onMousePress(pt: Point) -> Done {
    // ????
  }

  // Determine which button was pressed or if none, return noButton
  method getButtonPressed(pt: Point) -> NoisyButton is confidential {
     // ???
  }

  method setUpButtons -> Done is confidential {
    def audioList: List<Audio> = list.empty
    for (1..4) do {i:Number ->
      def nextAudio: Audio = audio.url(
        "http://www.cs.pomona.edu/~kim/CSC051GF14/demos/SimonNotes/tone.{i}.wav")
      audioList.add(nextAudio)
    }

    def buttonSize: Number = 100

    def maxColor: Number = 180
    def reddish: Color = color.r(maxColor)g(0)b(0)
    def greenish: Color = color.r(0)g(maxColor)b(0)
    def bluish: Color = color.r(0)g(0)b(maxColor)
    def yellowish: Color = color.r(maxColor)g(maxColor)b(0)

    def inset: Number = (canvas.width-2*buttonSize)/3
    buttonList.add(noisyButton.at(inset@inset) color(reddish) sound (audioList.at(1))
                           on (canvas))

    buttonList.add(noisyButton.at(inset@(2*inset+buttonSize)) color(greenish) sound (audioList.at(2
                           on (canvas))

    buttonList.add(noisyButton.at((2*inset+buttonSize)@inset) color(bluish) sound (audioList.at(3))
                           on (canvas))

    buttonList.add(noisyButton.at((2*inset+buttonSize)@(2*inset+buttonSize))
                           color(yellowish) sound (audioList.at(4)) on (canvas))

  }

  startGraphics
}
```

Table 1: Grading Guidelines

Value	Feature
	Design preparation (4 points total)
1 point	instance variables & constants
1 point	constructors
1 point	methods
1 point	`onMousePress` method
	Readability (5 points total)
2 points	Descriptive comments
1 points	Good names
1 points	Good use of constants
1 point	Appropriate formatting
	Code Quality (7 points total)
1 point	Conditionals and loops
2 points	General correctness/design/efficiency issues
1 point	Parameters, variables, and scoping
2 points	Good correct use of lists
1 point	Miscellaneous
	Correctness (4 points total)
1 point	Playing songs
1 point	Comparing user input with songs
1 point	Restarting correctly when user makes mistake
1 point	Lengthening song correctly when user is right