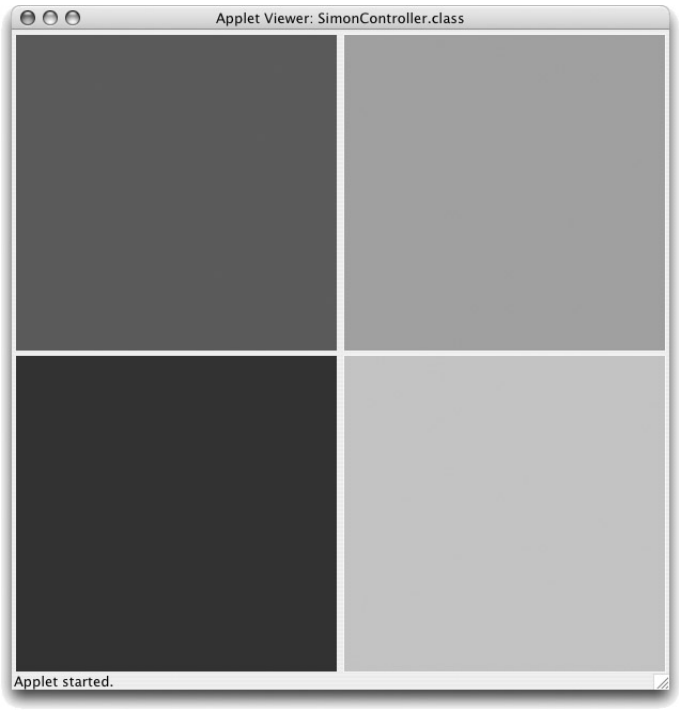
CS 134 Programming Exercise 8:
Simon

Objective: To gain experience working with arrays.

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 plastic base with four colored plastic buttons on top. Each button has a different color and 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.

For this laboratory exercise, we would like you to write a Java 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. Given the limitations of Java's layout managers, we will keep the graphics simple by simply placing the four buttons in a 2 by 2 grid as shown below.



The online version of this lab handout includes a demo version of this program. You can play with it to see what we have in mind. You should also visit the online version of this handout for instructions on obtaining a copy of the starter folder.

As soon as the buttons are displayed, your program should generate a sequence consisting of a single note/button. It should "play" a sequence by briefly highlighting the buttons that belong to the sequence in order. After a sequence 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 sequence correctly, the program should randomly pick a button to add to the end of the sequence 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.

Your program will consist of five main classes:

SimonController will extend `Controller` rather than `WindowController`. Recall that the only difference between a `Controller` and a `WindowController` is that the latter comes with the `canvas` installed. The `canvas` is not needed for this program as we will not be doing any drawing.

NoisyButton will describe buttons that act like those found on a Simon game. We will provide a complete implementation of the NoisyButton class as part of the starter folder for this lab. The details of how to use our code are provided in the following section.

ButtonCollection will manage the set of four `NoisyButton`s that form the game board.

Song will manage the sequence of buttons/tones corresponding to the "song" played by the game which the player needs to repeat.

SongPlayer will be a class that extends `ActiveObject`. It will be used to actually play the `Song`.

NoisyButton To complete this lab, you will use a class we have defined named `NoisyButton`. While `NoisyButton` is not part of the standard Swing GUI library, objects of the `NoisyButton` class can be used as GUI components. You can `add` a `NoisyButton` to your `Controller`'s content pane or to a `JPanel` just as you could add a `JButton` or a `JComboBox`. You can also specify how to react when someone clicks a `NoisyButton` by providing a listener for such events.

The `NoisyButton` class provides two methods that you will use in your program. The first method is named "`flash()`". It makes the button flash and plays the sound associated with the button. The second method is named `addListener` and is used to identify the object that should be notified when the user clicks on a `NoisyButton`.

The header for the method used to add a listener to a `NoisyButton` is

```
public void addListener(NoisyButtonListener listener)
```

An object that listens to a `NoisyButton` must implement an interface, NoisyButtonListener, which is included in our starter project and defined as:

```
public interface NoisyButtonListener
\{
        // Method invoked when a NoisyButton is clicked
    public void noisyButtonClicked(NoisyButton source);
\}
```

When someone clicks on a `NoisyButton`, the button will invoke the `noisyButtonClicked` method of the object that has been added as a listener. We expect you to use your `SimonController` as the listener, so you should define a `noisyButtonClicked` method in that class. When this method is invoked, the `NoisyButton` that has been clicked will be passed as a parameter.

The constructor for the `NoisyButton` class expect two parameters: an `AudioClip` for the sound the button should make when pressed and its `Color`. The colors you use should be somewhat dull shades of red, blue, yellow and green. We suggest using `new Color(180,0,0)`, `new Color(0,180,0)`, `new Color(0,0,180)`, and `new Color(180,180,0)`.

In the starter folder we have included four audio files you can use for the noises the buttons make and a fifth file for the razzing sound made when the player goofs. The files "tone.0.au", "tone.1.au", "tone.2.au" and "tone.3.au" describe the sounds the `NoisyButton`s should make. The file "razz.au" contains the unpleasant noise your program should make when the user goofs. (We will leave it to you to guess which of the CS professors sat around making funny noises to produce this file.) You can use the "`getAudio`" method associated with the "`Controller`" class to access these files. Like "`getImage`", this method expects a string that names a file as a parameter. The button sounds will be used by the code we have provided within the `NoisyButton` class. You will use the built-in `play` method to produce the razzing sound. If you declare a variable as:

```
AudioClip nastyNoise;
```

and in your `Controller` you assign it a value using:

```
nastyNoise = getAudio("razz.au");
```

then you can say:

```
nastyNoise.play();
```

when you want to make a razzing sound.

Collection classes The goal of this lab is to exercise your knowledge of arrays. From this point of view, the most interesting aspect of this assignment will be the implementation of two classes that will use arrays to maintain collections of `NoisyButtons`.

The first of these two classes will be aptly named `ButtonCollection`. It will simply hold the four buttons that appear on the screen in an array. Its most important feature will be a method that will return a randomly chosen button from the collection.

The other collection class you implement will be used to hold the sequence of buttons the user is currently being asked to repeat. This class will be named `Song`.

One method you will need to include in your `ButtonCollection` is a method that will return a `NoisyButton` randomly chosen from the collection. Each time you start a new song or the user correctly repeats the existing song, you will use this method to select a random button and then add this button to the current `Song`.

Both the `ButtonCollection` and `Song` class will need to provide methods that can be used to add `NoisyButton`s to a collection. As you create the `NoisyButton`s in your `begin` method and add them to the display you will also add them to your `ButtonCollection`. The standard Simon game only has four buttons, so an array of four elements will be sufficient to implement the `ButtonCollection` class.

On the other hand, you may add more than four buttons to the current `Song`. You will add a button to the `Song` whenever the player correctly repeats the current sequence. It is hard to predict how often a good player might do this, but an upper bound like 100 or 150 is almost certainly safe. To make your `Song` class as flexible as possible, the size of the array to be used should be included as a parameter to the `Song` constructor.

The `Song` class will need to provide several other methods that can be used to step through the sequence of buttons the user is supposed to repeat. These methods are most easily understood in the context of the classes that will use them, the `SongPlayer` and `SimonController` classes, so we will discuss them in the following sections.

The `SongPlayer` class Your `Song` class will manage the sequence of tones corresponding to the "song" played by the game, but you will need a separate `SongPlayer` class to play the song. The reason you need to create a separate class to play all the notes in a `Song` is that you need to pause between the individual notes (and it will be best if you also pause for a second or so before beginning to play the song). The `pause` method can only be used within an `ActiveObject` andthe `Song` will not extend `ActiveObject`. The `SongPlayer` class will be a class that extends `ActiveObject`. Therefore, whenever you want to play a `Song`, you will create a `SongPlayer` to actually do the work. The `SongPlayer` constructor will take a `Song` as a parameter. The `SongPlayer`'s run method will contain a loop that alternately `flash`es a `NoisyButton` and then pauses for a short period.

To write this loop, you will need a way to access the `NoisyButton`s stored in the `Song` from the `SongPlayer` class. The array in which the `NoisyButton`s are actually stored will be a `private` instance variable within the `Song` class. It cannot be accessed directly from the `SongPlayer`. Instead, you will need to include methods within the `Song` class that provide sufficient access to the `NoisyButtton`s to write the desired loop.

As we have repeatedly emphasized, there are three components that determine the iterative behavior of a loop: the initialization of the loop variables, the condition that determines termination, and the update

of the loop variables performed to move to the next iteration. You will need to provide a method in your `Song` class to facilitate the construction of each of these three components of the loop in the `SongPlayer`'s `run` method.

There will be one method to get the "next" note that should be played. This will be used to update a loop variable when you are ready to move on to playing the next node. To make it possible to implement this method in the `Song`, the `Song` class will have to include an instance variable that keeps track of which element of the array of `NoisyButton`s should be played next.

There will be a boolean method to test whether there are any `NoisyButton`s left to play. This method will be used to determine when to terminate the loop in the `SongPlayer`. It will also depend on the variable in the `Song` class that keeps track of the array entry holding the next `NoisyButton` to play. Finally, there will be a method to "rewind" the song so that the first (zeroth?) `NoisyButton` in the array becomes the next to play.

The `SimonController` class To complete this program, you will need to construct a class that extends `Controller` that will act as your "main program". The `begin` method for this class will create the four `NoisyButton`s. It will also establish the controller class as a listener for the buttons and add them to the display and a `ButtonCollection`. In addition, it should create and play a `Song` containing just one note.

The other important method in the `SimonController` will be the `noisyButtonClicked` method. Each time a button is clicked you will need to determine whether or not the right button was clicked. To do this, you will have to know which button is expected next. If the user clicks the wrong button, you will make a nasty noise and start over by creating a new one-note song. If the user clicks the right button, you will have to determine whether the user has repeated the entire `Song` so that you can decide whether to wait for additional clicks or to add a note to the `Song`.

Basically, to write `noisyButtonClicked`, you need a way to step through the notes of the `Song` one by one, much as you do this to write the loop that plays the song in `SongPlayer`. You will find that the same methods described in our discussion of the `SongPlayer` class should provide the access you need to make the appropriate tests in the `noisyButtonClicked` method.

Design. This week we will again require that you prepare a written "design" for your program before lab. At the beginning of the lab, the instructor will briefly examine each of your designs to make sure you are on the right track. At the same time, the instructor will assign a grade to your design.

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

- A list of the non-final instance variables you expect to include in the class 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. The size of your applet should be 400 pixels wide by 400 pixels tall.

- Constructing the appropriate screen display is a good place to start. Write and test the portion of the `begin` method needed to create the four `NoisyButtons` and add them to the display. You will need to get the audio clips to do this. Don't try to add listeners or put the `NoisyButton` in a `ButtonCollection` at this point.

- Once the buttons are displayed, you should make sure they can flash. Test this by adding a `noisyButtonClicked` method to your `SimonController` that simply flashes the button passed to it as a parameter. Add the controller as the listener for the `NoisyButton`s. Then, click and see if it works.

- Next implement the `ButtonCollection` class and see how good it is at picking random buttons. Modify your `begin` method to place your four `NoisyButtons` in a `ButtonCollection`. Then modify your `noisyButtonClicked` method so that each time you click it flashes a button randomly chosen by the `ButtonCollection` rather than the one you clicked on.

- Now, define the `Song` class. It is a bit hard to test this class one piece at a time because of the close relationship between the `add` method and the three methods that rewind the `Song`, get the next `NoisyButton` in the `Song`, and test to see if you are at the end. In fact, it is probably best to define and test the `Song` and `SongPlayer` classes at the same time. Once you think both classes have been written, you can change your `begin` method so that it creates an empty `Song` and change the `noisyButtonClicked` method so that each time you click a button, you add that button to the `Song` and then play the entire `Song`. It the `Song` and `SongPlayer` classes are correct, each time you click a button, the program should repeat the sequence of all the buttons you have clicked.

  Of course, this is backwards. The player is supposed to repeat what the computer did, not the other way around! So...

- Once you think the `ButtonCollection`, `Song` and `SongPlayer` classes are functioning correctly, it is time to construct versions of the `begin` and `noisyButtonPressed` methods that will play Simon as expected. In `begin`, you will need to construct a `Song` consisting of just one note and play it. In `noisyButtonPressed`, you need to check whether the expected button was pressed and whether it was the last button in the `Song` and then either wait for the next click, make a nasty noise and start over or add a note to make the song longer and play it.

- As an extra touch:

  One odd bit of behavior your program will exhibit is that it will get very confused if you click a button before the computer has finished playing the sequence it wants you to complete. Try it and think about what is going wrong.

  To fix this behavior, you have to ignore clicks on the buttons while a `SongPlayer` is actively playing a song. You can do this by adding a `boolean` instance variable to your `SimonController` that keeps track of when a `SongPlayer` is active. It is easy to set this `boolean` to `true` when you create a `SongPlayer`. It is a bit harder to have it become `false` when the `SongPlayer` is done.

Submitting Your Work The lab is due at 11 PM on Wednesday for the Monday lab section and Thursday for the Tuesday lab section. When your work is complete you should deposit it in the appropriate dropoff folder a folder that contains your program and all of the usual files needed by Eclipse. 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 each Java source file.

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.

Skeleton of the `NoisyButton` class provided in the starter

```
public class NoisyButton ...
{
        // construct a new NoisyButton
        // noise determines the tone played when the button is clicked
        // shade determines the unhighlighted color of the button
```

```
        public NoisyButton(AudioClip noise, Color shade) { ... }

        // Assign an object to listen for when someone clicks on the button
        public void addListener(NoisyButtonListener listener) { ... }

        // Make the button flash by creating an ActiveObject that does the work
        public void flash() { ... }
}
```

Startup file for SimonController We will provide you with the following start-up file to help you get going with the `SimonController` class:

```
import objectdraw.*;
import java.awt.*;
import javax.swing.*;
import java.applet.AudioClip;

// name, lab, etc

public class SimonController extends Controller implements NoisyButtonListener {
   private static final int BUTTONCOUNT = 4; // the number of distinct sounds
                                             // corresponding to the game buttons

   private static final int COLORINTENSITY = 180; // how bright to make buttons

   private AudioClip nastyNoise; // a razzing noise


   // create the display of four buttons on the screen
   public void begin() {
      // buttons should appear in a grid
      getContentPane().setLayout(new GridLayout(2,2));

      // load the nasty noise
      nastyNoise = getAudio("razz.au");

      // create an array of colors for the buttons
      Color shades[] = new Color[BUTTONCOUNT];
      shades[0] = new Color(COLORINTENSITY, 0, 0);
      shades[1] = new Color(0, COLORINTENSITY, 0);
      shades[2] = new Color(0, 0, COLORINTENSITY);
      shades[3] = new Color(COLORINTENSITY, COLORINTENSITY, 0);


      for (int buttonNum = 0; buttonNum < BUTTONCOUNT; buttonNum++) {
         AudioClip curSound = getAudio("tone." + buttonNum + ".au");

            // create and add buttons
      }

      // add the panel of buttons to the window
      validate();
```

```
    }

    // Check to see if the player clicked the expected button
    public void noisyButtonClicked(NoisyButton theButton) {

        /*
        if (theButton == "the expected button") {
             // player got it right
        } else {
             // player goofed

        }
        */
    }
}
```

Table 1: Grading Guidelines

Value	Feature
	Design preparation (4 points total)
1 point	instance variables & constants
1 point	constructors
1 point	methods
1 point	`noisyButtonClicked` method
	Syntax style (5 points total)
2 points	Descriptive comments
1 points	Good names
1 points	Good use of constants
1 point	Appropriate formatting
	Semantic style (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 arrays
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