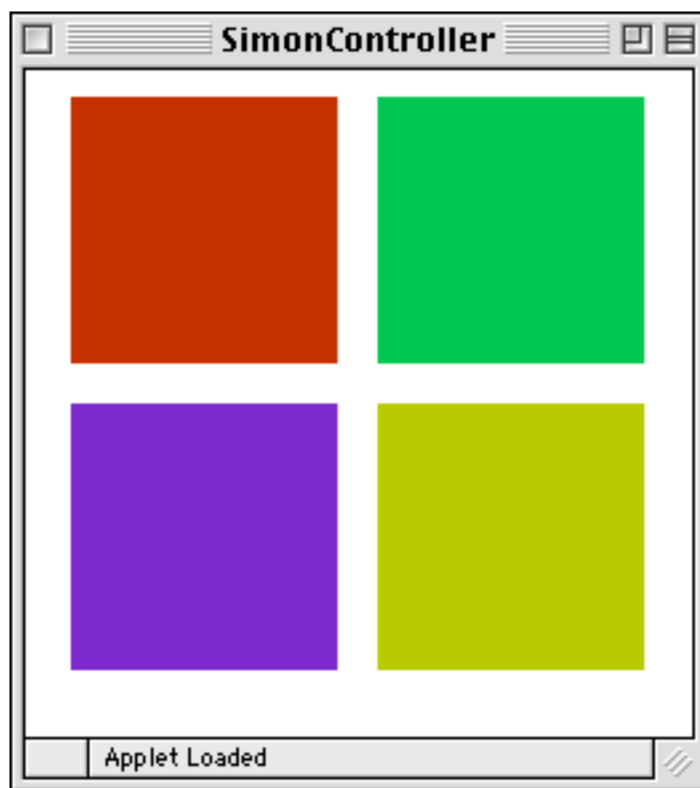
# CS 51 Laboratory # 8
## Simon

**Objective:** To gain experience working with arrays.

—

Many of you may be 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 DDR may recognize its resemblence to Simon . . . what additional functionality would be needed to program DDR?)

For this laboratory, you will write a Java program that allows 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 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.

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, play the sequence and again wait for the player to repeat 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'll recap everything you need to know about making sounds in Java.)

Your program will consist of five main classes:

**NoisyButton** will describe buttons that act like those found on a Simon game.

**ButtonPanel** will create and manage the set of `NoisyButtons` that form the game board.

**SimonController** will be your main class which extends `Controller` (rather than `WindowController` as in previous labs). 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.

**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`.

The good news is that we will provide complete implementations of the first two classes as part of the starter for this lab. The details of how to use our code are described in the following section.

**AudioClips, ButtonPanels and NoisyButtons** To complete this lab, you will need to work with our `ButtonPanel` and `NoisyButton` classes and one new feature of the Java system, support for manipulating audio files.

Working with audio in Java is straightforward (we've seen one example in the BouncingBasketball demo). There is a method named "`getAudio`" associated with the "`Controller`" class you extend when defining your main class. Like "`getImage`", this method expects a string that names a file as a parameter. This file must be in a standard format that represents audio segments. If it is, the system will read the file and return an object of type `AudioClip`. We will include five audio files in the starter folder for this lab. The files "tone.0.au", "tone.1.au", "tone.2.au" and "tone.3.au" describe the sounds the NoisyButtons should make. The file "razz.au" contains the unpleasant noise your program should make when the user messes up.

There is only one method of the `AudioClip` class 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:

```
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 funny sound.

The NoisyButton class produces buttons that look and act like those found on a Simon game. These buttons know how to beep and flash. The NoisyButton class provides one method which you will use in your program. The method is named "flash()". It makes the button flash and plays the sound associated with the button. In the begin method of SimonController we have already created the buttons and set them up with the appropriate sounds.

Like other GUI components, a NoisyButton needs to have some other object that "listens" for events that involve the button. The NoisyButtonListener is used by a class that wants to respond to a NoisyButton being clicked. The interface only has one method and is defined as:

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

To listen for (and repsond to) NoisyButton events, an object must provide a noisyButtonClicked method. The SimonController class implements this listener. We have included the method, but you will need to fill in the details. When the noisyButtonClicked method is invoked, the NoisyButton that has been clicked will be passed as a parameter.

The other class we will provide is called ButtonPanel. It creates the collection of NoisyButtons that form the game board. The ButtonPanel class is also a GUI component and can be added to the main panel, which we have done for you already in the SimonController class.

The ButtonPanel class provides two methods. The first method is used to assign a listener to all of the buttons in the panel. It is named addListener and expects an object that implements the NoisyButtonListener interface as a parameter. The other method will be used when you need to add a button to your "song". It is named getRandomButton and it simply returns a randomly chosen button from the panel.

The ButtonPanel constructor requires that you pass it the AudioClips for the sounds the buttons will make. Rather than expecting four separate parameters, the class is defined to expect as the only parameter to its constructor one array of AudioClips that refers to the four button sounds.

**The Song, SongPlayer and SimonController classes** To complete this program, you will need to construct a class that extends Controller that will act as your "main program" and two classes that will manipulate the "song" played by the game.

The Song class will manage the sequence of tones corresponding to the "song" currently played by the game. Internally, this class will represent the song using an array of NoisyButtons. The actual number of notes in the song will vary depending on just how good the player is at the game. So, your Song class will need to construct an array big enough to hold a sequence longer than any player is likely to remember (100 is certainly safe!) and then use a separate int variable to keep track of how many notes are currently in the sequence. You will also need to use another int 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 only 8 of the possible 100 notes are in the current song, and it 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 must provide methods to

- play the song

- determine the next button the player is expected to click
- 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 `Song` class is used to keep track of what song the user should be playing as well as where the user is in the song. You will also need to create a `SongPlayer` class that is used to "play" the song to the user so that the user knows what pattern/song he/she is supposed to copy.

The song is represented by an array of `NoisyButton`s. Each `NoisyButton` knows how to play itself (i.e. each will respond to the invocation of its `play` method). The problem is that in order to play the song correctly, you will 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`. The `SongPlayer` class will be a class that extends `ActiveObject`. Whenever the `Song` class is asked to play itself, it will create a `SongPlayer` to actually do the work. The array that holds the song and the song's current length will be passed as parameters to the `SongPlayer` constructor. The run method of the `SongPlayer` will simply play all the notes (with appropriate pauses) and then terminate.

Finally, you will need to fill in a few details of the `SimonController` class. In the `begin` method, you will need to create a new `Song` and play it for the user to get things going. In the `noisyButtonClicked` method you will need to handle the user button clicks appropriately. You can determine which button was clicked on by by examining the button that is passed as a parameter to the method. 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.

Most of this functionality should actually be implemented in the `Song` class. You will just be utilizing it appropriately in the `noisyButtonClicked` method.

**Design.** This week we will again require that you prepare a written "design" for your program before lab, which will be graded at the beginning of the lab. You will need a design for the `SimonController`, `Song` and `SongPlayer` classes. Follow the specification stated in class about what is required of a design. Think about how the different classes interact with eachother!

**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 250 pixels wide by 250 pixels tall.

- First, see if you can make the buttons flash. Modify the `noisyButtonClicked` method in the `SimonController` class that simply flashes the button passed to it as a parameter. Then, click and see if it works.
- Now, fill in the constructor and `play` method for the `Song` class and define the `SongPlayer` class. You won't be able to test these, however, unless you have a way to add notes to your song. So,

you should also define the method needed to add a random note to a song at this point. Then, to test if it 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 song.

- Finally, add a variable to the `Song` class to keep track of the note the player should click next. Then add methods to the `Song` class so that the `noisyButtonClicked` method of your controller 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 `noisyButtonClicked` method so that it reacts as the rules of Simon dictate when the player clicks on the buttons.

**Submitting Your Work** Before submitting your work, make sure that each of the .java files includes a comment containing your name. Also, 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. Use the Format command in the Source menu to make sure your indentation is consistent. Refer to the lab style sheet for more information about style.

Turn in your project the same as in past weeks; though make sure that the folder name begins with your last name (and includes the lab number).

This lab is due Monday at 11 p.m.

**Sketch of classes provided for this program**

```
public class ButtonPanel extends JPanel {
        //  Construct button panel.
        //  sounds is an array containing the tones played when the buttons flash
        public ButtonPanel(AudioClip[] sounds);

        // add a listener to all four buttons
        public void addNoisyButtonListener(NoisyButtonListener listener);

        // return a random button from the panel
        public NoisyButton getRandomButton();
}
```

```
public class NoisyButton extends JPanel implements MouseListener {
        // 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();

}

public class AudioClip {
        public void play();
}
```

**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.applet.AudioClip;

// Name:
// Lab:

public class SimonController extends Controller implements NoisyButtonListener {
    private static final int NUM_SOUNDS = 4;    // the number of distinct sounds
                                                // corresponding to the game buttons
    private AudioClip nastyNoise;               // a razzing noise

    private Song theSong;                       // a sequence of buttons/tones that
                                                // the user must reproduce
    // create the display of four buttons on the screen and
    // associate appropriate noises with those buttons
    public void begin() {

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

        // create the array of audio clips for the buttons
        AudioClip[] buttonSounds = new AudioClip[NUM_SOUNDS];
        for (int i = 0; i < NUM_SOUNDS; i++) {
                buttonSounds[i] = getAudio("tone."+i+".au");
        }

        // create the button panel
```

```
        ButtonPanel theButtons = new ButtonPanel(buttonSounds);
        // add a listener for user clicks on the buttons
        theButtons.addNoisyButtonListener(this);

        // add the panel of buttons to the window
        getContentPane().add(theButtons);
        validate();

        // add code to start a new song and play it for the user

    }

    public void noisyButtonClicked(NoisyButton theButton) {

        // Uncomment and use this if/else statement

        //if () if the expected button was clicked
        //{  // worry about whether it was the end of the song or not
        //}
        //else  the player made a mistake
        //{
        //}
     }
}
```

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 |
| | **Extra Credit (2 points maximum)** |
| .5 point | Better graphics |
| .5 point | Keeping score |