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

**Scenario**  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 pressing 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, you will write a Java program 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 handouts web page includes a demo version of this program. You can play with it to see what we have in mind. That page also includes 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. 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 NoisyButtons 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.

These five classes are described below.

**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

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

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

```java
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 expects two parameters: a String describing the note the button should play when pressed and its Color. Notes are just the musical notes “A” through “G”. 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).

(We are actually using a fairly powerful music synthesizer library to generate the noises in this program. The documentation for the library can be found at http://www.jfugue.org. Feel free to experiment with the synthesizer options and instruments in the NoisyButton class once you are finished with the entire lab. You can talk to us for more details on how to change the code in that class.)
**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 NoisyButton.

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. 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 to extend the song by one more note.

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

Both the *ButtonCollection* and *Song* class will need to provide methods that can be used to add NoisyButtons to a collection. As you create the NoisyButtons 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. The *ButtonCollection* will also need to keep track of how many buttons have been added to its array, so that it knows where in the array the next button will go.

You will certainly need to store more than four buttons for the current *Song* since you will add a new button every time 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. As with the *ButtonCollection*, the *Song* will also need to remember how many notes are in the current sequence in an int instance variable.

The *Song* class will provide a play method that will create a new *SongPlayer* object to play the song. The class must also include several other methods for stepping through the sequence of buttons the user is supposed to repeat. These methods are most easily understood in the context of the *SimonController* class below, so we will discuss them in that section.

**The *SongPlayer* class**  Although the *Song* class will manage the sequence of tones corresponding to the “song” played by the game, 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 you should also pause for a second or two before beginning to play the song). The pause method can only be used within an *ActiveObject* and the *Song* will not extend *ActiveObject*. Instead, the *SongPlayer* class will be the class to extend *ActiveObject*. Whenever you want to play a *Song*, you will create a *SongPlayer* to actually do the work. When the *Song*’s play method creates a *SongPlayer*, it will need to pass its NoisyButton array and the song’s length as parameters to the constructor. The *SongPlayer*’s run method should flash each NoisyButton in the song, pausing for a short period between each one.

**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 NoisyButtons. 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. You can play the “nasty” noise with the following command:

```java
NoisyButton.playNastyNoise();
```

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 new note to the *Song*.

To write noisyButtonClicked, you need a way to step through the notes of the *Song* one by one, as the user clicks buttons. This is a bit trickier than the *SongPlayer*, since we need to keep track of
where we are in the song throughout a whole sequence of button clicks. Also, the array in which the 
NoisyButtons are actually stored will be a private instance variable within the Song class, and it 
cannot be accessed directly from the SimonController.

Therefore, you will need to extend your Song class to provide additional methods that facilitate 
accessing the notes as buttons are clicked.

In particular, you will need to add a method to Song for getting the “next” note that should be 
matched by the user. This will be used in noisyButtonClicked to see whether the user got the next 
ote note right. To make it possible to implement this method in the Song, the Song class will need an 
int instance variable that keeps track of which element in its the array of NoisyButtons should be 
matched by the user’s next click. This instance variable will be set to 0 when the program is waiting 
for the user to match the first note of the Song, 1 when the program is waiting for the second note, and 
so on.

There will also be a boolean method to test whether there are NoisyButtons left to match. This 
method will be used to determine when the user has reached the end of the song. This method will 
also depend on the variable in the Song class that keeps track of the array entry holding the next 
NoisyButton to match.

Finally, there will be a method to “rewind” the song so that the first (zeroth?) NoisyButton in the 
array becomes the next to match. This method is used when the user gets to the end of the song and 
must start playing notes from the beginning of the song’s sequence again.

---

**Design**

This week we will again require that you prepare a written “design” for your program before lab. 
As always, we will briefly examine each of your designs to make sure you are on the right track, and 
to assign a grade to it.

Keep in mind that a good design includes the following items for each class:

- 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. Don’t 
try to add listeners or put the NoisyButtons 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 NoisyButtons. Then, click and 
see if it works.

- Next implement the ButtonCollection class and see how good it is at picking random buttons. 
To create and add a new class to your project, perform the following:
1. Click on the Simon project in the Package Explorer panel on the left side of the Eclipse window.
2. Open the File menu. Select “New” and then “Class.”
3. Enter “ButtonCollection” as the name of the class.
4. Click Finish.
5. You should now have a ButtonCollection.java file inside the Simon project. Modify the file so that it starts with “import objectdraw.*;” and “import java.awt.*;”.

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. In fact, it is probably best to define and test the Song’s add and play methods and the SongPlayer class at the same time.

The Song and SongPlayer classes are created in the same way as ButtonCollection above. Be sure to add “extends ActiveObject” to the SongPlayer declaration.

To test these classes, 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. If 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 backward. 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 noisyButtonClicked 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 noisyButtonClicked, 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.

At this stage, you should also complete the Song class by writing the three methods to “rewind” the Song, get the next NoisyButton in the Song, and test to see if you are at the end.

• 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. Make sure the folder name includes your name and the phrase “Lab8”. 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.
Grading Guidelines

<table>
<thead>
<tr>
<th>Value</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design preparation (4 points total)</td>
<td></td>
</tr>
<tr>
<td>1 point</td>
<td>instance variables &amp; constants</td>
</tr>
<tr>
<td>1 point</td>
<td>constructors</td>
</tr>
<tr>
<td>1 point</td>
<td>methods</td>
</tr>
<tr>
<td>1 point</td>
<td>noisyButtonClick method</td>
</tr>
<tr>
<td>Syntax style (5 points total)</td>
<td></td>
</tr>
<tr>
<td>2 points</td>
<td>Descriptive comments</td>
</tr>
<tr>
<td>1 points</td>
<td>Good names</td>
</tr>
<tr>
<td>1 points</td>
<td>Good use of constants</td>
</tr>
<tr>
<td>1 point</td>
<td>Appropriate formatting</td>
</tr>
<tr>
<td>Semantic style (7 points total)</td>
<td></td>
</tr>
<tr>
<td>1 point</td>
<td>Conditionals and loops</td>
</tr>
<tr>
<td>2 points</td>
<td>General correctness/design/efficiency issues</td>
</tr>
<tr>
<td>1 point</td>
<td>Parameters, variables, and scoping</td>
</tr>
<tr>
<td>2 points</td>
<td>Good correct use of arrays</td>
</tr>
<tr>
<td>1 point</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Correctness (4 points total)</td>
<td></td>
</tr>
<tr>
<td>1 point</td>
<td>Playing songs</td>
</tr>
<tr>
<td>1 point</td>
<td>Comparing user input with songs</td>
</tr>
<tr>
<td>1 point</td>
<td>Restarting correctly when user makes mistake</td>
</tr>
<tr>
<td>1 point</td>
<td>Lengthening song correctly when user is right</td>
</tr>
</tbody>
</table>

Starter Code

Skeleton of the NoisyButton class

```java
public class NoisyButton {
    // Construct a new NoisyButton
    // note determines the tone played when the button is clicked
    // It should be one of "A", ..., "G".
    // shade determines the unhighlighted color of the button.
    public NoisyButton(String note, 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() { ... }

    // Play a "nasty" noise. This method is static, which means that
    // you call it as follows:
    //
    // NoisyButton.playNastyNoise();
    //
    public static void playNastyNoise() { ... }
}
```
Starter code for `SimonController`. We will provide you with the following starter file to help you get going with the `SimonController` class:

```java
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

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

        // 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);

        // create an array of notes for the buttons
        String notes[] = new String[BUTTONCOUNT];
        notes[0] = "C";
        notes[1] = "A";
        notes[2] = "B";
        notes[3] = "D";

        for (int buttonNum = 0; buttonNum < BUTTONCOUNT; buttonNum++) {
            // create and add buttons
        }

        // add the panel of buttons to the window
        container.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
        }
        */
    }
}
```