Handout 10
CSCI 134: Spring, 2008
4 April

Lab 8

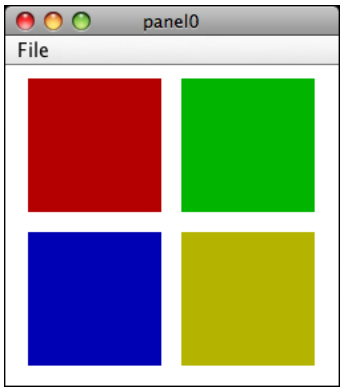
Simon

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.

This week, you will write a Java program to play a simple game like "Simon." Like the original, our game will involve four buttons that the player will 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. We provide almost all of the first two, and you will write the remaining three.

`NoisyButton`: This class describes buttons that act like those found in the 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.

`ButtonPanel`: This class creates and manages the set of four `NoisyButton`s that form the game board. We provide the source code for this class (except for the `getRandomButton` method described below).

`SimonController`: This is your main class. It extends `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.

`Song`: This class manages the sequence of buttons/tones corresponding to the "song" played by the game which the player needs to repeat.

`SongPlayer`: This class is an `ActiveObject` that will actually play the `Song`.

These five classes are described in detail below.

AudioClips, NoisyButtons, and ButtonPanels. This section describes the `ButtonPanel` and `NoisyButton` classes, and it also quickly reviews how to use `AudioClip`s.

- **AudioClip:** Our starter folder includes five audio files. 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 loses.

  Recall that you read audio clips with the `getAudio` command in your `SimonController` class, as in:

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

  You then play with noise with the play command:

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

- **NoisyButton**: The `NoisyButton` class produces buttons that look and act like those found on a Simon game. These buttons know how to beep and flash.

  Like other GUI components, a NoisyButton needs to have some other object that "listens" for events that involve the button. The object you choose to use as a listener for your game's buttons must implement an interface we have designed named `NoisyButtonListener`. We have included the definition of this interface in the starter file for this lab. The interface definition is:

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

That is, to listen for `NoisyButton` events, an object must provide a `noisyButtonClicked` method. 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 `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.

ButtonPanel: The other class we will provide is called `ButtonPanel`. It creates the collection of `NoisyButton`s that form the game board. The `ButtonPanel` class is also a GUI component. So, after creating a `ButtonPanel`, it can be added to your `SimonController`'s content pane. Our starter code does this for you.

The `ButtonPanel` class provides two methods:

- `addNoiseButtonListener`: This method assign a listener to all of the buttons in the button panel. It expects an object that implements the `NoisyButtonListener` interface as a parameter.
- `getRandomButton`: This method simply returns a randomly chosen button from the panel. It will be useful when you need to add a randomly-chosen button to your "song." You will write this method.

The `ButtonPanel` constructor requires that you pass it the `AudioClip`s 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 `AudioClip`s that refers to the four button sounds. You will have to create this array in `SimonController`'s begin method.

Simon Classes. To complete this program, you will need to construct the `SimonController` class to act as your "main program" and two classes that will manipulate the "song" played by the game.

- **Song**: 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 an array of `NoisyButton`s. 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!). You will need to use an integer instance variable to keep track of how many notes are currently stored in the sequence. The `Song` class should also have a `play` method that works with the `SongPlayer` class described below to play the notes stored in the array.

  You will also need a method for getting the "next" note that should be matched by the user when the user is playing back the song. This method will be used in the controller to check whether the user got the next note right. To make it possible to implement this method in the `Song` class, you will need a second integer instance variable that keeps track of which element in the song's array of `NoisyButton`s 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 should also be a boolean method to test whether there are `NoisyButton`s left to match, and a method to "rewind" the song so that the first `NoisyButton` in the array becomes the next to match.

- **SongPlayer**: In addition to the `Song` class, you will need a separate `SongPlayer` class to assist the `Song` class when it is told to play the song. This may be surprising because playing the song is fairly simple for the most part: 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 `flash` method). The problem is that in order to play the song correctly, you must 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`. Thus, the `SongPlayer`

class will be a class that extends `ActiveObject`. Whenever the `Song` class is asked to play itself, it will create a `SongPlayer` object 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.

- **SimonController**: This class ties together the whole game. The `begin` method should create a `ButtonPanel` and a `Song`. We have already written the code to load the audio clips into an array of sounds.

  This class should include a `noisyButtonClicked` method so that it can be used as the listener that determines how to react when the player clicks on a button.

  The `noisyButtonClicked` method is only called after the user clicks on a button. The actual button that the user pressed is passed as a parameter to that method. What happens next depends on whether or not the user clicked on the button corresponding to the next note in the song:

  - If the user clicked the wrong button, the program should make a nasty noise and start a new game (by creating the first note and playing it).
  - If the user clicked the right button, there are two possibilities. The first is that it was the last note of the song.

    If that button corresponds the last note of the song, the controller should add a new note to the song and play the entire song. After the song finishes, the user will be required to repeat the entire sequence again.

    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 `noisyButtonClicked` method will be executed again when the user clicks the next button.

  In summary, the `noisyButtonClicked` 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. As always, we will briefly examine each of your designs to make sure you are on the right track. Our design template is on the handouts page, if you wish to use it.

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

- A list of the 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;
- a sketch of the code used in the body of each method and constructor.

We would like you to design the `Song`, `SongPlayer`, and `SimonController` classes. Part of this design should be a sketch of how the `SimonController`'s `noisyButtonClicked` method will work.

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.

- Constructing the appropriate screen display is a good place to start. Write and test the portion of the `begin` method needed to create the `ButtonPanel` and add it to the display.
- 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 `ButtonPanel`'s `getRandomButton` method. Modify your `noisyButtonClicked` method so that each time you click it flashes a button randomly chosen by the `ButtonPanel` 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.

  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...
- Finally, complete the `Song` class by writing the three methods to "rewind" the `Song`, to get the next `NoisyButton` in the `Song`, and to test to see if you are at the end of the song.

  Once these methods are written, modify the `noisyButtonClicked` method so that it reacts as the rules of Simon dictate when the player clicks on the buttons.

As an extra touch: Your program will exhibit somewhat confusing behavior when the user clicks a button while the computer is playing a song. To avoid the confusion, you can have the game ignore clicks on the buttons while a `SongPlayer` is actively playing a song. As an small, optional extension, change your program to behave in this way.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder.
- Click "Go" and select "Connect to Server."
- For the server address, type in "Cortland" and click "Connect."
- Select the button next to "Guest" and click "Connect."
- A selection box should appear. Select "Courses" and click "Ok."
- You should now see a Finder window with a "cs134" folder. Open this folder.
- You should now see the drop-off folders for the three labs sections. Drag your program folder into the appropriate lab section folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click "OK."
- Log off of the computer before you leave.

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Labs, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before the 11 p.m. deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like "v2") and the resubmission will work fine.

Grading Guidelines

As on labs, we will evaluate your program for both style and correctness. Here are some specific items to keep in mind and focus on while writing your program:

Style
Descriptive comments
Good names
Good use of constants
Appropriate formatting

Design
Good use of boolean expressions, loops, conditionals
General correctness/design/efficiency issues
Parameters, variables, scoping
Proper use of arrays

Correctness
Initial set up
Playing songs
Comparing user input to remembered song
Restarting correctly when use makes mistake
Extending song correctly when user gets it right

Starter Code

NoisyButton Class

You will not need to modify this class.

```
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()
}
```

ButtonPanel Class

You will need to write `getRandomButton` in this class.

```
public class ButtonPanel extends Panel {

    // Number of buttons on board
    private static final int BUTTONCOUNT = 4;

    // Color value to use for buttons
    private static final int COLORINTENSITY = 180;

    // Array to hold buttons
    private NoisyButton[] buttons;

    /*
     * Construct button panel.  The sounds array should contain the
     * tones played when the buttons flash
     */
    public ButtonPanel(AudioClip[] sounds) {

        // 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 the four buttons and store them in the buttons array.
        buttons = new NoisyButton[BUTTONCOUNT];

        for (int buttonNum = 0; buttonNum < BUTTONCOUNT; buttonNum++) {
            buttons[buttonNum] =
               new NoisyButton(sounds[buttonNum], shades[buttonNum]);
            add(buttons[buttonNum]);
        }
    }

    /*
     * Add a listener to all four buttons
     */
    public void addNoisyButtonListener(NoisyButtonListener listener) {
        for (int buttonNum = 0; buttonNum < BUTTONCOUNT; buttonNum++)
            buttons[buttonNum].addListener(listener);
    }

    /*
     * Return a random button from the panel
     */
    public NoisyButton getRandomButton() {
        return null; // YOU MUST WRITE THIS
    }
}
```

SimonController Class

We provide the following starter file to help you get started:

```
public class SimonController extends Controller implements NoisyButtonListener {

    // the number of distinct sounds corresponding to the game buttons
    private static final int NUM_SOUNDS = 4;

    // a razzing noise
    private AudioClip nastyNoise;

    // the button panel
    private ButtonPanel buttons;

    public void begin() {
        // make window be a decent size for Simon.
        this.resize(250, 284);

        // 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");
        }

        // Add code to create the button panel and to add it
        // to the BorderLayout.CENTER of the content pane.

        Container contentPane = getContentPane();

        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 when its
           time to implement the rules of the game
        if (theButton == "the expected button") {
             // player got it right
        } else {
           // player goofed

        }
        */
    }
}
```