Bonus: Tetrominos User Interface with Swing

What you need to know about the built-in Java user interface library to write Tetrominos.

User interface coding is an involved task. You will not need to write any user interface code yourself for Tetrominos, since I will provide this portion for you. However, it's nice to know what types of things you can do with a user interface library, and this section will give a very brief introduction.

There are several user interface libraries for Java. We will use Swing, which is provided with Java. The user interface library handles putting windows, buttons, menus, and other elements on the screen, and allows for user input such as mouse motion, clicking, or key presses.

Let's start with an example. First, create a new project with Eclipse. Call it whatever you like; for example, `SwingExamples`. Create a new class, called SwingTest. Then copy-paste the following code into SwingTest. (This code is based on Oracle's HelloWorldSwing example.) Do not worry too much about the details of how the code works yet.

Run the code. If everything is working properly, a small window should appear somewhere on your screen containing the words "Hello, World". You can click the close button on that window to stop the program.

```java
import javax.swing.*;

public class SwingTest {
    private static void createAndShowGUI() {
        // create a new window, called a "frame"
        JFrame frame = new JFrame("Welcome to Swing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // create a label and add it to the window
        JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {

        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Starting the Swing program from main

Like any other Java program, the code starts executing in `main`. There's a fairly complicated command in `main`. Although we will not go into the details of how this works, the effect of the call to `invokeLater` is to call the method `createAndShowGUI()`. No other code should go in the main method; we may treat `createAndShowGui` as the entry point for the program.

The reasons `invokeLater` is needed are somewhat arcane, and have to do with the interaction between Swing and the operating system. The operating system captures events like key presses and mouse clicks from the user, and must pass those events along to a Swing program for handling. The call to the `invokeLater` method initializes Swing in such a way that it can properly receive and handle those events. To create a new Swing program, you can copy this `main` method in, without worrying about the details too much.

Creating a window

The Swing library is built around the idea of *components*: items like windows, labels, buttons, or drawing areas are created by created appropiate objects. The first line of code executed in `createAndShowGUI` creates a JFrame object representing a window on the screen. A window often has a title, shown at the top, and the constructor sets the title of the new window to `"Welcome to Swing"`.

```
    // create a new window, called a "frame"
    JFrame frame = new JFrame("Welcome to Swing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Methods on Swing objects can be used to change their behavior or properties. The call to `setDefaultCloseOperation` in this case indicates that the program should exit when the window is closed. (If the program allowed mutliple windows to be open, you might want some other behavior, and allow individual windows to be closed without quitting the program.)

Creating and adding a JLabel component

Components can act as containers. A window might contain a collection of buttons, some of which might contain text labels. In Swing, there are methods to add components to other components. For example:

```
// create a label and add it to the window
JLabel label = new JLabel("Hello World");
frame.getContentPane().add(label);
```

Although the `Frame` is a window, a window has different parts, including the title of the window, scrollbars, etc. `frame.getContentPane()` fetches the main region of the window where custom components might be added. Then `add` is called on that region to add the label to it.

Notice the chaining of method calls: `frame.getContentPane().add(label);` should be read left-to-right, recognizing that the result of `getContentPane` is itself a reference to an object upon which methods may be called.

Displaying the user interface

The size of the window should be determined by the components it contains. `frame.pack()` resizes the window appropriately so that the window is just big enough to hold the label.

It takes time to draw user interface components on the screen, and it is distracting if the user sees individual elements being added to a window one at a time, or if the window is suddenly resized on the screen without warning. By default, the window is invisible, so that initial set-up can be completed. Once ready, you can call `frame.setVisible(true)` to display the window.

Timer events

It can be useful for a program to take a certain action every once in a while. For example, in a game of Tetris, blocks must move down a row every 300 milliseconds or so. Swing provides the capabilities to do this, using a `Timer` object. If you add the following code to the end of `createAndShowGUI` after the frame has been displayed, the `println` method will be called every 300 milliseconds.

```
        Timer timer = new Timer(300, new ActionListener() {
                        public void actionPerformed(ActionEvent evt) {
                                System.out.println("Time is up!");
                        }

                });
```

The structure of the code above relies on some Java concepts that have not been introduced in these lessons. For now, it is enough to experiment with the results.

Keyboard events

It is also useful if a program can execute some piece of code every time a particular key is pressed. In the Tetrominos game, pressing the `s` key should trigger a method to be called that moves the piece to the left. Take a look at the provided code for the Tetrominos user interface in the next section to see how keys can be captured using a KeyListener object.

Example: the Tetrominos class

The Tetrominos class is the main entry point for the Tetrominos program, and you should already have it in Eclipse. Let’s take a high-level look at it:

`Tetrominos` opens a window with `new JFrame()` and displays it.

We use `frame.getContentPane().add(board);` to add `board` as a component to the window. Primarily, this means that the Board class can use graphics commands to draw onto the window.

Then a `Timer` is created that will call the method `nextTurn()` of the board class every 300 milliseconds. You'll write the body of the `nextTurn()` method later on.

Finally, keyboard input handling is set up. When the user presses one of the keys `s`, `d`, `f`, or `g`, certain methods of the Board class will be called.