Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: Java Applets with Swing

Most of our efforts so far this semester have involved Java applications: programs that interact with their users through the keyboard, a terminal window, and possible by reading and writing files.

One of the nice features of Java, however, is a rich set of support for programs with *graphical user interfaces (GUIs)*. These are programs that can also get input from a pointer device such as a mouse, and can display information with things like buttons and menus. We use such programs all the time.

In Java, such GUI programs are often developed as *applets*. Applets can be run in a standalone mode, through an IDE like BlueJ, or can be embedded in web pages. This ability to embed Java applets in web pages was a motivating factor in its design and certainly helped the language catch on.

We will study Java's *Swing components*, which improved upon the original AWT components from early Java versions.

The Gaddis text describes Java Swing components primarily using programs that construct objects of or extend the `JFrame` class, but we will focus on classes that extend the `JApplet` class. All of the concepts are and much of the code is the same.

Let's look at a simple `JApplet` much like the one you developed for the first mini-lab.

See Example: HelloWorldApplet

Even for this simple example, we can see several differences from the Java applications we have been using.

- The class header includes an extra directive `extends JApplet`. This tells Java that the class we are writing should *inherit* a collection of default functionality (such as the ability to create the GUI window) from another class called `JApplet`.
- Since `JApplet` is a Java GUI class, we need an appropriate `import` statement at the top of our program.
- Our program does not have a `main` method. The analog here is the `init` method, which will be called when we run the program. It is used to set up our GUI components.
- In this case, we are setting up just one simple GUI component: a *label*. We do this by constructing a `JLabel` object (after adding an appropriate `import` at the top). The `JLabel` constructor takes a `String` parameter which specifies the text to display in the label.

- In order for Java to know where we want this label, we need to tell the `JApplet` about it. We do this by passing the `JLabel` we construct to the `JApplet`'s `add` method. Since we are not specifying an object name before the method name as we often do, Java implicity adds `this.` and looks for a method in the current class. But we never wrote an `add` method! That's fine - it's provided by the `JApplet` class. Since our class `extends JApplet` it already knows how to do all of the things a `JApplet` knows how to to.

When we run this program, instead of choosing `main` from BlueJ's menu, we will choose "Run Applet". For now, we can accept the defaults in the window BlueJ pops up before running our program.

Multiple Contols in a Frame

Our first example is unrealistically simple – we'll want more than one Swing component in our window. Here's an example that adds a few:

See Example: ManyJLabels

When we run this, we find that we can only see the last one!

We need to give Java a bit more information about exactly where these objects are supposed to be placed.

See Example: ManyJLabelsContentPane

See the comments therein for information about the *content pane*, and how we can add items to the pane using its default `BorderLayout` configuration.

This is still very limited and quite ugly. We will see a number of other, more flexible mechanisms that we can use individually or in combinations to specify the layouts of our components. Rather than grouping them all together in a series of examples with boring `JLabel`s, we'll introduce them in subsequent examples when they become necessary.

Buttons and Event Driven Programming

Of course, we want to be able to interact with our applets in meaningful ways. We first must realize that applets are *event-driven programs*. An event-driven program responds to actions such as a mouse click or a key press by performing some specific action, then waits for the next event.

Java was designed with events in mind, and we will take advantage of this. It means we can write programs that respond to mouse movements and clicks, and we will use those programs to display and manipulate the components in our window.

Contrast this with the standard applications we have been working with for the most part to this point. Those programs start by executing a `main` method. When the `main` method returns, the program is finished. Here, we have the `init` method, but we will now see that the life of the program does not end when the `init` method returns.

To create an event-driven program, we'll need a program with a few new constructs and ideas.

See Example: ButtonClickCounter

There are a number of new constructs and important things to consider here.

First, we see a new Swing component, the `JButton`. These are the *buttons* you see all the time in dialog boxes in your favorite programs and on web pages and elsewhere.

When you press a button in a program, you usually expect the program to react somehow. There are a few steps we need to take to get our program to respond to clicks of our button.

First, we need to provide a class that has a method that will be called when someone presses the button. Such a class is called a *listener* class, in this case an `ActionListener`. An `ActionListener` class must do two things:

1. Specify `implements ActionListerner` in the class header.
2. Provide an *event handler* method `actionPerformed` that takes a parameter of type `ActionEvent`.

Doing the first will cause Java to force us to do the second. `ActionListener` is a Java *interface* that gives Java a list of methods (in this case, just the one) that the class promises to implement in exchange for being recognized by Java as a valid instance of that interface. Much more on this later.

Once we have a valid `ActionListener`, we can tell our button to call it when someone presses. This is done with the `addActionListener` method.

A few other notes about this example:

- Notice that the `init` method never explicitly calls the `actionPerformed` method. It only sets up a listener so that it will be called when an appropriate event occurs. In this case, when someone clicks on the button.
- The class has 2 instance variables and the `init` method has 2 local variables. The only variables that are declared as instance variables are those whose values need to be available in both methods and retain their value across method calls.
- We can update the text displayed by a `JLabel` using the `setText` method.

Admittedly, the window that comes up is kind of ugly. The label is off to the left and in a pretty small font, the window is huge for our purposes, and the button is stretched across the bottom of the window.

We can make some simple enhancements to this program to make it look nicer:

See Example: ButtonClickCounterPanels

Items of note:

- We use the `setSize` method in `init` to change the size of the window (in pixels).
- We place the components in `JPanel` objects, and add the `JPanel`s to the pane. This has the effect here of centering the components and making the `JButton`'s size just large enough for the text it contains.
- We use the `setFont` method of the `JLabel` to make it bigger and bold.
- There is a conditional in the `actionPerformed` method to handle the special singlular case: "1 click".

Events from Multiple Components

In the event-driven examples so far, there has been just one component that could trigger an event. More meaningful programs will have many components that trigger events.

Our next example tracks votes in a 2-candidate election, where votes are cast by clicking on one of the buttons.

See Example: Voting

What's new here?

- We have two `JButton` objects that both will trigger `actionPerformed` method calls when pressed. Since we have just one `actionPerformed` method, we need to remember references to our `JButton` objects as instance variables, and check to see which button triggered the event. Fortunately, we can get this information from the `ActionEvent` object that is sent to our `actionPerformed` method. The `getSource` method of this event will return a reference to the component that triggered the event. We can then check to see which button that is.
- We have multiple components being placed into each `JPanel` we create. Notice that they are placed side by side in the center of the portion of the window to which we add the panel.
- To change the font size, we retrieve the current `Font` and use its information to construct a new `Font` with the same font name and style, but a point size of 30.

Let's expand this to allow voting for more candidates.

See Example: VotingMany

Here, we have arrays of `JButton`, `JLabel`, and `int` to track the buttons, labels, and vote counts for each candidate, and use loops to construct and search among them as votes are cast.

This program produces a window that is still kind of ugly, but we can improve on it by introducing a new *layout manager*:

See Example: VotingManyLayout

Here, we set the layout of the content pane, so it no longer expects to have components in the north, south, east, west, and center. The `GridLayout` we create will have 6 rows and 1 column, which will be populated with components 1 at a time with the default `add` method. Note further that we place a `JPanel` containing the 5 `JButton` objects in the last row.

Text fields and Exception Handling

The next Swing component we consider is the `JTextField` – an editable box, or *text field*, where the user can enter text.

The example we will use is a program which computes the prime factorization of an integer.

See Example: PrimeFactors

We see that a `JTextField` can be constructed with an initial value (a `String`) and an `int` that specifies the width of the field (*i.e.*, the number of characters that will be visible in the field).

In the `actionPerformed` method, we begin by retrieving the `String` currently displayed in the field, using its `getText` method.

In this case, we further need to convert this to an `int` value. We have done this before, but here we see how to add some error checking to the process. In previous examples of `Integer.parseInt`, the program would crash with a nasty message if the `String` passed in was not able to be converted to an `int`. That crash was caused by the method *throwing an exception* – meaning that the method encountered a situation where it was unable to complete its task successfully, so it cannot return a valid answer. If we don't do anything about the thrown exception, it causes Java to kill our program.

But... we can do something: we can *catch* the exception. The example shows the code that can potentially cause an exception to be thrown inside of a `try` block, followed by a `catch` block that lists the type of exception we would like to handle. In this case, it is a `NumberFormatException`. If that kind of exception is thrown by any statements inside the `try` block, the statements inside the matching `catch` block are executed. Here, we just pop up an error message and let the user try again, but we could do whatever we want. In this case, we also return from the method, but nothing stops us from having the execution continue with the statements after the `try`..`catch` blocks, if that is what is appropriate.

Other items of interest here:

- We can call `setText` of a `JTextField` to change its contents.
- We can call `addActionListener` on a `JTextField` and have an `actionPerformed` method called to handle an action event. It will be triggered if the user hits Enter while focus is in the text field.

A Game

We now know enough to build some playable games. Here is a Tic-Tac-Toe game that uses `JButton`s in a `GridLayout`.

See Example: TicTacToe

The comments describe the program fairly thoroughly, but a few things worth pointing out here:

- We retain the default `BorderLayout` for the base content pane. In each of the `NORTH` and `SOUTH`, we place a `JPanel` with one component each. In the `CENTER`, we place a `JPanel` but then set it's layout to be a 3×3 `GridLayout`.
- The 9 `JButton`s that are used to represent the Tic-Tac-Toe squares are placed into the `JPanel`'s grid.
- Since we only want to be able to restart the game when there is not an unfinished game in progress, we disable the "Play Again" button when the game starts, and enable it when someone wins or a tie is detected. This is done with the `setEnabled` method.
- In addition to the X's and O's displayed on the game buttons, there is also a 2D array of `int` maintained with one of 3 values to represent empty cells, cells with an X, and cells with an O.
- The program uses a number of private helper methods to simplify the code. While none of these are strictly necessary, notice how they keep the code much more readable. Each helper method takes care of a very specific task.

Lots More!

Once you have the basics of the Java Swing GUI components down, learning more is really a matter of looking up the details of exactly what you want to do and writing the code to do it. There are IDE tools that can help set up complex windows. However, we will look at one more, much larger example, that is developed manually.

See Example: FiveGuys

This applet is intended to simulate taking orders at a fast food restaurant, modeled roughly on the menu at Five Guys.

The comments in the example and the API documentation have the details, but the example includes many items we had not previously seen:

- GUI concepts and constructs: text areas, scrolling areas, combo boxes/drop-down menus, checkboxes, radio buttons, spinners, sliders
- Swing components and related methods: `JTextArea`, `JScrollPane`, `JComboBox`, `JCheckBox`, `JRadioButton`, `JSpinner`, `JSlider`, and `JSeparator`
- Layouts: `BoxLayout`

- Listeners: `ItemListener`, `ChangeListener`

There are others, but having seen these, you can certainly learn about those on your own.