CSCI 479 **Developing GUIs in Java** Fall 2009

# Developing GUIs in Java

**Objectives:**

1. To explore GUIs in Java.
2. To write Java applications that have windows, simple graphics, GUI components, and menus.
3. To use Java listeners for event handling.

**Preparation:** Before attempting the exercises, read the following chapters in *Java: How to Program* by Deitel and Deitel, eighth edition, Chap. 14 *GUI Components: Part 1*, Chap. 15 *Graphics and Java 2D*, Chap. 25 *GUI Components: Part 2*. The sixth and seventh editions have similar chapters but are numbered differently.

**Assignment:**

These exercises gets you started in writing window-based Java applications, i.e., GUIs.

**Exercise 1. Using the Java API:**

Java is a smaller and cleaner language than C++. Chapters 1-11, and 16 of the Java text cover most of the language except threads (Chapter 26). The reason why programmers like Java is the HUGE standard *Application Programming Interface* (API). Sun's API includes classes for developing *Graphical User Interfaces* (GUIs), multimedia, networking, web-based computing, database connectivity, distributed objects (RMI and CORBA), security and others. This is the fun part of programming in Java!

Take a few minutes and explore Sun's API for Java 2, Standard Edition 6 at URL:

`http://java.sun.com/javase/6/docs/api/`

You should become comfortable looking up stuff in API. Make it a habit. Don't memorize. Look it up!

Many other Java APIs are available from third party sources. You only need to Goggle search the Web with "java api".

**Exercise 2. A Window-based Java Application:**

Below is the bare bones of a Java application that opens a frame (window). The application uses **JFrame** class which is part of the **swing** API. See pages 561 and beyond in Java text.

```
import javax.swing.JFrame;

public class Viewer {

   public static void main(String[] args) {

      JFrame frame = new JFrame();

      // Set size of frame
      frame.setSize(250, 250);
```

```
        // Set title in top bar
        frame.setTitle("View a frame");

        // Action to do when frame is closed
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Insert components here

        // Make frame visible
        frame.setVisible(true);
    }
}
```

Copy the code to a file, compile and run it.

The above skelton is an excellent start for any window-based Java application.

**Exercise 3. Adding GUI Components to the Frame:**

GUIs are built by adding *GUI components* to a frame. Some possible components are text fields, labels, buttons, check boxes and radio buttons. See Chapter 14 of Java text.

Copy the file from Exercise 2 to a new file and set the layout for the window to **FlowLayout** by

```
frame.setLayout(new FlowLayout());
```

To the frame add two **JLabel** objects initialized to some text.

**FlowLayout** says to place the components one after the other until the components no longer fit across the window then start a new row. Layouts in Java take a little getting use to. The idea is that when a user resizes the window the components flow around to fit the new window size.

After displaying the two **JLabel** objects, adjust the size and shape of the window to see the behavior.

**Exercise 4. Adding JTextField and a Listener:**

In this exercise we will add a **listener** to capture the text typed in a **JTextField**. **Listener**s are the way Java's API does event handling. See Chapter 14 of Java text.

Copy the file from Exercise 3 to a new file. Create a second class, e.g., MyComponent1, where you pass the frame object as a parameter to the constructor.

```
MyComponent1 forTextField = new MyComponent1(frame);
```

The reason for doing this is that you can't create instance fields (variables) and access them in a main method because it is static.

In the class MyComponent1 add an instance field **JTextField** object of width of 20 characters In the constructor, add the **JTextField** object to the frame, create a new **TextFieldHandler** object and add the **JTextField** object to **ActionListener**. See page 568 in Java text for information on **JTextField**. Create your own inner class to handle the event and call it **TextFieldHandler**. Display what is typed in the **JTextField** object in the shell window using **System.out.println()**. Since the listener inner class needs to access the **JTextField** object, we made it an instance field.

**Exercise 5. Adding a Menu:**

Menus are an important part of GUIs. See section 25.4 starting on page 1019 of Java text.

Copy the file from Exercise 4 into a new file. Add a menu bar with the label "File" that has an "Exit" item on it to quit the program. This code can be added to the Viewer class.

When you run it, notice that the new menu bar shifts the **JLabel** and **JTextField** components down to make room.

**Exercsie 6. Adding Simple Graphics to the Window:**

Drawing lines, rectangles, and circles are easy in Java but a bit tricker if you want to draw as well as have other graphical components on the screen. See Chapter 15 of Java text.

Copy the file from Exercise 5 to a new file.

You should *avoid* the older approach of **AWT** which used the **paint()** method. We strongly urge you to use the newer and much improved **swing** approach which uses **paintComponent()**. For example, if you have a Java program with an animation, **paintComponent()** will refresh the screen automatically for you while **paint()** does not.

To use **paintComponent()**, you must create a **JPanel** object. A **JPanel** creates a drawing area for graphics. See pages 138-141 about **JPanel**. A good way to do this is to create a second file with a class that **extends** the **JPanel** class, e.g., **MyJPanel**. Inside this extended class insert your **paintComponent()** method. You will need to add **super.paintComponent(g);** as the **first line** of your **paintComponent()** method, otherwise you will not see the graphical components like JTextfield.

Add lines in the **paintComponent()** method to display an orange rectangle and some blue text. See Chapter 15 of Java text for details.

BEFORE you create an object of **MyJPanel** and add it to the frame, you need to be careful with your Java layout. In Java the default layout is **BorderLayout**, which has five regions, NORTH (top), SOUTH (bottom), EAST (right), WEST (left) and CENTER. If you don't specify when you add a component, it goes in the CENTER. If you add two components to the CENTER, the second overwrites the second. It is very easy to do this and find yourself cursing "Where in the Heck is my drawing?"

Since **JPanel**s and FlowLayout don't seem to get along, change your program to use **BorderLayout** and place the panel in the CENTER, the two **JLabel**s to NORTH and WEST and the **JTextField** to SOUTH.

To create a line border around the panel, use the following line right after you create the panel.

```
panel.setBorder(BorderFactory.createLineBorder(Color.black));
```

If you want to use methods from Graphics2D library, cast g to g2 such as

```
// Recover Graphics2D
Graphics2D g2 = (Graphics2D) g;
```

When you run your **paintComponent()** method you override the **paintComponent()** method in the superclass **JPanel**. **JPanel** automatically calls the **paintComponent()** method after creating the frame and after any *expose window event*. Your Java frame receives an expose window event when the window is minimized (made an icon) and then maximized (icon opened). An expose event also happens when the window is redrawn after another window has overlapped it. Try both of these situations to see what happens.

**Hand in**

For Exercise 6, hand in the java code and a snapshot of the screen. Use the Linux tool `xv` to take a snapshot of the window. Print the java code using the **a2ps** command.