Graphical User Interfaces in Java

Using the Java Swing Library

COLLEGE OF COMPUTER STUDIES, DE LA SALLE UNIVERSITY

August 13, 2009
Authored by: Paul Inventado

Graphical User Interfaces in Java

Using the Java Swing Library

INTRODUCTION

Graphical User Interfaces

Up until now, we have been retrieving and displaying information by using the command line. This is usually called the command line interface (CLI). However, as you already know, this type of interface is quite difficult for a user to use, not very intuitive and not very appealing to the user. This called for a more visual or graphical interface which is now commonly seen as windows, icons, pictures, etc… on your desktop and even on websites. Thus through graphical user interfaces (GUI) we hope to improve the user experience.

GUI in Java

Java allows it's developers to create graphical interfaces for their applications through the libraries that they provide. Initially, the Java Abstract Window Toolkit (AWT) was Java's flagship library for GUI. However, it was later deemed quite heavy on the resources and not very compatible with some operating systems. This called for another GUI library which is now known as Swing.

The Java Swing Library

The Java Swing Library was not actually built from scratch but was built on top of Java AWT. The library contained modifications of some components from AWT and still use some of its components. You will notice that as you use Swing, you will see it use AWT components from time to time.

Users of the Java Swing Library enjoy its benefits over AWT such as having more lightweight components, more control over components, more features, and more active support from its developers.

USING THE JAVA SWING LIBRARY FOR CREATING GUI

The basics

The simplest way to design and create your GUI is by creating a class specifically for it. This will require you to have one class which will serve as a Driver for your application which contains the *public static void main(String[] args)* method. Another class is then needed to hold the actual GUI and is simply created and called from the Driver.

Let's start by creating a very simple application which simply displays a window on the screen. First, we will create the class for handling the GUI. This is shown in the code below.

```
import javax.swing.JFrame;
public class GUI extends JFrame
{
  public GUI()
  {
     setTitle("My GUI");
     setSize(200,200);
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     setVisible(true);
  }
}
```

The first step in creating the GUI application is importing the components from the Swing library. In this case, we are using the JFrame class which allows us to create frames or windows. When we create our class, we *extend* the JFrame class so we can reuse the methods that it already provides. We then create a constructor, which you can visualize as a special type of method, which simply contains the operations that will be done when we create the GUI object. A constructor is created by using the following syntax:

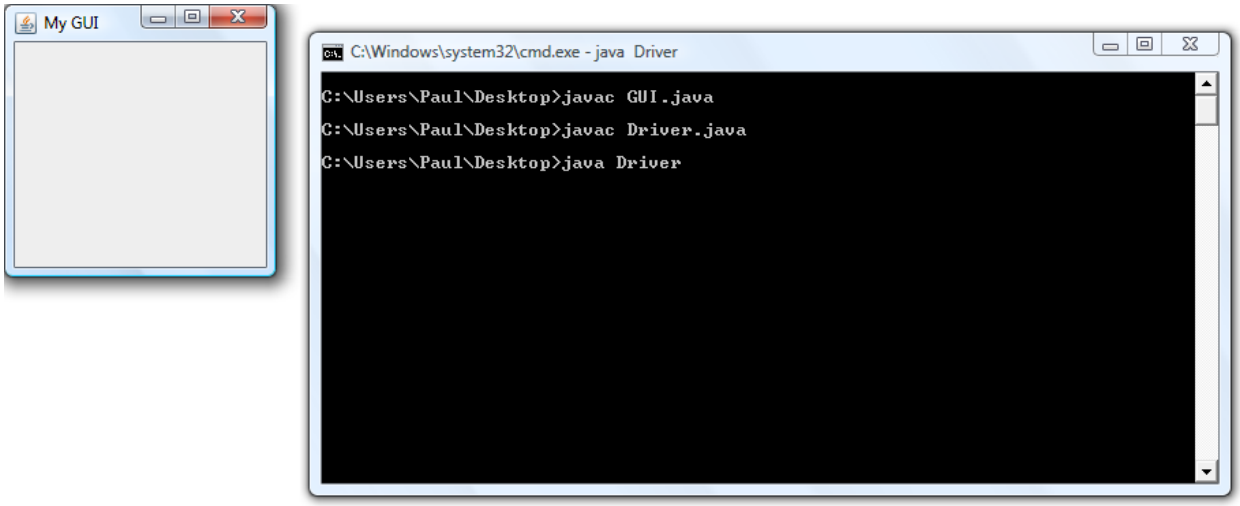
```
<access modifier> <class name>(){
      <statement>;
}
```

The access modifier we are using is *public* which means it is accessible to anyone, and the class name is simply provided. Inside the constructor we then call the methods that are already present in the JFrame class. In this case, we use the *setTitle()* method to change the title of the window. *setSize()* to specify the dimensions of the window, *setDefaultCloseOperation()* indicates what happens when the close button is pressed which in this case exits the program and *setVisible()* shows the window on the screen. You can take a look at JFrame class in the Java API to see all methods that it supports.

Now that the GUI class has been created, we simply have to create the GUI object through the Driver. This is shown in the source code below.

```
public class Driver{
  public static void main(String[] args)
  {
    GUI myWindow=new GUI();
  }
}
```

Save the two files as GUI.java and Driver.java. Compile both files to produce their respective byte codes. Java will only require the class containing the *public static void main(String[] args)* to run the program. In this case, just run the Driver class.



As shown in the illustration above, compiling and running the Driver class will result in displaying a simple window with features as specified by the methods in the GUI class.

Creating Graphical Components

Graphical components make up the actual graphical interface. These are components which the user interacts with to either provide inputs or view outputs. There are a lot of components in the Java Swing library so we will only focus on those that are more commonly used.

JLabel (javax.swing.JLabel)

The JLabel component is used to display text labels. It provides many constructors that can be used but the more commonly used is the *JLabel(String text)* constructor. It simply allows you to create the JLabel with a specific String value as shown below.

```
JLabel nameLabel = new JLabel("Name:");
```

Name:

JTextField (javax.swing.JTextField)

The JTextField component is used to create an input box where the user can input values. Again, it provides different constructors but the most commonly used is the *JTextField()* constructor which allows you to create an empty input box as shown below.

```
JTextField input = new JTextField();
```

JButton (javax.swing.JButton)

The JButton component is used to create button which allows the user to prompt the program to perform an action like "Ok" or "Cancel." Like the others, it has many constructors but the most used is the *JButton(String text)* constructor which creates a button with the indicated String provided as shown below.

```
JButton submitButton = new JButton("Submit");
```

Submit

Getting and setting the properties of graphical components

A very good advantage in the design of the Java Swing library is that it allows the programmer to easily modify the properties of a component. It is possible for example to set the text in a JLabel to a different value by using it's *setText(String text)* method. The opposite is also possible where you can get the current value of the JLabel's text by

using it's *getText()* method. setText() and getText() are not the only methods provided in the library. There are methods that allow the changing of background colors, providing pictures or icons and many more. All of these can be found in the respective component's API.

Other Graphical Components

Apart from the JLabel, JTextField and JButton classes, there are a lot more components that you may opt to use depending on the interface you would like to provide for your user. Some of them are shown in the table below.

Swing Component	Description
JCheckBox	Allows the user to select multiple options
JRadioButton	Allows the user to select an option from a group
JList	Allows a user to select single or multiple items in a list
JPanel	Used for containing a set of components or other panels
JScrollPane	A Panel which automatically provides scrollbars when content does not fit the container

Adding Graphical Components into the Window

Now that we know how to create a window and how to create the component, the next challenge would be placing the components into the window.

Setting the size and Location of the components

As was mentioned earlier, Swing provides many methods to manipulate components. In particular, the *setSize(int width, int height)* and *setLocation(int x, int y)* methods can be used to control the size and location of the components. These methods can be called for each component to position them in the window. Unlike how we usually imagine, the x and y locations however are no longer based on the Cartesian plane. Given an x value of 0 and a y value of 0, this now refers to the upper left most position inside the window. As x increases, the position moves to the right, and as y increases the position moves down. Thus, having negative values for x and y would mean that the object is outside the viewing area and is outside of the window. Below is an example of how the size and location of a JLabel object can be set. In particular, the JLabel's width is set to 100 pixels, its height to 20 pixels, its position 10 pixels from the left and its position 20 pixels from the top.

```
JLabel name = new JLabel("Name:");
name.setSize(100,20);
name.setLocation(10,20);
```

Setting the layout

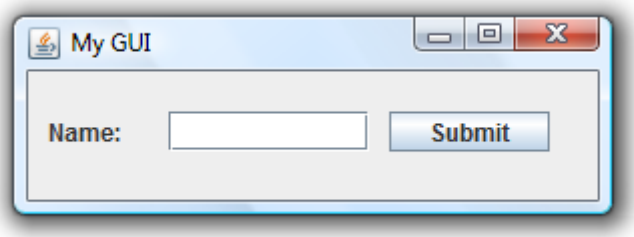
Swing automatically uses a layout manager which does not use the specified size and location in the object. In this case, to indicate that Swing uses the size and location we provided, we set the layout manager to null as shown in the code below.

```
public GUI()
{
  setTitle("My GUI");
  setSize(200,200);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setLayout(null);
}
```

Putting it all together

The only thing left would be to add each of the components into the window. This is done by using the *add()* method of the JFrame class, which allows the addition of any component into the window. If we compile all the things we learned from the previous sections, we will have the following source code and outputs.

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JButton;
public class GUI extends JFrame
{
  public GUI()
  {
     setTitle("My GUI");
     setSize(300,100);
     setDefaultCloseOperation(EXIT_ON_CLOSE);

     setLayout(null);

     JLabel name = new JLabel("Name:");
     name.setSize(50,20);
     name.setLocation(10,20);

     JTextField inputBox = new JTextField();
     inputBox.setSize(100,20);
     inputBox.setLocation(70,20);

     JButton submitButton=new JButton("Submit");
     submitButton.setSize(80,20);
     submitButton.setLocation(180,20);

     add(name);
     add(inputBox);
     add(submitButton);

     setVisible(true);
  }
}
```



HANDLING USER INTERACTION

Event-based Programming Paradigm

The Java Swing library uses the event-based programming paradigm which means, the developer is given a set of events that the library is able to detect, and is required to simply provide the action that will be performed when the event happens or is triggered. Examples of such events are the clicking of the button, the movement of the mouse, typing into an input box and so on.

Action Listeners (java.awt.event.ActionListener)

One of the most important event handler is the *ActionListener* interface. This interface allows users to handle events related to interaction with components in the window called an *ActionEvent*. Using the *ActionListener* requires three things: implementing the *ActionListener* interface, creating an *actionPerformed()* method and adding the current class as the *ActionListener* to each component we that we expect the user to interact with. This is shown in the code snippet below.

```
import javax.swing.JFrame;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class GUI extends JFrame implements ActionListener
{
  public GUI()
  {
    //operations
    JButton submit = new JButton("Submit information");
    JButton cancel = new JButton("Cancel submission");
    submit.addActionListener(this);
    cancel.addActionListener(this);
    //other operations
  }
  public void actionPerformed(ActionEvent event)
  {
      //operations
  }
}
```

The next step in handling an *ActionEvent* would be identifying which component triggered the event. This can be done by using the *getSource()* method. This method returns a reference to the actual object that triggered it. The result of the method can then be compared with the actual object. Below is a program which attempts to apply this approach.

```
import javax.swing.JFrame;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class GUI extends JFrame implements ActionListener
{
  public GUI()
  {
    //operations
    JButton submit = new JButton("Submit information");
    JButton cancel = new JButton("Cancel submission");
    submit.addActionListener(this);
```

```
    cancel.addActionListener(this);
  }
  public void actionPerformed(ActionEvent event)
  {
    if(event.getSource() == submit)
      System.out.println("The submit button was pressed");
    else
      if(event.getSource() == cancel)
        System.out.println("The cancel button was pressed");

  }
}
```

Unfortunately when this is run, the submit JButton object is not recognized. This is because of the scope of the submit button variable which only lies within the constructor. This problem is solved by making the declaration of the submit and cancel buttons outside of the methods so they are accessible both by the constructor and the *actionPerformed()* method. The complete code is shown below.

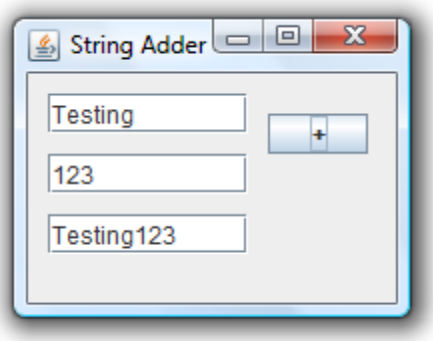
```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class GUI extends JFrame implements ActionListener
{
  JButton submit;
  JButton cancel;

  public GUI()
  {
    setTitle("My GUI");
    setSize(450,100);
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    setLayout(null);

    submit = new JButton("Submit information");
    submit.setSize(200,20);
    submit.setLocation(10,20);

    cancel = new JButton("Cancel submission");
    cancel.setSize(200,20);
    cancel.setLocation(220,20);

    submit.addActionListener(this);
    cancel.addActionListener(this);

    add(submit);
    add(cancel);

    setVisible(true);
  }
```

```
  public void actionPerformed(ActionEvent event)
  {
    if(event.getSource() == submit)
      System.out.println("The submit button was pressed");
    else
      if(event.getSource() == cancel)
        System.out.println("The cancel button was pressed");
  }
}
```

Having the *ActionListener* set up correctly, it is now easy to provide logic in your program. Below is a sample program which adds two numbers provided by the user in a text field and displays the result in another text field. The screen output is also shown.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JTextField;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class Concatenator extends JFrame implements ActionListener
{
  JTextField string1;
  JTextField string2;
  JTextField sum;
  JButton addButton;

  public Concatenator()
  {
    setTitle("String Adder");
    setSize(200,150);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(null);
    string1  = new JTextField();
    string1.setSize(100,20);
    string1.setLocation(10,10);
    string2 = new JTextField();
    string2.setSize(100,20);
    string2.setLocation(10,40);
    addButton = new JButton("+");
    addButton.setSize(50,20);
    addButton.setLocation(120,20);
    sum = new JTextField();
    sum.setSize(100,20);
    sum.setLocation(10,70);

    addButton.addActionListener(this);
    add(string1);
    add(string2);
    add(addButton);
    add(sum);
    setVisible(true);
```

```
  }
  public void actionPerformed(ActionEvent event)
  {
    if(event.getSource() == addButton)
    {
      String concatenated = string1.getText() + string2.getText();
      sum.setText(concatenated);
    }
  }
}

public class Driver
{
  public static void main(String[] args)
  {
    Concatenator myWindow=new Concatenator();
  }
}
```



Other Event Handlers

Apart from the ActionListener, other event handlers are provided to support other types of events. Examples of such are the MouseListener to detect the click and interaction of the mouse with a specified component, MouseMotionListener to detect the movement of the mouse and the KeyListener which detects the keys pressed on the keyboard. All of these handlers follow the same format as the ActionListener where we implement the interface, provide the methods required by the interface and add them as a listener to specific components.