Head First Java: Chapter 12 Notes

September 6, 2019

Graphical User Interface in Java

Many toolkits and packages are available in Java to implement GUIs. Among these, the package `Swing` and `AWT` are both robust and easy to use. Like any other aspects in Java programing, elements in `Swing`-implemented GUIs are objects, and the properties of each elements can be adjusted and modified with the object's instance variables and methods. To use `Swing` in Java, `java.swing.*` must first be imported. In a single window implementation of a `Swing` GUI, all desired elements desired take place in a `JFrame`; an object that represents a window in the user's OS:

```
JFrame frame = new JFrame();
```

Here the object reference variable `frame` is assigned to a new `JFrame` object (also named `frame`). The `JFrame` object has many methods available to adjust its appearance:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(300, 300);
frame.setVisible(true);
```

These method calls sets the closing-behavior, dimension in pixels, and visibility respectively. Additional elements cannot be directly added to the `JFrame`, but rather on its `ContentPane`. These elements can be buttons, check boxes, text field, radio buttons, and so forth, collectively referred to as *widgets*. All widgets are objects, and must be instantiated and assigned first before being added to the content pane:

```
JFrame frame = new JFrame;
JButton button = new JButton(''Text on the button'');

frame.getContentPane().add(BorderLayout.CENTER, button);
```

A couple of things of note here. Firstly, as mentioned the newly instantiated `button` (of the `JButton` class) cannot be directly added to `frame`. Within the `JFrame` object `frame`, there exist a private instance variable that references the actual content pane of the `JFrame` object in which widgets can be added onto. In Java, it is possible to use chained `.`-operator to access methods of a returned variable, which is how the code above was able to access the `add()` method of the content pane object within `frame`. The second thing of note is that `BorderLayout` is the default layout manager for `JFrame`, however accessing it requires importing `java.awt.*`. Because code exist within

`JFrame` object constructor to account for `BorderLayout` usage, if a new instance of `JFrame` object has already been instantiated then there is no need to instantiate `BorderLayout` itself to use it as a positional argument for widget placements. To specify position placements using `BorderLayout`, `CENTER, NORTH, SOUTH, EAST`, and `WEST` can be used.

Swing GUI Dynamics, Events, and Listeners

Widgets on a GUI can be interacted by the user to effect changes to the program. To do this, the GUI has to know what event would trigger a change, as well as the nature of the change itself. In `Swing`, most widgets are programed to generate an `ActionEvent` object when interacted with (*e.g.* a `JButton` object being clicked on). The generated event object carries the data about the nature of the event itself, but doesn't affect the program in anyways unless the program has mechanisms in place to detect the object and react to it. This is called *listening*, and there are numerous class in `java.awt.event.*` that is programed to listen to different types of events. For example, `ActionListener` detects and reacts to general interaction with a widget, `KeyListener` detacts and reacts to keyboard keys being pressed and released. All listener classes are interfaces containing abstract methods that must be implemented. For example:

```
import javax.swing.*;
import java.awt.event.*;

public class SimpleGui implements ActionListener {

    JFrame frame = new JFrame();
    JButton button = new JButton(''A Button'');

    public void actionPerformed(ActionEvent event) {
        button.setText(''I have been clicked.'');
    }

    public static void main(String[] args) {
        SimpleGui gui = new SimpleGui();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.getContentPane().add(button);
        button.addActionListener(this);
        frame.setVisible(true);
    }
}
```

The above code creates a window with a button, that when clicked on the text changes from "A Button" to "I have been clicked." The

code within the `main()` method can be examined line-by-line:

1. `SimpleGui gui = new SimpleGui();`

   Instantiate a `SimpleGui` object, which also declare and assign the variables `frame` and `button` their respective instance of `JFrame` and `JButton`.

2. `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`

   Set `frame` to exit the program when closed.

3. `frame.setSize(300, 300);`

   Set `frame` dimensions to 300 by 300 pixels.

4. `frame.getContentPane().add(button);`

   Place `button` on the content pane of `frame`.

5. `button.addActionListener(this);`

   Here is the key line of code that acts dynamics to the window. Recall that `Swing` widgets just as `JButton` objects are *event* sources, and many different types of events can be generated. In this case, because one is only interested in detecting when the button was clicked on, the program should be able to *listen* for `ActionEvent` (the type of event object generated when the button was meaningfully interacted upon; ActionEvent object generation is widget dependent.) The corresponding listening is `ActionListener`. Recall that all listeners are *interfaces*, which contain only abstract methods that must be implemented. The `button` object is firing `ActionEvent` objects whenever it is being clicked on, but it doesn't know what `ActionListening` is receiving the event. There are a number of ways to go about this at this point, but here `SimpleGui` itself as a class implements the `ActionListener` interface, which means `SimpleGui` IS-A `ActionListener`.

6. `frame.setVisible(true)`

   Set `frame` to be visible.

When `main()` is running via the JVM, the `JFrame` object `frame` is created (technically the `SimpleGui` object is created first; `frame` is an instance variable of `gui`) and displayed, along with all widget objects defined in the method. Aside from the `main()` method, the class `SimpleGui` also has another method `actionPerformed()`. The `actionPerformed()` method is inherited from the implementation of the `ActionListener` interface, and because it is an abstract it must be implemented explicitly here. The method `actionPerformed()` is not called directly in the written code, but rather it is called automatically when an `ActionEvent` object is received by the `ActionListener`,

which in this case is `gui`. This has the practical effect of running the code body of `actionPerformed()` every time `button` is clicked on. In summary, here is the order and relationship of event source and event listeners:

1. A class of widget fires event objects based on certain conditions being met. *E.g.* `JButton` fires `ActionEvent` if clicked on.
2. Implement the appropriate event listener interface to a class. *E.g.* implement `ActionListener` to `SimpleGui`.
3. Explicitly implement all abstract methods inherited from the listener interface (there may be more than one) within the class where the interface was implemented on. *E.g.* Implement `actionPerfromed()` within `SimpleGui`.
4. Establish event source and event listener link by calling the appropriate add listener method from the widget (the event source) while passing the event listener as the argument. *E.g.* Calling `addActionListening()` from `button` while passing `gui` as the argument (or, in this case, passing `this` as the argument because the method call was performed within the cod body of `gui`.)

Graphical Widgets in GUI

One of the common ways to add graphical elements to a GUI (such as a `JFrame`) is to use objects from the `java.awt.*` package. In `Swing`, a `JPanel` is a widget that can be added onto a `JFrame` that has a high level of flexibility to customize its graphical appearance. This can be achieved by extending a subclass of `JPanel`, and using graphical objects from `AWT` to place geometrical shapes, graphical files (.jpg, etc.), or even simple animations. For example:

```
import java.awt.*;
import javax.swing.*;

class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {
        g.setColor(Color.orange);
        g.fillRect(20, 50, 100, 100);
    }
}
```

The `paintComponent()` method is inherited from `JPanel`, and its code body should be overridden — as it was here — add the desired graphical element to the `MyDrawPanel` (A subclass of `JPanel`. The `paintComponent` method itself do not need to be called anywhere

else; once the `MyDrawPanel` is initialized and added to the `JFrame` content pane, the `paintComponent()` method is called automatically behind the scenes. There is some complexity for the argument of `paintComponent()`. This method is `protected`, so the argument it accepts cannot be changed from `Graphics g`. The class `Graphics` is an object from `AWT` that contains many methods for adjusting graphical properties and behavior (such as `setColor()` and `fillRect()` in this case.) However, more advanced graphical methods is not available in `Graphics` but rather in its subclasses, such as `Graphics2D`. Because the argument declaration of `paintComponent()` is protected, the argument must be passed as `Graphics` even if more advanced graphical methods is needed that is unavailable. The solution to this is to cast the `g` parameter from `Graphics` class to `Graphics2D` within the method call itself. For example:

```
class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        GradientPaint gradient = new GradientPaint(70, 70 Color.blue, 150, 150, Color.orange);

        g2d.setPaint(gradient);
        g2d.fillOval(70, 70, 100, 100);
    }
}
```

Inner Classes

Classes can be nested within a class. These are known as inner class. In the memory heap, there do not existing within each other *per se*, as the relationship between an outer class and an inner class is not the same as that of a superclass and a subclass. A object created as some superclass can be cast into one of its subclass later, but no new object was created; any interaction with the superclass version of the object or the subclass of the object with with the *same* object on the memory heap (they may have different object reference variables, however, which may exist on the heap or on the stack). Any inner class object can only be instantiated *after* an object of its outer class has been instantiated. The inner class object and the outer class object, are *separate* objects that exist on different memory address in the heap. However, the two objects are linked, so that the they can use each other's instance variables. Consider:

```
class MyOuter {

    private int x;
    MyInner inner = new MyInner();
```

```
    class MyInner {

        void go() {
            x = 42;
        }
    }

    public void doStuff() {
        inner.go();
    }

}
```

The (outer) class `MyOuter` has a private (it cannot be accessed by anything outside of the class) instance variable `x`. It also defines an inner class `MyInner`, that has a method `go()`. Because `MyInner` is an inner class of `MyOuter`, any object instance of `MyInner` can access instance variables of `MyOuter`, in this case `MyInner`'s `go()` method takes the instance variable of `MyOutter` and assign the integer `42` to it.