**B”H**

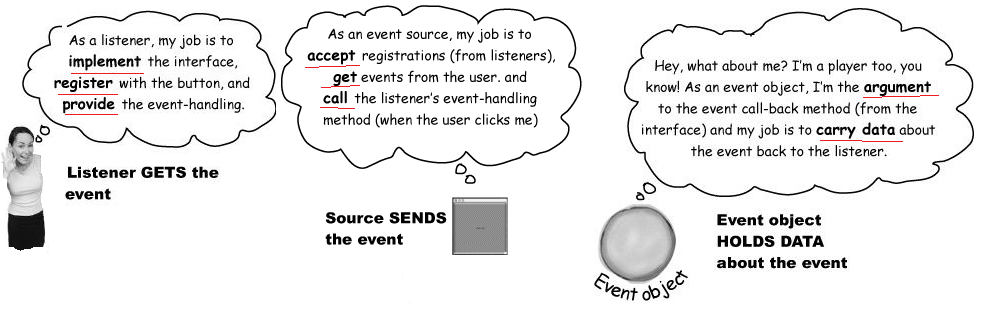
**Chapter 12**

**Getting GUI**

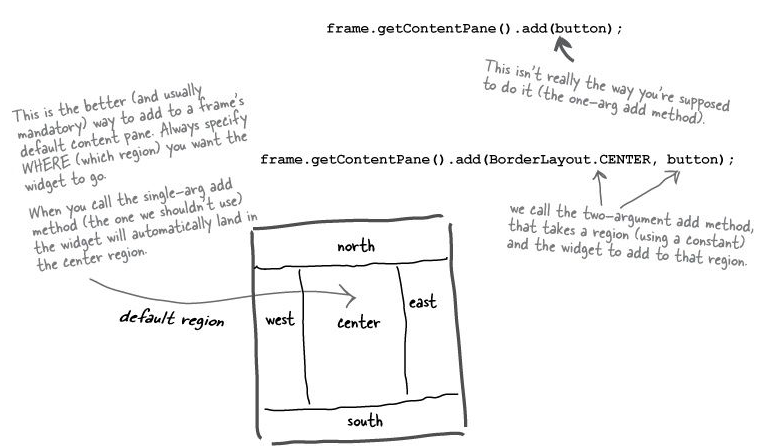
* A JFrame is the object that represents a window on the screen.
* It’s where you put all the interface things like buttons, checkboxes, text fields, etc., menu bar with menu items, windowing icons for minimizing, maximizing, and closing the window, etc.
* The JFrame looks different depending on the platform you’re on.
* Once you have a JFrame, you can put things (‘widgets’) in it by adding them to the JFrame.
* There are a ton of Swing components you can add; look for them in the javax.swing package.
  + The most common include:
    - JButton
    - JRadioButton
    - JCheckBox
    - JLabel
    - JList
    - JScrollPane
    - JSlider
    - JTextArea
    - JTextField
    - JTable
* You don’t add things to the frame directly. Think of the frame as the trim around the window, and you add things to the window pane.
* See Chapter12Sample01.java
* Not all look-and-feels are available on every platform.

**Event-handling (Note: see the Observer Pattern in chapter 2 of Head First Design Patterns)**

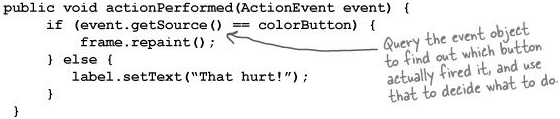
* There are many different event types in Java, although most involve GUI user actions.
* If you care about the button’s events**, implement an interface** that says, “I’m **listening** for your events.”
* A listener interface is the bridge between the **listener** (you) and **event source** (the button).
* The Swing GUI components are event sources.
  + In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events.
  + An event is represented as an object. An object of some event class.
  + If you scan through the java.awt.event package in the API, you’ll see a bunch of event classes (easy to spot — they all have Event in the name). You’ll find MouseEvent, KeyEvent, WindowEvent, ActionEvent, and several others.
* Most of the code you write (and all the code in this book) will receive events rather than create events.
* Every event type has a matching listener interface. If you want MouseEvents, implement the MouseListener interface. Want WindowEvents? Implement WindowListener. Etc.
* Some interfaces have more than one method because the event itself comes in different flavors.
  + For example MouseListener has events for mousePressed, mouseReleased, mouseMoved, etc.
  + Each of those mouse events has a separate method in the interface, even though they all take a MouseEvent.
  + So for mouse events, there’s only one event object, MouseEvent, but several different event methods, representing the different types of mouse events.



* See Chapter12Sample02.java
* For most designs, you don’t need the event object. It’s nothing more than a little data carrier, to send along more info about the event.
  + - But sometimes you might need to query the event for specific details about the event.
    - For example, if you want to know the X and Y screen coordinates for where the mouse was pressed?
    - Or sometimes you might want to register the same listener with multiple objects. An onscreen calculator, for example, has 10 numeric keys and since they all do the same thing, you might not want to make a separate listener for every single key. Instead, you might register a single listener with each of the 10 keys, and when you get an event (because your event call-back method is called) you can call a method on the event object to find out who the real event source was. In other words, which key sent this event.
  + How do you KNOW if an object is an event source? Look in the API. A method that starts with ‘add’, ends with ‘Listener’, and takes a listener interface argument. There’s a naming pattern.
  + If you want to put your own graphics on the screen, your best bet is to make your own paintable widget.
  + Make a subclass of JPanel and override one method, paintComponent(). All of your graphics code goes inside the paintComponent() method. Think of the paintComponent() method as the method called by the system to say, “Hey widget, time to paint yourself.”
  + When the frame holding your drawing panel is displayed, paintComponent() is called and your graphics appears.
    - If the user iconifies/ minimizes the window, the JVM knows the frame needs “repair” when it gets de-iconified, so it calls paintComponent() again.
    - You never call this method yourself! However, that you can ask the system to refresh the display using repaint()), which ultimately leads to paintComponent() being called.
  + See Chapter12Sample03.java and Chapter12Sample04.java
  + The argument to paintComponent() is declared as type Graphics (java.awt.Graphics).
    - public void paintComponent(Graphics g) {}
  + The object referenced by the ‘g’ parameter is actually an instance of the Graphics2D class (a subclass of Graphics)
  + If you need to use a method from the Graphics2D class, you can’t use the the paintComponent parameter (‘g’) straight from the method. But you can cast it with a new Graphics2D variable.
    - Graphics2D g2d = (Graphics2D) g;
  + Some of the methods you can call on a Graphics reference:
    - drawImage()
    - drawLine()
    - drawPolygon
    - drawRect()
    - drawOval()
    - fillRect()
    - fillRoundRect()
    - setColor()
  + Some of the methods you can call on a Graphics2D reference:
    - fill3DRect()
    - draw3DRect()
    - rotate()
    - scale()
    - shear()
    - transform()
    - setRenderingHints()
  + See Chapter12Sample05.java for GradientPaint example
  + By default, a frame has five regions you can add to. You can add only one thing to each region of a frame. However that one thing might be a panel that holds three other things including a panel that holds etc.



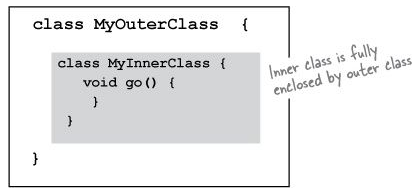
* If we have more than one button how do you get two events when you have only one actionPerformed() method?
  + You can’t implement two actionPerformed() methods – that’s not possible – won’t compile.
  + You don’t want to register the same listener with both buttons – it’s not very OO and usually it hurts maintainability and extensibility.



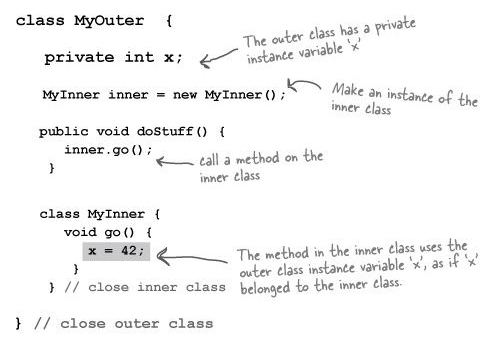
* + You don’t want to create two separate ActionListener classes. These classes won’t have access to the variables they need to act on, such as the ‘frame’ and ‘label’. You could fix it, but you’d need to add a reference variable, getters, setters, constructors, etc. It gets messier and more complicated.
* Inner class to the rescue!

**Inner Classes**

* You can have one class nested inside another. Just make sure that the definition for the inner class is inside the curly braces of the outer class.



* An inner class can use all the methods and variables of the outer class, even the private ones - as if they were declared within the inner class.
* An inner class instance must be tied to an outer class instance.
* An inner object must be tied to a specific outer object on the heap.
* If you instantiate an inner class from code within an outer class, the instance of the outer class is the one that the inner object will ‘bond’ with.



* See TwoButtons.java
* Benefits of Inner Classes:
  + You can’t implement a method more than once in a normal Java class. But using inner classes, each inner class can implement the same interface, so you can have all these different implementations of the very same interface methods.
  + What if you need to pass the IS-A test for two different classes? Classes that aren’t in the same inheritance hierarchy? You can always implement more than one interface, but you can extend only one class. Inner classes solves this.
* See SimpleAnimation.java where its inner class MyDrawPanel extends JPanel – the SimpleAnimation class can now extend from a different inheritance tree if needed