12 getting gui

A Very Graphic Story

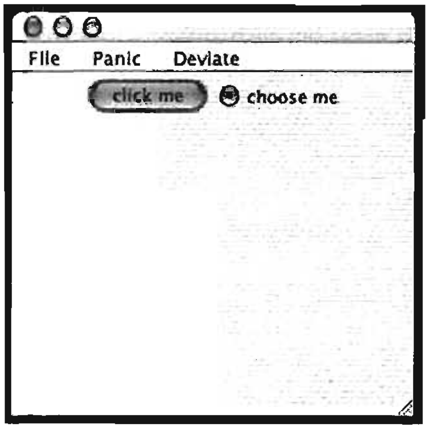
Wow! This looks great. I guess presentation really *is* everything.
I heard your ex-wife could only cook command-line meals.

Face it, you need to make GUIs. If you're building applications that other people are going to use, you *need* a graphical interface. If you're building programs for yourself, you *want* a graphical interface. Even if you believe that the rest of your natural life will be spent writing server-side code, where the client user interface is a web page, sooner or later you'll need to write tools, and you'll want a graphical interface. Sure, command-line apps are retro, but not in a good way. They're weak, inflexible, and unfriendly. We'll spend two chapters working on GUIs, and learn key Java language features along the way including **Event Handling** and **Inner Classes**. In this chapter, we'll put a button on the screen, and make it do something when you click it. We'll paint on the screen, we'll display a jpeg image, and we'll even do some animation.

It all starts with a window

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, and so on. It can have an honest-to-goodness menu bar with menu items. And it has all the little windowing icons for whatever platform you're on, for minimizing, maximizing, and closing the window.

The JFrame looks different depending on the platform you're on. This is a JFrame on Mac OS X:



a JFrame with a menu bar and two 'widgets' (a button and a radio button)

Put widgets in the window

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, and JTable. Most are really simple to use, but some (like JTable) can be a bit more complicated.

Making a GUI is easy:

1. Make a frame (a JFrame)

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

2. Make a widget (button, text field, etc.)

```
JButton button = new JButton("click me");
```

3. Add the widget to the frame

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

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.

4. Display it (give it a size and make it visible)

```
frame.setSize(300,300);
frame.setVisible(true);
```

Your first GUI: a button on a frame

```
import javax.swing.*;
```

don't forget to import this swing package

```
public class SimpleGui1 {
    public static void main (String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton(“click me”);
```

make a frame and a button

(you can pass the button constructor the text you want on the button)

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

this line makes the program quit as soon as you close the window (if you leave this out it will just sit there on the screen forever)

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

add the button to the frame's content pane

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

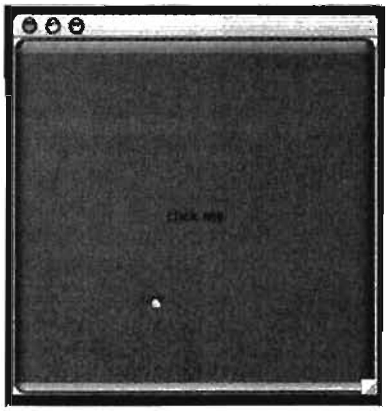
give the frame a size, in pixels

```
        frame.setVisible(true);
    }
}
```

finally, make it visible!! (if you forget this step, you won't see anything when you run this code)

Let's see what happens when we run it:

```
%java SimpleGui1
```



Whoa! That's a Really Big Button.

The button fills all the available space in the frame. Later we'll learn to control where (and how big) the button is on the frame.

But nothing happens when I click it...

That's not exactly true. When you press the button it shows that 'pressed' or 'pushed in' look (which changes depending on the platform look and feel, but it always does *something* to show when it's being pressed).

The real question is, "How do I get the button to do something specific when the user clicks it?"

We need two things:

① A **method** to be called when the user clicks (the thing you want to happen as a result of the button click).

② A way to **know** when to trigger that method. In other words, a way to know when the user clicks the button!

When the user clicks, we want to know.

We're interested in the user-takes-action-on-a-button event.

there are no Dumb Questions

Q: Will a button look like a Windows button when you run on Windows?

A: If you want it to. You can choose from a few "look and feels"—classes in the core library that control what the interface looks like. In most cases you can choose between at least two different looks: the standard Java look and feel, also known as ***Metal***, and the native look and feel for your platform. The Mac OS X screens in this book use either the OS X ***Aqua*** look and feel, or the ***Metal*** look and feel.

Q: Can I make a program look like Aqua all the time? Even when it's running under Windows?

A: Nope. Not all look and feels are available on every platform. If you want to be safe, you can either explicitly set the look and feel to Metal, so that you know exactly what you get regardless of where the app is running, or don't specify a look and feel and accept the defaults.

Q: I heard Swing was dog-slow and that nobody uses it.

A: This was true in the past, but isn't a given anymore. On weak machines, you might feel the pain of Swing. But on the newer desktops, and with Java version 1.3 and beyond, you might not even notice the difference between a Swing GUI and a native GUI. Swing is used heavily today, in all sorts of applications.

Getting a user event

Imagine you want the text on the button to change from *click me* to *I've been clicked* when the user presses the button. First we can write a method that changes the text of the button (a quick look through the API will show you the method):

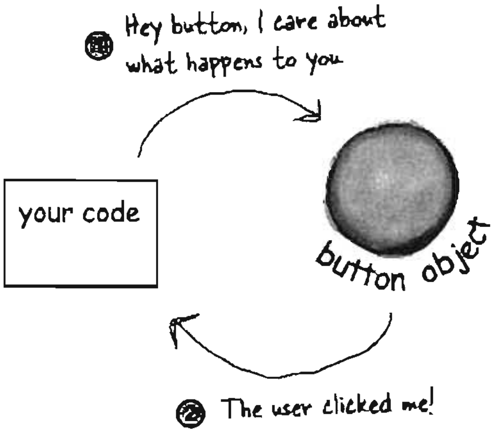
```
public void changeIt() {
     button.setText("I've been clicked!");
}
```

But *now* what? How will we *know* when this method should run? ***How will we know when the button is clicked?***

In Java, the process of getting and handling a user event is called *event-handling*. There are many different event types in Java, although most involve GUI user actions. If the user clicks a button, that's an event. An event that says "The user wants the action of this button to happen." If it's a "Slow Tempo" button, the user wants the slow-tempo action to occur. If it's a Send button on a chat client, the user wants the send-my-message action to happen. So the most straightforward event is when the user clicked the button, indicating they want an action to occur.

With buttons, you usually don't care about any intermediate events like button-is-being-pressed and button-is-being-released. What you want to say to the button is, "I don't care how the user plays with the button, how long they hold the mouse over it, how many times they change their mind and roll off before letting go, etc. ***Just tell me when the user means business!*** In other words, don't call me unless the user clicks in a way that indicates he wants the darn button to do what it says it'll do!"

First, the button needs to know that we care.



Second, the button needs a way to call us back when a button-clicked event occurs.

BRAIN POWER

1) How could you tell a button object that you care about its events? That you're a concerned listener?

2) How will the button call you back? Assume that there's no way for you to tell the button the name of your unique method (changeIt()). So what else can we use to reassure the button that we have a specific method it can call when the event happens? [hint: think Pet]

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).

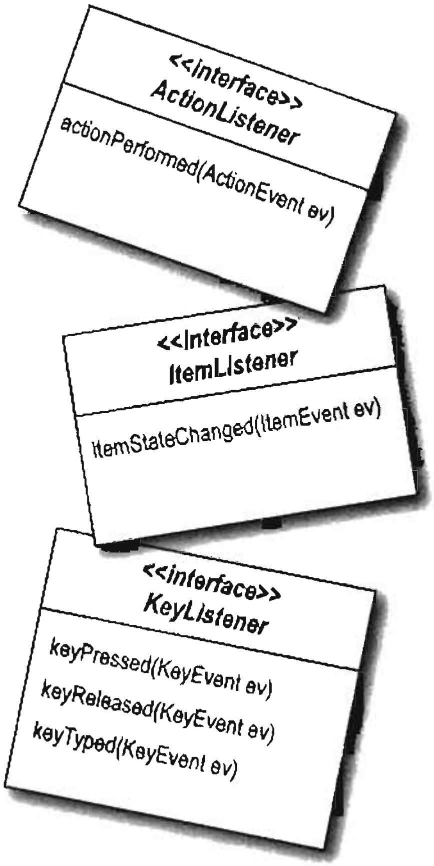
The Swing GUI components are event sources. In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events. And like virtually everything else in Java, an event is represented as an object. An object of some event class. If you scan through 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.

An event *source* (like a button) creates an *event object* when the user does something that matters (like *click* the button). Most of the code you write (and all the code in this book) will *receive* events rather than *create* events. In other words, you'll spend most of your time as an event *listener* rather than an event *source*.

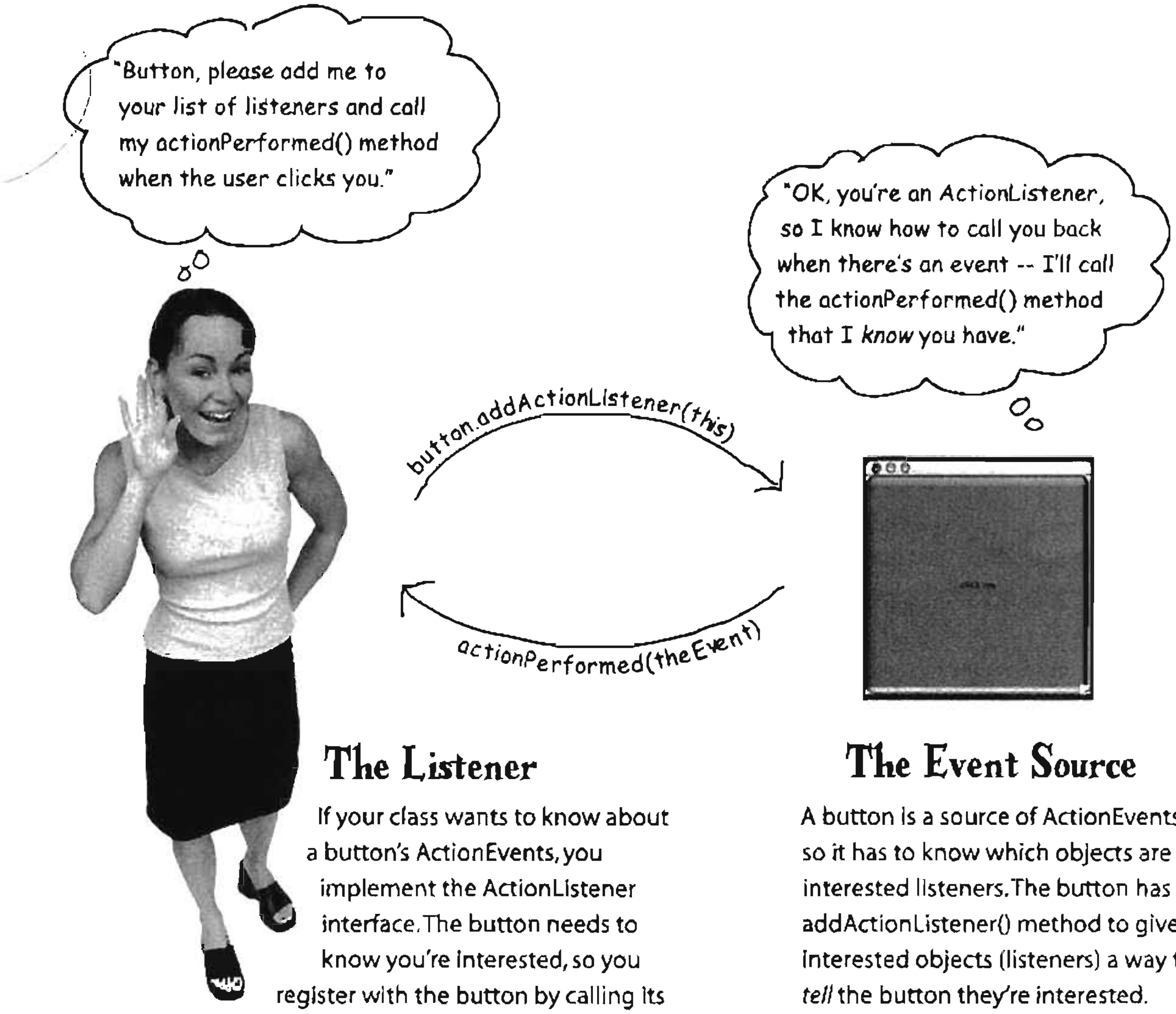
Every event type has a matching listener interface. If you want MouseEvents, implement the MouseListener interface. Want WindowEvents? Implement WindowListener. You get the idea. And remember your interface rules—to implement an interface you *declare* that you implement it (class Dog implements Pet), which means you must *write implementation methods* for every method in the interface.

Some interfaces have more than one method because the event itself comes in different flavors. If you implement MouseListener, for example, you can get 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. If you implement MouseListener, the mousePressed() method is called when the user (you guessed it) presses the mouse. And when the user lets go, the mouseReleased() method is called. So for mouse events, there's only one event *object*, MouseEvent, but several different event *methods*, representing the different *types* of mouse events.

When you implement a listener interface, you give the button a way to call you back. The interface is where the call-back method is declared.



How the listener and source communicate:


"Button, please add me to your list of listeners and call my actionPerformed() method when the user clicks you."
"OK, you're an ActionListener, so I know how to call you back when there's an event -- I'll call the actionPerformed() method that I *know* you have."
button.addActionListener(this)
actionPerformed(theEvent)

The Listener

If your class wants to know about a button's ActionEvents, you implement the ActionListener interface. The button needs to know you're interested, so you register with the button by calling its addActionListener(this) and passing an ActionListener reference to it (in this case, *you* are the ActionListener so you pass *this*). The button needs a way to call you back when the event happens, so it calls the method in the listener interface. As an ActionListener, you *must* implement the interface's sole method, actionPerformed(). The compiler guarantees it.

The Event Source

A button is a source of ActionEvents, so it has to know which objects are interested listeners. The button has an addActionListener() method to give interested objects (listeners) a way to *tell* the button they're interested.

When the button's addActionListener() runs (because a potential listener invoked it), the button takes the parameter (a reference to the listener object) and stores it in a list. When the user clicks the button, the button 'fires' the event by calling the actionPerformed() method on each listener in the list.

Getting a button's ActionEvent

1. Implement the ActionListener interface
2. Register with the button (tell it you want to listen for events)
3. Define the event-handling method (implement the actionPerformed() method from the ActionListener interrface)

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

a new import statement for the package that ActionListener and ActionEvent are in.

```
public class SimpleGui1B implements ActionListener {
    JButton button;
```

❶ Implement the interface. This says, "an instance of SimpleGui1B IS-A ActionListener".

(The button will give events only to ActionListener implementers)

```
    public static void main (String[] args) {
       SimpleGui1B gui = new SimpleGui1B();
       gui.go();
     }

    public void go() {
       JFrame frame = new JFrame();
       button = new JButton("click me");

       button.addActionListener(this);
```

❷ register your interest with the button. This says to the button, "Add me to your list of listeners". The argument you pass MUST be an object from a class that implements ActionListener!!

```
       frame.getContentPane().add(button);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
       button.setText("I've been clicked!");
    }
}
```

❸ Implement the ActionListener interface's actionPerformed() method.. This is the actual event-handling method!

The button calls this method to let you know an event happened. It sends you an ActionEvent object as the argument, but we don't need it. Knowing the event happened is enough info for us.

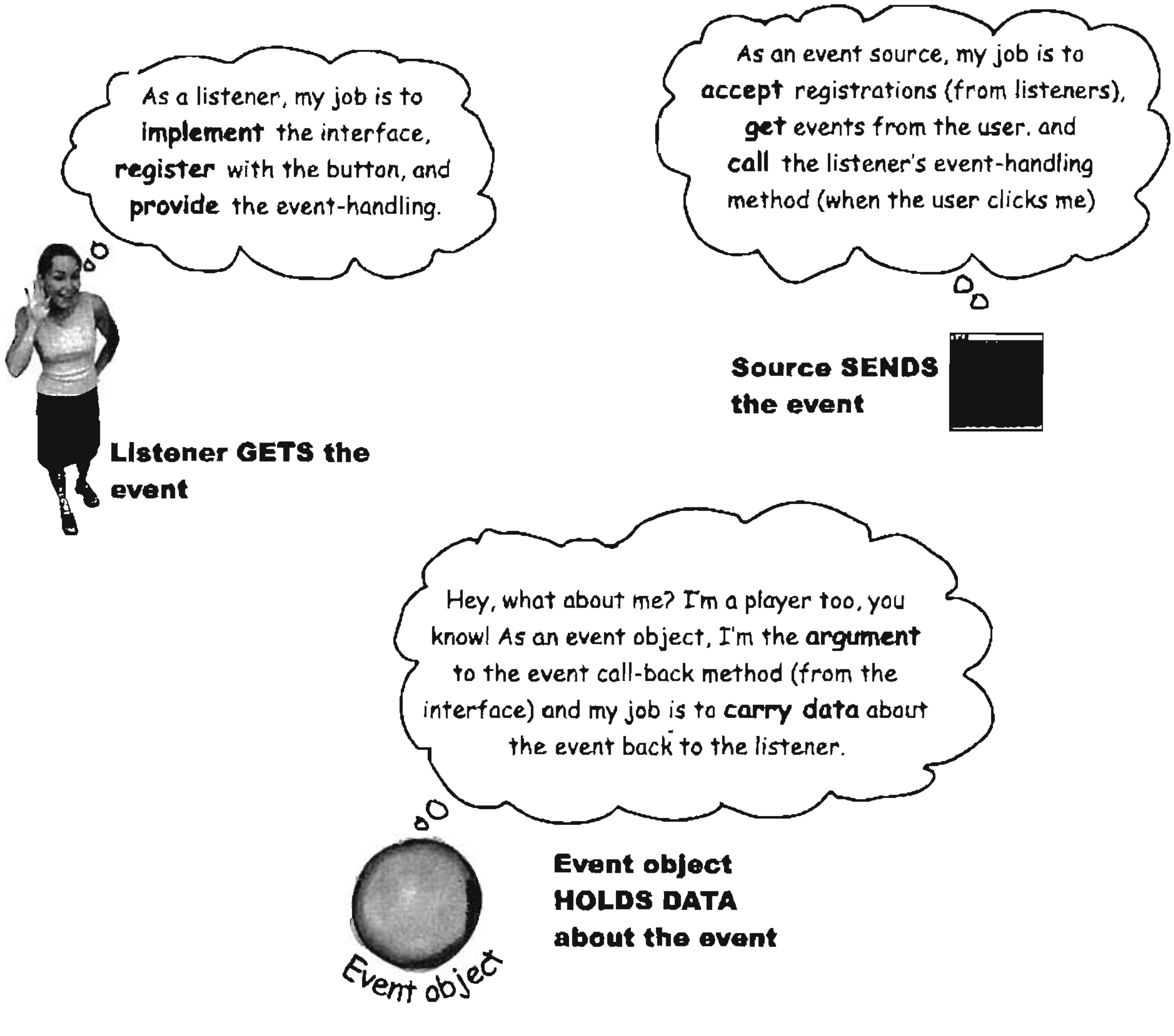
Listeners, Sources, and Events

For most of your stellar Java career, *you* will not be the *source* of events.

(No matter how much you fancy yourself the center of your social universe.)

Get used to it. ***Your job is to be a good listener.***

(Which, if you do it sincerely, *can* improve your social life.)


As a listener, my job is to **implement** the interface, **register** with the button, and **provide** the event-handling.
As an event source, my job is to **accept** registrations (from listeners), **get** events from the user. and **call** the listener's event-handling method (when the user clicks me)
Hey, what about me? I'm a player too, you know! As an event object, I'm the **argument** to the event call-back method (from the interface) and my job is to **carry data** about the event back to the listener.
Event object

Listener GETS the event

Source SENDS the event

Event object HOLDS DATA about the event

there are no Dumb Questions

Q: Why can't I be a source of events?

A: You CAN. We just said that *most* of the time you'll be the receiver and not the originator of the event (at least in the *early* days of your brilliant Java career~~). Most of~~ the events you might care about are 'fired' by classes in the Java API, and all you have to do is be a listener for them. You might, however, design a program where you need a custom event, say, StockMarketEvent thrown when your stock market watcher app finds something it deems important. In that case, you'd make the StockWatcher object be an event source, and you'd do the same things a button (or any other source) does—make a listener interface for your custom event, provide a registration method (addStockListener()), and when somebody calls it, add the caller (a listener) to the list of listeners. Then, when a stock event happens, instantiate a StockEvent object (another class you'll write) and send it to the listeners in your list by calling their stockChanged(StockEvent ev) method. And don't forget that for every *event type* there must be a *matching listener interface* (so you'll create a StockListener interface with a stockChanged() method).

Q: I don't see the importance of the event object that's passed to the event call-back methods. If somebody calls my mousePressed method, what other info would I need?

A: A lot of the time, 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 your mousePressed() method is called, you know the mouse was pressed. But what if you want to know exactly where the mouse was pressed? In other words, what 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*.

Sharpen your pencil

Each of these widgets (user interface objects) are the source of one or more events. Match the widgets with the events they might cause. Some widgets might be a source of more than one event, and some events can be generated by more than one widget.

Widgets	Event methods
check box	windowClosing()
text field	actionPerformed()
scrolling list	itemStateChanged()
button	mousePressed()
dialog box	keyTyped()
radio button	mouseExited()
menu item	focusGained()

How do you KNOW if an object is an event source?

Look in the API.

OK. Look for what?

A method that starts with 'add', ends with 'Listener', and takes a listener interface argument. If you see:

addKeyListener(KeyListener k)

you know that a class with this method is a source of KeyEvents. There's a naming pattern.

Getting back to graphics...

Now that we know a little about how events work (we'll learn more later), let's get back to putting stuff on the screen. We'll spend a few minutes playing with some fun ways to get graphic, before returning to event handling.

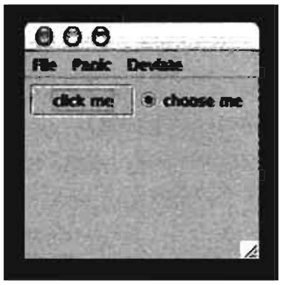
Three ways to put things on your GUI:

1. **Put widgets on a frame**

   Add buttons, menus, radio buttons, etc.

   `frame.getContentPane().add(myButton);`

   The javax.swing package has more than a dozen widget types.

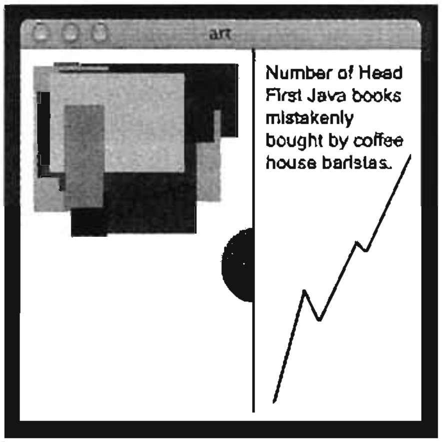


2. **Draw 2D graphics on a widget**

   Use a graphics object to paint shapes.

   `graphics.fillOval(70,70,100,100);`

   You can paint a lot more than boxes and circles; the Java2D API is full of fun, sophisticated graphics methods.


art
Number of Head First Java books mistakenly bought by coffee house baristas.

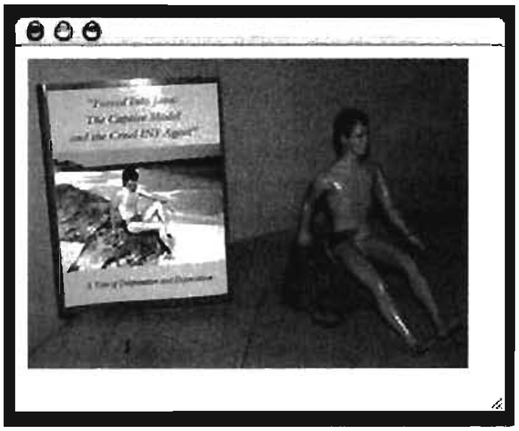
art, games, simulations, etc.

charts, business graphics, etc.

3. **Put a JPEG on a widget**

   You can put your own images on a widget.

   `graphics.drawImage(myPic,10,10,this);`

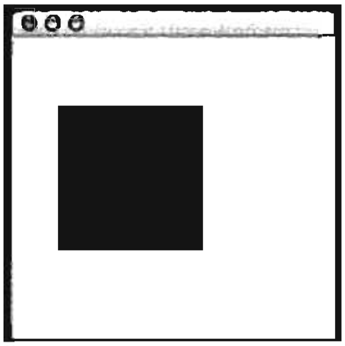


Make your own drawing widget

If you want to put your own graphics on the screen, your best bet is to make your own paintable widget. You plop that widget on the frame, just like a button or any other widget, but when it shows up it will have your images on it. You can even make those images move, in an animation, or make the colors on the screen change every time you click a button.

~~It's a piece~~ of cake.

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." If you want to draw a circle, the paintComponent() method will have code for drawing a circle. When the frame holding your drawing panel is displayed, paintComponent() is called and your circle 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. Anytime the JVM thinks the display needs refreshing, your paintComponent() method will be called.

One more thing, *you never call this method yourself!* The argument to this method (a Graphics object) is the actual drawing canvas that gets slapped onto the *real* display. You can't get this by yourself; it must be handed to you by the system. You'll see later, however, that you *can* ask the system to refresh the display (repaint()), which ultimately leads to paintComponent() being called.

```
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);
      }
}
```

you need both of these

Make a subclass of JPanel, a widget that you can add to a frame just like anything else. Except this one is your own customized widget

This is the Big Important Graphics method. You will NEVER call this yourself. The system calls it and says, "Here's a nice fresh drawing surface, of type Graphics, that you may paint on now.".

Imagine that 'g' is a painting machine. You're telling it what color to paint with and then what shape to paint (with coordinates for where it goes and how big it is)

Fun things to do in paintComponent()

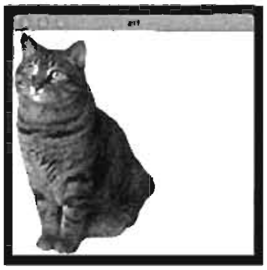
Let's look at a few more things you can do in paintComponent(). The most fun, though, is when you start experimenting yourself. Try playing with the numbers, and check the API for class Graphics (later we'll see that there's even *more* you can do besides what's in the Graphics class).

Display a JPEG

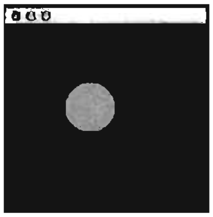
```
public void paintComponent(Graphics g) {

    Image image = new ImageIcon("catzilla.jpg").getImage();

    g.drawImage(image,3,4,this);

}
```

your file name goes here

The x,y coordinates for where the picture's top left corner should go. This says "3 pixels from the left edge of the panel and 4 pixels from the top edge of the panel". These numbers are always relative to the widget (in this case your JPanel subclass), not the entire frame.



Paint a randomly-colored circle on a black background



```
public void paintComponent(Graphics g) {

   g.fillRect(0,0,this.getWidth(), this.getHeight());


   int red = (int) (Math.random() * 255);
   int green = (int) (Math.random() * 255);
   int blue = (int) (Math.random() * 255);

   Color randomColor = new Color(red, green, blue);
   g.setColor(randomColor);
   g.fillOval(70,70,100,100);
}
```

fill the entire panel with black (the default color)

The first two args define the (x,y) upper left corner, relative to the panel, for where drawing starts, so 0, 0 means "start 0 pixels from the left edge and 0 pixels from the top edge." The other two args say, "Make the width of this rectangle as wide as the panel (this.width()), and make the height as tall as the panel (this.height)"

You can make a color by passing in 3 ints to represent the RGB values.

start 70 pixels from the left, 70 from the top, make it 100 pixels wide, and 100 pixels tall.

Behind every good Graphics reference is a Graphics2D object.

The argument to paintComponent() is declared as type Graphics (java.awt.Graphics).

```
    public void paintComponent(Graphics g) { }
```

So the parameter 'g' IS-A Graphics object. Which means it *could* be a *subclass* of Graphics (because of polymorphism). And in fact, it *is*.

The object referenced by the 'g' parameter is actually an instance of the* Graphics2D *class.

Why do you care? Because there are things you can do with a Graphics2D reference that you can't do with a Graphics reference. A Graphics2D object can do more than a Graphics object, and it really is a Graphics2D object lurking behind the Graphics reference.

Remember your polymorphism. The compiler decides which methods you can call based on the reference type, not the object type. If you have a Dog object referenced by an Animal reference variable:

```
Animal a = new Dog();
```

You can NOT say:

```
a.bark();
```

Even though you know it's really a Dog back there. The compiler looks at 'a', sees that it's of type Animal, and finds that there's no remote control button for bark() in the Animal class. But you can still get the object back to the Dog it really *is* by saying:

```
Dog d = (Dog) a;
d.bark();
```

So the bottom line with the Graphics object is this:

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;
```

Methods you can call on a Graphics reference:

- drawImage()
- drawLine()
- drawPolygon
- drawRect()
- drawOval()
- fillRect()
- fillRoundRect()
- setColor()

To cast the Graphics2D *object* to a Graphics2D *reference*:

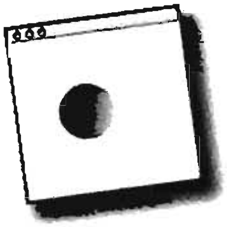
```
Graphics2D g2d = (Graphics2D) g;
```

Methods you can call on a Graphics2D reference:

- fill3DRect()
- draw3DRect()
- rotate()
- scale()
- shear()
- transform()
- setRenderingHints()

(these are not complete method lists, check the API for more)

Because life's too short to paint the circle a solid color when there's a gradient blend waiting for you.



it's really a Graphics2D object masquerading as a mere Graphics object

```
public void paintComponent(Graphics g) {

    Graphics2D g2d = (Graphics2D) g;
```

cast it so we can call something that Graphics2D has but Graphics doesn't

```
    GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);
```

starting point — starting color — ending point — ending color

this sets the virtual paint brush to a gradient instead of a solid color

```
    g2d.setPaint(gradient);

    g2d.fillOval(70,70,100,100);
```

the fillOval() method really means "fill the oval with whatever is loaded on your paintbrush (i.e. the gradient)"

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);
    Color startColor = new Color(red, green, blue);

    red = (int) (Math.random() * 255);
    green = (int) (Math.random() * 255);
    blue = (int) (Math.random() * 255);
    Color endColor = new Color(red, green, blue);

    GradientPaint gradient = new GradientPaint(70,70,startColor, 150,150, endColor);
    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
```

this is just like the one above, except it makes random colors for the start and stop colors of the gradient. Try it!

BULLET POINTS

EVENTS

- To make a GUI, start with a window, usually a JFrame
  ```
  JFrame frame = new JFrame();
  ```
- You can add widgets (buttons, text fields, etc.) to the JFrame using:
  ```
  frame.getContentPane().add(button);
  ```
- Unlike most other components, the JFrame doesn't let you add to it directly, so you must add to the JFrame's content pane.
- To make the window (JFrame) display, you must give it a size and tell it be visible:
  ```
  frame.setSize(300,300);
  frame.setVisible(true);
  ```
- To know when the user clicks a button (or takes some other action on the user interface) you need to listen for a GUI event.
- To listen for an event, you must register your interest with an event source. An event source is the thing (button, checkbox, etc.) that 'fires' an event based on user interaction.
- The listener interface gives the event source a way to call you back, because the interface defines the method(s) the event source will call when an event happens.
- To register for events with a source, call the source's registration method. Registration methods always take the form of: ***add<EventType>Listener***. To register for a button's ActionEvents, for example, call:
  ```
  button.addActionListener(this);
  ```
- Implement the listener interface by implementing all of the interface's event-handling methods. Put your event-handling code in the listener call-back method. For ActionEvents, the method is:
  ```
  public void actionPerformed(ActionEvent
                                     event) {
      button.setText("you clicked!");
  }
  ```
- The event object passed into the event-handler method carries information about the event, including the source of the event.

GRAPHICS

- You can draw 2D graphics directly on to a widget.
- You can draw a .gif or .jpeg directly on to a widget.
- To draw your own graphics (including a .gif or .jpeg), make a subclass of JPanel and override the paintComponent() method.
- The paintComponent() method is called by the GUI system. YOU NEVER CALL IT YOURSELF. The argument to paintComponent() is a Graphics object that gives you a surface to draw on, which will end up on the screen. You cannot construct that object yourself.
- Typical methods to call on a Graphics object (the paintComponent paramenter) are:
  ```
  graphics.setColor(Color.blue);
  g.fillRect(20,50,100,120);
  ```
- To draw a .jpg, construct an Image using:
  ```
  Image image = new ImageIcon("catzilla.
  jpg").getImage();
  ```
  and draw the imagine using:
  ```
  g.drawImage(image,3,4,this);
  ```
- The object referenced by the Graphics parameter to paintComponent() is actually an instance of the Graphics2D class. The Graphics 2D class has a variety of methods including:
  fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform()
- To invoke the Graphics2D methods, you must cast the parameter from a Graphics object to a Graphics2D object:
  ```
  Graphics2D g2d = (Graphics2D) g;
  ```

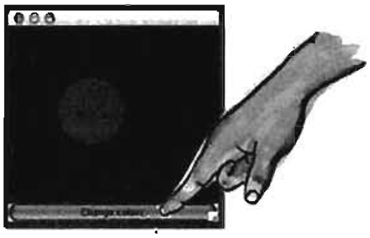
We can get an event.
We can paint graphics.
But can we paint graphics *when* we get an event?

Let's hook up an event to a change in our drawing panel. We'll make the circle change colors each time you click the button. Here's how the program flows:

Start the app

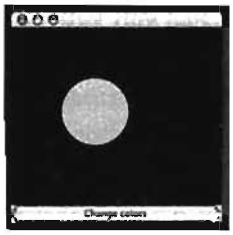
Change colors

1 The frame is built with the two widgets (your drawing panel and a button). A listener is created and registered with the button. Then the frame is displayed and it just waits for the user to click.



2 The user clicks the button and the button creates an event object and calls the listener's event handler.

3 The event handler calls repaint() on the frame. The system calls paintComponent() on the drawing panel.



4 Voila! A new color is painted because paintComponent() runs again, filling the circle with a random color.

GUI layouts: putting more than one widget on a frame

We cover GUI layouts in the *next* chapter, but we'll do a quickie lesson here to get you going. By default, a frame has five regions you can add to. You can add only *one* thing to each region of a frame, but don't panic! That one thing might be a panel that holds three other things including a panel that holds two more things and... you get the idea. In fact, we were 'cheating' when we added a button to the frame using:

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

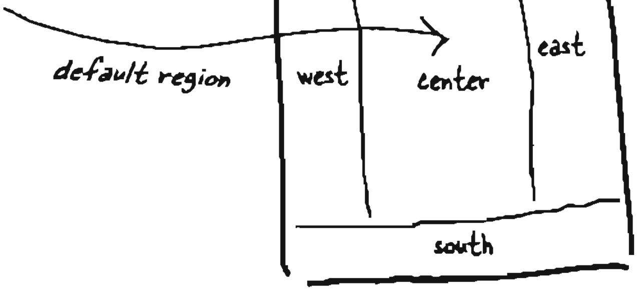
This isn't really the way you're supposed to do it (the one-arg add method).

This is the better (and usually mandatory) way to add to a frame's default content pane. Always specify WHERE (which region) you want the widget to go.

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

we call the two-argument add method, that takes a region (using a constant) and the widget to add to that region.

When you call the single-arg add method (the one we shouldn't use) the widget will automatically land in the center regsion.

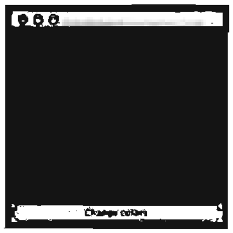


Sharpen your pencil

Given the pictures on page 351, write the code that adds the button and the panel to the frame.

The circle changes color each time you click the button.

The custom drawing panel (instance of MyDrawPanel) is in the CENTER region of the frame.



Button is in the SOUTH region of the frame

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


public class SimpleGui3C implements ActionListener {

    JFrame frame;

    public static void main (String[] args) {
       SimpleGui3C gui = new SimpleGui3C();
       gui.go();
    }

    public void go() {
       frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton button = new JButton("Change colors");
       button.addActionListener(this);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(BorderLayout.SOUTH, button);
       frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

Add the listener (this) to the button.

Add the two widgets (button and drawing panel) to the two regions of the frame.

When the user clicks, tell the frame to repaint() itself. That means paintComponent() is called on every widget in the frame!

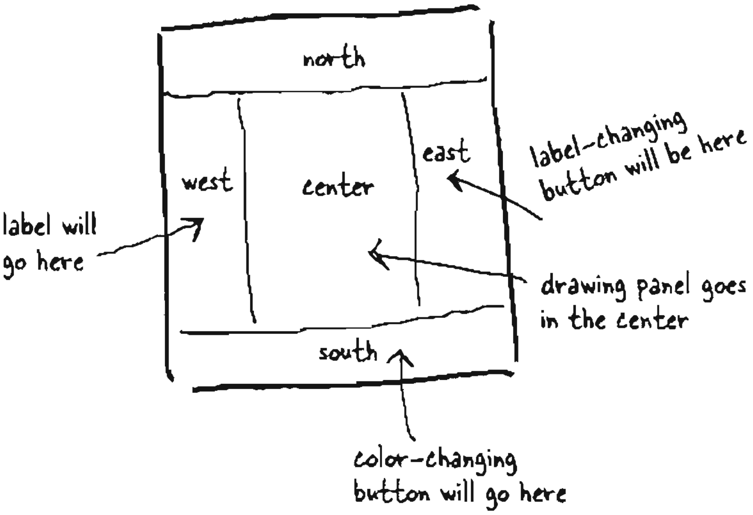
```
class MyDrawPanel extends JPanel {

   public void paintComponent(Graphics g) {
     // Code to fill the oval with a random color
     // See page 347 for the code
   }

}
```

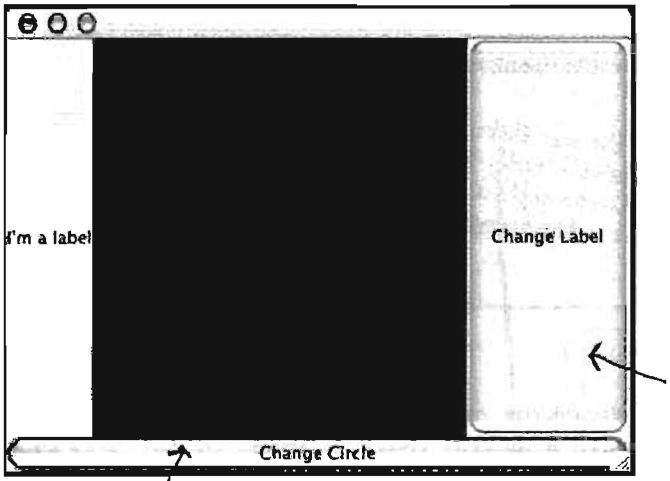
The drawing panel's paintComponent() method is called every time the user clicks.

Let's try it with TWO buttons

The south button will act as it does now, simply calling repaint on the frame. The second button (which we'll stick in the east region) will change the text on a label. (A label is just text on the screen.)

So now we need FOUR widgets


north
west
center
east
south
label will go here
label-changing button will be here
drawing panel goes in the center
color-changing button will go here



This button changes the text on the opposite side

This button changes the color of the circle

And we need to get TWO events

Uh-oh.

Is that even possible? How do you get *two* events when you have only *one* actionPerformed() method?

How do you get action events for two different buttons, when each button needs to do something different?

option one
Implement two actionPerformed() methods

```
class MyGui implements ActionListener {
    // lots of code here and then:

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }

    public void actionPerformed(ActionEvent event) {
        label.setText(“That hurt!”);
    }
}
```

But this is impossible!

Flaw: You can't! You can't implement the same method twice in a Java class. It won't compile. And even if you *could*, how would the event source know *which* of the two methods to call?

option two
Register the same listener with both buttons.

```
class MyGui implements ActionListener {
    // declare a bunch of instance variables here

    public void go() {
       // build gui
       colorButton = new JButton();
       labelButton = new JButton();
       colorButton.addActionListener(this);
       labelButton.addActionListener(this);
       // more gui code here ...
    }

   public void actionPerformed(ActionEvent event) {
        if (event.getSource() == colorButton) {
            frame.repaint();
        } else {
           label.setText(“That hurt!”);
        }
    }
}
```

Register the same listener with both buttons

Query the event object to find out which button actually fired it, and use that to decide what to do.

Flaw: this does work, but in most cases it's not very OO. One event handler doing many different things means that you have a single method doing many different things. If you need to change how *one* source is handled, you have to mess with *everybody's* event handler. Sometimes it *is* a good solution, but usually it hurts maintainability and extensibility.

How do you get action events for two different buttons, when each button needs to do something different?

③ option three
Create two separate ActionListener classes

```
class MyGui  {
   JFrame frame;
   JLabel label;
   void gui() {
      // code to instantiate the two listeners and register one
      // with the color button and the other with the label button
   }
} // close class
```

```
class ColorButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

Won't work! This class doesn't have a reference to the 'frame' variable of the MyGui class

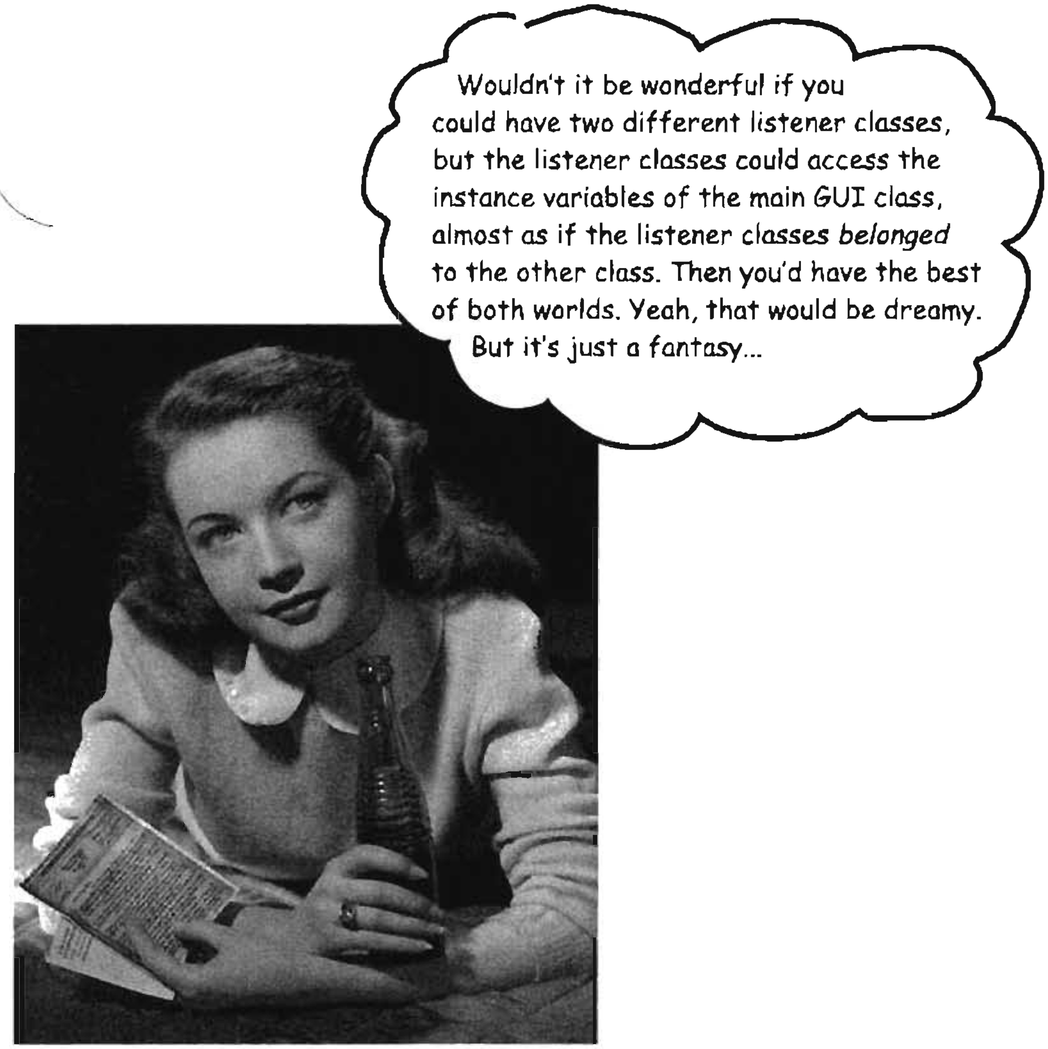
---

```
class LabelButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        label.setText("That hurt!");
    }
}
```

Problem! This class has no reference to the variable 'label'

Flaw: these classes won't have access to the variables they need to act on, 'frame' and 'label'. You could fix it, but you'd have to give each of the listener classes a reference to the main GUI class, so that inside the actionPerformed() methods the listener could use the GUI class reference to access the variables of the GUI class. But that's breaking encapsulation, so we'd probably need to make getter methods for the gui widgets (getFrame(), getLabel(), etc.). And you'd probably need to add a constructor to the listener class so that you can pass the GUI reference to the listener at the time the listener is instantiated. And, well, it gets messier and more complicated.

There has got to be a better way!


Wouldn't it be wonderful if you could have two different listener classes, but the listener classes could access the instance variables of the main GUI class, almost as if the listener classes *belonged* to the other class. Then you'd have the best of both worlds. Yeah, that would be dreamy. But it's just a fantasy...

Inner class to the rescue!

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

Simple inner class:

```
class MyOuterClass  {

   class MyInnerClass {
      void go() {
      }
   }

}
```

Inner class is fully enclosed by outer class

An inner class can use all the methods and variables of the outer class, even the private ones.

The inner class gets to use those variables and methods just as if the methods and variables were declared within the inner class.

An inner class gets a special pass to use the outer class's stuff. *Even the private stuff.* And the inner class can use those private variables and methods of the outer class as if the variables and members were defined in the inner class. That's what's so handy about inner classes—they have most of the benefits of a normal class, but with special access rights.

Inner class using an outer class variable

```
class MyOuterClass  {

   private int x;

  class MyInnerClass {
     void go() {
        x = 42;
     }
  } // close inner class

} // close outer class
```

use 'x' as if it were a variable of the inner class!

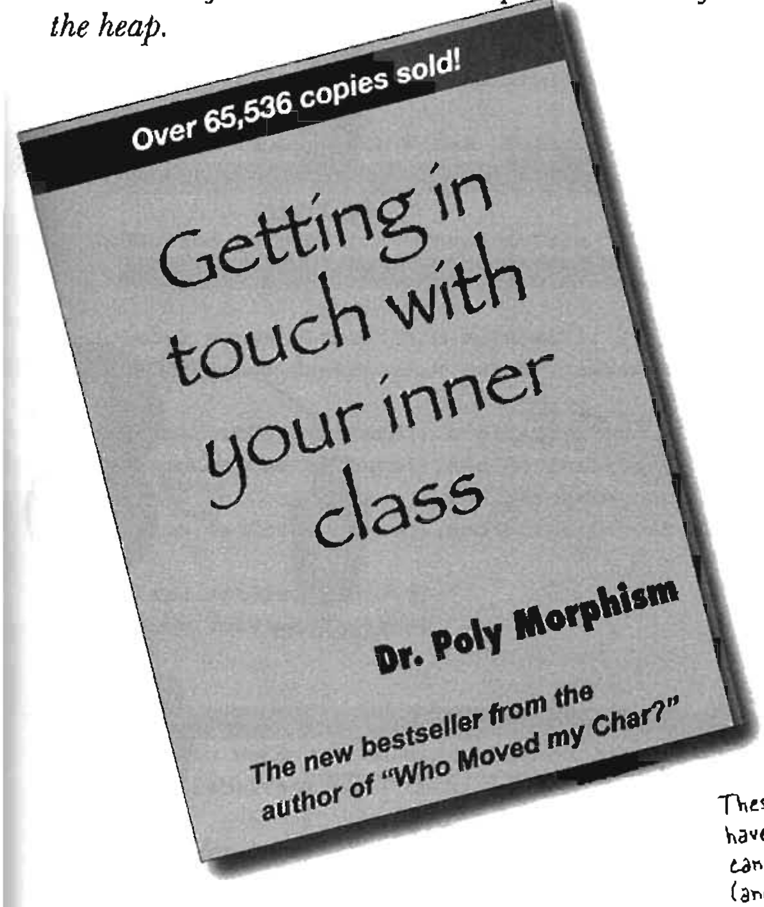
An inner class instance must be tied to an outer class instance*.

Remember, when we talk about an inner *class* accessing something in the outer class, we're really talking about an *instance* of the inner class accessing something in an *instance* of the outer class. But *which* instance?

Can *any* arbitrary instance of the inner class access the methods and variables of *any* instance of the outer class? **No!**

*An **inner** object must be tied to a specific **outer** object on the heap.*

An inner object shares a special bond with an outer object. ♥



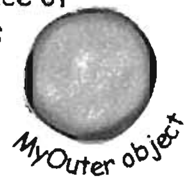
Over 65,536 copies sold!

Getting in touch with your inner class

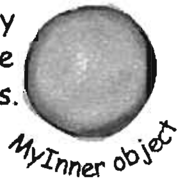
Dr. Poly Morphism

The new bestseller from the author of "Who Moved my Char?"

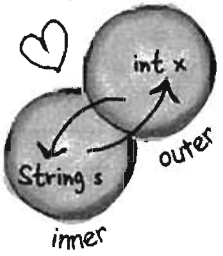
1. Make an instance of the outer class



2. Make an instance of the inner class, by using the instance of the outer class.



3. The outer and inner objects are now intimately linked.



These two objects on the heap have a special bond. The inner can use the outer's variables (and vice-versa).

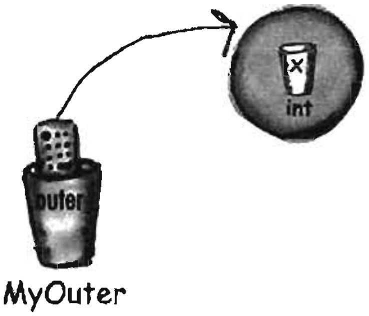
*There's an exception to this, for a very special case—an inner class defined within a static method. But we're not going there, and you might go your entire Java life without ever encountering one of these.

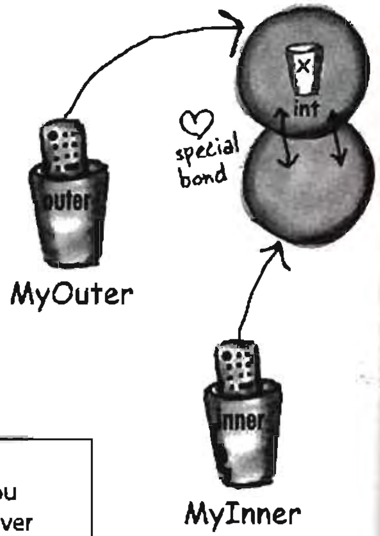
How to make an instance of an inner class

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. For example, if code within a method instantiates the inner class, the inner object will bond to the instance whose method is running.

Code in an outer class can instantiate one of its own inner classes, in exactly the same way it instantiates any other class... **new MyInner()**

```
class MyOuter  {

   private int x;  ← The outer class has a private
                     instance variable 'x'

   MyInner inner = new MyInner(); ← Make an instance of the
                                    inner class

   public void doStuff() {
      inner.go();  ← call a method on the
    }                inner class

   class MyInner {
      void go() {
         x = 42;  ← The method in the inner class uses the
      }             outer class instance variable 'x', as if 'x'
    } // close inner class   belonged to the inner class

} // close outer class
```



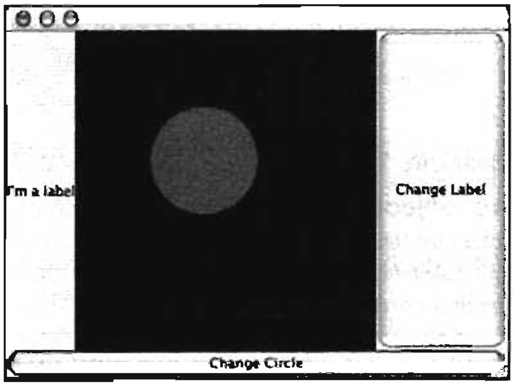


Side bar

You *can* instantiate an inner instance from code running *outside* the outer class, but you have to use a special syntax. Chances are you'll go through your entire Java life and never need to make an inner class from outside, but just in case you're interested...

```
class Foo {
    public static void main (String[] args) {
       MyOuter outerObj = new MyOuter();
       MyOuter.MyInner innerObj = outerObj.new MyInner();
    }
 }
```

Now we can get the two-button code working



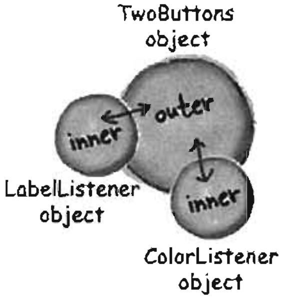
```
public class TwoButtons {

    JFrame frame;
    JLabel label;

    public static void main (String[] args) {
       TwoButtons gui = new TwoButtons ();
       gui.go();
    }
```

the main GUI class doesn't implement ActionListener now

```
    public void go() {
       frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton labelButton = new JButton(“Change Label”);
       labelButton.addActionListener(new LabelListener());

       JButton colorButton = new JButton(“Change Circle”);
       colorButton.addActionListener(new ColorListener());
```

instead of passing (this) to the button's listener registration method, pass a new instance of the appropriate listener class.

```
       label = new JLabel(“I'm a label”);
       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
       frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
       frame.getContentPane().add(BorderLayout.EAST, labelButton);
       frame.getContentPane().add(BorderLayout.WEST, label);

       frame.setSize(300,300);
       frame.setVisible(true);
    }
```



Now we get to have TWO ActionListeners in a single class!

```
    class LabelListener implements ActionListener {
       public void actionPerformed(ActionEvent event) {
           label.setText(“Ouch!”);
       }
    } // close inner class
```

inner class knows about 'label'

```
    class ColorListener implements ActionListener {
       public void actionPerformed(ActionEvent event) {
           frame.repaint();
       }
    }  // close inner class

}
```

the inner class gets to use the 'frame' instance variable, without having an explicit reference to the outer class object.

Java Exposed

This weeks interview: Instance of an Inner Class

HeadFirst: What makes inner classes important?

Inner object: Where do I start? We give you a chance to implement the same interface more than once in a class. Remember, 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.

HeadFirst: Why would you ever *want* to implement the same method twice?

Inner object: Let's revisit GUI event handlers. Think about it... if you want *three* buttons to each have a different event behavior, then use *three* inner classes, all implementing ActionListener—which means each class gets to implement its own actionPerformed method.

HeadFirst: So are event handlers the only reason to use inner classes?

Inner object: Oh, gosh no. Event handlers are just an obvious example. Anytime you need a separate class, but still want that class to behave as if it were part of *another* class, an inner class is the best—and sometimes *only*—way to do it.

HeadFirst: I'm still confused here. If you want the inner class to *behave* like it belongs to the outer class, why have a separate class in the first place? Why wouldn't the inner class code just be *in* the outer class in the first place?

Inner object: I just *gave* you one scenario, where you need more than one implementation of an interface. But even when you're not using interfaces, you might need two different *classes* because those classes represent two different *things*. It's good OO.

HeadFirst: Whoa. Hold on here. I thought a big part of OO design is about reuse and maintenance. You know, the idea that if you have two separate classes, they can each be modified and used independently, as opposed to stuffing it all into one class yada yada yada. But with an *inner* class, you're still just working with one *real* class in the end, right? The enclosing class is the only one that's reusable and separate from everybody else. Inner classes aren't exactly reusable. In fact, I've heard them called "Reuseless—useless over and over again."

Inner object: Yes it's true that the inner class is not *as* reusable, in fact sometimes not reusable at all, because it's intimately tied to the instance variables and methods of the outer class. But it—

HeadFirst: —which only proves my point! If they're not reusable, why bother with a separate class? I mean, other than the interface issue, which sounds like a workaround to me.

Inner object: As I was saying, you need to think about IS-A and polymorphism.

HeadFirst: OK. And I'm thinking about them because...

Inner object: Because the outer and inner classes might need to pass *different* IS-A tests! Let's start with the polymorphic GUI listener example. What's the declared argument type for the button's listener registration method? In other words, if you go to the API and check, what kind of *thing* (class or interface type) do you have to pass to the addActionListener() method?

HeadFirst: You have to pass a listener. Something that implements a particular listener interface, in this case ActionListener. Yeah, we know all this. What's your point?

Inner object: My point is that polymorphically, you have a method that takes only one particular *type*. Something that passes the IS-A test for ActionListener. But—and here's the big thing—what if your class needs to be an IS-A of something that's a *class* type rather than an interface?

HeadFirst: Wouldn't you have your class just *extend* the class you need to be a part of? Isn't that the whole point of how subclassing works? If B is a subclass of A, then anywhere an A is expected a B can be used. The whole pass-a-Dog-where-an-Animal-is-the-declared-type thing.

Inner object: Yes! Bingo! So now what happens if you need to pass the IS-A test for two different classes? Classes that aren't in the same inheritance hierarchy?

HeadFirst: Oh, well you just... hmmm. I think I'm getting it. You can always *implement* more than one interface, but you can *extend* only *one* class. You can only be one kind of IS-A when it comes to *class* types.

Inner object: Well done! Yes, you can't be both a Dog and a Button. But if you're a Dog that needs to sometimes be a Button (in order to pass yourself to methods that take a Button), the Dog class (which extends Animal so it can't extend Button) can have an *inner* class that acts on the Dog's behalf as a Button, by extending Button, and thus wherever a Button is required the Dog can pass his inner Button instead of himself. In other words, instead of saying x.takeButton(this), the Dog object calls x.takeButton(new MyInnerButton()).

HeadFirst: Can I get a clear example?

Inner object: Remember the drawing panel we used, where we made our own subclass of JPanel? Right now, that class is a separate, non-inner, class. And that's fine, because the class doesn't need special access to the instance variables of the main GUI. But what if it did? What if we're doing an animation on that panel, and it's getting its coordinates from the main application (say, based on something the user does elsewhere in the GUI). In that case, if we make the drawing panel an inner class, the drawing panel class gets to be a subclass of JPanel, while the outer class is still free to be a subclass of something else.

HeadFirst: Yes I see! And the drawing panel isn't reusable enough to be a separate class anyway, since what it's actually painting is specific to this one GUI application.

Inner object: Yes! You've got it!

HeadFirst: Good. Then we can move on to the nature of the *relationship* between you and the outer instance.

Inner object: What is it with you people? Not enough sordid gossip in a serious topic like polymorphism?

HeadFirst: Hey, you have no idea how much the public is willing to pay for some good old tabloid dirt. So, someone creates you and becomes instantly bonded to the outer object, is that right?

Inner object: Yes that's right. And yes, some have compared it to an arranged marriage. We don't have a say in which object we're bonded to.

HeadFirst: Alright, I'll go with the marriage analogy. Can you get a *divorce* and remarry something *else*?

Inner object: No, it's for life.

HeadFirst: Whose life? Yours? The outer object? Both?

Inner object: Mine. I can't be tied to any other outer object. My only way out is garbage collection.

HeadFirst: What about the outer object? Can it be associated with any other inner objects?

Inner object: So now we have it. This is what you *really* wanted. Yes, yes. My so-called 'mate' can have as many inner objects as it wants.

HeadFirst: Is that like, serial monogamy? Or can it have them all at the same time?

Inner object: All at the same time. There. Satisfied?

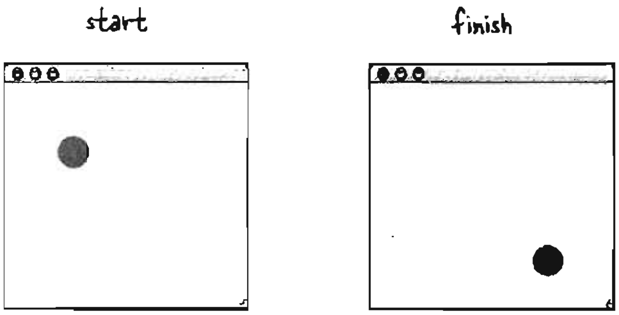
HeadFirst: Well, it does make sense. And let's not forget, it was *you* extolling the virtues of "multiple implementations of the same interface". So it makes sense that if the outer class has three buttons, it would need three different inner classes (and thus three different inner class objects) to handle the events. Thanks for everything. Here's a tissue.

He thinks he's got it made, having *two* inner class objects. But *we* have access to all his private data, so just imagine the damage we could do...

Using an inner class for animation

We saw why inner classes are handy for event listeners, because you get to implement the same event-handling method more than once. But now we'll look at how useful an inner class is when used as a subclass of something the outer class doesn't extend. In other words, when the outer class and inner class are in different inheritance trees!

Our goal is to make a simple animation, where the circle moves across the screen from the upper left down to the lower right.



How simple animation works

1. Paint an object at a particular x and y coordinate

```
g.fillOval(20,50,100,100);
```

20 pixels from the left, 50 pixels from the top

2. Repaint the object at a different x and y coordinate

```
g.fillOval(25,55,100,100);
```

25 pixels from the left, 55 pixels from the top

(the object moved a little down and to the right)

3. Repeat the previous step with changing x and y values for as long as the animation is supposed to continue.

there are no Dumb Questions

Q: **Why are we learning about animation here? I doubt if I'm going to be making games.**

A: You might not be making games, but you might be creating simulations, where things change over time to show the results of a process. Or you might be building a visualization tool that, for example, updates a graphic to show how much memory a program is using, or to show you how much traffic is coming through your load-balancing server. Anything that needs to take a set of continuously-changing numbers and translate them into something useful for getting information out of the numbers.

Doesn't that all sound business-like? That's just the "official justification", of course. The real reason we're covering it here is just because it's a simple way to demonstrate another use of inner classes. (And because we just *like* animation, and our next Head First book is about J2EE and we *know* we can't get animation in that one.)

What we really want is something like...

```
class MyDrawPanel extends JPanel {
      public void paintComponent(Graphics g) {
         g.setColor(Color.orange);
         g.fillOval(x,y,100,100);
      }
}
```

each time paintComponent() is called, the oval gets painted at a different location

Sharpen your pencil

But where do we get the new x and y coordinates?

And who calls repaint()?

See if you can **design a simple solution** to get the ball to animate from the top left of the drawing panel down to the bottom right. Our answer is on the next page, so don't turn this page until you're done!

Big Huge Hint: make the drawing panel an inner class.

Another Hint: don't put any kind of repeat loop in the paintComponent() method.

Write your ideas (or the code) here:

The complete simple animation code

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


public class SimpleAnimation {

    int x = 70;
    int y = 70;
```

make two instance variables in the main GUI class, for the x and y coordinates of the circle.

```
    public static void main (String[] args) {
       SimpleAnimation gui = new SimpleAnimation ();
       gui.go();
    }

    public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);
```

Nothing new here. Make the widgets and put them in the frame.

This is where the action is!

```
       for (int i = 0; i < 130; i++) {

          x++;
          y++;

          drawPanel.repaint();

          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }
```

repeat this 130 times

increment the x and y coordinates

tell the panel to repaint itself (so we can see the circle in the new location)

Slow it down a little (otherwise it will move so quickly you won't SEE it move). Don't worry, you weren't supposed to already know this. We'll get to threads in chapter 15.

```
    }// close go() method


    class MyDrawPanel extends JPanel {

       public void paintComponent(Graphics g) {
          g.setColor(Color.green);
          g.fillOval(x,y,40,40);

       }
    } // close inner class
} // close outer class
```

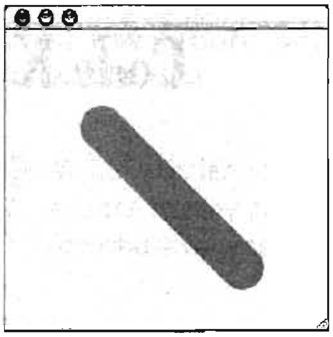
Now it's an inner class.

Use the continually-updated x and y coordinates of the outer class.

Uh-oh. It didn't move... it *smeared.*

What did we do wrong?

There's one little flaw in the paintComponent() method.



Not exactly the look we were going for.

We forgot to <u>erase</u> what was already there! So we got trails.

To fix it, all we have to do is fill in the entire panel with the background color, before painting the circle each time. The code below adds two lines at the start of the method: one to set the color to white (the background color of the drawing panel) and the other to fill the entire panel rectangle with that color. In English, the code below says, "Fill a rectangle starting at x and y of 0 (0 pixels from the left and 0 pixels from the top) and make it as wide and as high as the panel is currently.

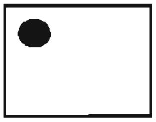
```
public void paintComponent(Graphics g) {
   g.setColor(Color.white);
   g.fillRect(0,0,this.getWidth(), this.getHeight());

   g.setColor(Color.green);
   g.fillOval(x,y,40,40);

}
```

getWidth() and getHeight() are methods inherited from JPanel.

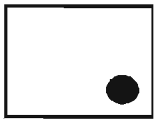
Sharpen your pencil (optional, just for fun)

What changes would you make to the x and y coordinates to produce the animations below? (assume the first one example moves in 3 pixel increments)

1



start

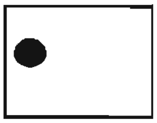


finish

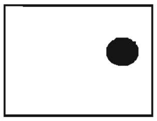
X +3

Y +3

2



start

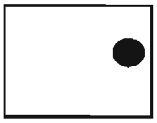


finish

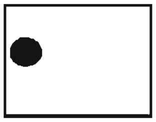
X_____

Y_____

3



start

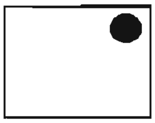


finish

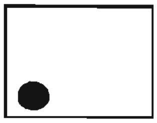
X_____

Y_____

1



start

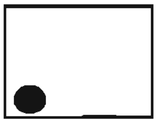


finish

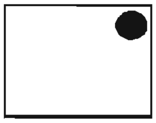
X_____

Y_____

2



start

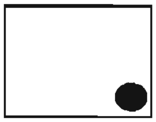


finish

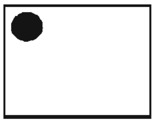
X_____

Y_____

3



start

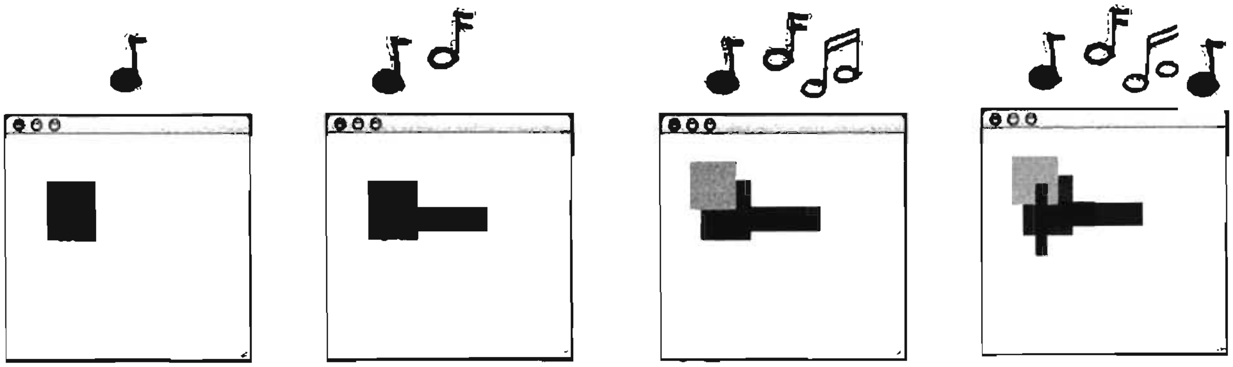


finish

X_____

Y_____

Code Kitchen



beat one

beat two

beat three

beat four ...

Let's make a music video. We'll use Java-generated random graphics that keep time with the music beats.

Along the way we'll register (and listen for) a new kind of non-GUI event, triggered by the music itself.

Remember, this part is all optional. But we think it's good for you. And you'll like it. And you can use it to impress people.

(Ok, sure, it might work only on people who are really easy to impress, but still...)

Listening for a non-GUI event

OK, maybe not a music video, but we *will* make a program that draws random graphics on the screen with the beat of the music. In a nutshell, the program listens for the beat of the music and draws a random graphic rectangle with each beat.

That brings up some new issues for us. So far, we've listened for only GUI events, but now we need to listen for a particular kind of MIDI event. Turns out, listening for a non-GUI event is just like listening for GUI events: you implement a listener interface, register the listener with an event source, then sit back and wait for the event source to call your event-handler method (the method defined in the listener interface).

The simplest way to listen for the beat of the music would be to register and listen for the actual MIDI events, so that whenever the sequencer gets the event, our code will get it too and can draw the graphic. But... there's a problem. A bug, actually, that won't let us listen for the MIDI events *we're* making (the ones for NOTE ON).

So we have to do a little work-around. There is another type of MIDI event we can listen for, called a ControllerEvent. Our solution is to register for ControllerEvents, and then make sure that for every NOTE ON event, there's a matching ControllerEvent fired at the same 'beat'. How do we make sure the ControllerEvent is fired at the same time? We add it to the track just like the other events! In other words, our music sequence goes like this:

BEAT 1 - NOTE ON, CONTROLLER EVENT

BEAT 2 - NOTE OFF

BEAT 3 - NOTE ON, CONTROLLER EVENT

BEAT 4 - NOTE OFF

and so on.

Before we dive into the full program, though, let's make it a little easier to make and add MIDI messages/events since in *this* program, we're gonna make a lot of them.

What the music art program needs to do:

1. Make a series of MIDI messages/events to play random notes on a piano (or whatever instrument you choose)
2. Register a listener for the events
3. Start the sequencer playing
4. Each time the listener's event handler method is called, draw a random rectangle on the drawing panel, and call repaint.

We'll build it in three iterations:

1. Version One: Code that simplifies making and adding MIDI events, since we'll be making a lot of them.
2. Version Two: Register and listen for the events, but without graphics. Prints a message at the command-line with each beat.
3. Version Three: The real deal. Adds graphics to version two.

An easier way to make messages / events

Right now, making and adding messages and events to a track is tedious. For each message, we have to make the message instance (in this case, ShortMessage), call setMessage(), make a MidiEvent for the message, and add the event to the track. In last chapter's code, we went through each step for every message. That means eight lines of code just to make a note play and then stop playing! Four lines to add a NOTE ON event, and four lines to add a NOTE OFF event.

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

Things that have to happen for each event:

1. Make a message instance

```
ShortMessage first = new ShortMessage();
```

2. Call setMessage() with the instructions

```
first.setMessage(192, 1, instrument, 0)
```

3. Make a MidiEvent instance for the message

```
MidiEvent noteOn = new MidiEvent(first, 1);
```

4. Add the event to the track

```
track.add(noteOn);
```

Let's build a static utility method that makes a message and returns a MidiEvent

the four arguments for the message

The event 'tick' for WHEN this message should happen

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
       ShortMessage a = new ShortMessage();
       a.setMessage(comd, chan, one, two);
       event = new MidiEvent(a, tick);

    }catch(Exception e) { }
    return event;
}
```

whoo! A method with five parameters.

make the message and the event, using the method parameters

return the event (a MidiEvent all loaded up with the message)

Example: how to use the new static makeEvent() method

There's no event handling or graphics here, just a sequence of 15 notes that go up the scale. The point of this code is simply to learn how to use our new makeEvent() method. The code for the next two versions is much smaller and simpler thanks to this method.

```
import javax.sound.midi.*; <—— don't forget the import

public class MiniMusicPlayer1 {

   public static void main(String[] args) {

      try {

         Sequencer sequencer = MidiSystem.getSequencer(); <—— make (and open) a sequencer
         sequencer.open(); <————————


         Sequence seq = new Sequence(Sequence.PPQ, 4); <— make a sequence
         Track track = seq.createTrack();         <———— and a track

      for (int i = 5; i < 61; i+= 4) { <——make a bunch of events to make the notes keep
                                          going up (from piano note 5 to piano note 61)

         track.add(makeEvent(144,1,i,100,i));          call our new makeEvent() method to make the
         track.add(makeEvent(128,1,i,100,i + 2));      message and event, then add the result (the
                                                       MidiEvent returned from makeEvent()) to
       } // end loop                                   the track. These are NOTE ON (144) and
                                                       NOTE OFF (128) pairs
          sequencer.setSequence(seq);       }
          sequencer.setTempoInBPM(220);     } start it running
          sequencer.start();                }
      } catch (Exception ex) {ex.printStackTrace();}
   } // close main


   public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
          MidiEvent event = null;
          try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

          }catch(Exception e) { }
         return event;
       }
} // close class
```

Version Two: registering and getting ControllerEvents

We need to listen for ControllerEvents, so we implement the listener interface

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {

     public static void main(String[] args) {
         MiniMusicPlayer2 mini = new MiniMusicPlayer2();
         mini.go();
     }
   public void go() {

      try {
         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();

         int[] eventsIWant = {127};
         sequencer.addControllerEventListener(this, eventsIWant);
```

Register for events with the sequencer. The event registration method takes the listener AND an int array representing the list of ControllerEvents you want. We want only one event, #127.

```
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         for (int i = 5; i < 60; i+= 4) {
            track.add(makeEvent(144,1,i,100,i));

            track.add(makeEvent(176,1,127,0,i));

            track.add(makeEvent(128,1,i,100,i + 2));
         } // end loop

         sequencer.setSequence(seq);
         sequencer.setTempoInBPM(220);
         sequencer.start();
      } catch (Exception ex) {ex.printStackTrace();}
   } // close
```

Here's how we pick up the beat -- we insert our OWN ControllerEvent (176 says the event type is ControllerEvent) with an argument for event number #127. This event will do NOTHING! We put it in JUST so that we can get an event each time a note is played. In other words, its sole purpose is so that something will fire that WE can listen for (we can't listen for NOTE ON/OFF events). Note that we're making this event happen at the SAME tick as the NOTE ON. So when the NOTE ON event happens, we'll know about it because OUR event will fire at the same time.

```
   public void controlChange(ShortMessage event) {
       System.out.println("la");
   }
```

The event handler method (from the ControllerEvent listener interface). Each time we get the event, we'll print "la" to the command-line.

```
   public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
          MidiEvent event = null;
          try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
            event = new MidiEvent(a, tick);

          }catch(Exception e) { }
         return event;
       }
} // close class
```

Code that's different from the previous version is highlighted in gray. (and we're not running it all within main() this time)

Version Three: drawing graphics in time with the music

This final version builds on version two by adding the GUI parts. We build a frame, add a drawing panel to it, and each time we get an event, we draw a new rectangle and repaint the screen. The only other change from version two is that the notes play randomly as opposed to simply moving up the scale.

The most important change to the code (besides building a simple GUI) is that we make the drawing panel implement the ControllerEventListener rather than the program itself. So when the drawing panel (an inner class) gets the event, it knows how to take care of itself by drawing the rectangle.

Complete code for this version is on the next page.

The drawing panel inner class:

The drawing panel is a listener

```
class MyDrawPanel extends JPanel implements ControllerEventListener {
```

We set a flag to false, and we'll set it to true only when we get an event.

```
    boolean msg = false;

    public void controlChange(ShortMessage event) {
       msg = true;
       repaint();
    }
```

We got an event, so we set the flag to true and call repaint()

```
    public void paintComponent(Graphics g) {
     if (msg) {
```

We have to use a flag because OTHER things might trigger a repaint(), and we want to paint ONLY when there's a ControllerEvent

```
       Graphics2D g2 = (Graphics2D) g;

       int r = (int) (Math.random() * 250);
       int gr = (int) (Math.random() * 250);
       int b = (int) (Math.random() * 250);

       g.setColor(new Color(r,gr,b));

       int ht = (int) ((Math.random() * 120) + 10);
       int width = (int) ((Math.random() * 120) + 10);
       int x = (int) ((Math.random() * 40) + 10);
       int y = (int) ((Math.random() * 40) + 10);
       g.fillRect(x,y,ht, width);
       msg = false;

     } // close if
   } // close method
 }  // close inner class
```

The rest is code to generate a random color and paint a semi-random rectangle.

Sharpen your pencil

This is the complete code listing for Version Three. It builds directoy on Version Two. Try to annotate it yourself, without looking at the previous pages.

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

     static JFrame f = new JFrame(“My First Music Video”);
     static MyDrawPanel ml;

     public static void main(String[] args) {
           MiniMusicPlayer3 mini = new MiniMusicPlayer3();
           mini.go();
     } // close method


     public  void setUpGui() {
        ml = new MyDrawPanel();
        f.setContentPane(ml);
        f.setBounds(30,30, 300,300);
        f.setVisible(true);
    } // close method

    public void go() {
       setUpGui();

       try {

          Sequencer sequencer = MidiSystem.getSequencer();
          sequencer.open();
          sequencer.addControllerEventListener(ml, new int[] {127});
          Sequence seq = new Sequence(Sequence.PPQ, 4);
          Track track = seq.createTrack();

          int r = 0;
          for (int i = 0; i < 60; i+= 4) {

             r = (int) ((Math.random() * 50) + 1);
             track.add(makeEvent(144,1,r,100,i));
             track.add(makeEvent(176,1,127,0,i));
             track.add(makeEvent(128,1,r,100,i + 2));
          } // end loop

          sequencer.setSequence(seq);
          sequencer.start();
          sequencer.setTempoInBPM(120);
       } catch (Exception ex) {ex.printStackTrace();}
    } // close method
```

Who am I?

A bunch of Java hot-shots, in full costume, are playing the party game "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one guy, then write down all for whom that sentence applies. Fill in the blanks next to the sentence with the names of one or more attendees.

Tonight's attendees:

Any of the charming personalities from this chapter just might show up!

I got the whole GUI, in my hands.

Every event type has one of these.

The listener's key method.

This method gives JFrame its size.

You add code to this method but never call it.

When the user actually does something, it's an _____ .

Most of these are event sources.

I carry data back to the listener.

An addXxxListener( ) method says an object is an _____ .

How a listener signs up.

The method where all the graphics code goes.

I'm typically bound to an instance.

The 'g' in (Graphics g), is really of class.

The method that gets paintComponent( ) rolling.

The package where most of the Swingers reside.

BE the compiler

The Java file on this page represents a complete source file. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it, and if it does compile, what would it do?

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

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                 JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    frame.getContentPane().add(
                 BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

Exercise Solutions

Who am I?

I got the whole GUI, in my hands.	JFrame
Every event type has one of these.	listener interface
The listener's key method.	actionPerformed( )
This method gives JFrame its size.	setSize( )
You add code to this method but never call it.	paintComponent( )
When the user actually does something, it's an ____	event
Most of these are event sources.	swing components
I carry data back to the listener.	event object
An addXxxListener( ) method says an object is an ___	event source
How a listener signs up.	addActionListener( )
The method where all the graphics code goes.	paintComponent( )
I'm typically bound to an instance.	inner class
The 'g' in (Graphics g), is really of this class.	Graphics2d
The method that gets paintComponent( ) rolling.	repaint( )
The package where most of the Swingers reside.	javax.swing

BE the compiler

Once this code is fixed, it will create a GUI with a button that toggles between A and B when you click it.

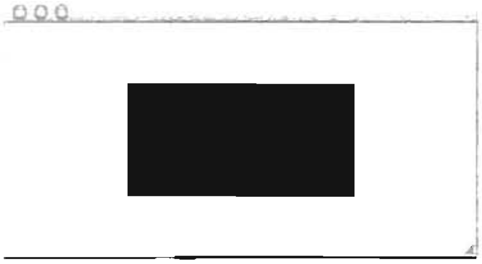
```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener( new BListener() );

    frame.getContentPane().add(
                    BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

The addActionListener( ) method takes a class that implements the ActionListener interface

ActionListener is an interface, interfaces are implemented, not extended

Pool Puzzle

The Amazing, Shrinking, Blue Rectangle.



```
import javax.swing.*;
import java.awt.*;
public class Animate {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animate gui = new Animate ();
       gui.go();
   }
   public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
       MyDrawP drawP = new MyDrawP();
       frame.getContentPane().add(drawP);
       frame.setSize(500,270);
       frame.setVisible(true);
       for (int i = 0; i < 124; i++,x++,y++ ) {
          x++;
          drawP.repaint();
          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }
   }
   class MyDrawP extends JPanel {
       public void paintComponent(Graphics g ) {
          g.setColor(Color.white);
          g.fillRect(0,0,500,250);
          g.setColor(Color.blue);
          g.fillRect(x,y,500-x*2,250-y*2);
       }
   }
}
```