that was overridden? No. If we really want to execute the original `turnLeft`, we should not have overridden it. Instead, we should have simply created a new method, perhaps called `fastTurnLeft`.

2.6.3 Side Effects

`FastTurnBot` has a problem, however. Suppose that Listing 2-12 contained the statement `speedy.setSpeed(20);` just before line 12. This statement would speed `speedy` up dramatically. Presumably, the programmer wanted `speedy` to be speedier than normal all of the time. After its first `turnLeft`, however, `speedy` would return to its normal pace of 2 moves per second.

This phenomenon is called a **side effect**. Invoking `turnLeft` changed something it should not have changed. Our programmer will be very annoyed if she must reset the speed after every command that turns the robot. Ideally, a `FastTurnBot` returns to its previous speed after each turn.

KEY IDEA

Not only are side effects annoying, they can lead to errors. Avoid them where possible; otherwise, document them.

The programmer can use the `getSpeed` query to find out how long the robot currently takes to turn. This information can be used to adjust the speed to its original value after the turn is completed. The new version of `turnLeft` should perform the following steps:

set the speed to 10 times the current speed
turn left
set the speed to one-tenth of the (now faster) speed

The query `this.getSpeed()` obtains the current speed. Multiplying the speed by 10 and using the result as the value to `setSpeed` increases the speed by a factor of 10. After the turn, we can do the reverse to decrease the speed to its previous value, as shown in the following implementation of `turnLeft`:

LOOKING AHEAD

Another approach is to remember the current speed. When the robot is finished turning, set the speed to the remembered value. More in Chapters 5 and 6.

```
public void turnLeft()
{ this.setSpeed(this.getSpeed() * 10);
  super.turnLeft();
  this.setSpeed(this.getSpeed() / 10);
}
```

Using queries and doing arithmetic will be discussed in much more detail in the following chapters.

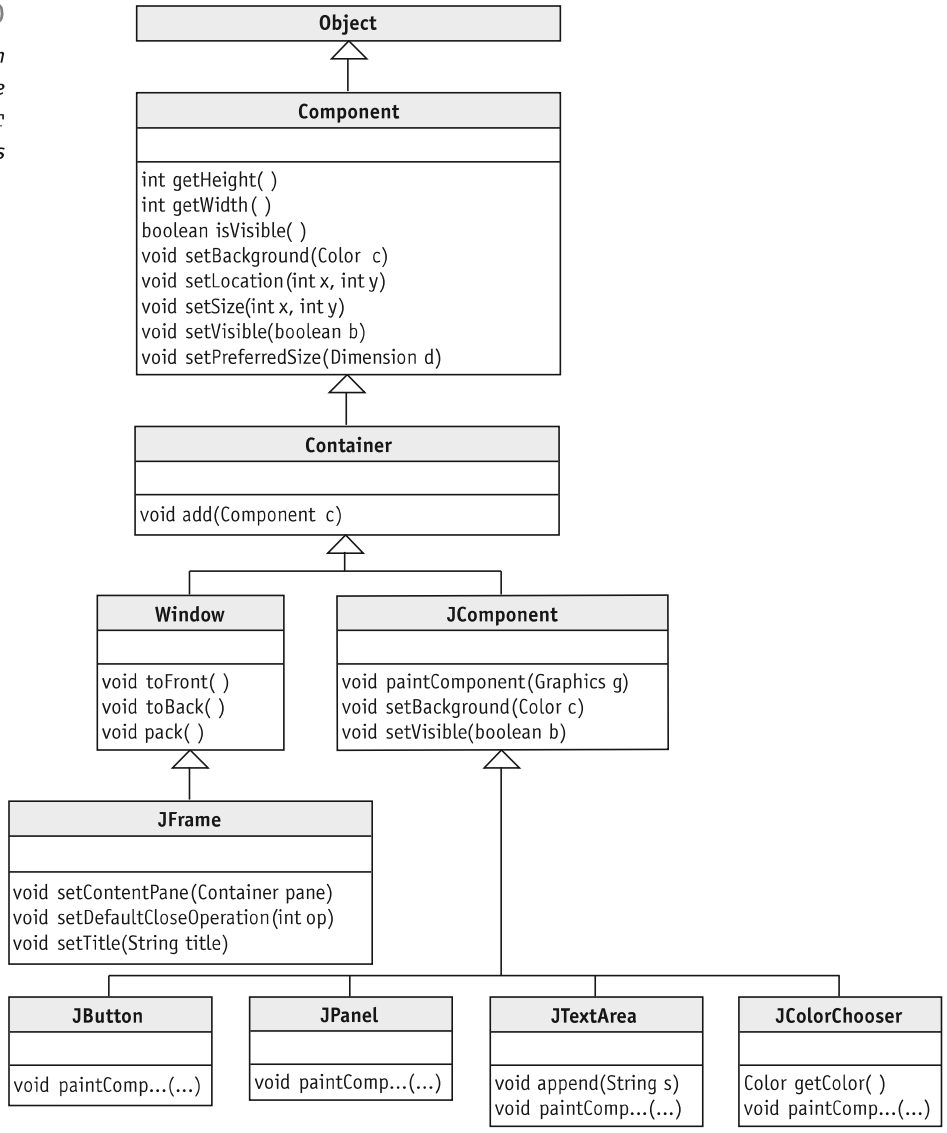
2.7 GUI: Extending GUI Components

The Java package that implements user interfaces is known as the **Abstract Windowing Toolkit** or **AWT**. A newer addition to the AWT is known as **Swing**. These packages contain classes to display components such as windows, buttons, and textboxes. Other classes work to receive input from the mouse, to define colors, and so on.

The AWT and Swing packages make extensive use of inheritance. Figure 2-14 contains a class diagram showing a simplified version of the inheritance hierarchy. Many classes are omitted, as are many methods and all attributes.

(figure 2-14)

Simplified class diagram showing the inheritance hierarchy for some `AWT` *and* `Swing` *classes*



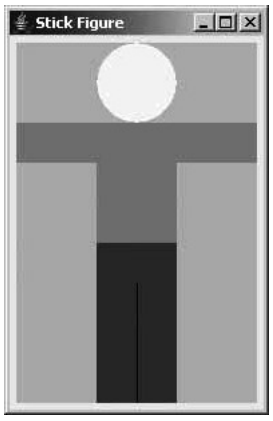
One new aspect of this class diagram is that some classes have two or more subclasses. For example, `Container` is the superclass for both `Window` and `JComponent`. The effect is that `Window` objects and `JComponent` objects (and their subclasses) have much in common—the set of services they inherit from the `Container` and `Component` classes.

The class diagram reveals several new pieces of information:

- When we implemented the `FramePlay` program in Listing 1-6, we sent six different messages to a `JFrame` object: `setContentPane`, `setTitle`, `setDefaultCloseOperation`, `setLocation`, `setSize`, and `setVisible`. We now realize that only three of these are actually declared by the `JFrame` class. The other three services are offered by `JFrame` because they are inherited from `Component`.
- Because `JFrame`, `JPanel`, `JButton`, and so on, all indirectly extend `Component`, they can all answer queries about their width, height, and visibility, and can all[2] set their background color, position, size, and visibility.
- The `JComponent` class overrides two of the services provided by `Component`. `JComponent` must be doing something extra for each of those services.
- The statement `contents.add(saveButton);` in the `FramePlay` program added a button to an instance of `JPanel`. We now see that `add` is actually a service of the `Container` class, inherited by `JPanel`.
- Each of the classes extending `JComponent` inherits the method `paintComponent`. Perhaps if this method were overridden, we could affect how the component looks. This result would, indeed, be the case and is the topic of the next section.

2.7.1 Extending `JComponent`

In this section we will write a program that paints a picture. Figure 2-15 shows a simple stick figure. When viewed in color, the pants are blue, the shirt is red, and the head is yellow.



(figure 2-15)

Simple stick figure

[2] There is, unfortunately, some fine print. The statements above are true, but in some circumstances you can't see the results. For example, setting the background color of a `JFrame` doesn't appear to have an effect because the content pane completely covers the `JFrame`, and you see the content pane's color.

Our strategy is to create a new class, `StickFigure`, which extends `JComponent`. We choose to extend `JComponent` because it is the simplest of the components shown in the class diagram, and it doesn't already have its own appearance. We will extend it by overriding `paintComponent`, the method responsible for the appearance of the component. As we did with the several components in the `FramePlay` program in Listing 1-6, the stick figure component will be placed in a `JPanel`. The `JPanel` will be set as the content pane in a `JFrame`.

Listing 2-13 shows the beginnings of the `StickFigure` class. It provides a parameterless constructor and nothing more. The constructor doesn't need parameters because `JComponent` has a constructor that does not need parameters. Our constructor calls `JComponent`'s constructor by invoking `super` without parameters.

The constructor performs one important task: in lines 13–14 it specifies a preferred size for the stick figure component. The preferred size says how many pixels wide and high the component should be, if possible. Line 13 creates a `Dimension` object 180 pixels wide and 270 pixels high. The next line uses this object to set the preferred size for the stick figure.

FIND THE CODE

ch02/stickFigure/

PATTERN

Extended Class Constructor

Listing 2-13: *An extended* `JComponent`

```
1  import javax.swing.*;       // JComponent
2  import java.awt.*;          // Dimension
3
4  /** A new kind of component that displays a stick figure.
5  *
6  *  @author Byron Weber Becker */
7  public class StickFigure extends JComponent
8  {
9    public StickFigure()
10   { super();
11
12     // Specify the preferred size for this component
13     Dimension prefSize = new Dimension(180, 270);
14     this.setPreferredSize(prefSize);
15   }
16 }
```

It is also possible to reduce lines 13 and 14 down to a single line:

```
this.setPreferredSize(new Dimension(180, 270));
```

This creates the object and passes it to `setPreferredSize` without declaring a variable. We can avoid declaring the variable if we don't need to refer to the object in the future (as with `Wall` and `Thing` objects), or we can pass it to the only method that requires it as soon as it's created, as we do here.

KEY IDEA

Sometimes we don't need a variable to store object references.

Now would be a good time to implement the `main` method for the program. By compiling and running the program early in the development cycle, we can often catch errors in our thinking that may be much more difficult to change later on. Listing 2-14 shows a program for this purpose. Running it results in an empty frame as shown in Figure 2-16. It follows the Display a Frame pattern and consequently it is similar to the `FramePlay` program in Listing 1-6.

FIND THE CODE

ch02/stickFigure/

Listing 2-14: *A program that uses a class extending* `JComponent`

```
1  import javax.swing.*;
2
3  /** Create a stick figure and display it in a frame.
4  *
5  *  @author Byron Weber Becker */
6  public class Main
7  {
8    public static void main(String[] args)
9    {  // Declare the objects to show.
10     JFrame frame = new JFrame();
11     JPanel contents = new JPanel();
12     StickFigure stickFig = new StickFigure();
13
14     // Add the stick figure to the contents.
15     contents.add(stickFig);
16
17     // Display the contents in a frame.
18     frame.setContentPane(contents);
19     frame.setTitle("Stick Figure");
20     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21     frame.setLocation(250, 100);
22     frame.pack();
23     frame.setVisible(true);
24   }
25 }
```

One difference between this program and the `FramePlay` program and the pattern is in how the frame is sized. The previous program explicitly set the size of the frame using the `setSize` method. This version uses the method `pack` in line 22. This method uses the preferred sizes of all the components to calculate the best size for the frame.

The result of running this program is shown in Figure 2-16. It looks exactly like an empty `JFrame` because the `JComponent` is invisible until we override `paintComponent` to change its appearance.

(figure 2-16)

Result of running the program in Listing 2-14 with the incomplete `StickFigure` *class from Listing 2-13*

2.7.2 Overriding `paintComponent`

To actually draw the stick figure, we need to override `paintComponent` to provide it with additional functionality. We know from both the class diagram in Figure 2-14 and the online documentation that `paintComponent` has a parameter of type `Graphics`. This parameter is often named simply `g`. We will have much more to say about parameters in later chapters. For now, we will just say that `g` is a reference to an object that is used for drawing. It is provided by the client that calls `paintComponent` and may be used by the code contained in the `paintComponent` method.

LOOKING AHEAD

We are practicing incremental development: code a little, test a little, code a little, test a little. For more on development strategies, see Chapter 11.

The superclass's implementation of `paintComponent` may have important work to do, and so it should be called with `super.paintComponent(g)`. It requires a `Graphics` object as an argument, and so we pass it `g`, the `Graphics` object received as a parameter. Doing so results in the following method. The method still has not added any functionality, but adding it to Listing 2-13 between lines 14 and 15 still results in a running program.

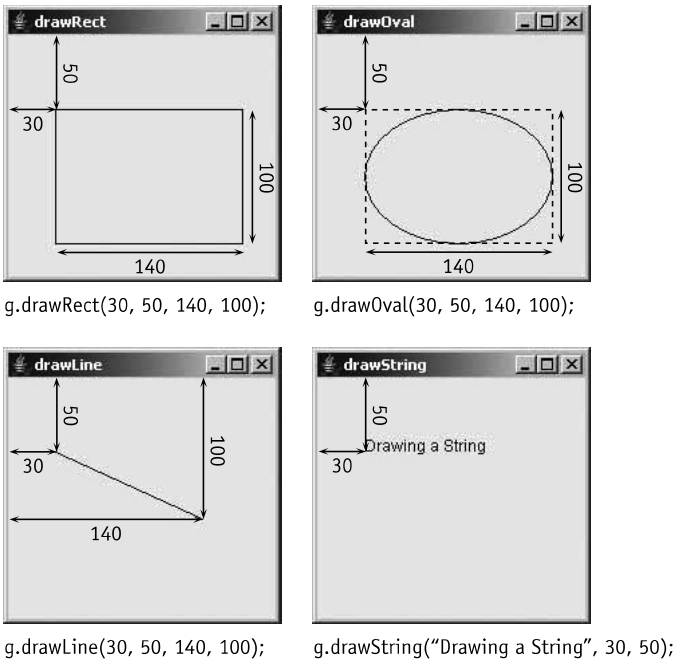
```
public void paintComponent(Graphics g)
{ super.paintComponent(g);
}
```

The `Graphics` parameter, `g`, provides services such as `drawRect`, `drawOval`, `drawLine`, and `drawString`, each of which draw the shape described in the service's name. A companion set of services includes `fillRect` and `fillOval`, each of which also draws the described shape and then fills the interior with a color. The color used is

determined by the most recent `setColor` message sent to `g`. The color specified is used until the next `setColor` message.

All of the `draw` and `fill` methods require parameters specifying where the shape is to be drawn and how large it should be. Like positioning a frame, measurements are given in relation to an origin in the upper-left corner, and are in pixels.

Figure 2-17 shows the relationship between the parameters and the figure that is drawn. For `drawRect` and `drawOval`, the first two parameters specify the position of the upper left corner of the figure, while the third and fourth parameters specify the width and height. For an oval, the width and height are of the smallest box that can contain the oval. This box is called the **bounding box** and is shown in Figure 2-17 as a dashed line. Of course, the bounding box is not actually drawn on the screen.



g.drawRect(30, 50, 140, 100);

g.drawOval(30, 50, 140, 100);

g.drawLine(30, 50, 140, 100);

g.drawString("Drawing a String", 30, 50);

(figure 2-17)

Relationship between the arguments and the effects of four drawing methods

In each of these methods, the order of the arguments is *x* before *y* and *width* before *height*.

The parameters for a line are different from the parameters for rectangles and ovals. The first two parameters specify one end of the line in relation to the origin, while the last two parameters specify the other end of the line in relation to the origin.

The `drawString` method takes a string as the first parameter and the position of the first letter as the second and third parameters.

With this background information, we can finally add the statements to draw the stick figure. The complete code for the `stickFigure` class is given in Listing 2-15. Running it with the `main` method in Listing 2-14 produces the image shown in Figure 2-15.

FIND THE CODE

ch02/stickFigure/

Listing 2-15: *Overriding* `paintComponent` *to draw a stick figure*

```
1  import javax.swing.*;       // JComponent
2  import java.awt.*;          // Dimension
3
4  /** A new kind of component that displays a stick figure.
5  *
6  *  @author Byron Weber Becker */
7  public class StickFigure extends JComponent
8  {
9    public StickFigure()
10   { super ();
11     Dimension prefSize = new Dimension(180, 270);
12     this.setPreferredSize(prefSize);
13   }
14
15   // Paint a stick figure.
16   public void paintComponent(Graphics g)
17   { super.paintComponent(g);
18
19     // Paint the head.
20     g.setColor(Color.YELLOW);
21     g.fillOval(60, 0, 60, 60);
22
23     // Paint the shirt.
24     g.setColor(Color.RED);
25     g.fillRect(0, 60, 180, 30);
26     g.fillRect(60, 60, 60, 90);
27
28     // Paint the pants.
29     g.setColor(Color.BLUE);
30     g.fillRect(60, 150, 60, 120);
31     g.setColor(Color.BLACK);
32     g.drawLine(90, 180, 90, 270);
33   }
34 }
```

Constructor

Parameterless Command

2.7.3 How `paintComponent` Is Invoked

You may have noticed that the `paintComponent` method is *not* called from anywhere in Listing 2-15 or the client code shown in Listing 2-14. Look all through the code, and you will not find an instance of the Command Invocation pattern `stickFig.paintComponent(g);`. Yet we know it is invoked because it paints the stick figure. How?

In the Sequential Execution pattern in Chapter 1, we described statements as being executed one after another, as if they were strung on a thread of string. A computer program can have two or more of these **threads**, each with their own sequence of statements. The program we just wrote has at least two threads. The first one is in the `main` method. It creates a `JFrame` and invokes a number of its commands such as `setDefaultCloseOperation` and `setVisible`. When it gets to the end of the `main` method, that thread ends.

When a `JFrame` is instantiated, a second thread begins. This is *not* a normal occurrence when an object is instantiated; `JFrame`'s authors deliberately set up the new thread. `JFrame`'s thread monitors the frame and detects when it has been damaged and must be repainted. A frame can be damaged in many ways. It is damaged when the user resizes it by dragging a border or clicking the minimize or maximize buttons. It's damaged when it is first created because it hasn't been drawn yet. It's also damaged if another window is placed on top of it and then moved again. In each of these cases, the second thread of control calls `paintComponent`, providing the `Graphics` object that `paintComponent` should draw upon.

LOOKING AHEAD

We will use this capability in Section 3.5.2 to make two or more robots move simultaneously.

KEY IDEA

`paintComponent` *is called by "the system." We don't call it.*

2.7.4 Extending `Icon`

We learned in Section 2.3.1 that `Icon` is the class used to represent images of things in the robot world—robots, intersections, things, flashers, walls, and so on—all use icons to display themselves. As you might expect, `Icon` has been extended a number of times to provide different icons for different kinds of things. The documentation references classes named `FlasherIcon`, `RobotIcon`, `WallIcon`, and so on.

You, too, can extend the `Icon` class to create your own custom icons. The example shown in Figure 2-18 was produced by the code shown in Listing 2-16.

As with any other subclass, it gives the name of the class it extends (line 5). Before that are the packages it relies on. In this case, it imports the `Icon` class from the `becker.robots.icons` package and the `Graphics` class from `java.awt`.