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`

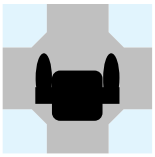
We learned in Section 2.3.1 that `Icon` is the class used to represent images of things in the robot world—robots, intersections, things, flashers, walls, and so on—all use icons to display themselves. As you might expect, `Icon` has been extended a number of times to provide different icons for different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

(figure 2-18)

Custom robot icon



One difference, when compared to extending `JComponent`, is that we must override a method named `paintIcon` instead of `paintComponent`. This fact can be gleaned from reading the documentation for `Icon`. Like `paintComponent`, `paintIcon` has a parameter of type `Graphics` to use for the actual drawing.

An icon is always painted in a standard 100 × 100 pixel space facing north. Lines 14–23 in Listing 2-16 draw the robot in this position. Other parts of the robot system scale and rotate the icons, as necessary.

FIND THE CODE

ch02/extendIcon/

Listing 2-16: *Code for a customized robot icon*

```
1  import becker.robots.icons.*;      // Icon
2  import java.awt.*;                 // Graphics, Color
3
4  /** Create a robot icon that has arms. */
5  public class ArmRobotIcon extends Icon
6  {
7     /** Create a new icon for a robot. */
8     public ArmRobotIcon()
9     { super();
10    }
11
12    /** Paint the icon. */
13    public void paintIcon(Graphics g)
14    { g.setColor(Color.BLACK);
15
16       // body
17       g.fillRoundRect(35, 35, 30, 30, 10, 10);
18       // shoulders
19       g.fillRect(25, 45, 10, 10);
20       g.fillRect(65, 45, 10, 10);
21       // arms
22       g.fillOval(25, 25, 10, 30);
23       g.fillOval(65, 25, 10, 30);
24    }
25 }
```

Use the `setIcon` method to change the icon used to display a robot. One way to call `setIcon` is to create a new class of robots, as shown in Listing 2-17.

Listing 2-17: *An* `ArmRobot` *uses an* `ArmRobotIcon` *to display itself*

```
1  import becker.robots.*;
2
3  /** A robot with an icon that shows arms. */
4  public class ArmRobot extends Robot
5  {
6    /** Construct a new ArmRobot.
7     * @param aCity          The City where the robot will reside.
8     * @param aStreet        The robot's initial street.
9     * @param anAvenue       The robot's initial avenue.
10    * @param aDirection     The robot's initial direction. */
11   public ArmRobot(City aCity, int aStreet, int anAvenue,
12                   Direction aDirection)
13   { super(aCity, aStreet, anAvenue, aDirection);
14     this.setIcon(new ArmRobotIcon());
15   }
16 }
```

FIND THE CODE

ch02/extendIcon/

2.8 Patterns

In this chapter we've seen patterns to extend a class, write a constructor, and implement a parameterless command. These are all extremely common patterns; so common, in fact, that many experienced programmers wouldn't even recognize them as patterns. We've also seen a much less common pattern to draw a picture.

2.8.1 The Extended Class Pattern

Name: Extended Class

Context: You need a new kind of object to provide services for a program you are writing. An existing class provides objects with closely related services.

Solution: Extend the existing class to provide the new or different services required. For example, the following listing illustrates a new kind of robot that provides a service to turn around.

```
import becker.robots.*;
public class TurnAroundBot extends Robot
{
```