What are the actions we must repeat? To plant a line of three things beginning with the next intersection, for example, we must perform the following actions:

move
plant a thing
move
plant a thing
move
plant a thing

The actions that are repeated are moving and planting a thing. They form the body of the `while` statement. We want them to be performed a specific number of times, as specified by the parameter. This is an ideal application for a count-down loop. This method is, in fact, identical to the `step` method developed earlier except that we also need to plant a `Thing` on the intersection. The code for the method follows:

PATTERN
Count-Down Loop

```
/** Plant a line of Things beginning with the intersection in front of the robot.
*   @param length        The length of the line. */
protected void plantLine(int length)
{ while (length > 0)
  { this.move();
    this.plantIntersection();
    length = length - 1;
  }
}
```

Finally, `plantIntersection` is a method to make future change easy. It contains a single call to `putThing`, as shown in the following code:

```
/** Plant one intersection. */
protected void plantIntersection()
{ this.putThing();
}
```

This completes the implementation of the `RectanglePlanter` class. In the course of its development we have demonstrated:

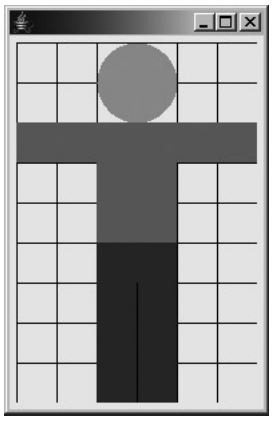
- A method with more than one parameter
- Passing parameters as arguments to helper methods
- A count-down loop with a more complex set of actions

4.7 GUI: Scaling Images

The graphical user interface section of Chapter 2 introduced us to drawing pictures similar to the one shown in Figure 4-16. In this section, we'll see how to use queries in the `JComponent` class to make our image adapt to different sizes.

(figure 4-16)

Stick figure and a grid used to design it



The code for the main method is shown in Listing 2-13. `StickFigure` is the class that does the actual drawing. It was originally shown in Chapter 2 and is reproduced in Listing 4-6. Notable points are that it sets the preferred size for the component in the constructor at line 8 and 9, and overrides `paintComponent` to draw the actual image.

FIND THE CODE

ch02/stickFigure/

Listing 4-6: *A class to draw a stick figure*

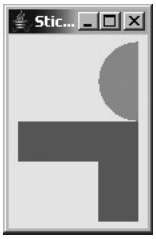
```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class StickFigure extends JComponent
5  {
6    public StickFigure()
7    { super();
8      Dimension prefSize = new Dimension(180, 270);
9      this.setPreferredSize(prefSize);
10   }
11
12   // Draw a stick figure.
13   public void paintComponent(Graphics g)
14   { super.paintComponent(g);
15
16     // head
17     g.setColor(Color.YELLOW);
18     g.fillOval(60, 0, 60, 60);
19
20     // shirt
21     g.setColor(Color.RED);
22     g.fillRect(0, 60, 180, 30);
23     g.fillRect(60, 60, 60, 90);
```

Listing 4-6: *A class to draw a stick figure* (continued)

```
24
25        // pants
26        g.setColor(Color.BLUE);
27        g.fillRect(60, 150, 60, 120);
28        g.setColor(Color.BLACK);
29        g.drawLine(90, 180, 90, 270);
30     }
31  }
```

Now, suppose that we wanted the image to be a different size. The preferred size set in the `StickFigure` constructor could be replaced with, for example, `new Dimension(90, 135)` to make the image half as big in each dimension.

There is a problem, however. Making only this one change results in an image similar to the one shown in Figure 4-17. Unfortunately, all of the calculations to draw the stick figure were based on the old size of 180 pixels wide and 270 pixels high.



(figure 4-17)

Naively changing the size of the stick figure

4.7.1 Using Size Queries

We can make the image less sensitive to changes in size by drawing it relative to the current size of the component. We can obtain the current size of the component, in pixels, with the `getWidth` and `getHeight` queries. To paint a shape that covers a fraction of the component, multiply the results of these queries by a fraction. For example, to specify an oval that is two-sixths of the width of the component and two-ninths of the height, we can use the following statement:

```
g.fillOval(0, 0, this.getWidth()*2/6, this.getHeight()*2/9);
```

The first two parameters will place the oval at the upper-left corner of the component.

The original stick figure was designed on a grid six units wide and nine units high. Figure 4-16 shows this grid explicitly and makes it easy to figure out which fractions to multiply by the width or the height. For example, the head can be painted with

```
g.fillOval(this.getWidth()*2/6, this.getHeight()*0/9,
           this.getWidth()*2/6, this.getHeight()*2/9);
```

The first pair of parameters says the head's bounding box should start 2/6th of the component's width from the left edge and 0/9th of the component's height from the top. The second parameter could be replaced by 0.

Converting the remaining method calls to use the `getWidth` and `getHeight` queries follows a similar pattern. It is tedious, however. Fortunately, there is a better approach.

4.7.2 Scaling an Image

Using `getWidth` and `getHeight` to scale an image follows a very predictable pattern, as shown in the last section. Fortunately, the designers of Java have provided a way for us to exploit that pattern with much less work on our part. They have provided a way for the computer to automatically multiply by the width or the height of the component and divide by the number of units on our grid. All we need to do is supply the numerator of the fraction that places the image or says how big it is. For example, in the previous section we wrote the following statements:

```
g.fillOval(this.getWidth()*2/6, this.getHeight()*0/9,
           this.getWidth()*2/6, this.getHeight()*2/9);
```

Java's drawing methods can be set up so that this method call is replaced with the following statement:

```
g.fillOval(2, 0, 2, 2);
```

Using this approach requires three things: Using a more capable version of the `Graphics` object, setting the scale to be used in drawing, and scaling the width of the lines to use in drawing. All are easy and follow a pattern consisting of the following four lines inserted at the beginning of `paintComponent`:

Scale an Image

```
1 // Standard stuff to scale the image
2 Graphics2D g2 = (Graphics2D)g;
3 g2.scale(this.getWidth()/6, this.getHeight()/9);
4 g2.setStroke(new BasicStroke(1.0F/this.getWidth()));
```

Explaining this code in more detail requires advanced concepts; however, the overview is as follows:

- Line 2 makes a larger set of capabilities in `g` available. More about this in Chapter 12.
- Line 3 tells the `Graphics` object how to multiply values to scale our paintings appropriately.
- Line 4 makes the width of a line, also called a stroke, proportional to the scaling performed in line 3.

Whether or not we understand exactly what these lines of code do, using them is easy:

- Import the package `java.awt.*`.
- Copy these four lines to the beginning of your `paintComponent` method.
- Decide on the size of your grid, and change the "6" and "9" in the call to `scale` accordingly. For a 50 x 100 grid, change the 6 to 50 and the 9 to 100.
- Use `g2` instead of `g` to do the painting.

The resulting `paintComponent` method is shown in Listing 4-7.

Listing 4-7: *Painting a stick figure by scaling the image*

```
1  public void paintComponent(Graphics g)
2  { super.paintComponent(g);
3
4    // Standard stuff to scale the image
5    Graphics2D g2 = (Graphics2D)g;
6    g2.scale(this.getWidth()/6, this.getHeight()/9);
7    g2.setStroke(new BasicStroke(1.0F/this.getWidth()));
8
9    // head
10   g2.setColor(Color.YELLOW);
11   g2.fillOval(2, 0, 2, 2);
12
13   // shirt
14   g2.setColor(Color.RED);
15   g2.fillRect(0, 2, 6, 1);
16   g2.fillRect(2, 2, 2, 3);
17
18   // pants
19   g2.setColor(Color.BLUE);
20   g2.fillRect(2, 5, 2, 4);
21   g2.setColor(Color.BLACK);
22   g2.drawLine(3, 6, 3, 9);
23 }
```

FIND THE CODE

ch04/stickFigure/