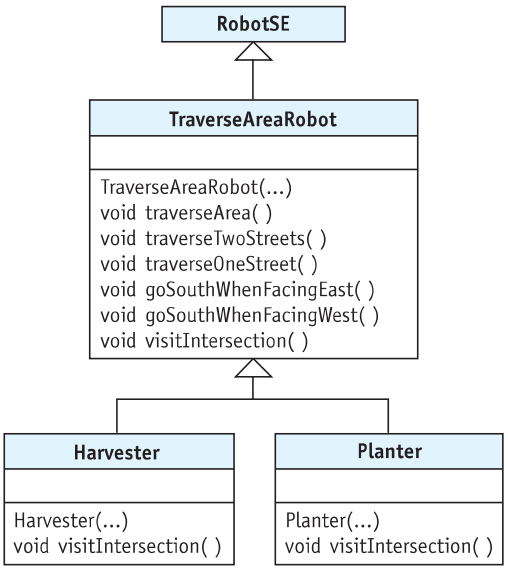
(figure 3-12)

*Class diagram for a group of classes for working with fields*




Of course, we could have solved the planting problem by extending the `Harvester` class and overriding `harvestIntersection`. The approach shown in Figure 3-12 differs from that in two ways. The first difference is that we planned for various tasks to occur at each intersection and named the methods accordingly. It is confusing to override a method named `harvestIntersection` so that it plants something instead of harvesting.

PATTERN

*Template Method*

The second difference is that the `TraverseAreaRobot` class deliberately does nothing when it visits an intersection. Instead, `visitIntersection` serves as an intentional point where subclasses can modify the behavior of `traverseArea`. In fact, the documentation for `visitIntersection` and `traverseArea` should explicitly describe the possibilities of overriding the method. In a sense, `traverseArea` is a template for a common activity, which is modified by overriding `visitIntersection`.

## 3.6 Private and Protected Methods

The `TraverseAreaRobot` class makes available six new services: `traverseArea`, `traverseTwoStreets`, `traverseOneStreet`, `goSouthWhenFacingEast`, `goSouthWhenFacingWest`, and `visitIntersection`. Should these all be available to all clients? For example, should a client such as `main` be able to invoke the `goSouthWhenFacingEast` method? After all, it was developed as a helper method, not as a service to be offered by a `TraverseAreaRobot`. Perhaps a client should not be allowed to invoke it.

Recall that a client is an object that uses the services of another object, called the server. The client uses the server's services by invoking its corresponding method with the Command Invocation pattern described in Section 1.7.3:

```
«objectReference».«methodName»(«parameterList»);
```

The client is the class that contains code, such as `karel.move()`, `joe.traverseArea()`, or even `this.goSouthWhenFacingEast()`. In these cases, `karel`, `joe`, and `this` are the *«objectReference»*s.

Java has a set of **access modifiers** that control which clients are allowed to invoke a method. The access modifier is placed as the first keyword before the method signature.

So far, we have used the access modifier `public`, as in `public void traverseArea()`. The keyword `public` allows any client to access the method. Like a public telephone, anyone who comes by can use it.

**KEY IDEA**

*Public methods may be invoked by any client.*

The access modifier `private` is at the other end of the scale. It says that no one except clients who belong to the same class, may invoke the method, and that the method may not be overridden. Staying private is what we want for many helper methods. `goSouthWhenFacingEast`, for example, was designed to help `traverseTwoStreets` do its work; it should not be called from outside of the class where it was declared. It should therefore be declared as follows:

```
private void goSouthWhenFacingEast()
```

**KEY IDEA**

*Private methods can only be invoked by methods defined in the same class.*

A middle ground is to use the `protected` access modifier. Protected methods may be invoked from clients that are also subclasses. Like all methods, protected methods can also be invoked from within the class defining them.

**KEY IDEA**

*Protected methods can be used from subclasses.*

Using `protected` on the `traverseOneStreet` and `visitIntersection` methods would be appropriate. It would allow us to override and use those methods in a subclass to traverse longer streets. We also did this in Section 3.3.4 when we overrode `harvestOneRow` to harvest a longer row. This approach is shown in Listing 3-6 and Listing 3-7. Listing 3-8 shows code that does *not* compile because it attempts to use `protected` and `private` methods.

**Listing 3-6:** *Using* `protected` *and* `private` *access modifiers in* `TraverseAreaRobot`

```
1 public class TraverseAreaRobot extends RobotSE
2 { public TraverseAreaRobot(...)          {     ...     }
3
4   public void traverseArea()             {     ...     }
5
6   private void traverseTwoStreets()      {     ...     }
7
```

**Listing 3-6:** *Using* `protected` *and* `private` *access modifiers in* `TraverseAreaRobot` (continued)

```
8    protected void traverseOneStreet()      {      ...      }
9
10   private void goSouthWhenFacingEast()    {      ...      }
11
12   private void goSouthWhenFacingWest()    {      ...      }
13
14   protected void visitIntersection()      {      ...      }
15 }
```

**Listing 3-7:** *Using* `protected` *methods in a subclass of* `TraverseAreaRobot`

```
1  public class TraverseWiderAreaRobot extends TraverseAreaRobot
2  { public TraverseWiderAreaRobot(...)      {      ...      }
3
4    protected void traverseOneStreet()
5    { super.traverseOneStreet();        // traverse first 5 intersections
6      this.move();                      // traverse one more
7      this.visitIntersection();
8    }
9  }
```

**Listing 3-8:** *A program that fails to compile because it attempts to use* `private` *and* `protected` *methods*

```
1  public class DoesNotWork
2  { public static void main(String[] args)
3    { ...
4      TraverseAreaRobot karel = new TraverseAreaRobot(...);
5      ...
6      karel.traverseArea();             // works—method is public
7      karel.traverseTwoStreets();       // compile error
8                                        // traverseTwoStreets is private
9      karel.visitIntersection();        // compile error
10                                       // visitIntersection is protected
11   }
12 }
```

It is also possible to omit the access modifier. The result is called "package" access. It restricts the use of the method to classes in the same package. The `becker.robots` package sometimes uses package access to make services available within all classes in the package that should not be available to students. For example, `Robot` actually has a `turnRight` method (contrary to what you read in Section 1.2.3), but it has package access, so most clients can't use it. `RobotSE`, however, is in the same package and thus has access to it. It makes `turnRight` publicly available with the following method, which overrides `turnRight` with a less restrictive access modifier.

```
public void turnRight()
{ super.turnRight();
}
```

Students should not need to use package access.

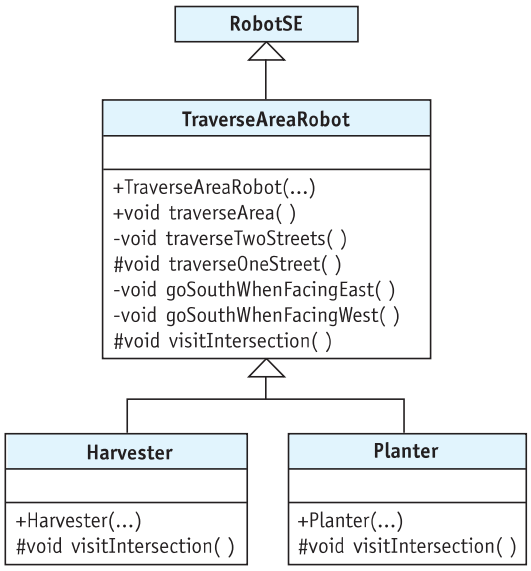
As a rule of thumb, beginning programmers should declare methods as `private` except in the following cases:

- The method is specifically designed to be a public service. In this case, you should declare it as `public`.
- The method is used only by a subclass. In this case, you should declare it as `protected`.

**KEY IDEA**

*Declare methods to be `private` unless you have a specific reason to do otherwise.*

Access modifiers are often shown in class diagrams with the symbols +, #, and –. They stand for `public`, `protected`, and `private` access, respectively. Figure 3-13 shows a class diagram for the `Harvester` class that includes these symbols.




**(figure 3-13)**

*Showing the accessibility of the helper methods in the `TraverseAreaRobot` class*

## 3.7 GUI: Using Helper Methods

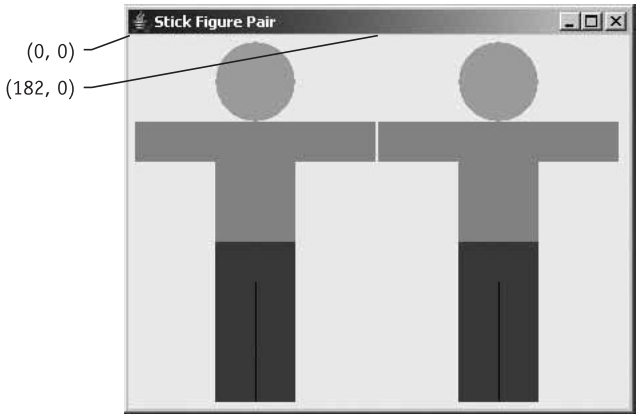
**KEY IDEA**

*Helper methods are a powerful way to help organize a class.*

Stepwise refinement and helper methods are useful in graphics programs, too. For example, consider the pair of stick figures in Figure 3-14. They are based on the stick figure program written in Section 2.7. The `paintComponent` method from that program is reproduced in Listing 3-9, but lines 17 to 29 need to somehow be executed twice to draw both figures. Simply executing the same code twice isn't enough—that would just draw one figure on top of the other. We also need to offset the second figure so that they stand side-by-side.

**(figure 3-14)**

*Pair of stick figures*




**FIND THE CODE**

*ch02/stickFigure/*

**Listing 3-9:** *The code to draw a single stick figure at a predetermined location*

```
12  // Paint a stick figure.
13  public void paintComponent(Graphics g)
14  { super.paintComponent(g);
15
16    // Paint the head.
17    g.setColor(Color.YELLOW);
18    g.fillOval(60, 0, 60, 60);
19
20    // Paint the shirt.
21    g.setColor(Color.RED);
22    g.fillRect(0, 60, 180, 30);
23    g.fillRect(60, 60, 60, 90);
24
25    // Paint the pants.
26    g.setColor(Color.BLUE);
27    g.fillRect(60, 150, 60, 120);
```

**Listing 3-9:** *The code to draw a single stick figure at a predetermined location* (continued)

```
28    g.setColor(Color.BLACK);
29    g.drawLine(90, 180, 90, 270);
30  }
```

One approach is to duplicate lines 17 to 29 inside the `paintComponent` method and adjust the arguments to offset the second figure. A much better approach is to place lines 17 to 29 inside a helper method. The `paintComponent` method calls the method twice to draw the two figures—except that we once again have the problem of offsetting the second figure to stand beside the first one. We could make two helper methods, one for each figure, but they would be almost identical.

The best solution is one helper method that uses parameters to specify the location of the figure. We have already made extensive use of parameters. For example, consider the method calls in lines 17 to 29 of Listing 3-9. They each pass arguments to the method's parameters indicating the location and size of the shape to draw. We will use the same strategy except that instead of drawing a simple oval or rectangle, our method will draw an entire stick figure. We will use parameters only for the location of the stick figure. Using such a helper method, the `paintComponent` method is simplified to the following:

```
1  /** Paint two stick figures
2  *   @param g The graphics context to do the painting. */
3  public void paintComponent(Graphics g)
4  { super.paintComponent(g);
5    this.paintStickFig(g, 0, 0);
6    this.paintStickFig(g, 182, 0);
7  }
```

Line 5 causes a stick figure to be drawn with its upper-left corner placed at (0, 0)—that is, the upper-left corner of the component. Figure 3-14 is annotated with this location. Line 6 causes the second figure to be painted at (182, 0), or 182 pixels from the left and 0 pixels down from the top. This location is also noted in Figure 3-14. The value of 182 was picked because each stick figure is 180 pixels wide, plus two pixels for a tiny gap between them.

Lines 5 and 6 also pass `g`, the `Graphics` object used for painting, as an argument because `paintStickFig` will need it to draw the required shapes.

## 3.7.1 Declaring Parameters

PATTERN

*Parameterized Method*

To use arguments such as `g`, `182`, and `0` inside our helper method, we need to declare corresponding parameters. These should look familiar because we have been declaring parameters in `Robot` constructors since the beginning of Chapter 2. The first line of the `paintStickFig` method should be:

```
private void paintStickFig(Graphics g2, int x, int y)
```

The first part of this line, `private void paintStickFig`, is the same as our Parameterless Command and Helper Method patterns.

**LOOKING BACK**

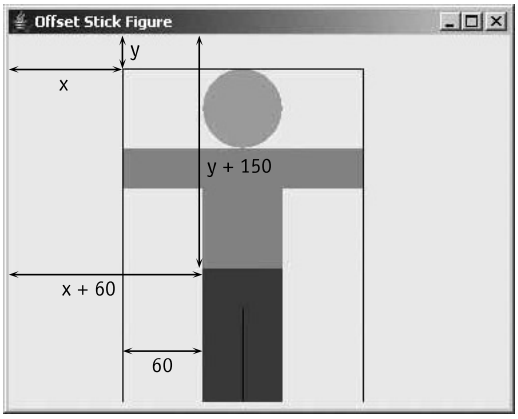
***Type** was defined in Section 1.3.1 as specifying a valid set of values for an attribute. Here it specifies the set of values for the parameter.*

Next come the three parameters. Each specifies a type and a name, and is separated from the next parameter with a comma. `Graphics` is the name of a class and specifies that the first argument to `paintStickFig` must be a reference to a `Graphics` object. This is similar to our `Robot` constructors. There, the first parameter has a type of `City`; consequently, we always pass a `City` object as the first argument. The next two parameters must always be passed integer arguments because they are declared with `int`.

Inside the method, the values passed as arguments will be given the name of the corresponding parameter. If the method is called with `this.paintStickFig(g, 182, 0)`, then inside `paintStickFig`, every time we use the name `x` it will be interpreted as `182`—the value passed to it.

## 3.7.2 Using Parameters

With this background, we can rewrite the method to use the parameters to specify the stick figure's position. Each time we refer to an $x$ or a $y$ location in drawing the stick figure, we add the appropriate `x` or `y` parameter. This action offsets the figure, as shown in Figure 3-15. Adding two numbers together uses the plus sign, and if one of the "numbers" happens to be a parameter, Java will use the number it represents (in this case, the number passed to it as an argument). The revised code for `paintStickFig` appears in Listing 3-10.




(figure 3-15)

*Offsetting the location of the stick figure with the* x *and* y *parameters*

Consider line 36 to paint the rectangle used for the pants. In the original code, we wrote `g.fillRect(60, 150, 60, 120)` to draw a rectangle 60 pixels from the left side and 150 pixels down from the top. The last two arguments specify that it should be 60 pixels wide and 120 pixels high. In line 36, this is changed to `g2.fillRect(x+60, y+150, 60, 120)`. Now the rectangle starts 60 pixels to the right of `x`. If `x` is passed 0, the pants are painted 60 pixels from the left side of the panel. If `x` is passed 182, the pants are painted 242 (182 + 60) pixels from the left side.

**Listing 3-10:** *A component that paints two stick figures, one beside the other*

```
1  import java.awt.*;                  // Graphics, Dimension, Color
2  import javax.swing.*;               // JComponent
3
4  public class StickFigurePair extends JComponent
5  {
6    public StickFigurePair()
7    { super ();
8      Dimension prefSize = new Dimension(2*180+5, 270);
9      this.setPreferredSize(prefSize);
10   }
11
12   /** Paint two stick figures
13   *  @param g The graphics context to do the painting. */
14   public void paintComponent(Graphics g)
15   { super.paintComponent(g);
16     this.paintStickFig(g, 0, 0);
17     this.paintStickFig(g, 182, 0);
18   }
19
20   /** Paint one stick figure at the given location.
```

FIND THE CODE

*ch03/stickFigure/*

**Listing 3-10:** *A component that paints two stick figures, one beside the other* (continued)

```
21     *  @param g2    The graphics context to do the painting.
22     *  @param x     The x coordinate of the upper-left corner of the figure.
23     *  @param y     The y coordinate of the upper-left corner of the figure. */
24    private void paintStickFig(Graphics g2, int x, int y)
25    { // Paint the head.
26      g2.setColor(Color.YELLOW);
27      g2.fillOval(x+60, y+0, 60, 60);
28
29      // Paint the shirt.
30      g2.setColor(Color.RED);
31      g2.fillRect(x+0, y+60, 180, 30);
32      g2.fillRect(x+60, y+60, 60, 90);
33
34      // Paint the pants.
35      g2.setColor(Color.BLUE);
36      g2.fillRect(x+60, y+150, 60, 120);
37      g2.setColor(Color.BLACK);
38      g2.drawLine(x+90, y+180, x+90, y+270);
39    }
40  }
```

Using a helper method helps keep the `paintComponent` method to a reasonable size. By adding parameters to the helper method, we allow the method to be used more flexibly with the result that we only need one helper method instead of two.

## 3.8 Patterns

This chapter introduced four patterns: Helper Method, Multiple Threads, Template Method, and Parameterized Method.

### 3.8.1 The Helper Method Pattern

**Name:** Helper Method

**Context:** You have a long or complex method to implement. You want your code to be easy to develop, test, and modify.