Chapter 5

Component Structure: Method and Class Building

In this chapter, we write classes that contain methods of our own design. Our objectives are

- *to write* public *and* private *methods that accept arguments (*parameters*) and reply with results;*
- *to write classes containing methods that we use over and over;*
- *to read and write* interface specifications*, which describe the behaviors of classes and their methods.*

These techniques will help us write applications whose component structure *consists of classes and methods completely of our own design.*

5.1 Methods

At the beginning of this text, we remarked that an object owns a collection of *methods* for accomplishing work. For example, a program for word processing will have methods for inserting and removing text, saving files, printing documents, and so on. A bank accounting program will have methods for depositing and withdrawing money from a bank account. And, a graphics-window object has methods for setting the window's size, setting its title bar, painting the window, and so on.

The term, "method," comes from real-life examples, such as the plumber (joiner) who has methods for fixing toilets, stopping leaks, and starting furnaces. A plumber's methods might be divided into "public" ones, that is, the methods that customers ask of the plumber, and "private" ones, that is, shortcuts and tricks-of-the-trade the plumber uses to complete a customer's request.

A plumber is a person, an entity, an *object*. In addition to plumbers, the world is full of other entities—carpenters, masons, electricians, painters, and so on. Each of these entities have methods, and the entities work together on major tasks, e.g., building a house.

In a similar way, a program consists of multiple objects that communicate and cooperate by asking one another to execute methods. For example, in the previous chapters, we wrote applications that sent `print` and `println` messages to the preexisting object, `System.out`. And, we constructed new objects, like `new GregorianCalendar()`, from preexisting classes in Java's libraries and sent messages to the new objects. These objects cooperated with our applications to solve problems.

We must learn to write our own classes with methods of our own design. So far, the methods we have written ourselves have been limited to two specific instances: First, we wrote many applications that contained the public method, `main`. When we start an application, an implicit message is sent to execute `main`. Second, we learned to write one form of graphics window, which owns a constructor method and a `paintComponent` method; the coding pattern looked like this:

```
import java.awt.*;
import javax.swing.*;
public class MyOwnPanel extends JPanel
{ public MyOwnPanel()
  { ...  // instructions for initializing field variables,
         //  constructing the panel's frame, and displaying it
  }

  public void paintComponent(Graphics g)
  { ...  // instructions for painting on the panel
  }
}
```

Recall that the constructor method executes when a `new MyOwnPanel()` object is constructed, and the `paintComponent` method executes each time the panel must be painted on the display.

In this chapter, we learn to write methods that are entirely our own doing and we learn how to send messages to the objects that own these methods. In this way, we can advance our programming skills towards writing programs like the word processors and accounting programs mentioned at the beginning of this Section. We begin by learning to write the most widely used form of method, the *public method.*

5.2 Public Methods

Consider this well-used statement:

```
System.out.println("TEXT");
```

The statement sends a `println("TEXT")` message to the object `System.out`, asking `System.out` to execute its method, `println`.

The object that sends the message is called the *client*, and the object that receives the message is the *receiver* (here, `System.out`). The receiver locates its method whose name matches the one in the message and executes that method's statements—we say that the method is *invoked.* (Here, `System.out` locates its `println` method and executes it, causing `TEXT` to appear in the command window; we say that `println` is invoked.)

An application can send `println` messages to `System.out` because `println` is a *public method*, that is, it is available to the "general public."

We have used many such public methods and have written a few ourselves, most notably, `main` and `paintComponent`. Consider again the format of the latter, which we use to paint graphics on a panel:

```
public void paintComponent(Graphics g)  // this is the header line
{ STATEMENTS  // this is the body
}
```

The syntax is worth reviewing: The method's header line contains the keyword, `public`, which indicates the general public may send `paintComponent` messages to the graphics window that possesses this method. (In this case, the "general public" includes the computer's operating system, which sends a `paintComponent` message each time the graphics window must be repainted on the display.) The keyword, `void`, is explained later in this chapter, as is the *formal parameter*, `Graphics g`. The statements within the brackets constitute the method's body, and these statements are executed when the method is invoked.

A public method must be inserted within a class. As we saw in Chapter 4, a class may hold multiple methods, and a class may hold fields as well as methods. When an object is constructed from the class, the object gets the fields and methods for its own. Then, other objects can send messages to the object to execute the methods.

We begin with some basic examples of public methods.

5.2.1 Basic Public Methods

A public method that we write should be capable of one basic "skill" or "behavior" that clients might like to use. Here is an example that illustrates this principle. (To to help us focus on crucial concepts, we dispense with graphics for the moment and work with basic techniques from Chapter 2.)

Pretend that you collect "Ascii art" insects, such as bees, butterflies, and ladybugs ("ladybirds"):

```
  ,-.
  \_/
>{|||}-
  / \
  `-^  hjw

   _ " _
  (_\|/_)
   (/|\)  ejm97

 `m'
 (|)  sahr
```

(Note: the initials next to each image are the image's author's.) From time to time, you might like to display one of the insects in the command window. Each time you do so, you might write a sequence of `System.out.println` statements to print the image, but this is foolish—it is better to write a method that has the ability to print such an image on demand, and then your applications can send messages to this method.

For example, here is the method we write to print the first image, a bee:

```
/** printBee prints a bee */
public void printBee()
{ System.out.println("  ,-.");
  System.out.println("  \\_/");
  System.out.println(">{|||}-");
  System.out.println("  / \\");
  System.out.println("  `-^  hjw");
  System.out.println();
}
```

We always begin a method with a comment—the method's *specification*—that documents the method's purpose or behavior. The method's header line comes next; it states that the method is public and gives the method's name, which we invent. For the moment, do not worry about the keyword, `void`, or the brackets, `()`; they will be explained later. The method's body contains the instructions for printing the bee.

A class that holds methods for printing the above images appears in Figure 1. The Figure shows the above method plus two others grouped into a class, `AsciiArtWriter`. There is also a fourth method, a *constructor method*, which we discuss momentarily.

When we compare `class AsciiArtWriter` to the graphics-window classes from Chapter 4, we see a similar format: we see a constructor method followed by public methods. And, we construct objects from the class in a similar fashion as well:

```
AsciiArtWriter writer = new AsciiArtWriter();
```

This statement constructs an object named `writer` that holds the three public methods. When the object is constructed, the constructor method is invoked, causing an empty line to print in the command window.

Think of `writer` as an object, like `System.out`. This means we can send messages to it:

```
writer.printBee();
```

Notice the syntax of the invocation: The format is the object's name, followed by the method name, followed by the parentheses. The parentheses help the Java compiler understand that the statement is indeed a method invocation—this is why the parentheses are required.

The above format is the usual way of sending a message to an object, and it is a bit unfortunate that the `paintComponent` methods we wrote in Chapter 4 were not invoked in this usual way—`paintComponent` is a special case, because it is normally invoked by the computer's operating system when a window must be repainted on the display.

Figure 2 shows an application that sends messages to an `AsciiArtWriter` object to print two bees and one butterfly.

The advantage of writing `class AsciiArtWriter` *is that we no longer need to remember the details of printing the Ascii images—they are saved in the class's methods. And, we can reuse the class over and over with as many applications as we like.* This is a primary motivation for writing methods and grouping them into a class. `class AsciiArtWriter` is a pleasant addition to our "library" of program components.

Exercises

1. Say that we alter the class in Figure 2 to look like this:

```
public class DrawArt2
{ public static void main(String[] args)
  { AsciiArtWriter w = new AsciiArtWriter();
    w.printButterfly();
```

Figure 5.1: class that draws Ascii art

```
/** AsciiArtWriter contains methods for drawing Ascii art  */
public class AsciiArtWriter
{ /** Constructor AsciiArtWriter does an ‘‘initialization.’’ */
  public AsciiArtWriter()
  { System.out.println(); }

  /** printBee prints a bee */
  public void printBee()
  { System.out.println("  ,-.");
    System.out.println("  \\_/");  // the \ must be written as \\
    System.out.println(">{|||}-");
    System.out.println("  / \\");
    System.out.println("  ‘-^  hjw");
    System.out.println();
  }

  /** printButterfly prints a butterfly */
  public void printButterfly()
  { System.out.println("   _ \"");  // the " must be written as \"
    System.out.println("  (_\\|/_)");
    System.out.println("   (/|\\)  ejm97");
    System.out.println();
  }

  /** printLadybug prints a ladybug */
  public void printLadybug()
  { System.out.println(" ‘m\’");  // the ’ must be written as \’
    System.out.println(" (|)  sahr");
    System.out.println();
  }
}
```

Figure 5.2: application that prints Ascii art

```
/** DrawArt prints some Ascii art and a message */
public class DrawArt
{ public static void main(String[] args)
  { AsciiArtWriter writer = new AsciiArtWriter();
    writer.printBee();
    System.out.println("This is a test.");
    writer.printButterfly();
    writer.printBee();
  }
}
```

```
    w.printButterfly();
  }
}
```

What is printed on the display?

2. Explain what appears in the command window when this application is executed. How many `AsciiArtWriter` objects are constructed?

```
public class TestArt
{ public static void main(String[] args)
  { AsciiArtWriter writer = new AsciiArtWriter();
    writer.printBee();
    new AsciiArtWriter().printButterfly();
    writer.printLadyBug();
  }
}
```

3. Here is a class with a public method:

```
import javax.swing.*;
public class HelperClass
{ public HelperClass()
  { }  // nothing to initialize

  /** computeSquareRoot reads an input integer and displays its square root. */
  public void computeSquareRoot()
  { String s = JOptionPane.showInputDialog("Type a number:");
    double d = new Double(s).doubleValue();
    double root = Math.sqrt(d);
    JOptionPane.showMessageDialog(null,
                     "The square root of " + d + " is " + root);
  }
}
```

Write an application whose `main` method invokes the method to help its user compute two square roots.

4. Write the missing method for this application:

```
import javax.swing.*;
/** NameLength calculates the length of two names.
  *  Input: two names, each typed into an input dialog
  *  Output: dialogs that display the names and their lengths.  */
public class NameLength
{ public static void main(String[] args)
  { HelperClass c = new HelperClass();
    c.readNameAndDisplayItsLength();
    c.readNameAndDisplayItsLength();
    JOptionPane.showMessageDialog(null, "Finished!");
  }

public class HelperClass
{ ...
  /** readNameAndDisplayItsLength  reads one name and displays the
    *  name with its length
    *  Hint: for a string,  x,  x.length()  returns  x's length  */
  ...
}
```

5.2.2 Constructor Methods

A constructor method is a special case of a public method. When we construct an object from a class, e.g.,

```
AsciiArtWriter writer = new AsciiArtWriter();
```

the object is constructed in computer storage and the class's constructor method is immediately invoked. For this reason, you should read the phrase, `new AsciiArtWriter()` in Figure 2, as both constructing a new object *and* sending a message to the method named `AsciiArtWriter()`.

`AsciiArtWriter`'s constructor does little, but as we saw repeatedly in Chapter 4, a constructor method is often used to "complete" the construction of an object: The constructor methods for all graphics windows in Chapter 4 contained statements that set the windows' sizes, background colors, framing, and visibilities. For example, recall Figure 18, Chapter 4, which constructs a graphics window that displays a count of the window's paintings:

```
import java.awt.*;
import javax.swing.*;
/** FieldExample displays how often a window is painted on the display */
public class FieldExample extends JPanel
{ private int count;  // this field variable holds the count of how
                      // often the window has been painted.

  /** FieldExample constructs the window. */
  public FieldExample()
  { count = 0;  // the window has never been painted
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    int height = 200;
    my_frame.setSize((3 * height)/2, height);
    my_frame.setVisible(true);
  }
  ... // See Figure 18, Chapter 4, for the rest of the class
}
```

The `FieldExample`'s constructor initializes `count` to zero, then frames, sizes, and displays the panel. Without the constructor to complete the `FieldExample` object's construction, the object would be useless. Review Chapter 4 to see again and again where constructor methods are used this way.

Because a constructor method is invoked when an object is constructed, it is usually labelled as a public method. A constructor method *must* have the same name as the class that contains it, and again, this is why a statement like `new AsciiArtWriter()` should be read as both constructing a new object and invoking its constructor method. For reasons explained later in this chapter, the keyword, `void`, never appears in the header line of a constructor method.

If the constructor has nothing to do, we can write it with an empty body:

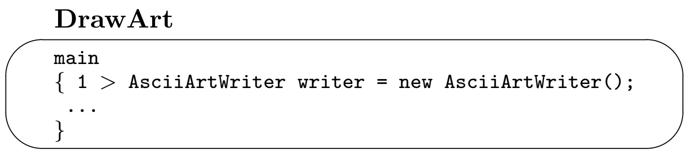
```
public AsciiArtWriter()
{ }
```

When a `new AsciiArtWriter()` object is constructed, the do-nothing constructor method is invoked and does nothing. The Java language allows you to write a class that has *no* constructor method at all, but this style of programming will not be followed in this text.

Execution Trace of Method Invocation

To reinforce our intuitions regarding method invocation, we now study the steps the Java interpreter takes when it executes the application in Figure 2.

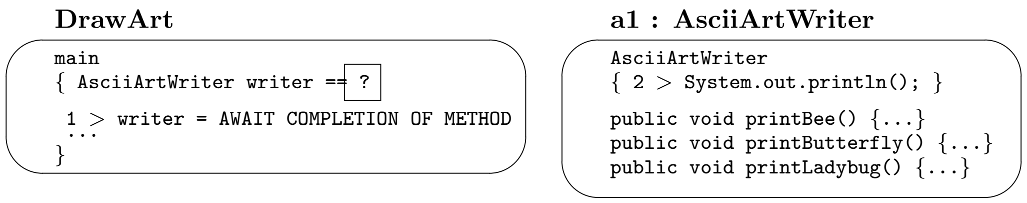
When the application is started, a `DrawArt` object appears in primary storage, and its `main` method is started:



```
DrawArt
main
{ 1 > AsciiArtWriter writer = new AsciiArtWriter();
  ...
}
```

We annotate the execution marker with the numeral, `1`, so that we can see the effects of method invocation.

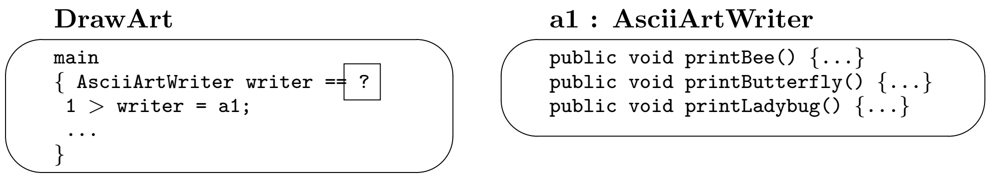
The first statement creates a storage cell named `writer` and starts construction of an `AsciiArtWriter` object:



```
DrawArt                                      a1 : AsciiArtWriter
main                                         AsciiArtWriter
{ AsciiArtWriter writer == ?                 { 2 > System.out.println(); }
 1 > writer = AWAIT COMPLETION OF METHOD     public void printBee() {...}
 ...                                         public void printButterfly() {...}
}                                            public void printLadybug() {...}
```

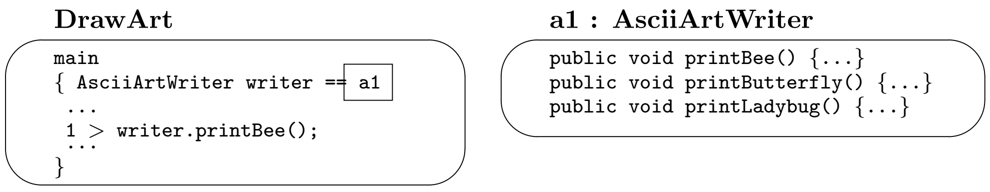
The object is constructed at an address, say `a1`, and its construction method is invoked. A new execution marker, `2>`, shows that execution has been paused at position `1>` and has started within the invoked method at `2>`.

Within the constructor, the `println` invocation executes next. When this statement completes, the constructor method finishes, *the address,* `a1`*, is returned as the result*, and execution resumes at point `1>`:



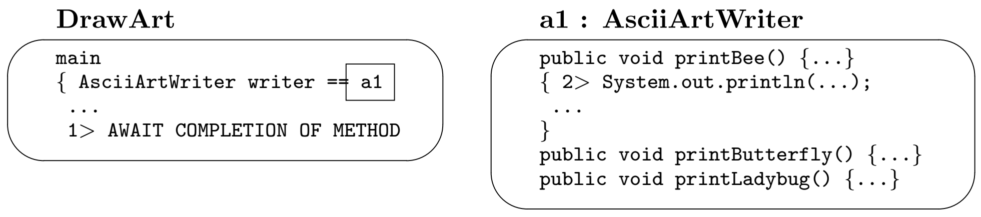
```
DrawArt                              a1 : AsciiArtWriter
main                                 public void printBee() {...}
{ AsciiArtWriter writer == ?         public void printButterfly() {...}
 1 > writer = a1;                    public void printLadybug() {...}
 ...
}
```

Once finished, a constructor method always, automatically returns the address of the object constructed and the method disappears from the object. Notice that `a1` is inserted at the position of the invocation; this lets the assignment complete and the execution proceed to the next statement:



```
DrawArt                              a1 : AsciiArtWriter
main                                 public void printBee() {...}
{ AsciiArtWriter writer == a1        public void printButterfly() {...}
 ...                                 public void printLadybug() {...}
 1 > writer.printBee();
 ...
}
```

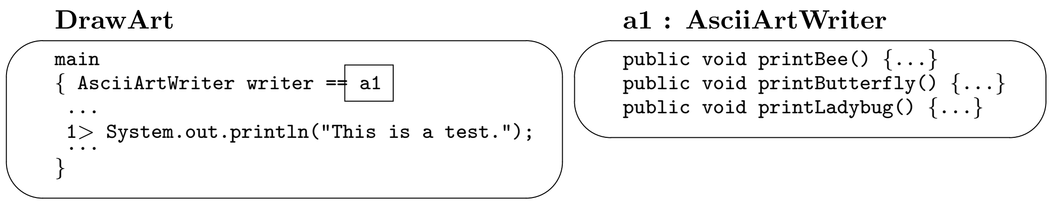
To execute, this invocation, the address of the receiver, `writer`, must be computed; it is `a1`, so the invocation computes to `a1.printBee()`, meaning that execution starts within `printBee` in `a1`:



```
main
{ AsciiArtWriter writer == a1
 ...
 1> AWAIT COMPLETION OF METHOD
```

```
public void printBee() {...}
{ 2> System.out.println(...);
 ...
}
public void printButterfly() {...}
public void printLadybug() {...}
```

The statements in `printBee` execute one by one. Once finished, execution restarts at `1>`, and `printBee` "resets" for future use:



```
main
{ AsciiArtWriter writer == a1
 ...
 1> System.out.println("This is a test.");
 ...
}
```

```
public void printBee() {...}
public void printButterfly() {...}
public void printLadybug() {...}
```

In this way, the messages to `writer` are executed one by one.

Exercises

1. Here is a class with a constructor and public method:

```
public class PrintClass
{ public PrintClass()
  { System.out.println("A"); }

  public void printB()
  { System.out.println("B"); }
}
```

What is printed by this application?

```
public class TestPrintClass
{ public static void main(String[] args)
  { PrintClass p = new PrintClass();
    p.printB();
    p.printB();
    new PrintClass().printB();
    PrintClass q = p;
  }
}
```

2. Here is a class with a constructor and public method:

```
public class Counter
{ private int count;

  public Counter()
  { count = 0; }

  public void count()
  { count = count + 1;
    System.out.println(count);
  }
}
```

What is printed by this application?

```
public class TestCounter
{ public static void main(String[] args)
  { Counter a = new Counter();
    a.count();
    a.count();
    Counter b = new Counter();
    b.count();
    a.count();
    Counter c = a;
    c.count();
    a.count();
  }
}
```

5.3 Parameters to Methods

When an object is sent a message, the message often contains additional information in parentheses, called *arguments*. We have used arguments in messages many times; the standard example is `System.out.println("TEXT")`, which sends the argument, `"TEXT"`, in its message to the `System.out` object.

The technical name for an argument is *actual parameter*, or *parameter*, for short. When an object receives a message that includes actual parameters, the object gives the parameters to the method named in the message, and the method *binds* the parameters to local variables. We can see this in the following example.

Say that we wish to attach names to the bees that we draw with the `AsciiArtWriter`. For example, this bee is named “Lucy”:

Figure 5.3: method with a parameter

```
/** printBeeWithName prints a bee with a name attached to its stinger.
  * @param name - the name attached to the stinger  */
public void printBeeWithName(String name)
{ System.out.println("  ,-.");
  System.out.println("  \\_/");
  System.out.println(">{|||}-" + name + "-");
  System.out.println("  / \\");
  System.out.println("  `-^  hjw");
  System.out.println();
}
```

```
  ,-.
  \_/
>{|||}-Lucy-
  / \
  `-^  hjw
```

To do this, we modify method `printBee` from Figure 1 so that it attaches a name to its bee by means of a parameter. The method's header line changes so that it contains information about the parameter within its brackets:

```
public void printBeeWithName(String name)
```

The information in the parentheses, `String name`, is actually a declaration of a local variable, `name`. The declaration is called a *formal parameter*—it binds to (is assigned the value of) the incoming argument (actual parameter).

To understand this, say that we add the modified method in the Figure to `class AsciiArtWriter`, and we invoke it as follows:

```
AsciiArtWriter writer = new AsciiArtWriter();
writer.printBeeWithName("Lucy");
```

The invocation, `writer.printBeeWithName("Lucy")`, gives the actual parameter, `"Lucy"`, to the method, `printBeeWithName`, which generates this initialization inside of a *new local variable* inside the method's body:

```
String name = "Lucy";
```

Then the variable, `name`, can be used within the body of `printBeeWithName` and it causes `"Lucy"` to be used. Figure 3 shows the modifications.

The main motivation for using parameters is that *a method can be invoked many times with different values of actual parameters and can perform related but distinct computations at each invocation.* For example,

```
writer.printBeeWithName("Fred Mertz");
```

prints a bee similar to Lucy's bee but with a different name.

We can write a method like `printBeeWithName` in advance of its invocations, because the method need not know exactly the name it attaches to the bee it draws, *but it assumes that the formal parameter—a variable—will contain the value when the time comes to draw the bee.*

As part of our documentation policy, the comments preceding the method include a short description of each parameter and its purpose. For reasons explained at the end of the chapter, we begin each parameter's description with the text, `* @param`.

When `printBeeWithName` is invoked, it *must* be invoked with exactly one actual parameter, which *must* compute to a string. This was the case with `writer.printBeeWithName("Lucy")`, but we might also state `writer.printBeeWithName("Fred" + "Mertz")`, whose argument is an expression that computes to a string, or we might use variables in the argument:

```
String s = "Ricardo":
writer.printBeeWithName("Ricky " + s);
```

In each of these cases, the actual parameter is an expression that computes to a string and binds to `printBee`'s formal parameter, `name`.

For example, the last invocation, `writer.printBeeWithName("Ricky " + s)`, causes the actual parameter, `"Ricky " + s`, to compute to `"Ricky " + "Ricardo"`, which computes to `"Ricky Ricardo"`, which binds to `name` within method `printBeeWithName`. In effect, the invocation causes this variant of `printBeeWithName` to execute:

```
{ String name = "Ricky Ricardo";
  System.out.println("  ,-.");
  System.out.println("  \\_/");
  System.out.println(">{|||}-" + name + "-");
  System.out.println("  / \\");
  System.out.println("  `-^  hjw");
}
```

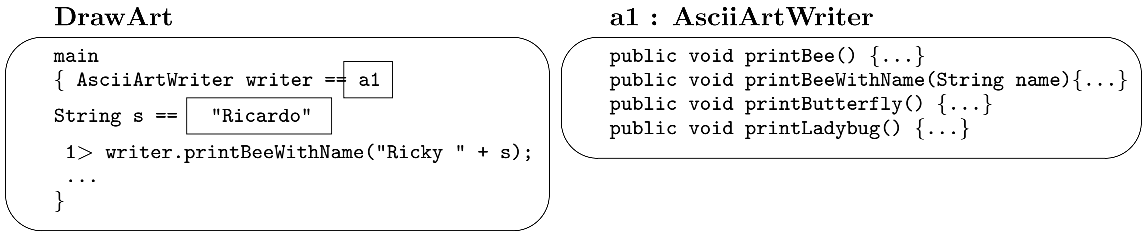
This example emphasizes that

Parameter binding works like initialization of a local variable—the actual parameter initializes the variable (formal parameter), and the local variable is used only within the body of the method where it is declared.

If we would attempt the invocation, `writer.printBeeWithName(3)`, the Java compiler would announce that a data-type error exists because the actual parameter has data type `int`, whereas the formal parameter has data type `String`. If we truly wanted to attach `3` as the name of the bee, then we must convert the integer into a string, say by, `writer.printBeeWithName(3 + "")`, which exploits the behavior of the `+` operator.

Execution Trace

To ensure that we understand the above example, we draw part of its execution trace. Say that an application is about to draw a bee with a name:

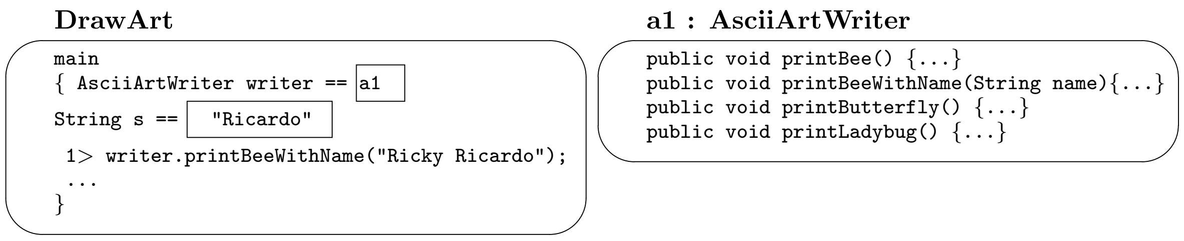


```
DrawArt                                          a1 : AsciiArtWriter
main                                             public void printBee() {...}
{ AsciiArtWriter writer == a1                    public void printBeeWithName(String name){...}
String s == "Ricardo"                            public void printButterfly() {...}
 1> writer.printBeeWithName("Ricky " + s);       public void printLadybug() {...}
 ...
}
```

First, the receiver object is computed; since `writer` holds address `a1`, this is the receiver:

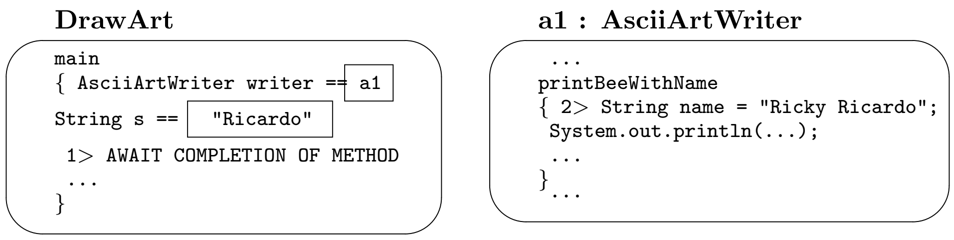
```
a1.printBeeWithName("Ricky " + s);
```

Next, the value of the argument is computed. Since `s` holds the string, `"Ricardo"`, the argument computes to `"Ricky Ricardo"`:



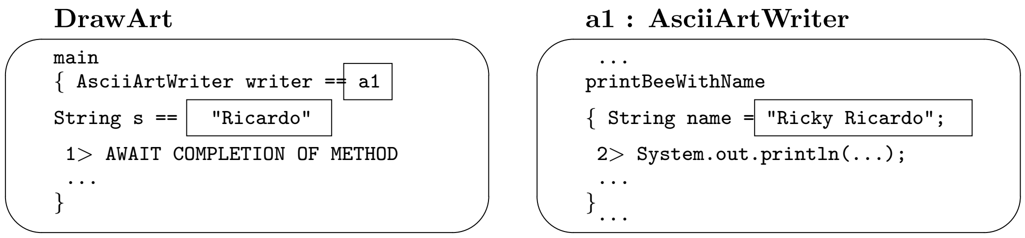
```
DrawArt                                             a1 : AsciiArtWriter
main                                                public void printBee() {...}
{ AsciiArtWriter writer == a1                       public void printBeeWithName(String name){...}
String s == "Ricardo"                               public void printButterfly() {...}
 1> writer.printBeeWithName("Ricky Ricardo");       public void printLadybug() {...}
 ...
}
```

Only after the argument computes to its result does the binding of actual to formal parameter take place. This creates an initialization statement in `printBeeWithName`:



```
DrawArt                               a1 : AsciiArtWriter
main                                   ...
{ AsciiArtWriter writer == a1         printBeeWithName
String s == "Ricardo"                 { 2> String name = "Ricky Ricardo";
 1> AWAIT COMPLETION OF METHOD         System.out.println(...);
 ...                                   ...
}                                     }...
```

The initialization executes as usual,



```
DrawArt                               a1 : AsciiArtWriter
main                                   ...
{ AsciiArtWriter writer == a1         printBeeWithName
String s == "Ricardo"                 { String name = "Ricky Ricardo";
 1> AWAIT COMPLETION OF METHOD         2> System.out.println(...);
 ...                                   ...
}                                     }...
```

and at this point, `printBee`'s body behaves like any other method.

Exercises

1. What will this application print?

```
public class PrintClass
{ public PrintClass() { }

  public void printName(String name)
  { String s = " ";
    System.out.println(name + s + name);
  }
}

public class TestPrintClass
{ public static void main(String[] args)
  { String s = "Fred";
    new PrintClass().printName(s + s);
  }
}
```

2. Write the missing bodies of this class's methods; see the application that follows for help, if necessary.

```
import javax.swing.*;
/** NameClass remembers a name and prints information about it. */
public class NameClass
{ private String name;  // the name that is remembered

  /** Constructor NameClass initializes a NameClass object
    * @param n - the name that is to be remembered  */
  public NameClass(String n)
  { ... }

  /** printName prints the name remembered by this object */
  public void printName()
  { ... }

  /** displayLength prints the integer length of the name remembered.
    * (Hint: use the  length  method from Table 5, Chapter 3.)  */
  public void displayLength()
  { ... }
}

public class TestNameClass
```

```
     { public static void main(String[] args)
       { NameClass my_name = new NameClass("Fred Mertz");
         System.out.print("my name is ");
         my_name.printName();
         System.out.print("my name has this many characters in it: ");
         my_name.displayLength();
       }
     }
```

5.3.1 Forms of Parameters

What can be a parameter? The answer is important:

Any value that can be assigned to a variable can be an actual parameter.

This means numbers, booleans, and even (addresses of) objects can be actual parameters. Of course an actual parameter must be compatible with the formal parameter to which it binds. To understand this, we study a series of examples based on this method for printing an inverse:

```
import java.text.*;  // Note: This statement is inserted at the beginning
                     // of the class in which the following method appears.

/** printInverse prints the inverse,  1.0/i,  of an integer, i,
  * formatted to three decimal places.
  * @param i - the integer that is inverted  */
public void printInverse(int i)
{ DecimalFormat formatter = new DecimalFormat("0.000");  // see Chapter 4
  double d = 1.0 / i;
  String s = formatter.format(d);  // formats  d  with three decimal places
  System.out.println(s);
}
```

Parameter `i` is declared as an integer, so any invocation of `inverseOf` must use an actual parameter that computes to an integer. (The Java compiler verifies this when it checks the data type of the actual parameter in the invocation.) The method computes the fractional inverse of `i` and formats it into three decimal places, using a helper object constructed from `class DecimalFormat`, found in the package `java.text` and introduced in Chapter 4. The formatted result is printed.

Say that we insert `inverseOf` into some new class, say, `class MathOperations`. This lets us do the following:

```
MathOperations calculator = new MathOperations();
calculator.printInverse(3);
```

Figure 5.4: method with multiple parameters

```
import java.text.*;  // Note: This statement is inserted at the beginning
                     // of the class in which the following method appears.

/** printInverse prints the inverse,  1.0/i,  of an integer, i.
  * @param i - the integer that is inverted
  * @param pattern - the pattern for formatting the fractional result */
public void printInverse(int i, String pattern)
{ DecimalFormat formatter = new DecimalFormat(pattern);  // see Chapter 4
  double d = 1.0 / i;
  String s = formatter.format(d);
  System.out.println(s);
}
```

which prints `0.333`. Because of the declaration of its formal parameter, we cannot invoke `printInverse` with doubles, e.g., `calculator.printInverse(0.5)` will be disallowed by the Java compiler—only integers can be actual parameters to the method.

But if `printInverse` had been written with this header line:

```
public void printInverse(double i)
```

Then the invocation,

```
calculator.printInverse(0.5);
```

would be acceptable, because the data type of the actual parameter is compatible with the data type of the formal parameter. Indeed, we could also perform

```
calculator.printInverse(3);
```

because integers can be used in any context where doubles are expected. We see this clearly when we remember that parameter binding is the same as variable initialization; therefore, the latter invocation generates this binding:

```
double i = 3;
```

which is acceptable in Java, because the `3` is converted into `3.0`.

Methods may receive multiple parameters, and we can use this extension to good effect in the above example. Say that we alter `printInverse` so that its client can specify the pattern of precision used to print the inverse. The method now receives two parameters; see Figure 4.

The parameters are stated in the header line, between the brackets, separated by commas. When the method is invoked, there *must* be exactly two actual parameters, and the first *must* be an integer and the second *must* be a string:

```
MathOperations calculator = new MathOperations();
calculator.printInverse(3, "0.00000");  // print with five decimal places
calculator.printInverse(5, "0.0");  // print with one decimal place
```

The important point to remember is

Actual parameters bind to formal parameters in order—the first actual parameter binds to the first formal parameter, the second actual parameter binds to the second formal parameter, and so on.

The particular variable names, if any, that are used in the actual parameters *do not matter*—it is the order in which the actual parameters are listed that determines their bindings to the formal parameters.

For example, the first invocation generates this execution of `printInverse`:

```
{ int i = 3;
  String pattern = "0.00000";
  DecimalFormat formatter = new DecimalFormat(pattern);
  double d = 1.0 / i;
  String s = formatter.format(d);
  System.out.println(s);
}
```

The order of the actual parameters proves crucial when there is a situation like this one,

```
public void printDivision(double n, double d)
{ System.out.println(n / d); }
```

where the data types of the two formal parameters are identical.

Finally, we modify Figure 4 to show that objects can also be parameters:

Begin Footnote: Indeed, since strings are implemented as objects in Java, we know already that objects can be parameters, but the example that follows shows that objects we construct with the `new` keyword can be parameters also. *End Footnote*

```
import java.text.*;  // Note: This statement is inserted at the beginning
                     // of the class in which the following method appears.

/** printInverse prints the inverse,  1.0/i,  of an integer, i.
  * @param i - the integer that is inverted
  * @param f - the object that formats the fractional result */
public void printInverse(int i, DecimalFormat f)
{ double d = 1.0 / i;
  String s = f.format(d);
  System.out.println(s);
}
```

Remember that every class name generates its own data type, hence there is a data type, `DecimalFormat`, and we use it to describe the second formal parameter, which will bind to an object we construct:

```
MathOperations calculator = new MathOperations();
DecimalFormat five_places = new DecimalFormat("0.00000");
calculator.printInverse(3, five_places);
```

The invocation proceeds like the others: The actual parameters compute to their results (in this case, the second parameter computes to the address of a `DecimalFormat` object) and bind to the corresponding formal parameters.

Now that we are acquainted with the various forms of parameters, we can solve a couple of mysteries. First, the header line of the `paintComponent` method one uses for graphics windows reads

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

The `Graphics g` part is a formal parameter, and whenever the operating system sends a message to `paintComponent`, the message will contain an actual parameter that is (the address of) the graphics-pen object that `paintComponent` uses for painting.

Second, method `main`'s header line also requires a parameter:

```
public static void main(String[] args)
```

The formal parameter, `args`, names the collection of program arguments that are submitted when an application starts. As noted in Chapter 3, the arguments are extracted from `args` by the names `args[0]`, `args[1]`, and so on. The data type, `String[]`, is read "string array"—it describes collections of strings. We study array data types in Chapter 8.

We finish our examination of parameters with two final observations:

- Constructor methods may use formal parameters in the same fashion as other public methods.

- A method whose header line has the form,

  ```
  public void METHODNAME()
  ```

  is a method with *zero* formal parameters and must be invoked by a statement that uses *zero* actual parameters within the parentheses—`RECEIVER_OBJECT.METHODNAME()`—this is why an empty bracket set is required with some invocations.

Exercises

1. Here is a helper class:

```
public class ArithmeticClass
{ private int base;

  public ArithmeticClass(int b)
  { base = b; }

  public void printMultiplication(String label, double d)
  { System.out.println(label + (base * d)); }
}
```

   (a) What does this application print? (Draw execution traces if you are uncertain about parameter passing.)

```
public class TestArithmeticClass
{ public static void main(String[] args)
  { ArithmeticClass c = new ArithmeticClass(2);
    c.printMultiplication("3", 4.5 + 1);
    int i = 4;
    c.printMultiplication("A", i);
    c.printMultiplication("A", i - 1);
  }
}
```

   (b) Explain the errors in this application:

```
public class Errors
{ public static void main(String[] args)
  { ArithmeticClass c = new ArithmeticClass();
    printMultiplication("A", 5);
    int s = 0;
    c.printMultiplication(s, 2 + s);
    c.printMultiplication(1, "A");
    c.printMultiplication("A", 9, 10);
    c.printSomething();
  }
}
```

2. Write the missing methods for this class:

```
/** RunningTotal helps a child total a sequence of numbers */
public class RunningTotal
{ private int total;  // the total of the numbers added so far
```

```
  /** Constructor RunningTotal initializes the object */
  public RunningTotal()
  { total = 0; }

  /** addToTotal  adds one more number to the running total
    * @param num - the integer to be added to the total  */
  ...

  /** printTotal  prints the current total of all numbers added so far */
  ...
}

public class AddSomeNumbers
{ public static void main(String[] args)
  { RunningTotal calculator = new RunningTotal();
    calculator.addToTotal(16);
    calculator.addToTotal(8);
    calculator.printTotal();
    calculator.addToTotal(4);
    calculator.printTotal();
  }
}
```

3. The `paintComponent` method of the clock-writing class in Figure 17, Chapter 4, can be rewritten so it uses this method:

```
/** paintClock paints a clock with the time
  * @param hours - the current hours, an integer between 1 and 12
  * @param minutes - the current minutes, an integer between 0 and 59
  * @param x - the upper left corner where the clock should appear
  * @param y - the upper right corner where the clock should appear
  * @param diameter - the clock's diameter
  * @param g - the graphics pen used to paint the clock  */
public void paintClock(int hours, int minutes, int x, int y,
                       int diameter, Graphics g)
```

   Write this method, insert it into `class ClockWriter` in Figure 17, Chapter 4, and rewrite `ClockWriter`'s `paintComponent` method to invoke it.

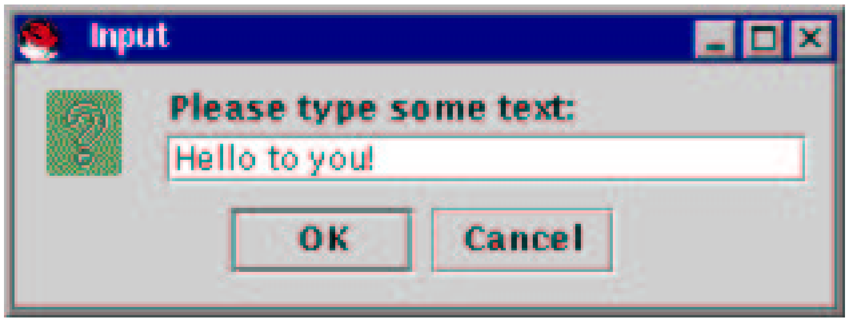
   Next, make `paintComponent` draw *two* clocks—one for your time and one for the current time in Paris.

   Note: If you find this exercise too demanding, read the next section and return to rework the exercise.

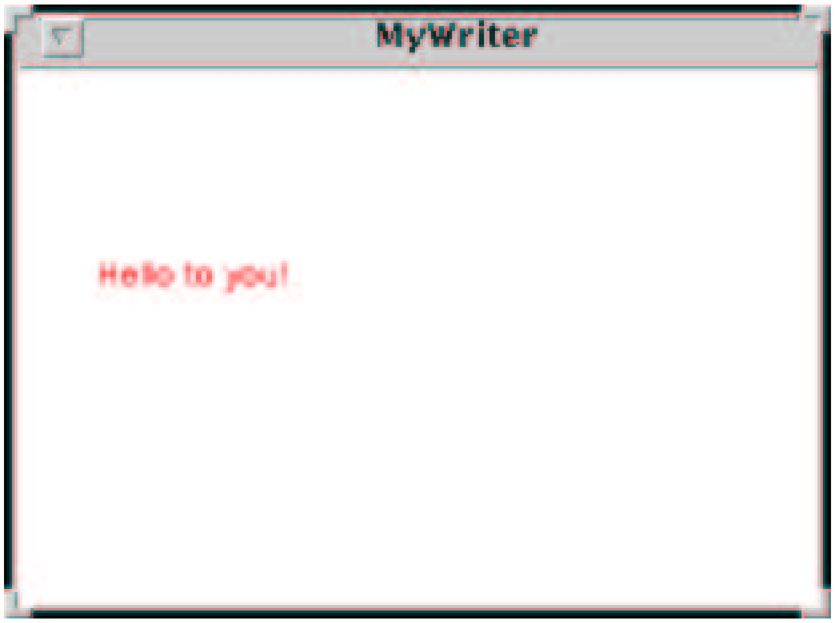
5.4 Case Study: General-Purpose Output Frame

In practice, we design public methods when we are designing a class: We consider what the responsibilities (behaviors) of the class might be, and we design public methods for each of the expected behaviors. We write the class with the necessary private fields and constructor methods so that the public methods operate properly.

A major component of our applications has been the "output view"—the part that displays the computation results. We can reduce our dependency on `System.out` as our output-view object if we design our own graphics window to display textual output. Perhaps we write an application that asks its user to type input text:



The input is fetched and displayed in our new graphics window, e.g.,



Of course, we can use a `JOptionPane`-generated dialog for the input-view, but we must design and write the output-view that has the ability to display a textual string. Keeping this simple, we design the graphics window so that it displays exactly one line of text and we can state where to position the text. (In the Exercises at the end of this section, we make the window more versatile.)

We follow these steps when we design a new class:

1. *List the class's methods (its "responsibilities" or "behaviors"), and for each method, give an informal description that states the method's behavior and the arguments the method requires to execute. This includes the constructor method as well.*

2. *List the private fields ("attributes") that will be shared by the the methods.*

3. *Write the methods' bodies so that they have the listed behaviors.*

All three items above are crucial to the person who *writes* the class; Item 1 is important to those who *use* the class.

Let's design the output frame; what methods should it have? Perhaps we decide on two: *(i)* we can send a message to the frame to print a sentence; *(ii)*we can tell the frame where it should print the sentence. Perhaps we can the first method, `writeSentence`; obviously, the method will require an argument — the sentence to be printed. The second method — call it, `positionSentence` — will require the x- and y-coordinates that state where the sentence should be printed on the frame.

Also, we will require a constructor method that constructs the frame with some fixed width and height.

Now, for the attributes: The initial descriptions of the methods suggest that the frame must have at private fields that remember the current sentence to display and the x- and y-coordinates of where to print it. Table 5 summarizes what we have developed so far.

The new class is named `MyWriter`, and the table is called its *interface specification* or *specification*, for short. The specification indicates the methods we must write and it also states the ways that others can use the methods we write, so we will retain the specification as useful documentation after the class is built. You might compare Table 5 to the ones that summarized the methods for `class Graphics` (Table 16, Chapter 4) and `JFrame` (Table 20, Chapter 4).

The notion of "specification" comes from real-life: For example, when you buy a tire for your car, you must know the correct size of tire—the size is the tire's specification. Your waist and inseam measurement is another example of an specification that you use when you sew or purchase a new pair of pants. Specifications help you construct and use objects in real life, and the same is true in computer programming.

Given the specification in Table 5, how do we write its methods? Since `MyWriter` is a "customized" graphics window, we can follow the techniques from Chapter 4. This suggests that we write a class whose `paintComponent` method paints the sentence on the window. How will `writeSentence` ask `paintComponent` do its work?

The solution is to declare the private field,

Figure 5.5: specification of an output frame

`MyWriter` creates a graphics window that displays a sentence.	
Constructor:	
`MyWriter(int w, int h)`	construct the window with the width of `w` pixels and the height of `h` pixels and display it.
Methods:	
`writeSentence(String s)`	display sentence `s` in the graphics window
`repositionSentence(int new_x, int new_y)`	redisplay the existing sentence at the new position, `new_x, new_y`
Private attributes:	
`sentence`	the sentence that is displayed
`x_position`	the x-coordinate of where the sentence will appear
`y_position`	the y-coordinate of where the sentence will appear

```
private String sentence;  // holds the sentence to be displayed
```

and have `paintComponent` display the field's contents:

```
public void paintComponent(Graphics g)
{ ...
  g.drawString(sentence, ... );
}
```

Now `writeSentence(String s)` has this algorithm:

1. assign `sentence = s`;
2. make `paintComponent` execute.

We do Step 2 by invoking a prewritten method already contained within `class JPanel`: It is called `repaint`, and it invokes `paintComponent` with the panel's the graphics pen object as the actual parameter, just like the computer's operating system does whenever a panel must be repainted. The invocation of `repaint` forces the panel to repaint, even if it is not iconified or moved or covered. The coding reads:

```
public void writeSentence(String s)
  { sentence = s;
    this.repaint();  // indirectly forces  paintComponent  to execute
  }
```

Because `repaint`'s coding lives within `class JPanel` and because `class MyWriter extends JPanel`, it means that every `MyWriter` object will have its own `repaint` method. To invoke the method within `MyWriter`, we write `this.repaint()`, because the receiver of the invocation is `this` object—the client object that sends the message is also the receiver!

The completed `class MyWriter` appears in Figure 6. The class uses a number of private fields, which remember the window's size, the position for displaying the sentence, and the sentence itself (which is initialized to an empty string). The constructor method uses two parameters to size the window.

The `paintComponent` method operates as expected, and `writeSentence` deposits the string to be displayed in field `sentence` and forces this object to `repaint`. Finally, `positionSentence` resets the fields that position the sentence and forces the sentence to be redisplayed. We use the invocation,

```
this.writeSentence(sentence);
```

to reinforce that an object can send messages to its own public methods.

Begin Footnote: The Java language allows the `this` pronoun to be omitted from invocations of an object's own methods, e.g., `writeSentence(sentence)` can be used in the previous example. *End Footnote*

Note also that the value of a field can be an argument in an invocation. Finally, `positionSentence` could also be written as

```
public void positionSentence(int new_x, int new_y)
{ x_position = new_x;
  y_position = new_y;
  this.repaint();
}
```

which has the same behavior.

Here is a small application that uses `class MyWriter` to interact with its user:

```
import javax.swing.*;
/** MyExample2 displays, in a graphics window, a sentence its user types */
public class MyExample2
{ public static void main(String[] args)
  { int width = 300;
    int height = 200;
    MyWriter writer = new MyWriter(width,height);
    writer.positionSentence(50, 80);  // set position to my liking
    String s = JOptionPane.showInputDialog("Please type some text:");
    writer.writeSentence(s);  // display  s
  }
}
```

When the application starts, the graphics window appears with an empty sentence, which is immediately repositioned to location, 50, 80. Then the interaction displayed at the beginning of this section occurs.

Here are the lessons learned from this case study:

Figure 5.6: graphics window for displaying text

```
import java.awt.*;  import javax.swing.*;
/** MyWriter creates a graphics window that displays a sentence */
public class MyWriter extends JPanel
{ private int width;  // the frame's width
  private int height; // the frame's height
  private String sentence = ""; // holds the sentence to be displayed
  private int x_position;  // x-position of sentence
  private int y_position;  // y-position of sentence

  /** Constructor MyWriter creates the Panel
    * @param w - the window's width
    * @param h - the window's height  */
  public MyWriter(int w, int h)
  { width = w;
    height = h;
    x_position = width / 5;  // set the sentence's position
    y_position = height / 2;
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("MyWriter");
    my_frame.setSize(width, height);
    my_frame.setVisible(true);
  }

  /** paintComponent paints the panel
    * @param g - the ``graphics pen'' that draws the items */
  public void paintComponent(Graphics g)
  { g.setColor(Color.red);
    g.drawString(sentence, x_position, y_position);
  }

  /** writeSentence displays a new string in the window
    * @param s - the sentence to be displayed  */
  public void writeSentence(String s)
  { sentence = s;
    this.repaint();  // indirectly forces  paintComponent  to execute
  }

  /** positionSentence redisplays the existing sentence in a new position
    * @param new_x - the new horizontal starting position
    * @param new_y - the new vertical starting position  */
  public void positionSentence(int new_x, int new_y)
  { x_position = new_x;
    y_position = new_y;
    this.writeSentence(sentence);  // force a rewrite of the existing sentence
  }
}
```

1. We designed and wrote a class that can be used as a component of many applications.
2. We designed a specification that helped us write the class.
3. The users of the class can read the specification to understand how to use the class's methods; *there is no need to read the class's coding.*
4. The coding used a constructor method and private fields to help the public methods behave correctly; the public methods invoked each other as needed.

Item 3 of the above list is so crucial that you should always write specifications, even "after the fact," for the classes you build. Indeed, there should be a close, if not exact, match between the specification and the Java comments you attach to the class and its methods, so the specifications really generate no additional effort.

In the next Chapter, we will appreciate two more benefits of designing and writing classes like `MyWriter`:

- When an application is built by several people, it is easier for distinct people to write, and test the pieces of the application if the application is divided into classes.
- If part an application must be rewritten or replaced, it is easier to do so if the part is one separate class.

Exercises

1. Explain the behavior of this application that uses `class MyWriter`:

```
import javax.swing.*;
public class AnotherExample
{ public static void main(String[] args)
  { MyWriter writer = new MyWriter(300, 200);
    String s = JOptionPane.showInputDialog("Please type some text:");
    writer.writeSentence(s);
    s = JOptionPane.showInputDialog("Try it again:");
    writer.writeSentence(s);
    writer.repositionSentence(0, 190);
    writer.writeSentence(s + s);
  }
}
```

2. Write an application that asks the user to type an integer, computes the square root of that integer, and uses `class MyWriter` to display the integer and its square root, the latter displayed to a precision of 6 decimal places.

3. Add this new method to `class MyWriter`:

   ```
   /** writeSecondSentence displays a sentence,  t,  underneath the first
     * sentence in the window.
     * @param t - the second sentence to be displayed  */
   public void writeSecondSentence(String t)
   ```

   When you add this method, must you revise `writeSentence`? `repositionSentence`?

4. Write a class that satisfies this specification:

`TextWriter` displays up to three lines of text in a graphics window	
Constructor:	
`TextWriter(int w, int h)`	Constructs the window with width `w` pixels and height `h` pixels and displays a window with three empty lines of text. Methods:
`print1(String s)`	Appends `s` to the end of the Line 1 text and displays the current values of all three lines of text.
`reset1(String s)`	Sets Line 1 of the text to `s` and displays the current values of all three lines of text.
`print2(String s)`	Appends `s` to the end of the Line 2 text and displays the current values of all three lines of text.
`reset2(String s)`	Sets Line 2 of the text to `s` and displays the current values of all three lines of text.
`print3(String s)`	Appends `s` to the end of the Line 3 text and displays the current values of all three lines of text.
`reset3(String s)`	Sets Line 3 of the text to `s` and displays the current values of all three lines of text.

Test the class with this application:

```
import javax.swing.*;
public class TestTextWriter
{ public static void main(String[] args)
  { TextWriter writer = new TextWriter(300, 200);
    String s = JOptionPane.showInputDialog("Please type some text:");
```

```
      writer.print1(s);
      s = JOptionPane.showInputDialog("Try it again:");
      writer.print1(s);
      s = JOptionPane.showInputDialog("Once more:");
      writer.print3(s);
      s = JOptionPane.showInputDialog("Last time:");
      writer.reset1(s);
    }
  }
```

Next, rewrite Figure 2, Chapter 4, to use a `TextWriter` object to display its output.

5.5 Results from Methods: Functions

When a client object sends a message, sometimes the client expects an answer in reply. We saw this in Figure 2, Chapter 4, where the controller asked the input-view to read an input string:

```
String input =
      JOptionPane.showInputDialog("Type an integer Celsius temperature:");
```

When `JOptionPane`'s `showInputDialog` method executes, it fetches the string the user types, and the string is the "reply," which is deposited into the position where the `readLine` message appears. Then, the string is assigned to variable `input`.

Yet another example appears in the same figure,

```
int c = new Integer(input).intValue();
```

where the object, `new Integer(input)`, is sent the message, `intValue()`, and replies with the integer that is assigned to `c`.

"Replies" from methods are called *results*. The idea comes from mathematics, where one speaks of the result from calculating a formula. Indeed, we can take the well-used temperature conversion formula, `f = (9.0 / 5.0) * c + 32`, and encode it as a method that returns a result; see Figure 7. The method illustrates the technique for returning a result:

- In the method's header line, replace the keyword, `void`, by the data type of the result the method should return. (Here, it is `double`.) We now note that all the previous uses of the term, `void`, meant "returns no result."

- At the end of the method's body, insert a `return E` statement, where `E` computes to the value that will be returned. `E` can be any expression whose data type matches the one listed in the header line as the result's data type.

  Indeed, the temperature-conversion method can be written just this tersely:

Figure 5.7: temperature conversion function

```
/** celsiusIntoFahrenheit translates degrees Celsius into Fahrenheit
  * @param c - the degrees in Celsius
  * @return the equivalent degrees in Fahrenheit */
public double celsiusIntoFahrenheit(double c)
{ double f = ((9.0 / 5.0) * c) + 32;
  return f;
}
```

```
public double celsiusIntoFahrenheit(double c)
{ return ((9.0 / 5.0) * c) + 32; }
```

- As part of our documentation policy, we include the line,

  ```
  * @return the equivalent degrees in Fahrenheit
  ```

  in the comments at the head of the method to describe the result computed.

A method that returns a result is sometimes called a *function.* A function's commentary includes a line labelled `@return`, which explains the nature of the result returned by the function.

If the `celsiusIntoFahrenheit` was included in some class, say, `class TemperatureConvertor`, then we might use it to convert temperatures as follows:

```
import javax.swing.*;
/** ConvertATemperature converts one Celsius temperature into Fahrenheit */
public class ConvertATemperature
{ public static void main(String[] args)
  { String input =
      JOptionPane.showInputDialog("Type an integer Celsius temperature:");
    int c = new Integer(input).intValue();  // convert  input  into an int

    TemperatureConvertor convert = new TemperatureConvertor();
    double f = convert.celsiusIntoFahrenheit(c);

    MyWriter writer = new MyWriter(300, 200);
    writer.writeSentence(c + " Celsius is " + f + " Fahrenheit");
  }
}
```

The two statements in the middle of the `main` method can be compressed into one, if desired:

```
double f = new TemperatureConvertor().celsiusIntoFahrenheit(c);
```

With a bit of work, we might collect together other conversion formulas for temperatures and save them in `class TemperatureConvertor`—see the Exercises that follow. As always, by writing functions and collecting them in classes, we can save and reuse the formulas in many applications. Indeed, in the Java package, `java.lang`, one finds `class Math`, which is exactly such a collection of commonly used mathematical functions. In addition, the next Chapter shows other crucial uses of functions.

What forms of results can a function return? The answer is: *any value that has a data type can be the result of a function*—use the data-type name in the function's header line, and use the value in the `return` statement. Therefore, we have the same freedom as we have with parameters when we define results of functions—primitive values as well as objects can be function results.

The Java compiler will let an application "discard" the result of a function, e.g.,

```
import javax.swing.*;
public class IgnoreConversion
{ public static void main(String[] args)
  { String input =
       JOptionPane.showInputDialog("Type an integer Celsius temperature:");
    int c = new Integer(input).intValue();
    TemperatureConvertor convert = new TemperatureConvertor();
    convert.celsiusIntoFahrenheit(c);
    System.out.println("The End");
  }
}
```

The penultimate statement invokes the `celsiusIntoFahrenheit` function, the function computes and returns a result, but the result is ignored and execution proceeds to the final statement.

This technique is more commonly used with a constructor method, which is a "function" that automatically returns the address of the constructed object. For example, `class CelsiusToFahrenheitFrame` in Figure 14, Chapter 4, is self-contained, so its start-up application merely reads:

```
public class Convert
{ public static void main(String[] args)
  { new CelsiusToFahrenheitFrame(); }
}
```

The sole statement in `main` invokes the constructor method and ignores the returned address. Indeed, remember that a constructor method must always return the address of the object constructed, and for this reason a constructor can never return any other value as its result—you will never see a return-data-type listed in the header line of a constructor method.

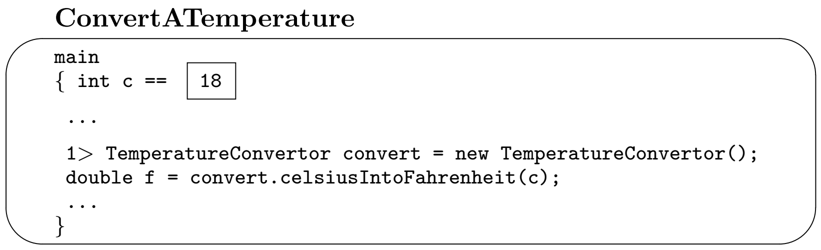
Finally, remember that it is good programming policy to make the `return` statement the final statement of a function. For example, the Java compiler will allow the following function to compiler and execute:

```
/** square incorrectly computes a number's square
  * @param num - the number
  * @return an incorrect compution of  num * num  */
public int square(int num)
{ int result = 0;
  return result;
  result = num * num;
}
```

Because execution proceeds from the first statement forwards, the badly placed `return` statement forces the function to quit prematurely and return the current value of `result`, which is 0—the third statement never executes.

Execution Trace

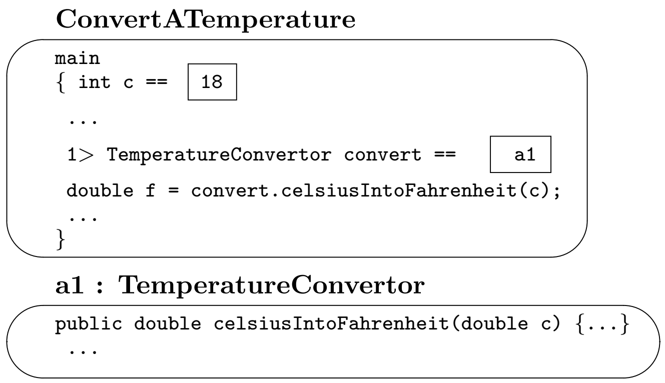
We finish the section with a few steps of the execution trace of the example that uses the temperature-conversion function in Figure 7. Say that the `main` method has reached this point in its execution:



ConvertATemperature

```
main
{ int c == [18]
 ...
 1> TemperatureConvertor convert = new TemperatureConvertor();
 double f = convert.celsiusIntoFahrenheit(c);
 ...
}
```

As seen before, the initialization constructs the object that holds the function:



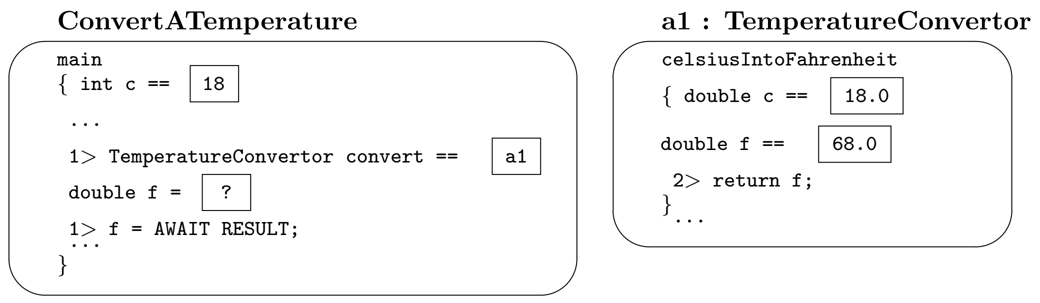
ConvertATemperature

```
main
{ int c == [18]
 ...
 1> TemperatureConvertor convert ==  [a1]
 double f = convert.celsiusIntoFahrenheit(c);
 ...
}
```

a1 : TemperatureConvertor

```
public double celsiusIntoFahrenheit(double c) {...}
 ...
```

The invocation binds the value of the actual parameter to the formal parameter, as usual, and the message to the object at `a1` starts execution of the function, which proceeds to its result:

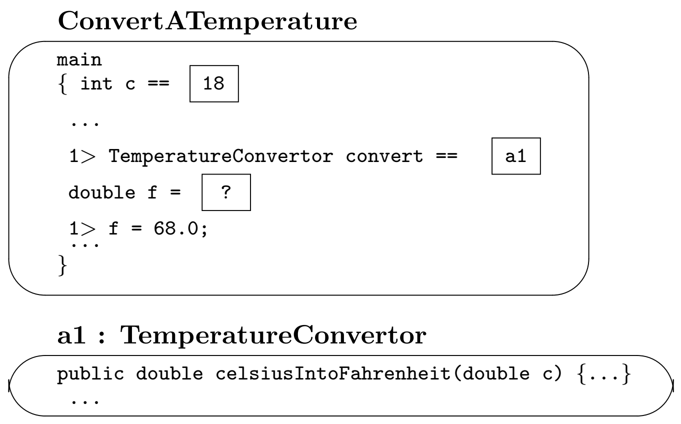


```
main
{ int c == 18
 ...
 1> TemperatureConvertor convert == a1
 double f = ?
 1> f = AWAIT RESULT;
 ...
}
```

```
celsiusIntoFahrenheit
{ double c == 18.0
double f == 68.0
 2> return f;
}
 ...
```

Notice that the variables, `c` and `f`, local to `celsiusIntoFahrenheit`, are *unrelated* to the local variables in `main`—no problems are introduced, no inadvertent connections are made, merely because the two program components chose similar names for their local variables.

Finally, the `return f` statement computes `68.0` and inserts it into the position where the invocation appeared. The function resets for later invocations:



```
main
{ int c == 18
 ...
 1> TemperatureConvertor convert == a1
 double f = ?
 1> f = 68.0;
 ...
}
```

```
public double celsiusIntoFahrenheit(double c) {...}
 ...
```

Because function invocation operates the same way as ordinary method invocation, the execution trace reinforces the intuition that a method whose "result type" is `void` "returns" no result at all.

Exercises

1. What does this application print?

```
public class Counter
{ private int count;

  public Counter(int i)
  { count = i; }
  public int countA()
  { count = count + 1;
    return count;
  }

  public double countB()
  { return count + 1.5; }

  public String countC()
  { return "*" + count; }
}

public class TestCounter
{ public static void main(String[] args)
  { Counter c = new Counter(3);
    int x = c.countA();
    System.out.println(x + " " + c.countB());
    System.out.println(c.countC() + c.countA());
  }
}
```

2. Place the function, `celsiusIntoFahrenheit`, seen in this section into a class, `class TemperatureConvertor`. Add this method to the class:

```
/** fahrenheitIntoCelsius translates degrees Fahrenheit into Celsius
  * @param f - the degrees in Fahrenheit, a double
  * @return the equivalent degrees in Celsius, a double */
```

(Hint: use algebra to compute the conversion formula.) Next, text the class with this application:

```
public class ConvertTemps
{ public static void main(String[] args)
  { TemperatureConvertor calculator = new TemperatureConvertor();
    int temp = new Integer(args[0]).intValue();  // get command-line input
    double ftemp = calculator.celsiusIntoFahrenheit(temp);
    System.out.println(temp + "C is " + ftemp + "F");
    System.out.println("Verify: " + ftemp + "F is "
                       + calculator.fahrenheitIntoCelsius(ftemp) + "C");
    double ctemp = calculator.fahrenheitIntoCelsius(temp);
    System.out.println(temp + "F is " + ctemp + "C");
    System.out.println("Verify: " + ctemp + "C is "
                       + calculator.celsiusIntoFahrenheit(ctemp) + "F");
  }
}
```

3. Write functions that match each of these specifications:

   (a)
   ```
   /** kilometersToMiles converts a kilometers amount into miles
     * using the formula:    Miles = 0.62137 * Kilometers
     * @param k - the kilometers amount
     * @return the corresponding miles amount  */
   public double kilometersToMiles(double k)
   ```
   Insert the function into an application that reads a kilometers value from the user and prints the value in miles.

   (b)
   ```
   /** compoundTotalOf computes the compounded total
     * that result from a starting principal, p,  an interest rate, i,
     * and a duration of n compounding periods, using this formula:
     *  total = p((1 + (i/n))<sup>n</sup>)
     * @param p - the starting principal, a dollars, cents, amount
     * @param i - the interest rate per compounding period,
     *   a fraction (e.g., 0.01  is 1%)
     * @param n - the compounding periods (e.g., if compounding is done
     *   monthly, then two years is 24 compounding periods)
     * @return the total of principal plus compounded interest  */
   public double compoundTotalOf(double p, double i, int n)
   ```

   (c)
   ```
   /** isDivisibleByNine checks if its argument is divisible by 9 with no remander.
     * @param arg - the argument to be checked
     * @return true, if it is divisible by 9; return false, otherwise.  */
   public boolean isDivisibleByNine(int arg)
   ```

   (d)
   ```
   /** pay  computes the weekly pay of an employee
     * @param name - the employee's name
     * @param hours - the hours worked for the week
     * @param payrate - the hourly payrate
     * @return a string consisting of the name followed by "$" and the pay */
   public String pay(String name, int hours, double payrate)
   ```

4. Here is an application that would benefit from a function. Rewrite it with one.

   ```
   /** Areas prints the areas of three circles */
   public class Areas
   { public static void main(String[] args)
     { System.out.println("For radius 4, area = " + (Math.PI * 4*4));
       System.out.println(Math.PI * 8*8);
       System.out.println((Math.PI * 19*19) + " is the area for radius 19");
     }
   }
   ```

   (Note: `Math.PI` is Java's name for the math constant, Pi.)

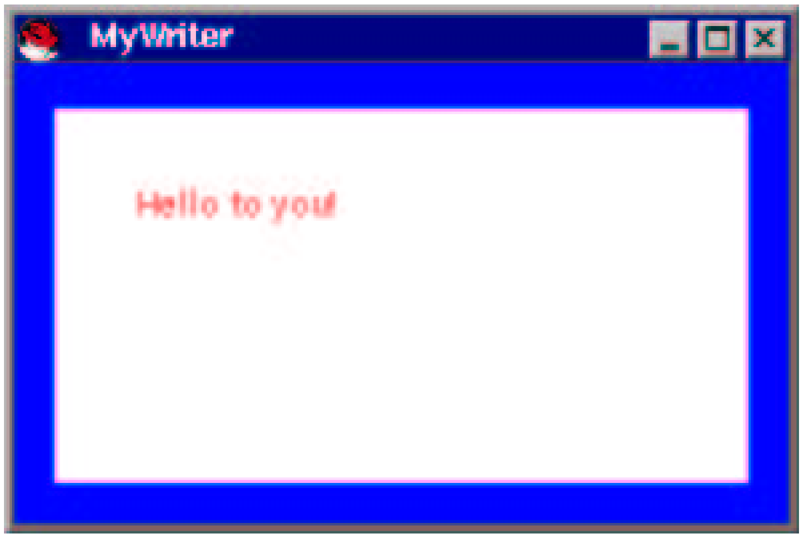
5.6 Private Methods

This Chapter's introduction mentioned that an object's methods can be classified as "public" and "private": Public methods can be invoked by the general public, but private methods can be invoked only within the class where they appear.

Why should we bother with writing private methods? Private methods help impose neat internal structure to a class: specific instruction sequences can be collected together, appropriately named, and they can be invoked multiple times within the class. We give two examples.

Naming a Subalgorithm with a Private Method

Say that we decide to improve `class MyWriter` in Figure 6 so that the graphics window appears with a blue border and a white center:



This adjustment does not change the class's specification in Figure 5—the programming changes are purely internal. But how do we paint a blue border and a white center? The algorithm takes a bit of thought:

1. Paint the entire window *blue*.
2. Calculate the size of a rectangle whose size is slightly smaller than the size of the entire window.
3. Paint, in the center of the blue window, a white rectangle of this slightly smaller size.

These steps accomplish the desired result. Because the algorithm is self contained, it makes sense to give it a name, say, `makeBorder`, and include it as a private method, which is used by `paintComponent`. Figure 8 shows the method and the revised class.

Figure 5.8: graphics window with private method

```
import java.awt.*;  import javax.swing.*;
/** MyWriter2 creates a graphics window that displays a sentence */
public class MyWriter2 extends JFrame
{ private int width;  // the frame's width
  private int height; // the frame's height
  private String sentence = ""; // holds the sentence to be displayed
  private int x_position = 50;  // x-position of sentence
  private int y_position = 80;  // y-position of sentence

  /** Constructor MyWriter2 creates the window and makes it visible
    * @param w - the window's width
    * @param h - the window's height  */
  public MyWriter2(int w, int h)
  { ... see Figure 6 ... }

  /** paintComponent paints the panel
    * @param g - the ``graphics pen'' that draws the items */
  public void paintComponent(Graphics g)
  { makeBorder(g);  // invoke private method to paint the border
    g.setColor(Color.red);
    g.drawString(sentence, x_position, y_position);
  }

  /** makeBorder paints the frame's border.
    * @param pen - the graphics pen used to paint the border */
  private void makeBorder(Graphics pen)
  { pen.setColor(Color.blue);
    pen.fillRect(0, 0, width, height);      // paint entire window blue
    int border_size = 20;
    int center_rectangle_width = width - (2 * border_size);
    int center_rectangle_height = height - (2 * border_size);
    pen.setColor(Color.white);
    pen.fillRect(border_size, border_size,  // paint center rectangle white
                 center_rectangle_width, center_rectangle_height);
  }

  /** writeSentence displays a new string in the window
    * @param s - the sentence to be displayed  */
  public void writeSentence(String s)
  { ... see Figure 6 ... }

  /** repositionSentence redisplays the existing sentence in a new position
    * @param new_x - the new horizontal starting position
    * @param new_y - the new vertical starting position  */
  public void repositionSentence(int new_x, int new_y)
  { ... see Figure 6 ... }
}
```

The private method, `makeBorder`, is written just like a public method, except its header line includes the keyword, `private`. Its parameter, `Graphics pen`, is the graphics pen it is given to do painting. The parameter is supplied by method `paintComponent`, which invokes `makeBorder` by merely stating its name—because the method is a private method, the receiver must be "this" object:

```
makeBorder(g);
```

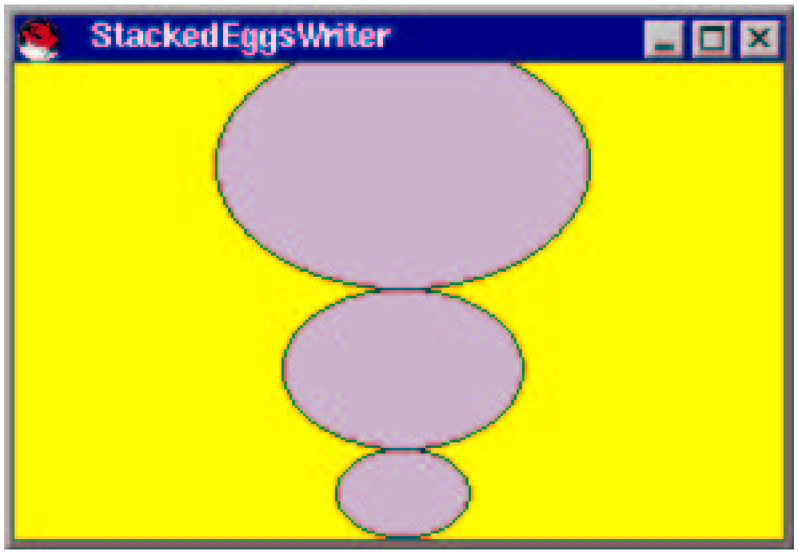
The private method is useful because it maintains the internal structure of the original class, leaves `paintComponent` almost exactly the same as before, and keeps together the "subalgorithm" we wrote for painting the border. If we choose later to paint the border differently, we need only change the private method.

The construction of `makeBorder` is driven by this standard motivation for private methods:

Where a formula or subalgorithm deserves a name of its own, make it into a private method.

Repeating an Activity with a Private Method

Chapter 4 showed us how to draw geometric figures in a graphics window. Say that our current passion is stacking "eggs" in a window, like this:



The window's `paintComponent` method must draw the three eggs, but our sense of economy suggests we should write a private method that knows how to draw one egg and make `paint` invoke the method three times. Because of our knowledge of parameters, we specify the method, `paintAnEgg`, like this:

```
/** paintAnEgg  paints an egg in 3-by-2 proportion
```

```
 * (that is, the egg's height is two-thirds of its width)
 * @param bottom - the position of the egg's bottom
 * @param width - the egg's width
 * @param pen - the graphics pen that will draw the egg
 * @return the y-position of the painted egg's top edge.  */
```

This method's algorithm goes as follows:

1. Calculate the egg's height as 2/3 of its width.
2. Calculate the egg's left edge so that the egg is centered in the window, and calculate the egg's top edge by measuring from its bottom.
3. Paint the egg so that its upper left corner is positioned at the left edge, top edge.
4. Return the value of the top edge, in case it is needed later to stack another egg on top of this one.

Using the parameters, we readily code the algorithm into the private method that appears in Figure 9. Method `paintComponent` merely invokes the private method three times to stack the three eggs. A small `main` method is included at the bottom of the class so that the graphics window can be executed directly.

This example illustrates a classic motivation for writing private methods:

Where there is repetition of a task, write one private method whose body does the task and invoke the method repeated times.

Stated more bluntly, the above slogan warns you, "If you are using the cut-and-paste buttons on your text editor to copy statements to do the same task multiple times, then you should be using a private method instead!"

Exercises

1. Write a testing application that constructs these variations of `StackedEggsWriter`:

   (a) `new StackedEggsWriter(30, 0, 500);`

   (b) `new StackedEggsWriter(30, -10, 30);`

   (c) `new StackedEggsWriter(-300, 300, 30);`

   You might find it helpful to maximize the window to view some of the results.

2. Add public methods `setEggSize1(int size)`, `setEggSize2(int size)`, and `setEggSize3(int size)` to `class StackedEggsWriter`. Of course, each method resets the size of the respective egg displayed and repaints the window. Test the modified class with this application:

Figure 5.9: repeated invocations of a private method

```
import java.awt.*;
import javax.swing.*;
/** StackedEggsWriter displays three eggs, stacked */
public class StackedEggsWriter extends JPanel
{ private int frame_width;
  private int frame_height;
  // the sizes (widths) of the three eggs stacked:
  private int egg1_size;
  private int egg2_size;
  private int egg3_size;

 /** Constructor StackedEggsWriter stacks three 3-by-2 eggs in a window
    * @param width - the width of the panel
    * @param height - the height of the panel
    * @param size1 - the width of the bottom egg
    * @param size2 - the width of the middle egg
    * @param size3 - the width of the top egg  */
  public StackedEggsWriter(int width, int height,
                           int size1, int size2, int size3)
  { frame_width = width;
    frame_height = height;
    egg1_size = size1;
    egg2_size = size2;
    egg3_size = size3;
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("StackedEggsWriter");
    my_frame.setSize(frame_width, frame_height);
    my_frame.setVisible(true);
  }

  /** paintComponent fills the window with the eggs
    * @param g - the graphics pen */
  public void paintComponent(Graphics g)
  { g.setColor(Color.yellow);
    g.fillRect(0, 0, frame_width, frame_height);  // paint the background
    // lay the first egg at the bottom of the frame:
    int egg1_top = paintAnEgg(frame_height, egg1_size, g);
    // stack the two remaining eggs on top of it:
    int egg2_top = paintAnEgg(egg1_top, egg2_size, g);
    int egg3_top = paintAnEgg(egg2_top, egg3_size, g);
  }
 ...
```

Figure 5.9: repeated invocations of a private method (concl.)

```
  /** paintAnEgg  paints an egg in 3-by-2 proportion (the egg's height
    *  is two-thirds of its width)
    * @param bottom - the position of the egg's bottom
    * @param width - the egg's width
    * @param pen - the graphics pen that will draw the egg
    * @return the y-position of the painted egg's top edge.  */
  private int paintAnEgg(int bottom, int width, Graphics pen)
  { int height = (2 * width) / 3;
    int top_edge = bottom - height;
    int left_edge = (frame_width - width) / 2;
    pen.setColor(Color.pink);
    pen.fillOval(left_edge, top_edge, width, height);
    pen.setColor(Color.black);
    pen.drawOval(left_edge, top_edge, width, height);
    return top_edge;
  }

  /** Test the window: */
  public static void main(String[] args)
  { int total_width = 300;
    int total_height = 200;
    new StackedEggsWriter(total_width, total_height, 50, 90, 140);
  }
}
```

```
import javax.swing.*;
public class StackSomeEggs
{ public static void main(String[] args)
  { StackedEggsWriter writer = new StackedEggsWriter(0, 0, 0);
    String s = JOptionPane.showInputDialog("Type size of bottom egg (an int):");
    writer.setEggSize1(new Integer(s).intValue());

    s = JOptionPane.showInputDialog("Type size of middle egg (an int):");
    writer.setEggSize2(new Integer(s).intValue());

    s = JOptionPane.showInputDialog("Type size of top egg (an int):");
    writer.setEggSize3(new Integer(s).intValue());
  }
}
```

3. Modify the `paintBorder` method in Figure 8 so that it paints a white-filled circle with diameter equal to the window's height in the center of the window in front of a yellow background.

4. This class can benefit from a private method; insert it.

```
import java.awt.*;
import javax.swing.*;
/** Circles draws three concentric circles */
public class Circles extends JPanel
{ public Circles()
  { JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("TextWriter");
    my_frame.setSize(200, 200);
    my_frame.setVisible(true);
  }

  public void paintComponent(Graphics g)
  { int x_pos = 100;  // x-position of center of circle
    int y_pos = 100;  // y-position of center of circle
    int diameter = 60;  // diameter of circle to draw
    g.setColor(Color.black);
    int radius = diameter / 2;
    g.drawOval(x_pos - radius, y_pos - radius, diameter, diameter);
    diameter = diameter + 20;
    radius = diameter / 2;
    g.drawOval(x_pos - radius, y_pos - radius, diameter, diameter);
    diameter = diameter + 20;
```

```
      radius = diameter / 2;
      g.drawOval(x_pos - radius, y_pos - radius, diameter, diameter);
    }
  }
```

5.7 Summary

We now summarize the main points of this chapter:

New Constructions

Here are examples of the new constructions encountered in this chapter:

- *public method* (from Figure 1):

```
/** printLadybug prints a ladybug */
public void printLadybug()
{ System.out.println(" ‘m\’"); // the ’ must be written as \’
  System.out.println(" (|)  sahr");
  System.out.println();
}
```

- *formal parameter* (from Figure 3):

```
/** printBeeWithName prints a bee with a name attached to its stinger.
  * @param name - the name attached to the stinger  */
public void printBeeWithName(String name)
{ System.out.println("  ,-.");
  System.out.println("  \\_/");
  System.out.println(">{|||}-" + name + "-");
  System.out.println("  / \\");
  System.out.println("  ‘-^  hjw");
  System.out.println();
}
```

- *function* (from Figure 7):

```
/** celsiusIntoFahrenheit translates degrees Celsius into Fahrenheit
  * @param c - the degrees in Celsius
  * @return the equivalent degrees in Fahrenheit */
public double celsiusIntoFahrenheit(double c)
{ double f = ((9.0 / 5.0) * c) + 32;
  return f;
}
```

- *private method* (from Figure 9):

```
/** paintAnEgg  paints an egg in 3-by-2 proportion (the egg's height
    *  is two-thirds of its width)
    * @param bottom - the position of the egg's bottom
    * @param width - the egg's width
    * @param pen - the graphics pen that will draw the egg
    * @return the y-position of the painted egg's top edge.  */
  private int paintAnEgg(int bottom, int width, Graphics pen)
  { int height = (2 * width) / 3;
    int top_edge = bottom - height;
    int left_edge = (FRAME_WIDTH - width) / 2;
    pen.setColor(Color.pink);
    pen.fillOval(left_edge, top_edge, width, height);
    pen.setColor(Color.black);
    pen.drawOval(left_edge, top_edge, width, height);
    return top_edge;
  }
```

New Terminology

- *method*: a named sequence of statements; meant to accomplish a specific task or responsibility (e.g., `writeSentence`) A method has a *header line* (e.g., `public void writeSentence(String s)` and a *body* (e.g., { `sentence = s; repaint();` })
- *invoking a method*: sending a message that includes a method's name (e.g., `writer.printBee()`). The message requests that the receiver object (`writer`) execute the method named in the message (`printBee`).
- *client*: the object that sends a message (an invocation).
- *receiver*: the object that receives a message; it executes the method named in the message.
- *actual parameters*: arguments that are listed with the method's name when invoking the method (e.g., `"Lucy"` in `writer.printBeeWithName("Lucy")`)
- *formal parameters*: variable names that are listed in the method's header line (e.g., `String name` in the header line, `public void printBeeWithName(String name)`. When the method is invoked, the actual parameters are assigned (or *bound*) to the formal parameters (e.g., the invocation, `writer.printBeeWithName("Lucy")`, assigns `"Lucy"` to variable `name`.)

- *result of a method*: an "answer" calculated by a method and given back to its client (e.g., `return f` in the body of `celsiusToFahrenheit` in Figure 7.
- *function*: a method that returns a result
- *scope of a formal parameter*: the statements where the parameter can be referenced and assigned. This is normally the method's body where the parameter is defined (e.g., the scope of formal parameter `s` of `writeSentence` consists of the two statements, `sentence = s; this.repaint();`).
- *(interface) specification*: a list of a class's methods and their behaviors; can also list the class's private attributes (e.g., Table 5 is the specification for `class MyWriter` in Figure 6).

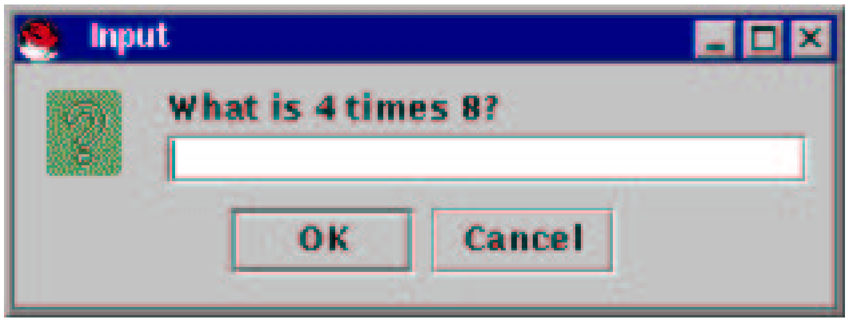
Points to Remember

- A method can be labelled *private* (for the use of only the other methods within the class in which it appears, e.g., `paintAnEgg`) or *public* (for the use of other objects, e.g., `writeSentence`).
- Private methods are written in response to two situations:
  1. *Where there is repetition of a task, write one private method whose body does the task and invoke the method repeated times.*
  2. *Where a fundamental concept, formula, or subalgorithm deserves a name of its own, make it into a private method.*
- Public methods are written in response to this situation: *A skill or responsibility that other objects depend upon should be written as a public method.*
- Often, we design classes and methods hand in hand, because a class possesses a collection of related "behaviors," where the methods encode the behaviors.
- A specification lists the names and describes the responsibilities of a class's public methods. Specifications are written for each class because:
  1. It becomes easier to use and reuse a class in several different applications.
  2. It becomes easier for distinct people to design, write, and test the distinct classes in an application;
  3. It becomes easier to rewrite or replace one class without rewriting all the others;
- Actual parameters bind to formal parameters in order—the first actual parameter binds to the first formal parameter, the second actual parameter binds to the second formal parameter, and so on.

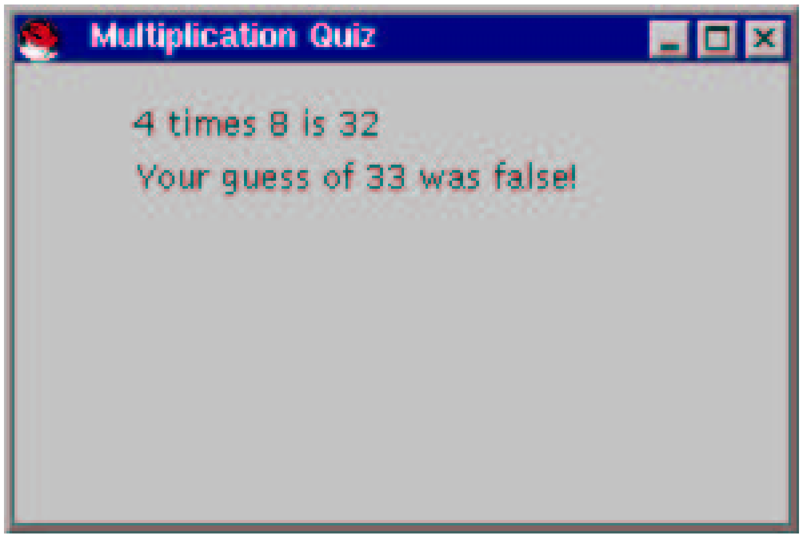
- Any value that can be assigned to a variable can be an actual parameter that binds to a formal parameter.

5.8 Programming Projects

1. Return to the Programming Projects for Chapter 4. For any of the projects that you completed, revise the application so that it displays its output information in a graphics window that you designed first with a specification and then coded. For example, if you built the application that helps a child learn multiplications, then the application might display an input view that asks,

   When the child replies, with say, `33`, the answer appears in a newly created graphics window:

   ```
   4 times 8 is 32
   Your guess of 33 was false!
   ```

2. For practice with private methods, write this application that counts votes for a small election:

   When started, the application asks Candidate 1 to type her name, followed by her address. The application asks Candidate 2 to do the same. Next, the application asks 5 voters to vote for either Candidate 1 or Candidate 2. The application finishes by displaying the candidates' names, addresses, and total vote counts.

3. Answers have more impact when displayed visually. Write an application that displays the distances one can travel on a full tank of gasoline at different velocities. Use this formula to calculate distance travelled for a positive velocity, v:

$$distance = (40 + 0.05v - (0.06v)^2) * capacity$$

   where *capacity* is the size of the automobile's fuel tank. (Note: This simplistic formula assumes that the car has a one-gear transmission.)

   The input to the application is the fuel tank's size; the output are the distances travelled (and the time taken to do so) for velocities 20, 40, 60, 80, and 100 miles per hour. Display the answers graphically, so that the answers are pictures like this:

```
For a 10-gallon fuel tank:
                             /A CAR\
40 m.p.h.: ================== -o--o--  362.4 miles in 9.06 hours

                         /A CAR\
60 m.p.h.: ============== -o--o--  300.4 miles in 5.007 hours
```

4. Recall that the formula to calculate the distance, `d`, that an automobile travels starting from initial velocity, `Vsub 0` , and acceleration, `a`, is defined

$$d = V_0 t + (1/2)a(t^2)$$

   Write an application that asks the user to submit values for V_0 and a; the application produces graphical output showing the distances travelled for times 0, 1, ..., 10.

5. Write a general purpose currency convertor program. The program is started by giving it three program arguments: the name of the currency the user holds and wishes to sell, the name of the currency the user wishes to buy, and the conversion rate from the first currency to the other. For example, the application might be given these arguments:

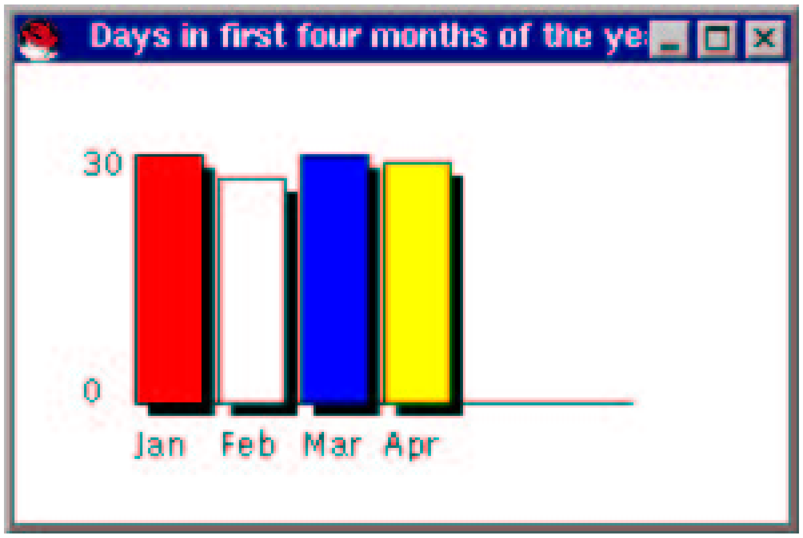
```
USDollar Euro 0.9428
```

   to tell it that one US Dollar can buy 0.9428 of one Euro.

   Once it is started, the application asks the user to type the amount of currency the user wishes to sell. When the user submits this information, the application computes the amount of currency purchased and displays the answer as two proportionally sized coins:

```
IMAGE OF TWO BALLOONS, LABELLED AS:  200 Dollars   =    188.56 Euros
```

6. Write a class that helps a user display a bar graph that displays up to 6 separate bars. Here is an example of such a graph: `class BarGraphWriter`:



The class should have these methods:

`class BarGraphWriter`	helps a user draw bar graphs
Methods	
`setAxes(int x_pos, int y_pos, String top_label, int y_height)`	draw the x- and y-axes of the graph. The pair, `x_pos`, `y_pos`, state the coordinates on the window where the two axes begin. The height of the y-axis, stated in pixels, is `y_height`. (The length of the x-axis will be exactly long enough to display 6 bars.) The label placed at the top of the y-axis is `top_label`. (The label placed at the bottom of the y_axis is always `0`.) See the picture above.
`setBar1(String label, int height, Color c)`	draw the first bar in the graph, where the label underneath the bar is `label`, the height of the bar, in pixels, is `height`, and the bar's color is `c`. See the picture above.
`setBar2(String label, int height, Color c)`	draw the second bar in the graph, where the arguments are used in the same way as in `setBar1`
`setBar3(String label, int height, Color c)`, `setBar4(String label, int height, Color c)`, `setBar5(String label, int height, Color c)`, `setBar6(String label, int height, Color c)`	draw the third through sixth bars of the graph

(a) Here is the application that drew the above graph:

```
import java.awt.*;
public class TestGraph
{ public static void main(String[] a)
  { BarGraphWriter e = new BarGraphWriter();
    e.setTitle("Days in first four months of the year");
    e.setAxes(20, 120, "30", 90);      // x- and y-axes start at  20, 120
                // graph is 90 pixels high; top of graph is labelled "30"
    int scale_factor = 3;
    e.setBar1("Jan", 31 * scale_factor, Color.red);   // Jan has 31 days
    e.setBar2("Feb", 28 * scale_factor, Color.white); // etc.
    e.setBar3("Mar", 31 * scale_factor, Color.blue);
    e.setBar4("Apr", 30 * scale_factor, Color.red);
  }
}
```

Test your coding of `class BarGraphWriter` with `TestGraph`.

(b) Here is table of interesting statistics about the first six planets:

	Mercury	Venus	Earth	Mars	Jupiter	Saturn
Distance from Sun (astronomical units)	0.387	0.723	1.00	1.524	5.203	9.539
Mass (relative to Earth)	0.05	0.81	1.00	0.11	318.4	95.3
Length of day (hours)	2106.12	718	23.93	24.62	9.83	10.03
Length of year (in Earth days)	88.0 days	224.7 days	365.3 days	687 days	11.86 years	29.46 years
Weight of 1kg. object	0.25	0.85	1.00	0.36	2.64	1.17

Plot some or all of these statistics in bar graphs.

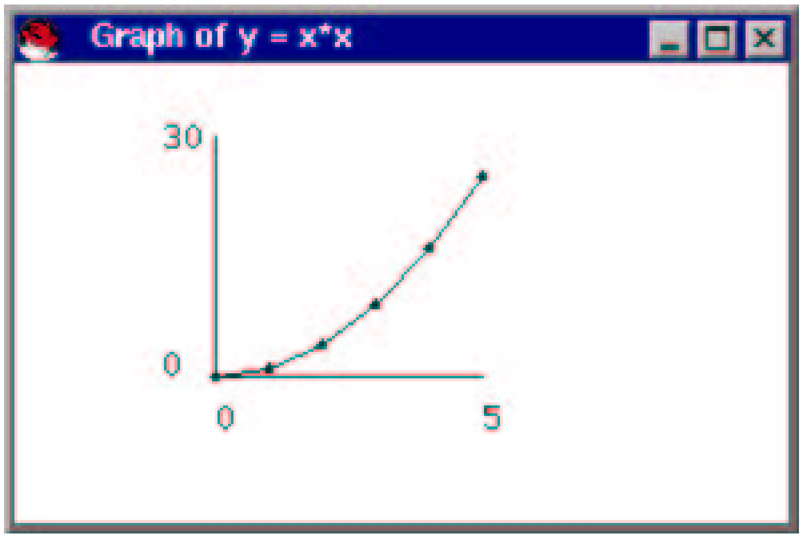
(c) Alter the `MakeChange` application in Figure 3, Chapter 3, so that it reads its input interactively and displays the amounts of change as a four-bar graph. (Think of these as "stacks" of coins!)

(d) Write an application that helps the user plot a bar graph of her own. The application first asks the user for the title of the graph. Next, the application asks for the largest value that will be plotted as a bar—this will be used as the label on the graph's y-axis. Then, the application asks the user for the values of the six bars to be drawn. (If the user does not want one of the six bars to appear, she types `0` for its value and *newline* by itself for the bar's label.) The application displays the bar graph as its answer.

Notice that the application decides for itself the location of the x- and y-axes. Here is how the interaction between the application and its user might go (the user's responses are stated in italics):

```
Please type the title of your graph: <em>Days in the months</em>
Please type the value of the largest bar you will draw: <em>31</em>
Please type the name of the first bar: <em>Jan</em>
Please type the value of the first bar: <em>31</em>
  ... etc. ...
```

7. Another form of graph is a sequence of plotted points, connected by lines. Here is a plot of the equation, `y = x*x`, where the values of `y` for `x` equals 0,1,...,5 are plotted and connected:



Write a class, `PointGraphWriter`, that generates a graph of exactly 6 plotted points. The class should meet this specification:

`class PointGraphWriter`	helps a user draw a graph of 6 plotted points Methods
`setAxes(int x_pos, int y_pos, int axis_length, String x_label, String y_label)`	draw the the vertical and horizontal axes of the graph, such that the intersection point of the axes lies at position `x_pos, y_pos`. Each axis has the length, `axis_length` pixels. The beginning labels of both the x- and y-axes are `0`; the label at the top of the y-axis is `y_label`, and the label at the end of the x-axis is `x_label`.
`setPoint1(int height)`	plot the first point of the graph, so that it appears at the 0-position on the x-axis and at the `height` position on the y-axis.
`setPoint2(int height)`	plot the second point of the graph, so that its x-position is at one-fifth of the length of the x-axis and at the `height` position on the y-axis.

`setPoint3(int height),`
`setPoint4(int height),`
`setPoint5(int height),`
`setPoint6(int height)`
	plot the third and subsequent points of the graph, so that they are equally spaced along the x-axis and appear at the specified `height` on the y-axis. Each point is connected to its predecessor by a straight line.

Test the class you write on these applications:

(a) the graph seen above. Here is the application that does this; test your coding of `class PointGraphWriter` with it.

```
public class TestPlot
{
  public static void main(String[] a)
  { PointGraphWriter e = new PointGraphWriter();
    e.setTitle("Graph of  y = x*x");
    e.setAxes(50, 110, 90, "5", "30");
      // axes start at position 50,110; the axes have length 90 pixels
      // x-axis is labelled  0..5
      // y-axis is labelled  0..30
    int scale_factor = 3;
    e.setPoint1(0 * scale_factor);   // 0*0 = 0
    e.setPoint2(1 * scale_factor);   // 1*1 = 1
    e.setPoint3(4 * scale_factor);  // 2*2 = 4
    e.setPoint4(9 * scale_factor);  // etc.
    e.setPoint5(16 * scale_factor);
    e.setPoint6(25 * scale_factor);
  }
}
```

(b) the values of `y`, for `x` equals 0, 2, 4, 6, 8, 10. (Note: set `y`'s scale to the range 0..100.)

i. $y = x^2 + 2x + 1$

ii. $y = 90 - (0.8x)^2$

iii. $y = 20x - (0.5x)^3$

iv. $y = 0.1(x^3) + x^2 - x$

(c) Write an application that helps the user plot a graph of her own. The application asks the user the maximum values for the x- and y- axes, and then the application asks the user for the values of the six points to be plotted. The application displays the plotted graph as its answer.

(d) Use `class PointGraphWriter` to plot the daily high temperatures at your home for the past six days.

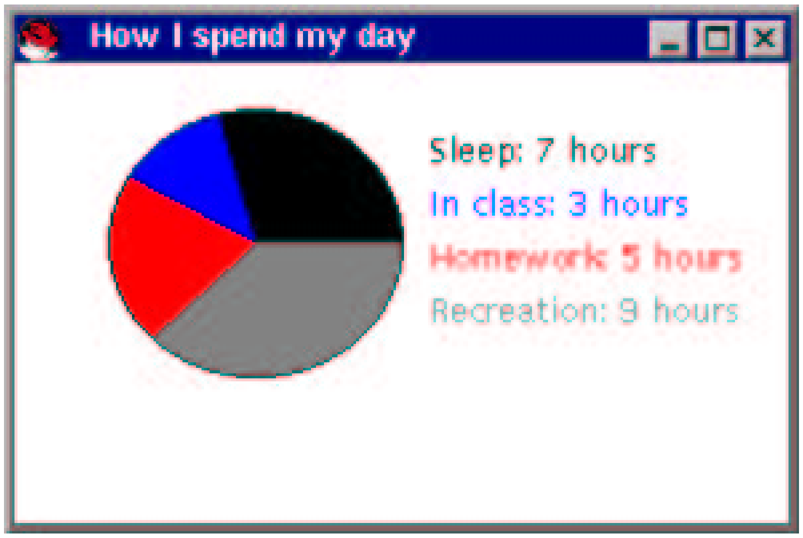
(e) Use `class PointGraphWriter` to plot the weekly share value over the past 6 weeks of a stock of your choosing

(f) Recall that the formula to calculate the distance, `d`, that an automobile travels starting from initial velocity, V_0, and acceleration, a, is defined

$$d = V_0 t + (1/2)a(t^2)$$

Write an application that asks the user to submit values for V_0 and a; the application produces as its output a plotted graph that shows the distances travelled for times 0, 2, 4, ..., 10.

8. Here is an example of a "pie chart":



The chart was generated from an output-view object with this specification:

`class PieChartWriter`	helps a user draw a pie chart of at most 6 "slices"
Methods	
`setSlice1(String label, int amount, Color c)`	draw the first slice of the chart, such that `amount` indicates the amount of the slice, and `c` is the slice's color. The `label` is printed to the right of the pie, and it is printed in the color, `c`.
`setSlice2(String label, int amount, Color c)`, `setSlice3(String label, int amount, Color c)`, `setSlice4(String label, int amount, Color c)`, `setSlice5(String label, int amount, Color c)`, `setSlice6(String label, int amount, Color c)`	draw the subsequent slices and their labels.

Use the class to plot

(a) the chart seen above. Here is the application that does this; test your coding of `class PieChartWriter` with it.

```
import java.awt.*;
public class TestPieChart
{  public static void main(String[] args)
   { PieChartWriter p = new PieChartWriter();
     p.setTitle("How I spend my day");
     p.setSlice1("Sleep: 7 hours", 7, Color.black);
     p.setSlice4("Recreation: 9 hours", 9, Color.gray);
     p.setSlice2("In class: 3 hours", 3, Color.blue);
     p.setSlice3("Homework: 5 hours", 5, Color.red);
  }
}
```

(b) these percentages about the income and outlays of the United States government in Fiscal Year 1997:

```
INCOME:
personal income taxes: 46%
social security and medicare taxes: 34%
corporate income taxes: 11%
excise and customs taxes: 8%
borrowing to cover deficit: 1%
```

```
OUTLAYS:
social security and medicare: 38%
national defense: 20%
social programs: 18%
interest on national debt: 15%
human and community development: 7%
general government: 2%
```

(c) the statistics regarding the masses of the planets in the table seen two exercises earlier.

(d) how you spend your monthly budget

5.9 Beyond the Basics

These optional sections build on the core materials in the chapter and show

- *how to invent names for methods, classes, parameters, and variables*
- *how to use the* `javadoc` *program to create API documentation for the classes you write*
- *how the* `main` *method can use* static *private methods*
- *how to state precisely the syntax and semantics of method declaration and invocation*

5.9.1 Naming Variables, Methods, and Classes

Here are the guidelines used in this text for devising names for variables, methods, and classes. The guidelines are devised so that we can read a name and deduce almost immediately whether it is a name of a variable, a method, or a class.

- The name of a class should begin with an upper-case letter, and the first letter of each individual word in the name should be upper case also. Examples are `MyWriter` and `GregorianCalendar`. Underscores, _, and dollar signs, `$`, although legal, are discouraged.

- The name of a method should begin with a lower-case letter, and the first letter of each individual word in the name should be upper-case, e.g., `writeSentence`. Underscores and dollar signs are discouraged. If the method returns no result (its return type is `void`), then use an imperative verb or predicate phrase for its name, (e.g., `printBee` or `writeSentence`). If the method is a function (its return type is non-`void`), then use a phrase that describes the result or how to compute the result (e.g., `celsiusIntoFahrenheit`).

- The names of formal parameters and variables should be should be descriptive but succinct. In this text we use all lower-case letters for variables, where the the underscore character separates individual words in a variable name, e.g., `answer` and `left_edge`. This distinguishes variable names from method and class names. A commonly used alternative is to use the same spelling for method names as for variable names—do as you please.

- When a field variable receives an initial value that never changes, it might be spelled with all upper-case letters and underscores, e.g., `FRAME_WIDTH`.

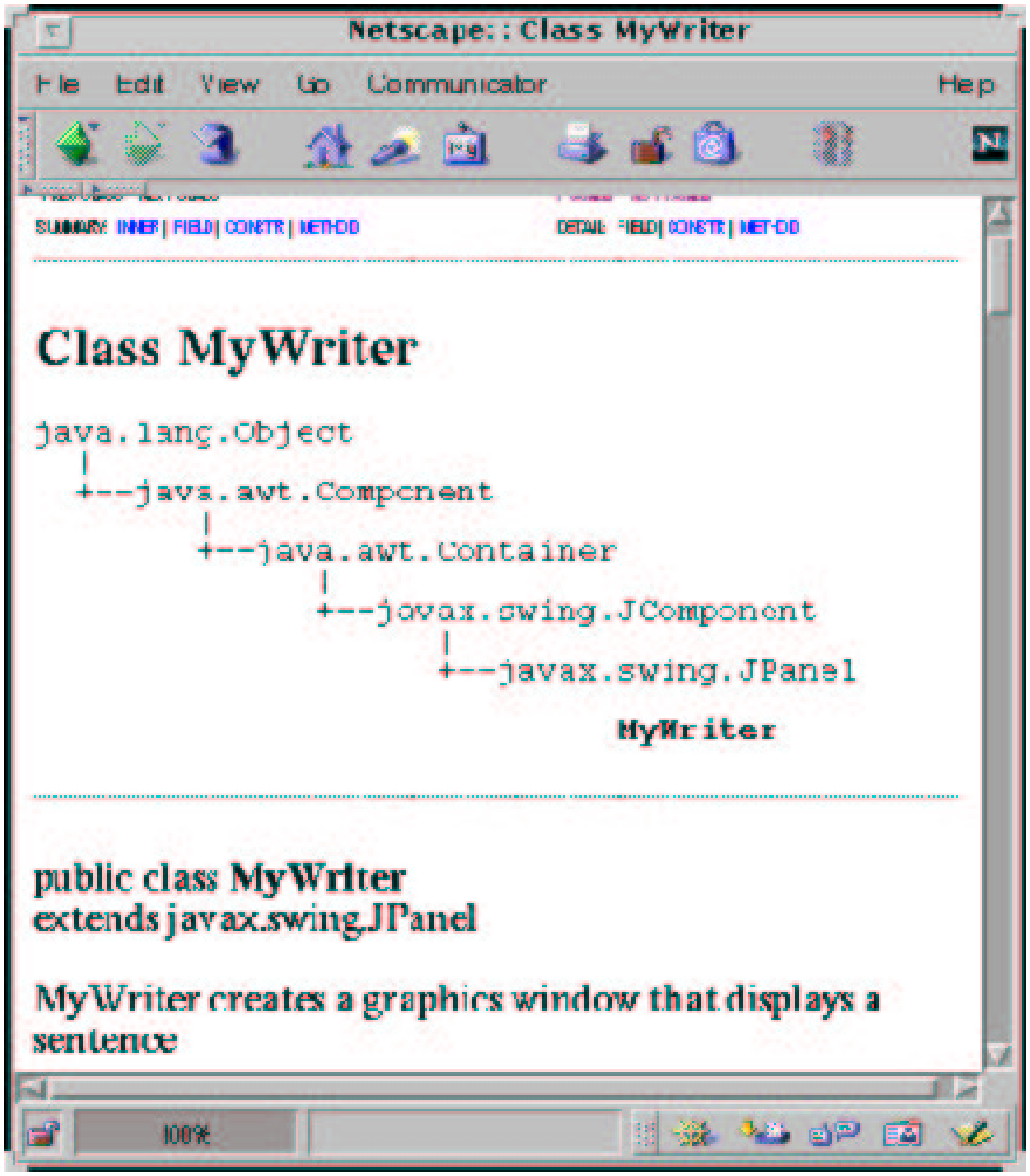
  Begin Footnote: Java provides a special variant of a field variable, called a `final` variable, which is a field whose value is set once and never changed thereafter. Final variables are studied in Chapter 9. *End Footnote*

5.9.2 Generating Web Documentation with `javadoc`

In Chapter 2, we observed that Java's packages, like `java.lang` and `java.util`, are documented with HTML-coded web pages. The web pages are called the *API (Application Programming Interface) documentation* and were generated automatically from the Java packages themselves by the tool, `javadoc`.

The current chapter suggested that every class should have a specification. Further, information from the specification should be inserted into the comments that prefix the class's methods. Every class's specification should be documented, ideally with an an HTML-coded Web page. We obtain the last step for free if the class's comments are coded in "javadoc style," because we can use the `javadoc` program to generate the Web pages.

`javadoc` is a program that reads a Java class, locates all comments bounded by the format `/** ... */`, and reformats them into Web pages. For example, we can apply `javadoc` to `class MyWriter`. by typing at the command line `javadoc MyWriter.java` (Note: The previous command works if you use the JDK. If you use an IDE, consult the IDE's user guide for information about `javadoc`.) Here is the page created for the class:


Netscape: : Class MyWriter
File
Edit
View
Go
Communicator
Help
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
100%

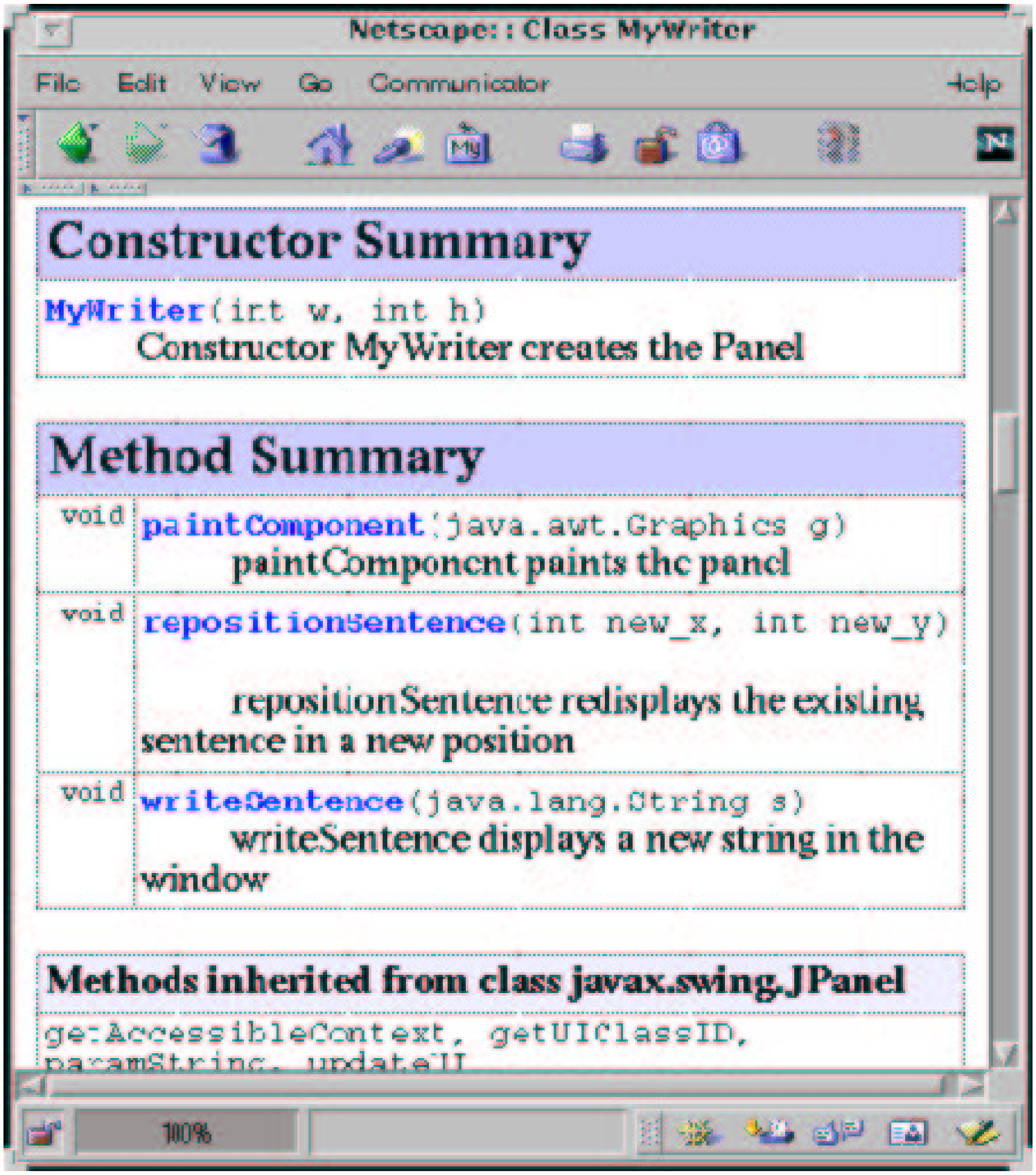
Class MyWriter

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
                          |
                          +--MyWriter
```

public class MyWriter
extends javax.swing.JPanel

MyWriter creates a graphics window that displays a sentence

The comments from `MyWriter` have been attractively formatted into a useful description and annotated with additional information about the built-in graphics classes that were included by inheritance with `MyWriter` If we scroll the page downwards, we encounter a summary of the class's public methods:


Netscape: : Class MyWriter
File Edit View Go Communicator Help
100%

Constructor Summary

`MyWriter(int w, int h)`
Constructor MyWriter creates the Panel

Method Summary

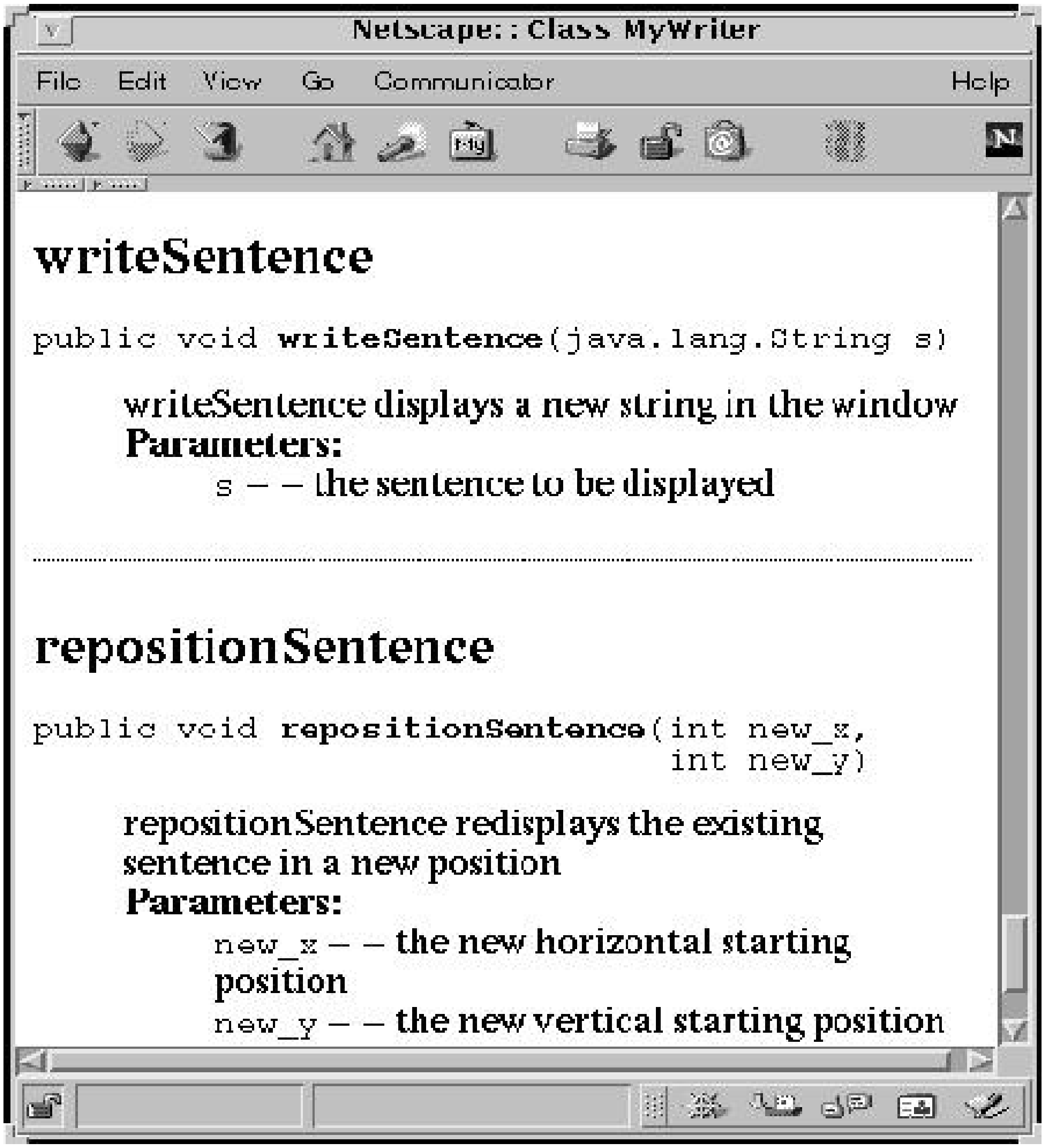
`void`	`paintComponent(java.awt.Graphics g)` paintComponent paints the panel
`void`	`repositionSentence(int new_x, int new_y)` repositionSentence redisplays the existing sentence in a new position
`void`	`writeSentence(java.lang.String s)` writeSentence displays a new string in the window

Methods inherited from class javax.swing.JPanel

`getAccessibleContext, getUIClassID, paramString, updateUI`

Now, a user of `MyWriter` need not read the program to learn its skills—the API documentation should suffice!

If we wish more information about one of the methods, say, `writeSentence`, we jump to the link named `writeSentence` and view the following:


Netscape: : Class MyWriter
File Edit View Go Communicator Help

writeSentence

```
public void writeSentence(java.lang.String s)
```

writeSentence displays a new string in the window
Parameters:
`s` – – the sentence to be displayed

repositionSentence

```
public void repositionSentence(int new_x,
                               int new_y)
```

repositionSentence redisplays the existing sentence in a new position
Parameters:
`new_x` – – the new horizontal starting position
`new_y` – – the new vertical starting position

The commentary from the `writeSentence` method has been extracted and formatted in a useful way.

The format of comments that `javadoc` reads is as follows:

- A class should begin with a comment formatted with `/** ... */`. The comment must begin with a one-sentence description that summarizes the purpose of the class. The comment can extend over multiple lines, provided that each line begins with `*`. If desired, lines of the form, `* @author ...` and `* @version ...` , can be included in this comment to indicate the author of the class and date or version number of the class.

- Prior to each method in the class, there is a comment stating the purpose of the method, the purposes of its parameters, and the purpose of the result that the method returns.
  - The comment begins with `/**`, and each subsequent line begins with `*`. The first sentence of the comment must state the purpose of the method.
  - Each parameter is documented on one or more lines by itself, beginning with `* @param`
  - The method's result is documented on one or more lines by itself, beginning with `* @return`
  - The comment is terminated with the usual `*/`
  - If a method can possibly "throw" an exception (e.g., a division by zero, as seen in Chapter 3), a comment line labelled `@throw` can be used to warn its user.
- If desired, comments of the form, `/** ... */`, can be included before the fields and private methods of the class.

One-line comments of the form, `// ...`, are ignored by `javadoc`.

Important: `javadoc` will *not* normally include commentary about private fields and methods in its web pages, because these components are not meant for public use. If you desire documentation of a class's private components, you obtain this by using the `-private` option, e.g., `javadoc -private TemperaturesWriter.java`.

`javadoc` is best used to document a *package* of classes; we organize applications into packages in in Chapter 9.

Exercise

Apply `javadoc` to `class TemperaturesWriter` in Figure 10, once with and once without the `-private` option. Do the same for `class DialogReader` in Figure 14.

5.9.3 Static Methods

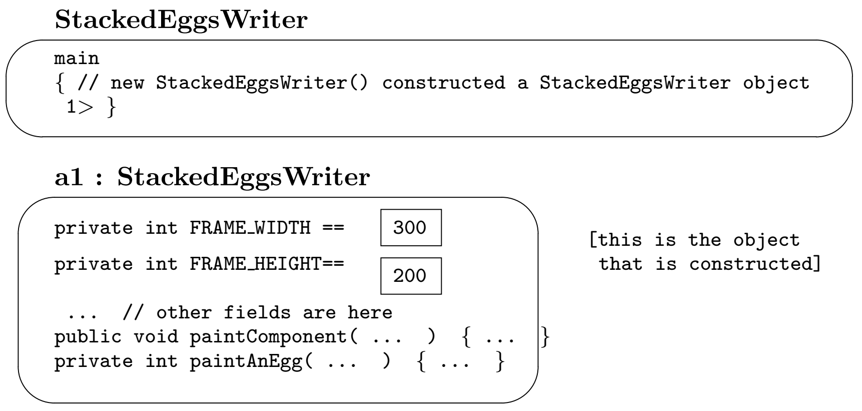
An application is typically designed as a collection of classes, where each class contains methods and private field variables. A class's methods and fields work together to perform the class's responsibilities—a good example is `class MyWriter` in Figure 6.

Occasionally, one writes a method that does not depend on field variables and can stand by itself, that is, the method need not be owned by any particular object and need not be coded in any particular class. For historical reasons, such a method is called *static*.

The `main` method, which starts every Java application, is the standard example of a static method: `main` need not belong to any particular object or class. In this text, for each application we typically write a separate "start up" class that holds `main`. But `main` need not be placed in a class by itself—it can be inserted in any one of an application's classes, as we saw in Figure 9, where a trivial `method` was inserted into `class StackedEggsWriter` for convenient start-up.

In Java, a static method like `main` is labelled `static` in its header line. The label, `static`, means that *the method can be invoked without explicitly constructing an object that holds the method.* This is indeed what happens when we start a Java application.

Here is a picture in computer storage after the `StackedEggsWriter` class from Figure 9 is started:


StackedEggsWriter
main
{ // new StackedEggsWriter() constructed a StackedEggsWriter object
1> }
a1 : StackedEggsWriter
private int FRAME_WIDTH ==
300
private int FRAME_HEIGHT==
200
... // other fields are here
public void paintComponent(...) { ... }
private int paintAnEgg(...) { ... }
[this is the object that is constructed]

The static portion embedded within `class StackedEggsWriter` is extracted and kept separate from the `new StackedEggsWriter()` objects that are subsequently constructed. The `new StackedEggsWriter()` object *never contains the static portion of the class.* Technically, the static portion of `class StackedEggsWriter` is not an "object" like the objects created by `new StackedEggsWriter()` (in particular, it does not have an address), but it does occupy storage and we will continue to draw it to look like an object.

There can be other static methods besides `main`. For example, here is a rewrite of the temperature-conversion application, where `main` relies on its own private method to convert the input temperature:

```
import java.text.*;
import javax.swing.*;
/**  CelsiusToFahrenheit2 converts an input Celsius value to Fahrenheit.
  *    input: the degrees Celsius, an integer read from a dialog
  *    output: the degrees Fahrenheit, a double */
```

```
public class CelsiusToFahrenheit2
{ public static void main(String[] args)
  { String input =
       JOptionPane.showInputDialog("Type an integer Celsius temperature:");
    int c = new Integer(input).intValue();  // convert  input  into an int
    double f = celsiusIntoFahrenheit(c);
    DecimalFormat formatter = new DecimalFormat("0.0");
    System.out.println("For Celsius degrees " + c + ",");
    System.out.println("Degrees Fahrenheit = " + formatter.format(f));
  }

  /** celsiusIntoFahrenheit translates degrees Celsius into Fahrenheit
    * @param c - the degrees in Celsius
    * @return the equivalent degrees in Fahrenheit */
  private static double celsiusIntoFahrenheit(int c)
  { return ((9.0 / 5.0) * c) + 32; }
}
```

The private method, `celsiusIntoFahrenheit`, must itself be labelled `static`, because the private method is itself invoked without explicitly constructing an object that holds it. As a rule, *all methods invoked by a* `static` *method must themselves be labelled* `static`*.*

There is another use of `static` methods in Java: We might argue that some methods, like the square-root, exponentiation, sine, and cosine methods, really do not need to be held within an explicitly constructed object. Of course, one might also argue that it is reasonable to construct an object that has the abilities to compute precisely these operations!

The Java designers followed the first argument, and they wrote `class Math`, which resides in the package `java.lang`. Inside `class Math` are the static methods, `sqrt`, `pow`, `sin`, and `cos`, mentioned above. Because these methods are labelled `static`, we invoke them differently than usual: Rather than constructing a `new Math()` object and sending `sqrt` messages to it, we invoke the method directly:

```
double d = Math.sqrt(2.0);
```

Instead of an object name, an invoked static method is prefixed by the name of the *class* where the method is coded. This is analogous to starting an application by typing the name of the class where the `main` method is coded.

Other examples of static methods are `showInputDialog` and `showMessageDialog` from `class JOptionPane`. Indeed, as you utilize more of Java's libraries, you will encounter classes holding just static methods (or classes with a mixture of static and normal—nonstatic—methods). Please remember that a static method is invoked by prefixing it with its class name; a normal method is invoked by constructing an object from the class and sending the object a message to invoke its method.

In this text, the only static method we ever write will be `main`*.*

Finally, we note that Java allows *fields* to be labelled `static` and `public` as well! We will not develop this variation at this time, other than to point out that `class Math` uses it to give a convenient name for the mathematical constant, Pi: When you write,

```
System.out.println("Pi = " + Math.PI);
```

you are using a public, static field, named `PI`, that is declared and initialized within `class Math`.

Static fields make later appearances in Chapters 8 and 10.

5.9.4 How the Java Compiler Checks Typing of Methods

The data typing information in a method's header line helps the Java compiler validate both the well formedness of the method's body as well as the correctness of invocations of the method. When the compiler reads a method like

```
public double celsiusIntoFahrenheit(double c)
{ double f = ((9.0 / 5.0) * c) + 32;
  return f;
}
```

the compiler verifies that the method's body uses parameter `c` the way doubles should be used. Also, all statements of the form `return E` must contain expressions, `E`, whose data type matches the information in the header line. In the above example, the compiler does indeed verify that `f` has type `double`, which matches the data type that the header line promises for the function's result.

When the compiler encounters an invocation, e.g., `convert.celsiusIntoFahrenheit(68)`, the compiler verifies that

- the data type of `convert` is a class that has a `celsiusIntoFahrenheit` method.
- the data type of the actual parameter, `68`, is compatible with the data type of the corresponding formal parameter of `celsiusIntoFahrenheit`. (More precisely, the data type of the actual parameter must be a subtype of the formal parameter's type; here, an actual parameter that is an integer is acceptable to a formal parameter that is a double.)
- the answer, if any, returned by the method can be used at the position where the invocation is located.

But most important, when the Java compiler studies an invocation, *the compiler does not reexamine the body of invoked method; it examines only the method's header line.*

When the compiler examines an entire class, it analyses the class's fields and methods, one by one. When it concludes, the compiler collects an interface specification for the class, based on the information it extracted from the header lines of the class's public components. The information is used when the compiler examines another class that references this first one. In this way, the Java compiler can systematically analyze an application built from multiple classes.

A more precise description of data-type checking of methods follows in the next section.

5.9.5 Formal Description of Methods

Because method writing is fundamental to programming, here is a formal description of the syntax and semantics of method definition and invocation. The steps behind type checking and executing method invocations are surprisingly intricate, mainly because Java allows one class to extend (inherit) another's structure.

The material that follows is provided as a reference and most definitely is nonrequired reading for the beginner.

Method Definition

The format of method introduced in this chapter extends the syntax of methods that appeared at the end of Chapter 2. We now have this form:

```
METHOD ::=  VISIBILITY  static?  RETURN_TYPE  METHOD_HEADER
            METHOD_BODY
```

Recall that `?` means that the preceding phrase is optional.

```
VISIBILITY ::=  public | private
RETURN_TYPE ::=  TYPE | void
METHOD_HEADER ::=  IDENTIFIER ( FORMALPARAM_LIST? )
```

A method can be public or private and can optionally return a result. The syntax of `TYPE` remains the same from Chapter 3.

```
FORMALPARAM_LIST ::=  FORMALPARAM [[ , FORMALPARAM ]]*
FORMALPARAM ::=  TYPE IDENTIFIER
METHOD_BODY ::=  { STATEMENT* }
```

A method can have a list of parameters, separated by commas. (Recall that `*` is read as "zero or more" and the double brackets are used for grouping multiple phrases.)

The syntax of `STATEMENT` is carried over from Chapter 2, but one form of statement changes:

```
INVOCATION ::=  [[ RECEIVER . ]]? IDENTIFIER ( ARGUMENT_LIST? )
```

because the `RECEIVER` can be omitted when an object invokes one of its own methods. Also, a new statement form is included:

```
RETURN ::=  return EXPRESSION ;
```

A method definition is well typed if

- All of the `FORMALPARAM`s in a method's header line are distinctly named identifiers.
- The `STATEMENT`s in the method's body are well typed, assuming use of these extra variables:
  - each formal parameter, `TYPE IDENTIFIER`, defines a variable, `IDENTIFIER` of type `TYPE`.
  - each field, `private TYPE IDENTIFIER`, in the class in which this method appears defines a variable, `IDENTIFIER` of type `TYPE`.

  In case a field has the same name as a local variable or formal parameter, the latter takes precedence in usage.
- In the method's body, the `E` component of every `return E` statement has the data type specified as the method's `RETURN_TYPE`. (If `RETURN_TYPE` is `void`, no `return E` statements are allowed.)

The semantics of a method definition is that the name, formal parameters, and body of the method are remembered for later use.

Method Invocation

Method invocations change little from their format in Chapter 2:

```
INVOCATION ::=  [[ RECEIVER . ]]? IDENTIFIER ( ARGUMENT_LIST? )
```

In this section, we consider invocations of only normal (non-static) methods; see the subsection below for comments about invocations of static methods.

Say that the Java compiler must check the data types within an invocation that looks like this:

```
[[ RECEIVER . ]]?  NAME0 ( EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn)
```

As noted in the preceding section, the data-type checking proceeds in several steps:

1. *Determine the data type of* `RECEIVER`*:* if `RECEIVER` is omitted, then use the class name where the invocation appears; if `RECEIVER` is a variable name, then use the variable's data type (which must be a class name); if `RECEIVER` is an arbitrary expression, calculate its data type (which must be a class name).

   Let `C0` be the data type that is calculated.

2. *Locate the method,* `NAME0`: Starting at `class C0`, locate the *best matching method* with the name, `NAME0`. (The precise definition of "best matching method" is given below.) The best matching method will have its own `METHOD_HEADER`, and say that the header line has this form:

```
VISIBILITY TYPE0 NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
```

3. *Attach the header-line information to the invocation*: The Java compiler attaches the data-type names, `TYPE1, TYPE2, ..., TYPEn`, to the actual parameters, `EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn`, for use when the method is executed by the Java Virtual Machine.

   If the located method, `NAME0`, is a private method (that is, `VISIBILITY` is `private`) and was found within `class Ck`, then the label, `private Ck`, is attached to the invocation as well. If `NAME0` is a public method, then the label, `public`, is attached.

   Therefore, when `NAME0` is a public method, the compiler annotates the invocation to look like this:

```
[[ RECEIVER . ]]?  NAME0 (EXPRESSION1: TYPE1, EXPRESSION2: TYPE2,
              ..., EXPRESSIONn:TYPEn) public;
```

   When `NAME0` is a private method, the invocation looks like this:

```
[[ RECEIVER . ]]?  NAME0 (EXPRESSION1: TYPE1, EXPRESSION2: TYPE2,
              ..., EXPRESSIONn:TYPEn) private Ck;
```

4. *Return the result data type*: The overall data type of the method invocation is `TYPE0`. (If `TYPE0` is `void`, then the invocation must appear where a statement is expected.)

We will see below how the compiler's annotations are used to locate and execute an invoked method.

Here is an example. Given this class,

```
public class Test
{ public static void main(String[] args)
  { FourEggsWriter w = new FourEggsWriter();  // see Figure 2
    String s = setTitle(2);
    w.setTitle(s);
  }

  private static String setTitle(int i)
  { return ("New " + i); }
}
```

the compiler must analyze the invocation, `setTitle(2)`. Since there is no prefix on the invocation, the data type of the receiver is this class, `Test`. A search of `class Test` locates the private method, `setTitle`, whose formal parameter's type matches the actual parameter type. This is the best matching method. The compiler annotate the invocation to read,

```
setTitle(2: int) private Test;
```

and it notes that the result will be a `String`, which is acceptable.

Next, the compiler analyzes `w.setTitle(s)`. The prefix, `w`, has type `FourEggsWriter`, so that class is searched for a public method named `setTitle`. None is found, but since `class FourEggsWriter extends JFrame`, the compiler searches `class JFrame` and finds the best matching method there, `JFrame`'s `setTitle`. The invocation is annotated as

```
w.setTitle(s: String) public;
```

Definition of Best Matching Method

As the two previous examples indicate, the "best matching method" for an invocation is the method, `NAME0`, whose `n` formal parameters have data types that match the data types of the `n` actual parameters. But we should state this more precisely, if only because the concept is extended in Chapter 9.

Here is a more precise definition:

Again, let

```
[[ RECEIVER . ]]?  NAME0 ( EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn)
```

be the invocation that is analyzed; we assume that only non-static methods are invoked. Let `C0` be the data type of the `RECEIVER`. (If `RECEIVER` is omitted, we take `C0` to be the name of the class where the invocation appears.) If the invocation appears within `class C0` also, then the *best matching method* is the method the compiler finds by `searching(all public and private methods of class C0)`; Otherwise, the best matching method is the method found by `searching(all public methods of class C0)`.

The algorithm for `searching(some methods of class C0)` is defined as follows:

1. Within `some methods of class C0`, find the method whose header line has the name and parameters,

   ```
   NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
   ```

   such that

   - There are exactly `n` formal parameters to match the `n` actual parameters.

   - The data type of each actual parameter, `EXPRESSIONi`, is a (sub)type of `TYPEi`, the data type of the corresponding formal parameter, `NAMEi`.

   If the method is found, the search is successful.

2. If there is no such method, `NAME0`, that fits these criteria, and if `class C0 extends C1`, then the best matching method comes from `searching(all public methods of class C1)`. But if `class C0` extends no other class, there is no best matching method, and the search fails.

We return to the earlier example; consider

```
w.setTitle(s);
```

The compiler finds the best matching method for the invocation; since the prefix, `w`, has type `FourEggsWriter`, but the invocation appears within `class Test`, the compiler does `searching(all public methods of class FourEggsWriter)`. The compiler finds no method in `FourEggsWriter` that matches, but since `FourEggsWriter extends JFrame`, it does `searching(all public methods of class JFrame)`. There, the compiler locates the method; its header line is

```
public void setTitle(String title)
```

The above definition of best matching method will serve us well until we encounter the difficult issue of method overloading based on parameter type; this is studied in Chapter 9.

Executing Method Invocations

Finally, we consider the execution semantics of method invocation. Thanks to the compiler's annotations, the invocation has this form:

```
[[ RECEIVER ]]?  NAME0(EXPRESSION1: TYPE1, EXPRESSION2:TYPE2,
                        ..., EXPRESSIONn:TYPEn) SUFFIX
```

(Recall that `SUFFIX` is either `private Ck` or `public`.) For simplicity, we consider invocations of only normal (non-static) methods.

Execution proceeds in the following steps:

1. *Calculate the address of the receiver object:* Compute `RECEIVER` to get the address of an object. (If `RECEIVER` is absent, use the address of the object in which this invocation is executing.) Call the address, `a`. Let `C0` be the run-time data type that is attached to the object at address `a`.

2. *Select the method in the receiver:*

   - If the `SUFFIX` is `private Ck`, select the private method from `class Ck`, whose header line has the form:

```
private ... NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
```

This method must exist for object `a`, because the Java compiler located it during the type-checking phase.

- If the `SUFFIX` is `public`, then use data type `C0` to search for the method: Starting with the public methods from `class C0`, search for a method whose header line has the form,

  ```
  public ... NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
  ```

  If the method is found, select it. Otherwise, search the public methods that come from `class C1`, where `class C0 extends C1`; repeat the search until the method is found. The method *must* be found because the compiler found it during type checking.

3. *Evaluate the actual parameters:* Each of `EXPRESSION1`, `EXPRESSION2`, ..., `EXPRESSIONn` is evaluated in turn to its result. Call the results `r1`, `r2`, ..., `rn`.

4. *Bind the parameters:* Variables are created with the names `TYPE1 NAME1`, `TYPE2 NAME2`, ..., `TYPEn NAMEn`, and these cells are assigned the values `r1`, `r2`, ..., `rn`, respectively.

5. *Execute the method:* The statements in the body of method `NAME0` are executed with the new variables. Any reference to a name, `NAMEi`, is evaluated as a lookup into the cell for variable `NAMEi`. Execution of the body terminates when the end of the body is reached or when a statement of the form `return EXPRESSION` is encountered: the `EXPRESSION` is evaluated to its result, `r`, and `r` is returned as the result.

Steps 4 and 5 are the same as as executing this code inside object `PREFIX`:

```
{ TYPE1 NAME1 = r1;
  TYPE2 NAME2 = r2;
   ...
  TYPEn NAMEn = rn;
  BODY
}
```

Step 2 is necessary so that there is no confusion when a programmer uses the same name, `p`, for a private method in one class and a public method in another class. Also, the "searching" that one does to select a public method is necessary to resolve overridden methods. (For example, `class JFrame` has a useless `paint` method, but in Figure 2 `class FourEggsWriter` has a newer `paint` method. When we create a `FourEggsWriter` object, it has two `paint` methods, but the newer one is always selected.)

Static Methods

A static method can be invoked by making the `RECEIVER` part of an invocation, `RECEIVER NAME0( ... )` be the name of a class, e.g., `Math.sqrt(2)`. Or, if the `RECEIVER` part is omitted, and the invocation appears in the body of a static method, then the invocation is treated as an invocation of a static method.

When the Java compiler encounters an invocation of a static method, it searches for the best matching method *only amongst the static methods.* As usual, the search for the best matching method begins at the class named by the `RECEIVER`. When the Java compiler locates the best matching static method, it uses the information in the method's header line to annotate the invocation as seen above. The `SUFFIX` part of the annotation is `static Ck`, where `Ck` is the class where the best matching method was located.

When the Java Virtual Machine executes the invocation, the suffix, `static Ck` tells it to execute the static method in `Ck`.

5.9.6 Revised Syntax and Semantics of Classes

Classes now have fields and constructor methods, so here are the augmentations to the "Syntax" and "Semantics" sections in Chapter 2.

The syntax of a class becomes

```
public class IDENTIFIER EXTEND?
           { FIELD*
             CONSTRUCTOR
             METHOD*
           }
EXTEND ::=  extends IDENTIFIER
FIELD ::=  private DECLARATION
CONSTRUCTOR ::=  public METHOD_HEADER
                 { STATEMENT* }
```

Data-type checking is performed on the fields, constructor method, and methods of the class. No two fields may be declared with the same name. Two methods *may* be declared with the same name if they have different quantities or data types of formal parameters; this issue is explored in Chapter 9.

The semantics of a class is that it is remembered for creating objects.

An object is constructed from a class, `class C0`, by an expression of the form

```
new C0( EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn )
```

The semantics of object construction goes as follows:

1. *Construct an object in computer storage:* `class C0` is located, and copy of its components are allocated in storage. If `C0` extends another class, `C1`, then `C1`'s components are copied into the object as well.

2. *Execute all the field declarations.*
3. *Execute the constructor method*: The phrase, `C0(EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn)`, is executed like a method invocation, as described in the previous section.
4. *Remove all constructor methods from the object*—they are no longer needed.
5. *Return as the result the address of the object.*