1 Objects

This chapter presents an overview of object-oriented programming in Java. Sections 1.1 through 1.4 show you how to control an object that can draw pictures. This object is called a turtle because it is based on the Logo programming language, in which a turtle icon does the drawing.

You will learn to create turtle objects and to send them messages to take actions. A turtle understands only eight elementary kinds of messages, but Java lets you teach the turtle new messages that are combinations of existing ones. You supply the artistic ability by deciding which messages to send in what order; the turtle carries out your requests.

The real purpose of the first part of this chapter, however, is to give you the opportunity to write programs that will impress your friends and relatives. Naturally, you cannot expect to be able to create an interesting program from scratch until you have been studying computer science for several weeks. But you can download the three-page turtle software from this book's website or type it in from the listings in Chapter Eight. Then, with the help of these turtle objects, you can create a program that draws complex pictures. When your friends and relatives ask you what you have learned to do in the course, you will have something good to show them by the end of your first week.

In this context, you will learn to define executable Java programs and to define instance methods without parameters or return values. Section 1.5 explains in detail how to compile and execute your programs in the terminal window.

Later sections give examples of programs using other kinds of objects. One is a program that sends messages to portfolio objects to perform financial-market tasks. Another is a program that sends messages to other kinds of objects to perform personnel database tasks. We even include a very simple program developed in Java from scratch, using only facilities that come with every Java compiler. This foretaste requires showing you language features whose full explanation is in Chapters Two through Four, so you are not responsible for remembering these features at this point.

1.1 Using Turtle Objects To Draw Pictures

Turtle objects are derived from the Logo programming language, which has been around for decades. When you run a Java program that uses Turtle objects, you have a drawing surface on which a Turtle is positioned, initially facing due East. The Turtle is carrying a bucket of paint and a paint brush (too tiny for you to see on the screen). The Turtle moves one little Turtle step at a time (these steps are also known as **pixels**). The three most useful requests that a Turtle understands are named `paint`, `move`, and `swingAround`. Examples of their use are as follows:

- `move(45,20)` sends a message that causes the Turtle to turn 45 degrees to its left and then walk 20 little Turtle steps, without leaving any marks. This message is used to move the Turtle from one position to another on the drawing surface.
- `paint(90,30)` sends a message that causes the Turtle to turn 90 degrees to its left and then walk 30 little Turtle steps, dragging its paintbrush behind it. This message is used to draw a straight line on the drawing surface, going off at the specified angle from the current position.
- `paint(-60,50)` sends a message that causes the Turtle to turn 60 degrees to its right and then walk 50 little Turtle steps, drawing a line as it goes. That is, a negative number for the angle causes a right turn; a positive number for the angle causes a left turn; and zero for the angle causes no turn at all. This applies to `move` as well.

- `swingAround(100)` sends a message that causes the Turtle to put the paintbrush on a rope and swing it around its head, thereby drawing a circle. The length of the rope is given as 100 Turtle steps (pixels). So the Turtle paints a circle of radius 100 with itself at the center.

Objects in Turtle programs

The programs to control the computer chip are loaded from a permanent storage space, such as a hard disk, into RAM (Random Access Memory). RAM storage gives very fast access to the programs and the data they use. Each time you run the program, it loads from permanent storage again and starts afresh.

A program that uses Turtles must create an internal description of each Turtle and put that data in RAM. This internal description is an **object**; it records the Turtle's current position and heading. The phrase `new Turtle()` in a program creates the Turtle object.

The phrase `Turtle sam` in a program sets aside a part of the RAM's data area, called a **variable**, which can refer to a Turtle object (the internal description of a drawing instrument). The phrase `Turtle sam` also **declares** that `sam` is the name for that variable. It is difficult to tell a particular Turtle to carry out some action if the Turtle does not have a name you can use. So Turtle programs often contain a sequence such as the following, which lets later parts of the program refer to the Turtle as `sam`:

```
Turtle sam;
sam = new Turtle();
```

When the Turtle is first created, it is in the center of the drawing surface, facing due East and carrying a can of black paint. The drawing surface is 760 pixels wide and 600 pixels tall (because some computer monitors are not much larger than that in size). Drawings outside of that range will not appear on the drawing surface.

Illustration of basic Turtle messages

Figure 1.1 shows how the drawing looks after a newly-created Turtle receives the three messages `paint (90, 7); move (0, 2); paint (0, 2);` in that order. The tiny figure indicates the position of the Turtle, and the Turtle's head indicates its heading. Each `paint` message colors in the pixel the Turtle starts on but not the one it ends up on.

If for instance `sue` refers to some Turtle, `sue.move(0,7)` is what you write in a Java program to tell that Turtle to move 7 little Turtle steps straight ahead (without first turning), and `sue.paint(180,30)` is what you write to tell that Turtle to turn completely around and then draw a line 30 little Turtle steps long. Later in this chapter you learn Turtle messages that write words and change the color of parts of the drawing. But we keep things simple for now with just these three kinds of messages.

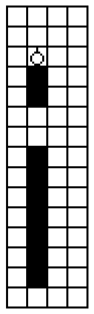
Figure 1.1 the effect of paint(90,7); move(0,2); paint(0,2);

Exercise 1.1 Write a sequence of messages that causes a Turtle named `sue` to draw a rectangle twice as wide as it is tall, with a height of 60 pixels.
Exercise 1.2 Write a sequence of messages that causes a Turtle named `sam` to draw a lowercase letter 'r' 12 pixels tall and 8 pixels wide. Include the angled part of the 'r'.
Answers to these exercises are at the end of this chapter.

1.2 A Complete Java Application Program Using Turtle Methods

You write a program to perform a complex task by breaking the task down into a combination of simple actions. An instruction to a Turtle to take an action is a **command**. In a program, you need a semicolon at the end of each command in a sequence of commands. The command and semicolon together are called a **statement**.

Suppose you want a Turtle to draw the capital letter 'H', 12 pixels tall and 6 pixels wide, with a circle around it. The following sequence of statements can do this. The remark at the right of each command explains the meaning of the command:

```
   Turtle sam;             // declare the variable named sam
   sam = new Turtle();     // create the object sam refers to
   sam.paint (90, 12);     // draw the left side of the H
   sam.move (-180, 6);     // return to the center of the H
   sam.paint (90, 6);      // draw the crossbar of the H
   sam.move (90, -6);      // move to the bottom of the right side
   sam.paint (0, 12);      // draw the right side of the H
   sam.move (150, 6);      // go to just above the center of the H
   sam.swingAround (9);    // draw a circle enclosing the H
```

The `//` symbol in a program indicates a **comment**. That symbol is a signal that everything on the rest of its line is to be ignored. Comments are only for humans to read; they do not affect the operation of a program.

The structure of an application program

The preceding sequence of nine statements describes a method for accomplishing a task. Before you can have the computer follow this method of doing things, you have to give the method structure. In Java, the way you do this is to put those statements between matching left and right **braces** `{` and `}` and put the following **heading** above them:

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

You have then constructed a **main method**. The words in the heading of the main method have meanings that will be explained in the next section. This section tells you what you do to make a program; the next section tells you why you do it. That way, you have an overview of the entire process before going into the details.

Some programs need hundreds of methods in order to do what they need to do. It would be very difficult to keep track of all of them, their meanings and relationships, if they were not organized in some reasonable way. The primary organizing unit in Java is called a **class**. The general idea is, you collect several methods together that are very closely related to each other and put them in a single class. For instance, `swingAround`, `paint`, `move`, and others are collected in the Turtle class. This is in line with the meaning of the word "class" as a number of things that have common attributes.

In Java, the normal way you signal that a number of methods belong to a particular class is to put the method(s) between matching left and right braces `{` and `}` and put a heading like the following above them, although you have a free choice of the name you use in place of SomeClass:

```
   public class SomeClass
```

A class containing a main method is usually called an **application program**. This book does not put any method in an application program other than the main method. Listing 1.1 shows a complete Java application program using the preceding sequence of nine statements to draw a letter 'H'.

Listing 1.1 An application program using a Turtle object

```
public class ProgramOne
{
   // Draw an uppercase letter 'H', 12 pixels tall and 6 wide.
   // Put a circle around the outside of the 'H'.

   public static void main (String[ ] args)
   {  Turtle sam;              // create the variable named sam
      sam = new Turtle();      // create the object sam refers to
      sam.paint (90, 12);      // draw the left side of the H
      sam.move (-180, 6);
      sam.paint (90, 6);       // draw the crossbar of the H
      sam.move (90, -6);
      sam.paint (0, 12);       // draw the right side of the H
      sam.move (150, 6);
      sam.swingAround (9);     // draw a circle enclosing the H
   }  // this right-brace marks the end of the main method
}     // this right-brace marks the end of the class
```

Programming Style Comments (the parts after the `//` symbols) are optional in a program, but you should always have at least a comment before the main method describing the purpose of the program. You should also make sure that each right brace is lined up vertically with the corresponding left brace, and that everything between the braces is indented by one tab position.

The result of executing ProgramOne is shown in Figure 1.2. Note that the capital 'H' actually spans a total of seven pixels horizontally, since `paint` colors in the pixel the Turtle starts on but not the pixel the Turtle ends on. The Turtle finishes just above the middle of the 'H' facing south-south-west, indicated by the tiny circle.

Edit, compile, and run

First you type the lines of this program into a plain-text file named `ProgramOne.java` (because ProgramOne is the name of the class) using a word processor. The next thing you do is submit this text file named `ProgramOne.java` to the **compiler** program to see if it is correctly expressed in the Java language. On the simplest systems, you do this by entering `javac ProgramOne.java` at the prompt in a **terminal window** (a window that only allows plain text input and output; often called the DOS window). However, you cannot compile a program that uses Turtles without first compiling this textbook's `Turtle.java` file (details in Section 1.5).

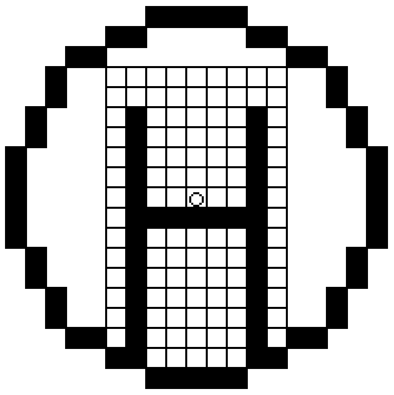
Figure 1.2 After execution of Listing 1.1

If the compiler does not detect any errors when you have it check out your program, it translates your text file to an executable file named `ProgramOne.class`. It does not translate anything after the `//` symbol on a line.

You then run the program by entering `java ProgramOne` at the prompt in the terminal window. This tells the **runtime system** to carry out the commands in the executable file. You also need a compiled form of the Turtle class, which is available on this book's website (or you could type it in from Section 8.11). This Turtle class lets you test the programs you write.

Caution You must be careful in programs to capitalize letters in words exactly as shown. The compiler program sees `paint` and `Paint` and `PAINT` all as three totally unrelated words, i.e., Java is case-sensitive. Of those three words, Turtles only understand `paint`.

If you replaced `sam` by `sue` in every statement of Listing 1.1, it would make no difference in the effect of the program. The choice of the name `sam` for the Turtle object is arbitrary, as long as you spell and capitalize it the same way throughout your program.

Application program to draw two squares

If you want a Turtle to draw two squares side by side, you could have the runtime system execute the program in Listing 1.2. The first two statements create the Turtle object and position it in the center of the drawing area facing East. They also make `sue` contain a reference to that Turtle object.

Listing 1.2 An application program using a Turtle object

```
public class TwoSquares
{
   // Draw two 40x40 squares side by side, 10 pixels apart.

   public static void main (String[ ] args)
   {  Turtle sue;
      sue = new Turtle();

      sue.paint (90, 40);     // draw the right side of square #1
      sue.paint (90, 40);     // draw the top of square #1
      sue.paint (90, 40);     // draw the left side of square #1
      sue.paint (90, 40);     // draw the bottom of square #1

      sue.move (0, 50);       // move 50 pixels to the right
      sue.paint (90, 40);     // draw the right side of square #2
      sue.paint (90, 40);     // draw the top of square #2
      sue.paint (90, 40);     // draw the left side of square #2
      sue.paint (90, 40);     // draw the bottom of square #2
   }  //======================
}
```

After the Turtle object is created, the next four statements draw the first square, 40 pixels on a side. The last five statements of the program draw the second square of the same size, 10 pixels away from the first square. Figure 1.3 shows the status of the Turtle object and the drawing after execution of this program.

The order of the commands is important: You cannot send a message to `sue` to perform an action before you store a Turtle object in the variable named `sue` (statement #2), and you cannot do that before you declare that variable (statement #1).

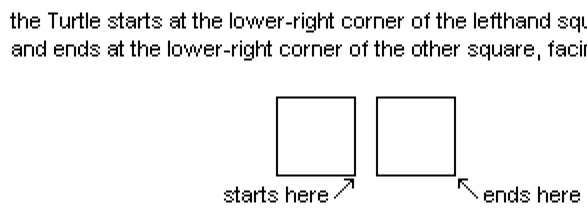
the Turtle starts at the lower-right corner of the lefthand square, facing east, and ends at the lower-right corner of the other square, facing east.

Figure 1.3 After execution of Listing 1.2

The meaning of some program elements

You are surely wondering why you need `public` and `void` and all the rest to make a simple program. Each of the parts of a Java program has a purpose. The following paragraphs, while not a full explanation, should give you some idea of what the purpose is. These paragraphs also preview what you will see in the first half of this book.

A **method** is so called because it describes the method by which some objective is achieved. For instance, the main method in Listing 1.2 describes a method of drawing two squares.

Question: Why the matched pair of braces? *Answer:* They tell the compiler where a class begins and ends and where a method begins and ends. This is needed because you can have more than one class within a file and you can have more than one method within a class. You will see an example of the latter in the next section.

Question: Why must one declare `Turtle sue` when the next command `sue = new Turtle()` makes it quite clear that `sue` is a Turtle variable? *Answer:* You will sometimes misspell a name. If the compiler were to accept the misspelled name as a different variable, that could cause hard-to-find errors in your programs. But because Java requires you to explicitly declare all names, you have little trouble finding misspellings; the compiler points them out to you.

Question: Why the word `public` in the headings? *Answer:* An alternative is `private`. If the main method were private, it could not be used by anything outside the class. The terminal window is outside the class. So the runtime system cannot execute the main method from the terminal window unless the main method is public. Similarly, the class should be public instead of private so it can be used by anything outside the class. You will see private methods in Chapter Three.

Question: Why the word `static` in the method heading? *Answer:* When a method heading does not include `static`, the compiler will not let you use the method unless you first create an object to send the message to. At the time the `java` command in the terminal window executes the main method, the runtime system has not yet created any object to send the message to. So the main method must be marked `static`. You will see other kinds of methods marked `static` in Chapter Five.

Question: Why the word `void` in the method heading? *Answer:* When you send a message, sometimes you get an answer back to use later in the program, and sometimes you do not. The word `void` signals that no answer will be sent back by this particular method. Since there is no "later in the program" after the main method is executed, the main method should be marked `void`. You will see non-void methods in Chapter Two.

Question: Why "void" instead of say "noAnswerGiven"? Why "main" instead of "programStartsHere"? Why semicolons instead of commas? *Answer:* The designers of Java decided that, where the choice was quite arbitrary, they would use the symbols and signals from the C programming language, because most professional programmers are familiar with it.

Question: Why the `(String[ ] args)` part? *Answer:* It can be used to get the user's input (though it does not do so in this particular program). For instance, you may have a Euchre-playing program you start by entering the basic `java Euchre` command in the terminal window followed by two extra words. You can start it by entering `java Euchre 4 English`; it then allows four players and communicates in English. Or you can start it by entering `java Euchre 3 French`, so it allows three players speaking French. The runtime system uses the `(String[ ] args)` part of the main heading to send those two extra pieces of information to the main method.

For now, it would not hurt to treat that phrase as just one of those things you have to have to make things work right in Java. It is like the "ne" in the French phrase "ne pouvez pas"; the "pas" means "not", but for some reason you have to tack on a "ne" to be speaking correct French.

Exercise 1.3 Write an application program that creates one Turtle and has it draw a lowercase 'b' 6 pixels wide and 12 pixels tall, without going over the same pixel twice.
Exercise 1.4 Write an application program that creates one Turtle and has it draw a lowercase 'm' 9 pixels wide and 7 pixels tall.
Exercise 1.5 Write an application program that creates one Turtle and has it draw a hexagon 50 pixels on a side.
Exercise 1.6* Write an application program that creates one Turtle and has it draw a lowercase 'g' 6 pixels wide and 6 pixels tall, descending 3 pixels below the baseline, without going over the same pixel twice.
Exercise 1.7* Write an application program that creates one Turtle and has it draw a house 200 pixels wide and 150 pixels tall, with a door and two windows.
Note: A star on an exercise means the answer is <u>not</u> in the book. Unstarred exercises have the answers at the end of the corresponding chapter.

1.3 A First Look At Inheritance: Defining Instance Methods In Turtle Subclasses

You can expect to draw a square in several different programs, with various Turtle objects receiving that sequence of four `paint(90,40)` messages. This was done twice for `sue` in Listing 1.2. Fortunately, you can invent new messages for the Turtle that are combinations of existing messages. This will simplify your programs.

For instance, you can define a new message named `makeBigSquare`:
`sue.makeBigSquare()` tells `sue` to execute those four `paint(90,40)` actions, and
`sam.makeBigSquare()` tells `sam` to execute those four `paint(90,40)` actions.

You may also want to draw a small square in several different situations. The sequence of four messages

```
   paint (90, 10);
   paint (90, 10);
   paint (90, 10);
   paint (90, 10);
```

could be quite common, sent to various Turtle objects. You can define a new message named `makeSmallSquare`: `sam.makeSmallSquare()` tells `sam` to carry out those four `paint(90,10)` actions, and `sue.makeSmallSquare()` tells `sue` to carry out those four `paint(90,10)` actions.

A simple Turtle object does not know the meaning of the words `makeBigSquare` and `makeSmallSquare`. They are not part of its vocabulary. You need a new class of objects that can understand these two messages plus all the messages a Turtle understands. Let us call this new kind of Turtle object a SmartTurtle. Then you could rewrite the main method in Listing 1.2 to do exactly the same thing but with a simpler sequence of statements, as follows:

```
   public static void main (String[ ] args)
   {  SmartTurtle sam;
      sam = new SmartTurtle();
      sam.makeBigSquare();
      sam.move (0, 50);
      sam.makeBigSquare();
   }  //======================
```

How to define a class of objects

You may define a class that provides new messages and a new kind of object that understands those messages. The class definition in Listing 1.3 says that, if you create an object using the phrase `new SmartTurtle()` instead of `new Turtle()`, that object will understand the `makeBigSquare` and `makeSmallSquare` messages as well as all of the usual Turtle messages. In a sense, a SmartTurtle object is better educated than a basic Turtle object.

Listing 1.3 The SmartTurtle class of objects

```
public class SmartTurtle extends Turtle
{
   // Make a 10x10 square; finish with the same position/heading.

   public void makeSmallSquare()
   {  paint (90, 10);
      paint (90, 10);
      paint (90, 10);
      paint (90, 10);
   }  //======================

   // Make a 40x40 square; finish with the same position/heading.

   public void makeBigSquare()
   {  paint (90, 40);
      paint (90, 40);
      paint (90, 40);
      paint (90, 40);
   }  //======================
}
```

A class definition that extends the capabilities of a Turtle object must have the heading

```
   public class WhateverNameYouChoose extends Turtle
```

followed by a matched pair of braces that contain some definitions. The SmartTurtle class definition contains two parts beginning `public void`. These two parts are **method definitions**. The names chosen here are `makeBigSquare` and `makeSmallSquare`, but they could be anything you choose. Just be sure that, when you send these messages to an object, as in `sam.makeBigSquare()` or `sue.makeSmallSquare()`, you always spell them the way the method definition shows, including capitalization, and finish with the empty pair of parentheses.

Each of the two method definitions in the SmartTurtle class describes one new message in terms of previously-known messages. In the heading of the method definition, `public` means that the statements in any class can send these methods to its SmartTurtles. `void` means that these are definitions of actions to be taken instead of questions to be answered; you will see how to use and define questions in the next chapter. The comments following the right braces (beginning with `//`) are dividers that help visually separate the various method definitions within the class definition.

Within the definition of this kind of method, you leave out the name of the variable that refers to the object that receives the message. This lets you use any variable name you like (such as `sue`, `sam`, or whomever) when you write the command outside the method. So the `makeSmallSquare` definition says that for any `x`, if you have previously defined `x = new SmartTurtle()`, then `x.makeSmallSquare()` has the same meaning as:

```
   x.paint (90, 10);
   x.paint (90, 10);
   x.paint (90, 10);
   x.paint (90, 10);
```

You can read the second method definition verbally as follows: Define an action method named `makeBigSquare` that any SmartTurtle object in any class can be asked to carry out. Using this method sends a message asking the object to draw the four sides of a square 40 pixels on a side, ending up with the same position and heading as at the start.

Programming Style You will find it much easier to understand a class definition you have written if you mark the end of each method with a comment that stands out clearly. This book puts "===========" at the end of each method definition. Some people prefer the phrase "End of method" or something more specific such as "End of makeBigSquare".

Executors and instances

An object created by `new Turtle()` is called an **instance** of the Turtle class, and an object created by `new SmartTurtle()` is called an **instance** of the SmartTurtle class.

You cannot use either of the two SmartTurtle methods of Listing 1.3 in an application program without mentioning an instance of the SmartTurtle class in front of the method name, separated from it by a dot (period). So these two methods are called **instance methods**. The absence of the word `static` in the heading signals this restriction -- a main method is not an instance method.

A command of the form `someObject.someMessage()` is a **method call**. We say that the object referenced before the dot is the **executor** of the method, since it executes the commands in the method. Every call of an instance method requires an executor.

This vocabulary applies to Turtle methods as well as to SmartTurtle methods. So `sue` refers to the executor in the method call `sue.swingAround(30)`. All three of `paint`, `move`, and `swingAround` require an executor when they are used. So these three are instance methods of the Turtle class. [Technical Note: Java has no official terminology for what this book calls the "executor"; other names some people use are "target object", "receiver", and "implicit parameter".]

The object that executes the `paint(90, 40)` commands inside the definition of `makeBigSquare` **defaults** to the executor of the `makeBigSquare` method call, since the executor of those commands is not explicitly stated. By contrast, you cannot call an instance method from the main method without stating its executor, because the main method itself has no executor to default to. The word `static` in the heading of the main method signals that it has no executor.

Listing 1.4 illustrates the use of these new SmartTurtle commands in a program that makes an X-shaped pattern of one large square with four small squares at its four corners. Figure 1.4 shows the result of running this program.

Listing 1.4 An application program using a SmartTurtle object

```
public class SquarePattern
{
   // Make an X shape with one big 40x40 square in the center
   // and a small 10x10 square in each corner.

   public static void main (String[ ] args)
   {  SmartTurtle sue;
      sue = new SmartTurtle();
      sue.makeBigSquare();       // draw the center square

      sue.move (-90, 15);        // go south
      sue.move (90, 15);         // move to the southeast corner
      sue.makeSmallSquare();     // draw the southeast square

      sue.move (90, 70);         // move to the northeast corner
      sue.makeSmallSquare();     // draw the northeast square

      sue.move (90, 70);         // move to the northwest corner
      sue.makeSmallSquare();     // draw the northwest square

      sue.move (90, 70);         // move to the southwest corner
      sue.makeSmallSquare();     // draw the southwest square
   }  //======================
}
```

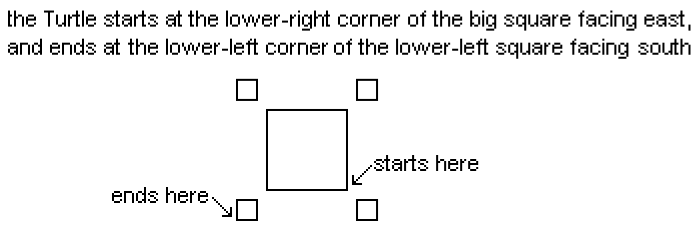
the Turtle starts at the lower-right corner of the big square facing east,
and ends at the lower-left corner of the lower-left square facing south
starts here
ends here

Figure 1.4 After execution of SquarePattern

Inheritance from the superclass

The SmartTurtle class in the earlier Listing 1.3 contains the `makeBigSquare` method definition and the `makeSmallSquare` method definition explicitly. The phrase `extends Turtle` in the heading of the class definition means the SmartTurtle class indirectly contains all the public method definitions it gets from the Turtle class. We say that SmartTurtle objects **inherit** the Turtle methods (`paint`, `move`, etc.).

Inheritance is what allows any SmartTurtle object to use all of the regular Turtle methods in addition to those directly defined in the SmartTurtle class. In short, a SmartTurtle is a kind of Turtle. Turtle is the **superclass** and SmartTurtle is the **subclass** in this inheritance relationship. You will see examples of inheritance in most of the chapters of this book.

When you decide what methods you need in a subclass of Turtle to accomplish a task, you are doing **object design**. Designing useful objects "from scratch" requires knowing a large number of language features, so it must wait until Chapter Four. Until then, our object classes will extend interesting classes such as Turtle.

"Object orientation, involving encapsulation, inheritance, polymorphism, and abstraction, is an important approach in programming and program design. It is widely accepted and used in industry and is growing in popularity in the first and second college-level programming courses. It facilitates the reuse of program components and the management of program complexity, allowing large and complex programs to be written more effectively and efficiently and with more confidence in their correctness than with the more traditional purely procedural approach." [AP Computer Science Ad Hoc Committee Recommendations, October 2000]

Programming Style It is good style to use indentation and spacing in a program to make the relationship of one line of the program to another clear. The convention in Java is to indent one tab position at each line inside the braces of a class definition. Indent another tab position at each line that is also inside a method definition. If you want to separate statements into groups that do separate tasks, do not use indentation to do so. Instead, use a blank line between the groups, as shown in Listing 1.4.

Exercise 1.8 Write an application program that uses a SmartTurtle to draw two large squares side by side, each with a small square centered inside it.
Exercise 1.9 Write a new instance method to be added to the SmartTurtle class: It has the executor carry out the last two commands of Listing 1.4. Then rewrite Listing 1.4 to call this new method three times, thereby shortening the logic.
Exercise 1.10 Write a `drawHexagon` instance method to be added to the SmartTurtle class: The executor draws a hexagon 30 pixels on a side in just six statements.
Exercise 1.11 Write an application program that uses a SmartTurtle object, as changed by the preceding exercise, to draw three hexagons such that any two of them meet along one side.
Exercise 1.12* Write a StarTurtle class with two instance methods: One draws a five-point star (hint: turn 144 degrees) and one draws a six-point star (two overlapping equilateral triangles with symmetry). Make them 60 pixels per line segment.
Exercise 1.13* Write an application program that uses a StarTurtle object, as defined in the preceding exercise, to make an interesting drawing with at least five stars in it.
Exercise 1.14* Write an application program that draws ten big squares in the same arrangement as the setup for bowling pins. Use a SmartTurtle object.

1.4 Additional Turtle Methods; Identifiers Versus Keywords

The Turtle carries ten cans of paint of various colors, not just one. If you want to switch to e.g. red, you can do so with the following message:

```
   sam.switchTo (Turtle.RED).
```

Thereafter, all drawings are made in red (until you switch to another color). The ten available colors are BLACK, GRAY, BLUE, GREEN, RED, YELLOW, ORANGE, PINK, MAGENTA, and WHITE. You use `Turtle.WHITE` when you want to erase something you drew earlier; this helps you do animations.

You have to spell the names of the colors entirely in capitals, because that is how they are defined in the Turtle class. You put "Turtle" in front of each color name so that the compiler knows to look for the definition of the color name in the Turtle class. But within an instance method in a subclass of Turtle, you can use the color names without the class name, just as you can use method names without mentioning the executor.

The Turtle class has four more kinds of messages that you can send to a Turtle object:

- `fillCircle(80)` is the same as `swingAround(80)` except that the circle is completely filled with whatever the current drawing color is.
- `fillBox(20,90)` draws a rectangle of width 20 and height 90 with the Turtle at the center, and fills it in with whatever the current drawing color is.
- `say("whatever")` prints what you have in quotes at the Turtle's current location.
- `sleep(70)` causes the Turtle to stop what it is doing for 70 milliseconds (0.070 seconds). This command lets you control the speed of animations.

These new methods do not change the position and heading of the object. If a number in parentheses is not positive, the methods simply do nothing. By contrast, the `move` and `paint` methods do something useful when the numbers within their parentheses are negative or zero: A negative angle indicates a turn to the right, and a negative distance indicates a Turtle walking backwards.

A program using a FlowerMaker

Listing 1.5 contains an application program that creates a FlowerMaker kind of Turtle and has it draw six flowers in a row, centered on the drawing surface. A FlowerMaker is capable of drawing two flowers next to each other when you ask it to, 60 pixels apart. After each pair of flowers the Turtle pauses for 300 milliseconds (0.3 seconds), because that makes the drawing a little more interesting (hopefully). Then it prints a message above the row of flowers. Figure 1.5 shows the result of executing this program.

Listing 1.5 An application program using most of the new Turtle methods

```java
public class GardenApp
{
   // Draw 6 flowers all in a row, with a word title.

   public static void main (String[ ] args)
   {  FlowerMaker florist;
      florist = new FlowerMaker();
      florist.drawTwoFlowers();    // the central two
      florist.sleep (300);

      florist.move (0, 120);
      florist.drawTwoFlowers();    // the two right of center
      florist.sleep (300);

      florist.move (0, -240);
      florist.drawTwoFlowers();    // the two left of center
      florist.sleep (300);

      florist.move (40, 130);
      florist.switchTo (Turtle.BLUE);
      florist.say ("My flower garden");  // above the flowers
   }  //======================
}
```

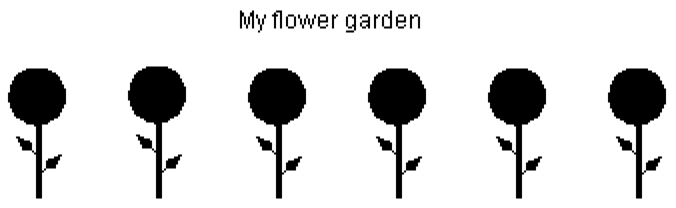

Figure 1.5 After execution of GardenApp

The FlowerMaker class

The class definition in Listing 1.6 says that objects declared as FlowerMakers have the ability to understand the three new messages there in addition to all Turtle messages. The `drawTwoFlowers` method in the top part of Listing 1.6 draws one flower, moves to the right 60 pixels, draws a second flower, and then returns to the original starting point and resumes the original heading. To do this right, you need to know that the `drawFlower` method leaves the Turtle one pixel to the left of where it started, facing south instead of east.

Listing 1.6 The FlowerMaker class of objects

```
public class FlowerMaker extends Turtle
{
   // Draw two flowers each 60 pixels tall.
   // Start and end facing east at the base of the left flower.

   public void drawTwoFlowers()
   {  drawFlower();
      move (90, 61);
      drawFlower();
      move (90, -59);
   }  //======================

   // Start facing east at the base of the flower, right side,
   // with the current drawing color being BLACK (for the stem).
   // End facing south at the base of the flower, center.

   public void drawFlower()
   {  paint (90, 50);    // right side of stem
      paint (90, 2);
      paint (90, 50);    // left side of stem
      paint (90, 1);
      paint (90, 10);    // one-fourth of the way up the stem
      paint (-45, 8);    // draw the twig for the right leaf
      drawLeaf();

      paint (45, 10);    // one-half of the way up the stem
      paint (45, 8);     // draw the twig for the left leaf
      drawLeaf();

      paint (-45, 30);   // to top of stem, in the center
      switchTo (RED);
      fillCircle (15);   // draw the flower petals
      switchTo (BLACK);
      move (180, 50);    // return to the base of the flower
   }  //======================

   public void drawLeaf()
   {  switchTo (GREEN);
      fillCircle (3);
      move (0, 3);
      fillCircle (2);
      move (0, 2);
      fillCircle (1);
      move (0, -13);
      switchTo (BLACK);
   }  //======================
}
```

A Turtle that executes the `drawFlower` method is initially facing east, assuming that the flower is to grow to the north, which normal flowers do. The Turtle starts by drawing a thick BLACK stem 3 pixels wide. As it makes the middle of the three strokes for the stem, it stops 10 pixels up to make a leaf off to the right, then stops again 20 pixels up to make a leaf off to the left. Then it draws a RED circle at the top of the stem to represent the flower. Finally, it returns to the base of the stem, ending up facing south.

A Turtle that executes the `drawLeaf` method draws three overlapping GREEN circles, each smaller than the one before. That will hopefully look rather like a leaf. Then it moves back to the base of the 8-pixel-long twig and switches the drawing color back to BLACK, which is what it was when the method was called. Note that the return to the base is made using `move(0,-13)` rather than `move(180,13)`, so that the Turtle's heading remains as it was initially.

The definition of `drawFlower` is in terms of `drawLeaf`, and the definition of `drawTwoFlowers` is in terms of `drawFlower`. You may define a method in any terms the object can understand. In particular, you may define a method in the FlowerMaker class as a sequence of messages any Turtle object or any FlowerMaker object can understand.

Your first thought is probably that this allows you to do something really silly, such as define `drawOne` to mean two messages of `drawTwo` and define `drawTwo` to mean two messages of `drawOne`. Yes, you can do that. If you then sent either of those messages to an object, it would cause the program to crash. It is your responsibility to avoid such silliness; objects are not smart enough to know when you make a logic error.

Two kinds of methods and classes

You have now seen two kinds of method definitions: A main method is a sequence of instructions normally initiated from the terminal window. The main method is normally called by `java Whatever` in the terminal window, where the Whatever class contains that main method. Its heading must be `public static void main (String[ ] args)` (except `args` could be any name you like). A method without `static` in the heading is an instance method, specifying a sequence of actions carried out by its executor. An instance method is typically called by putting an object value in front of the method name, separated by a dot. That object value must refer to an instance of the class the method belongs to, or a subclass of that class.

A method definition has two parts: the **method heading** (everything up to but not including the first left brace) and the **method body** (the matched pair of braces and its contents). For instance, the heading of the main method in Listing 1.5 is the line that begins with `public static` and ends with the right parenthesis; the body of that method is the thirteen statements that follow, together with the enclosing braces.

The heading `public class Whatever` signals a **class definition**. You have now seen two kinds of class definitions: An application program is a class that contains a main method. An **object class** is a class that contains one or more public instance methods, such as the SmartTurtle class or the FlowerMaker class, and no main method. These instance methods define what messages individual objects of that class can "understand". Most classes are object classes, i.e., they define a new class of object (hence the name "class").

Identifiers and keywords

The name you choose for a method, variable or class is its **identifier** (e.g., `makeLeaf`, `sam`, and `SmartTurtle`). You can use letters, digits, and underscores to form an identifier: `Flower_maker` and `Bring_3_back` are permissible names.

Caution You cannot have a blank within an identifier or have a digit as its first character. A very common mistake is to start a program with something like `public class Program Two`. A blank in the middle of the class name is not allowed.

You cannot change the spelling or capitalization of any of the **keywords** in a program, such as `public`, `class`, `extends`, `static`, `void`, and `new`. You never use capital letters for them, and you never name a method, variable, or class with one of them.

The Java developers at Sun Microsystems Inc. have developed a library of over one thousand classes for use by Java programmers. It comes with the standard installation of Java that Sun provides. It does not include the Turtle class -- that was developed for this book. We begin the serious use of the **Sun standard library** in Chapter Four, and introduce over 130 of those classes by the end of Chapter Fifteen.

The word "String" in the heading of the main method is the name of a class in the Sun standard library. The people who wrote the String class could choose whatever name they wanted; but now that they have, you have to spell it and capitalize it exactly the same way they did. Otherwise your program will not compile. Similarly, the word "Turtle" was chosen arbitrarily by the developer of the Turtle class, and you have to spell it that way to use it.

Programming Style The convention in Java is to name all classes starting with a capital letter and all methods and non-constant variables starting with a lowercase letter. It is good style to keep to that convention. You should also use **titlecase** for names -- capitalizing only the first letter of each word within a name (as in `useItNow`) -- or underscoring (as in `use_it_now`).

It is good programming style to choose a name for a method, variable, or class that conveys its meaning. Note, for instance, that the object variable in Listing 1.5 is named `florist`, so you can easily remember that it draws flowers. The names `sue` and `sam` have been used only for objects that are not particularly distinguishable.

Language elements

Beginning in Chapter Two, sections that introduce new features of the Java language usually conclude with a formal description of the new language elements. The following is an example of such a description for most of what you have seen so far in this chapter:

Language elements

A CompilableUnit can be:	public class ClassName { DeclarationGroup }	
or:	public class ClassName extends ClassName { DeclarationGroup }	
A DeclarationGroup is any number of consecutive Declarations.		
A Declaration can be:	public static void main (String [ ] args) { StatementGroup }	
or:	public void MethodName ( ) { StatementGroup }	
A StatementGroup is any number of consecutive Statements.		
A Statement can be:	ClassName VariableName ;	e.g., Turtle sam;
or:	VariableName = new ClassName ( ) ;	e.g., sam = new Turtle();
or:	VariableName . MethodName ( ) ;	e.g., sam.drawFlower();

These descriptions are compact, but they can be difficult to make sense of at times (examples of a principle often clarify the principle better than an explicit statement of the principle). We will analyze the notation used in this particular description so that you will be better able to understand similar descriptions later in this book. This notation is not used anywhere in the book except in these special descriptions and the language review at the end of Chapter Five.

Lines 1 and 2: The basic unit in a programming language is a CompilableUnit, i.e., the contents of a text file that you can submit to a compiler and have translated to an executable form. Line 1 says that one kind of CompilableUnit in Java is three words followed by material in a matched pair of braces. The three words are the two unalterable keywords "public class" followed by any name you choose for the class being defined. The material in braces is any number of Declarations you would like to have. Line 2 says that, in the class heading, you can insert just after the name of the class the word "extends" followed by the name of a superclass from which it inherits.

Lines 4 and 5: You have seen two kinds of Declaration so far, described in these two lines (you will see others later). The first line describes the structure of a main method declaration as shown in Listing 1.5. The second line describes the structure of an instance method declaration as shown in Listing 1.6; the choice of the MethodName is arbitrary for an instance method.

The notation used here should be clearer now. With the special exception of the heading of the main method:

- A word that begins with a capital letter and ends in "Name" indicates an identifier; you can have any name you choose for that category of declaration.
- A word that begins with a capital letter and does not end in "Name" indicates a language construct that is defined elsewhere.
- Any other item means that item itself must appear there.

Lines 7 through 9: Here you have descriptions of the three kinds of statements you have seen so far (there are many more). Examples are given at the right of each description. Line 7 says a statement may declare a variable to have a particular VariableName and be able to store a reference to a particular instance of the class named ClassName. Line 8 describes the general format of a statement that assigns a reference to a newly-created instance of the class named ClassName to a variable named VariableName. Line 9 describes the general format of a statement that calls a method named MethodName with an executor referred to by the variable named VariableName.

Exercise 1.15 Revise the `drawTwoFlowers` method to have the second flower not only 60 pixels to the right of the first but also 20 pixels higher than the first.
Exercise 1.16 Write a `drawSmallFlower` method to be added to the FlowerMaker class: The executor makes a tiny red flower 10 pixels tall with no leaves.
Exercise 1.17 How many statements would the `drawTwoFlowers` method have if it did exactly the same thing but did not call on any other methods outside the Turtle class (so the statements from `drawFlower` and `drawLeaf` are in `drawTwoFlowers`)?
Exercise 1.18 Write an application program that draws a target: a solid black circle inside a solid blue circle inside a solid yellow circle inside a solid red circle.
Exercise 1.19* Revise the `drawFlower` method to have four leaves on alternating sides, branching off at 10, 15, 20, and 25 pixels up the stem.
Exercise 1.20* Revise the `drawFlower` method to have six smaller circular yellow petals distributed around the outside of the red center, overlapping it somewhat.
Exercise 1.21* Write an application program that draws five circles of different colors and sizes at a variety of points on the drawing surface. Pause half a second between circles.
Exercise 1.22* Describe all the kinds of situations you have seen within a class in which you use parentheses.
Exercise 1.23** In Listing 1.5, which ones of the first twelve statements could be swapped with the statement directly following it without changing the effect of the program? List all of them that can be swapped.
Note: Double-starred exercises are the hardest ones. Their answers are not at the end of the chapter.

1.5 Compiling And Running An Application Program

This section goes into the technical details of writing, compiling, and running a program. First, make a folder on your hard disk for these Turtle programs and others you might use. Then copy all the files from this book's website or CD-ROM disk that end in `.java` into that one folder so you can use them. The following describes what you do in one common situation, using a Windows operating system and the free JDK you can download off the Internet. Your programming environment may be different. Even if so, you should be aware of what is going on behind the scenes, as described below.

You would try out ProgramOne as follows, assuming the program folder is `c:\cs1` and the JDK has been properly installed with path settings:

1. Obtain the terminal window: Under Windows, click Start, then click Programs, then click MS-DOS Prompt or Command Prompt or something equivalent. You should see a terminal window with probably the `C:\windows>` prompt.
2. Switch to the program folder: Type the command `cd C:\cs1` in the terminal window and press the Enter key. You should see the `C:\cs1>` prompt. If the programs from the book are in this folder, one of them is in a file named `Turtle.java` and another is in a file named `ProgramOne.java`.
3. Compile Turtlet: Type `javac Turtlet.java` and press the Enter key. You should see the `C:\cs1>` prompt reappear after a short wait. This indicates the compiler has translated `Turtlet.java` into `Turtlet.class`.
4. Compile Turtle: Same as Step 3 except use `javac Turtle.java`. Note: You will not need to repeat steps 3 and 4 for any additional programs that use Turtles.
5. Compile ProgramOne: Same as Step 3 except use `javac ProgramOne.java`.
6. Run that program: Type `java ProgramOne` and press the Enter key. You should see the Turtle window appear. ProgramOne from Listing 1.1 will run, drawing a capital letter 'H'.

Say you run GardenApp from Listing 1.5. The runtime system **links in** the FlowerMaker class it mentions. This causes the runtime system to link in the Turtle class FlowerMaker mentions, which causes it to link in the graphics classes that the Turtle class mentions. This is done automatically for you if all Turtle-related classes are in the same disk folder.

Computer storage

A computer has two kinds of places to store data. One is RAM, which gives very fast access to the data. However, when the computer is turned off, the data in RAM is lost. And when a program runs in a window and that window is closed, the data in RAM is lost. So RAM is called volatile memory.

The other kind of place to store data is permanent storage, such as the computer's hard disk, a floppy disk, or a CD-ROM. Data stored in these places is not lost even when the computer is turned off. However, the computer chip needs much more time to get data from permanent storage than to get it from RAM. The components of the computer (chip, RAM, disk, etc.) are **hardware**; the programs that control the computer are **software**.

Writing and running your own program

When you have worked out the logic for your own program, type it in a WordPad editor window (or any word-processor program with the spell/grammar check turned off). Save it in permanent storage, in the `c:\cs1` folder as a text file with the name `ProgramMine.java` (except change ProgramMine to whatever your class's name is). You may need quotes around the name the first time you save it; otherwise the word processor may add an extra word (e.g. ".txt") to the name that makes it uncompilable.

Once your program is saved on disk, open the terminal window alongside your editor window. Compile your program using `javac ProgramMine.java`. If the compiler detects errors, it prints descriptive messages in the terminal window. Correct them in the editor window, click Save, then compile the program again in the terminal window. Some general problems that can occur in compiling are as follows:

- Getting more than 100 error messages generally means you saved the file with all of the formatting codes that your word-processor uses for margins, fonts, etc. Go back and this time save it right, as an ordinary text file.
- "Bad command or file name" usually means your javac compiler is not properly installed including the path.
- "Can't read: X.java" usually means you have probably misspelled the name of the file, in capitalization if nothing else. The DOS command `dir p*` will list all of the files you have in the directory that start with the letter p, and similarly for other letters. Use the `dir` command to see what the true file name is.
- "Public class X must be defined in a file called X.java" means just what it says: The name of the file must match exactly the name of the class, except for the ".java" part.

Source code versus object code

When you have a successful compile (no error message before the prompt reappears), the compiler program translates the contents of your `ProgramMine.java` file, called the **source code**, into a form called the **object code**, stored on disk in a file named `ProgramMine.class`. You can then execute the object code in the file by entering `java ProgramMine`. This loads the program into RAM and begins its execution.

Downloading the JDK

If your computer system has not been set up properly for using Java, it may not recognize the `javac` or `java` command. You may download the JDK (which includes the compiler) from `http://java.sun.com/products`. Then enter the following two commands each time you open the terminal window for compiling or running (check your disk folders to see what is the exact name of the folder containing the javac program; it may not be jdk1.3.1_01):

```
set path=c:\jdk1.3.1_01\bin;%path%
set classpath=.
```

Of course, if you use a commercial software development package instead of the free JDK download, the process for writing, compiling, and running programs is different. Some other free Java development packages may be available at `http://www.bluej.org`, `http://www.realj.com`, `http://www.netbeans.org.`, or `http://www.eng.auburn.edu/grasp`.

If you install the **BlueJ** environment (details in Section 4.11), you can work with the programs in your cs1 folder as follows: Enter the BlueJ environment; click Project; click Open Non-BlueJ Project; Navigate through the folder structure until you click the cs1 folder; click Open In BlueJ. Now you have all the classes in BlueJ.

Caution The most common compiler errors beginning Java programmers make are: (a) capitalizing the "p" in `public`; (b) not capitalizing the "s" in `String`; (c) forgetting the pair of parentheses at the end of each message; and (d) forgetting the semicolon at the end of each statement. The most common non-compiler error is failure to make a backup copy of the source code files on floppy disk every hour or so.

Exercise 1.24* Find on your computer the folder that contains the file named `javac.exe`, the Java compiler.

1.6 Sending Messages To Objects

It is essential you understand the concept of sending messages to objects. Think of it this way: Each object is a person you can contact over the Internet. You can send email messages to them and receive email answers from them.

- When you declare `Turtle sam` in a program, that creates space for an entry in your email address book; `sam` is the name of the entry space.

- When you execute `sam = new Turtle()` in a program, that puts the email address of a particular person in the entry space with the name `sam`. Now your program can communicate with that person (or turtle).

- When you execute `sam.move(30,50)` in a program, that sends an email message to the person whose address is stored with the name `sam`. The message is a request to the person to move from one place to another. The subject line is `move` and the body of the email is `30,50`, which describes how to move. The `move` message does not get a response from the recipient. It simply produces a change in the state of the object to whom the message was sent. So do `paint` and `switchTo`.

- When you execute `pat.drawFlower()` in a program, having previously executed `pat = new FlowerMaker()`, that sends a message to the person whose address is stored with the name `pat`. The message is a request to the person to carry out the commands given in the definition of the `drawFlower` method. Nothing is in the body of the email. Only a FlowerMaker can understand this message; the compiler will not let you send it to a plain uneducated Turtle such as `sam`.

- When you put `bill = sam` in a program, having previously declared `Turtle bill`, that copies the address in `sam` into the entry space named `bill`. Then `sam` and `bill` contain the same address, so they refer to the same object. If you send a message to `sam`'s object, you are perforce sending a message to `bill`'s object, and vice versa. We will not have occasion to do this for Turtle objects, but it is quite useful for some other kinds of objects you will see later.

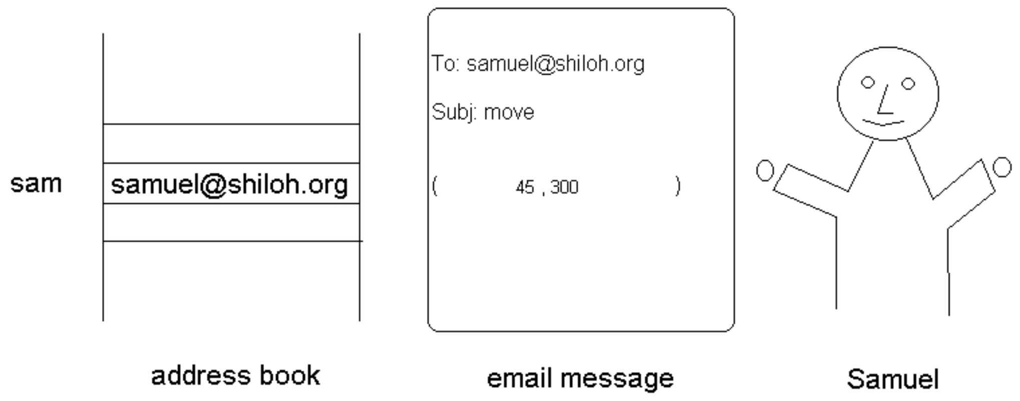

Figure 1.6 Sending an email message to a person

Remember, this is all just a metaphor. Technically, the runtime system creates space for all of a method's variables when the method begins execution.

Key Point: An object variable contains the address of an object, not the object itself. Putting `java.sun.com` in your address book is clearly not the same as putting the entire Sun Microsystems company itself in your address book.

Order of execution

When method X sends a message to an object O, X suspends all operations until O notifies X it is finished doing everything the message asked it to do. Specifically:

- X passes control of the execution of the program to O at the time the message is sent.
- O does whatever it is supposed to do; X is doing nothing at this point, just waiting.
- O returns control of execution back to X when O is finished.
- X executes the next operation after the messaging operation.

For instance, the message `florist.drawTwoFlowers()` in Listing 1.5 means the main method passes control of execution to `florist` and waits patiently until `florist` has drawn the two flowers. The main method does not send its next message, `florist.sleep(300)`, until after `florist` has drawn the two flowers and returned control to the main method.

If it were not pedagogically unsound to cascade metaphors, we would say you could think of it this way: X calls O up on the telephone and asks O to do something; O puts X on hold while he does it; O takes X off hold to report he has done it; X hangs up.

Variables versus values

There is a sharp difference between a variable and the value stored in it. You can have a variable refer to two different objects at different times, or you can have two different variables refer to the same object at the same time. This is illustrated by the following sequence of statements (although we normally try to avoid having two different variables refer to the same object within the same method):

```
Turtle pat;
pat = new Turtle();     // pat refers to Turtle object #1
pat.move (90, 50);      // that object is now north of the center

Turtle chris;
chris = pat;            // 2 variables refer to Turtle object #1
chris.move (90, 50);    // that object is now northwest of the center

pat = new Turtle();     // pat refers to a different Turtle object #2
pat.move (90, 50);      // Turtle object #2 is now north of the center
```

Exercise 1.25** Explain in your own words the difference between an object and a reference to the object.

Part B Enrichment And Reinforcement

1.7 Three Application Programs Using Other Kinds Of Objects

Future chapters present objects that are more complex to work with than Turtles. Several applications involving them are briefly described here so you can see how other kinds of objects are used. You should be able to get the general idea of what they do, although the details will have to wait until later chapters.

Evaluating the stock market

Suppose some investors want to know the range of possible outcomes if they invest in the stock market. That is, they are too smart to just bet on the average performance over the past few decades; they want to know reasonable estimates of the best and the worst possible outcomes for their investments over the next twenty years or so.

Listing 1.7 is an application program to model the financial markets and simulate their behavior over a period of twenty years. It is a slight variant of software you will learn to develop in Chapter Nine. The first statement declares an object variable named `wealth` for a portfolio of investments, a combination of stocks and bonds. The second statement creates a model of the portfolio and makes `wealth` refer to that model. The third statement prints out a description of the various mutual funds the portfolio is invested in.

Listing 1.7 An application program using a BuyAndHoldPortfolio object

```
public class InvestFor20
{
   public static void main (String[ ] args)
   {  BuyAndHoldPortfolio wealth;                            // 1
      wealth = new BuyAndHoldPortfolio();                    // 2
      wealth.describeInvestmentChoices();                    // 3
      wealth.waitForYears (20);                              // 4
      wealth.displayCurrentValues();                         // 5
   }  //======================
}
```

The fourth statement simulates the action of the financial markets every day for the next twenty years and keeps track of the changes in the values of the portfolio. The last statement displays the results at the end of the twenty years.

You could run this program many times and write down the highest and lowest outcomes obtained. Or the program itself can be revised to run 100 simulations and then print out the lowest and highest end results obtained. (Note: You cannot actually run the three programs in this section until after you study the later chapters, because you do not have the object classes the programs need).

The investment program has the same structure as a Turtle program: You declare an object variable (`sam` or `wealth`), have it refer to a newly-created object, then send the object whatever messages will accomplish the task to be performed by the program.

Combining two files into one

Suppose you have two separate files on a hard disk that contain retail sales information from two different cash registers. You need to combine them for further computer processing. Specifically, you need an application program that combines two files on the hard disk into one file so that all the lines from the first file (named "firstData.txt") are followed by all the lines from the second file (named "secondData.txt") in a new file named "combined.txt".

Listing 1.8 is an application program to do just that. You will learn how to write such programs in Chapter Twelve. The first two statements declare an object variable of the IO class, named `inputFile`, and have it refer to a representation of the physical file "firstData.txt". The IO class is built from Sun's standard library classes for disk files.

The next two statements declare an object variable of the FileOutputStream class, named `outputFile`, and have it refer to a representation of the physical combined file (replacing any existing file on the hard disk of the same name, creating a new one if needed). The FileOutputStream class is one of Sun's standard library classes that comes with the free download of JDK.

Listing 1.8 An application program using three disk file objects

```
public class Combiner
{
   public static void main (String[ ] args)
           throws FileNotFoundException    // explained in Ch 12
   {  IO inputFile;                                          // 1
      inputFile = new IO ("firstData.txt");                  // 2

      FileOutputStream outputFile;                           // 3
      outputFile = new FileOutputStream ("combined.txt");    // 4

      inputFile.copyTo (outputFile);                         // 5
      inputFile = new IO ("secondData.txt");                 // 6
      inputFile.copyTo (outputFile);                         // 7
      outputFile.close();                                    // 8
   }  //======================
}
```

The effect of the last sequence of four statements is as follows:

1. Statement #5 sends a message to the `inputFile` object to print all of its information to the file represented by the `outputFile` object.
2. Statement #6 changes the value of the `inputFile` variable so it refers to a representation of a completely different disk file named "secondData.txt".
3. Statement #7 sends the new object a message to print all of its information to the file represented by the `outputFile` object; this information is added after the information from "firstData.txt".
4. Statement #8 sends the combined file referred to by `outputFile` the message that it has been completed, which causes some clean-up activities.

The quotes around "firstData.txt" mean this is the name of a physical file on the hard disk. The runtime system uses the name to find an existing file. If you did not have the quotes, the compiler would interpret it as the name of a variable. That would not be acceptable, because you have not declared the variable or given it a value.

This program would be more useful if you could use it for any three files, not just the ones named. That could be done with the `String[] args` part: You would put something involving `args` in place of the three names in quotes, then run the program using

```
   java Combiner firstData.txt secondData.txt combined.txt
```

(If it were not far too early to introduce this detail now, we would tell you to just put `args[0]` in place of "firstData.txt", put `args[1]` in place of "secondData.txt", and put `args[2]` in place of "combined.txt" within the main method.)

Finding the highest-paid worker

Suppose a company stores information about all the current employees in a hard-disk file named "workers.txt". The number of employees in the company fluctuates weekly, but it is not expected to exceed 2800 or so. The company needs an application program that prints out the name of the highest-paid employee and what he/she is paid.

Listing 1.9 is an application program to do just that. Chapter Seven shows how to define a WorkerList object, which is a list of all the workers in the company, and how to write this program. The first two statements create a WorkerList object that can store up to 3,000 employees, each represented as a separate Worker object, and have `company` refer to the WorkerList object. The third statement sends a message to the `company` object to read in the information about the employees from the hard-disk file named "workers.txt".

Listing 1.9 An application program using WorkerList and Worker objects

```
public class FindHighest
{
   public static void main (String[ ] args)
   {  WorkerList company;                               // 1
      company = new WorkerList (3000);                  // 2
      company.addFromFile ("workers.txt");              // 3

      Worker highestPaid;                               // 4
      highestPaid = company.getWorkerWithHighestPay();  // 5
      System.out.println (highestPaid);                 // 6
   }  //======================
}
```

The fourth and fifth statements ask the object named `company` to find the Worker object representing the employee with the highest pay and then store the object in a Worker variable named `highestPaid`. The last statement sends a message to an object to print a line in the terminal window describing the employee.

The object that prints a line is named `out`. Note that the program does not declare that object variable. That is because another class named System already declares the object variable named `out` and assigns it a value. Your class can use the variable as long as it makes it clear to the compiler where to find it. The compiler will not look for such variables outside of the class you are in unless you explicitly tell it to, by giving the name of the other class before the name of the variable.

A prototype application using only Sun library classes

Sometimes you want to test out part of a software system before you have the whole system completed. An executable program that does only a small part of what the final product is intended to do is a **prototype**. Many of its methods may be incomplete. A common technique is to have such methods simply print a message to the terminal window that it was called. Consider the following statement:

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

It prints whatever is within the quotes to the terminal window. Using this statement, you can make a BuyAndHoldPortfolio class for the InvestFor20 application that allows you to run the program. This class is in Listing 1.10. It, together with the InvestFor20 class in Listing 1.7, forms a complete working program without relying on anything other than the Sun standard library. But it does not do much. By the time you complete Chapter Four, you will know how to write complete programs that do much more interesting tasks.

Listing 1.10 A prototype of the BuyAndHoldPortfolio class of objects

```
public class BuyAndHoldPortfolio
{
   public void describeInvestmentChoices()
   {  System.out.println ("You have 5 ways to invest.");
   }  //======================

   public void waitForYears (int years)
   {  System.out.println ("Wait for your money to grow.");
   }  //======================

   public void displayCurrentValues()
   {  System.out.println ("You now have lots of money.");
   }  //======================
}
```

Objects

You have seen four different concrete examples of object-oriented programming so far. Now you can look at what is common to them and to other similar situations.

Object-oriented software does its job primarily by sending messages to various objects. These messages call on the services provided by the objects. A class definition says what services an object of the class provides, defined as methods.

For the Turtle software, the objects represent the point of a pen that draws figures in colors. For the InvestFor20 software, the objects represent mutual funds and the portfolio as a whole. For the Combiner software, the objects represent physical files on a hard disk. For the WorkerList software, the objects represent the company as a whole and the individual workers.

The effect of a given message on different instances from the same class can vary. This is because each object stores individual information about itself. A Turtle object stores its own heading and coordinates. A Worker object stores its name, birth date, and week's pay. The values of the stored data affect what the object does in response to a message.

An analogy with CEOs

In olden times (thirty years ago), programmers developed software in a way that kept all of the information stored out in the open. A programmer had to keep track of perhaps hundreds of variables, not just a few Turtle or WorkerList variables. This was often overwhelming, and it limited the size and complexity of software that could be developed in a reasonable time. It is analogous to what happens in a private company run by a hands-on owner. When the owner supervises most of the details of everything that gets done, the company cannot grow past a certain modest size.

For object-oriented programming, the programmer creates objects that store the information. These objects supply or modify the information when requested via method calls. This gives the programmer much less to keep track of, and so larger and more complicated software can be developed efficiently. Leaving most of the work to individual objects is a classic use of the principle of delegation of responsibility.

Such a programmer acts more like a CEO of a large company than a hands-on owner of a small company. For instance, the CEO may have an assistant do a substantial part of the job and report the result, which the CEO passes on to two other assistants who between them complete the job. The CEO is there only to determine the overall structure of the solution to a problem and make sure that individual subtasks are given to assistants who can do the task well.

A CEO who needs a task done looks for an employee with the right qualifications. Similarly, when you design larger programs, you will look for classes of objects you already have in the Sun standard library or in your own library that can do the task, i.e., you try to **reuse software**. The CEO who has no suitable employee for a given task sees if one can easily be trained in the needed skills. Similarly, you may find that you need to add new methods to existing classes of objects (i.e., subclass them) to get your tasks done. The CEO who has no employee who can be retrained goes out and hires a new one who can do the job. Similarly, you will often find you need to create new classes of objects to perform the tasks that a piece of software needs to do.

Models and simulation

Objects in software typically provide a **model** of a portion of the real world. For instance, a Turtle object is a model of a hand holding a pen or crayon. Software for operating a bank may have many Account objects, SafeDepositBox objects, Teller objects, and Customer objects that model part of the bank operations. And software for finding efficient ways of manufacturing furniture may have many WoodShaper objects, Assembler objects, and Finisher objects that model people with specific tasks.

Execution of the software typically provides a **simulation** of reality. Simulation is defined in Merriam Webster's Collegiate Dictionary as: "a. the imitative representation of the functioning of one system or process by means of the functioning of another; b. examination of a problem often not subject to direct experimentation by means of a simulating device." That device would be the computer, or more precisely, a program running on the computer.

The simulation may be profitable because it can be done by the computer hundreds of times faster than in reality, or because it avoids the destruction of material and the use of manpower that reality would require. The computer does not shape real wood into furniture, only virtual wood (objects) into virtual furniture (more objects).

An important way in which computer models and simulations affect us all is through the use of software to model the weather around the world and simulate its development over the next few days. This software is what makes your daily weather report possible so you can plan your activities in the near future. It also provides fairly reliable predictions of severe weather that can kill people and damage property if no one is prepared for it. The objects used are Storm objects, Cloud objects, JetStream objects, and the like.

In general, an application program often works with a model of a real situation. A model contains various elements that represent parts of the model. These elements (objects) can be grouped into classes according to the kind of behavior they exhibit -- objects with the same kinds of behaviors are grouped into the same class. For instance, Turtles draw pictures; BuyAndHoldPortfolios of stocks make money; FileOutputStreams store textual data; and Workers provide hours of work in return for their week's pay.

Exercise 1.26 Revise Listing 1.7 to have it print the status of the portfolio at the end of each five-year period in the overall twenty-year period.
Exercise 1.27 Revise Listing 1.9 to also print the information for the worker with the lowest pay. Assume the existence of one additional obvious method.
Exercise 1.28* Revise Listing 1.8 to have it combine four files into a single file.
Exercise 1.29* Think of another real-life problem a computer program could be used to solve. Write an application program to solve it on the level illustrated here (that is, with all the messy details left for the objects to carry out).
Exercise 1.30* Describe a different situation in which you know computer software models and simulates something. What does it model, what does it simulate, and what are the elements in the situation?

1.8 Program Development: Analysis, Logic Design, Object Design, Refinement, Coding

When you develop software, you need a plan. You cannot just read the statement of a problem and start coding. This section outlines a process for developing software that has been found to be very effective. If you use it, you will take somewhat longer to get to where you have Java code that appears to solve the problem, but you will take much less time to get to where you have Java code that really does solve the problem.

When you worked out some of the Turtle problems, surely you found that it was much easier to come up with the right answer if you first drew a picture of what you were trying to accomplish, putting in pixel measurements where needed. That was part of your plan. Other programming problems are not so strongly graphics-based, so you will generally need to have most of the plan in writing rather than in pictures.

The process presented here has five stages: Analysis, Logic Design, Object Design, Refinement, and then Coding. The presentation in this section must of necessity be rather general, since you have not seen much in the way of language features yet. You will see applications of this process to problems in Chapter Two and Chapter Three. Late in Chapter Three, we will go over these points more concretely, once you have seen Java language features that let you make choices and perform actions repeatedly. And we will discuss them further in Chapter Eight after you have seen larger examples.

Analysis

Before you can make a plan, you have to analyze the problem to see exactly what is required. Drawing a picture is often helpful. Thinking of many different situations in which the software will have to perform helps you decide how you will react to the situation. The statement of a problem is usually incomplete, so you cannot develop a solution until you have the answers to some questions about it. Your objective is to have a clear, complete, and unambiguous specification of the problem before you start working on the solution to the problem.

Choose some test data that the software will have to react to and decide what it should do for that particular situation. Repeat this several times with different data. This helps you more precisely determine what you are supposed to do. Make a record of the data and the expected result so you can test your program when you complete it. The time to develop a test plan is during the analysis and design stages, not after the coding stage.

Logic Design

Decide the order in which tasks will be done. As you do, think of the helpers you would like to have carry out those tasks for you, to make your job easy. For instance, the developer of the program in Listing 1.9 might have worked it out as follows:

"I need a records-clerk helper that can store information about thousands of workers. I will ask this records-clerk helper to get information about all the workers in the company from the company records. Then I will ask this records-clerk helper to produce a worker helper who knows who is the highest paid of all. I will then ask that worker helper to tell me the highest-paid worker's name and pay."

The process you develop should be written out entirely in English in this stage (or whatever natural language you prefer). Do not code anything in Java yet.

Object Design

Decide on a name for each of the types of helpers (known as objects in Java) you need. Then decide how you will phrase the messages you will send to them. In our example of Listing 1.9, the developer decided (a) to name the records-clerk helper a WorkerList, (b) to name the message that asks the records-clerk helper to get information about the workers `addFromFile`, and (c) to send along with that message the file name, because the records-clerk helper needs the file name to perform the task.

Refinement

Study the logic and the objects you have developed so far and make sure that everything is correct. Run through several sets of test data in your mind or on paper to see how the logic and the objects handle it. Make any modifications you see are needed. Do not go further until you are quite sure that the logic is correct.

Coding

Translate your design to Java (this is called **coding** the design). Since this is your first course in software development, you should use the following trick: Only code what you are fairly sure is not going to produce more than three or four errors, if that. If a method requires more than one or two statements, you can leave the method body empty for now or just have it print a simple message, as shown in Listing 1.10. Compile your partially-done program before adding more. That way, when errors occur, you will find it much easier to figure them out and correct them.

For a program of some complexity, you are not done yet. Your design calls on objects to perform subtasks to get the overall task done, but you need to develop the logic for those subtasks. That usually requires that you repeat the process just described, but this time for one subtask at a time. And you will have to repeat the process on subtasks of the subtasks if they are at all complex.

The waterfall model

Just because you are very careful in the steps so far, you cannot be sure that your program is correct. You need to test the program with the test data you developed during analysis and design. This usually leads to changes in coding, sometimes also to changes in the analysis and design. When the program appears correct, you can distribute it. Thereafter, the program will need maintenance as the way in which it is used changes or more faults are found.

The classic **waterfall model** emphasizes these steps: analysis, design, coding, testing, and maintenance. This book concentrates primarily on the first three.

1.9 Placing Java In History

Now that you have some idea of what Java is, you should know something about its historical background. Each computer chip has a set of numeric codes it understands directly, called **machine code**. For instance, 31 65 31 83 31 75 might mean to print the word ASK. Actually, machine code is in binary notation (base 2) rather than decimal notation (base 10), so the instruction would be 11111 1000001 11111 1010011 11111 1001011. Reminder: Base 2 uses powers of 2, e.g., 1111 means $2^3 + 2^2 + 2^1 + 1$, which is 8+4+2+1 = 15; and 100101 means $2^5 + 2^2 + 1$, which is 32+4+1 = 37.

Programming in these machine codes is prohibitively difficult. So **assembler languages** were invented, along with programs to translate the assembly instructions into the chip's machine code. In some assembler language, `PRNT 'A' PRNT 'S' PRNT 'K'` might be how you tell the chip to print the word ASK. This matches up one-for-one with the binary machine code instructions, but it is easier to remember and use. But still, each kind of chip has its own assembler language. If you want to write a program that runs on five different kinds of chips, you have to write it five times in five different assembler languages. And the language is still quite tedious.

High-level programming languages

High-level languages were invented to let you give a command such as `write('ASK')` to print the word ASK for any computer chip. FORTRAN (for science and engineering), COBOL (for business) and Lisp (for artificial intelligence) were developed in the middle 1950s. BASIC (for students) and C (for operating systems) were developed in the 1960s and early 1970s. Pascal was developed in the 1970s and became the dominant language for teaching computer science in colleges and universities.

If you want to write a program that runs on five different chips, you just need to write it once if you use Pascal. Then you have it translated (compiled) to each chip's own machine code. You can do this because each of the chips has a compiler written in its own machine code that reads in a source code file written in Pascal and compiles it. The five compilers are written in different languages; but since the compilers have already been developed, additional programs can be written in just the one language Pascal.

Unfortunately, Niklaus Wirth, who invented Pascal, named its successor Modula instead of Pascal++, which is perhaps the primary reason Pascal lost its dominance in university instruction (sometimes marketing is everything). Ada (for government contracts) and C++ (a successor of C, for object-oriented programming) and others were developed in the 1980s. Then along came Java in the 1990s.

Java

Java is a high-level programming language that was originally developed to control electronic consumer appliances, such as CD organizers and home security systems. It is a simpler cleaner language than most other widely-used high-level languages. A key advantage is that it has only one compiled form regardless of the computer chip on which it is used. This is platform independence. The compiled form is called **bytecode**, and it runs on a **Java Virtual Machine** (**JVM**) (which is an "abstract design" of a machine, according to Sun Microsystems).

Sun Microsystems and others have created simulators of the JVM for all the common kinds of computer chips. A JVM simulator, called an **interpreter**, is written in the chip's native machine code. It takes one unit of bytecode at a time and executes it, then goes on to the next, etc. This is slower than executing the compiled form of e.g. Pascal, but chips today are fast enough that speed is not a problem in most cases.

The big advantage is that a program in compiled form (a `.class` file) can be retrieved over the internet and executed immediately. This is better than having to obtain the source code (as with Pascal) and have it compiled on your own machine before you can use it. Browsers have JVM simulators built in so they can execute applets on web pages. And security measures in the browsers prevent applets from harming your computer files. These security measures are not possible with programming languages such as Pascal.

You can spend hours reading about the history of computing on the web. Good places to start are the following web sites:

http://www.chac.org/chac/chhistpg.html Computer History Association of California
http://ei.cs.vt.edu/~history History of Computing from Virginia Tech University
http://www.livinginternet.com The Living Internet by William Stewart
http://www.computerhistory.org The Computer History Museum
http://www.thocp.net The History Of Computing Project

Exercise 1.31 Convert to decimal notation: 1100 and 10110 and 101010.
Exercise 1.32 Convert to decimal notation: 0011,0001,0110 and 1000,0000,1000,1000.

*1.10 Fractal Turtles (*Enrichment)*

Lest you think that such a simple concept as the Turtle only allows simple-minded drawings, the program in Listing 1.11 (see next page) illustrates the power of Turtles. The upper part contains the application program and the lower part contains the class of Turtle objects the program uses. The application begins by creating a FractalTurtle object, which knows how to draw a fractal. First it backs south by 240 pixels so as to have plenty of room to draw a tall tree. Then it draws a tree with a trunk of length 80 pixels.

This is a very special tree, called a Pythagorean tree, which is one of many fractal images Turtles can make. The directions for making the tree are in the `drawTree` method. It uses four language features you have not seen before (which is why this is an optional section). But you can probably make sense of the following step-by-step description.

How to draw a tree fractally

To draw a tree with a trunk of a given length, the Turtle first paints a straight line `trunk` pixels long. Second, it draws the branches that go off to the left. Specifically, it turns 30 degrees to its left and then draws a tree whose size is 70% of the tree it is in the midst of drawing. That is, the branches it is currently drawing are shaped like a somewhat smaller version of the tree it is currently drawing. This is indicated in the `drawTree` logic by multiplying the value of `trunk` by the number 0.7. Note that it is permissible for distances to be expressed as decimal numbers in the Turtle methods; the word `double` in the parentheses signals that the value to be used is a decimal. Note also that an asterisk is a multiplication sign.

After the Turtle finishes drawing the branches on the left side of the trunk, it swings back to its right 60 degrees, so it is now facing 30 degrees to the right of the trunk line. Then it draws the branches that go off to the right, which look like a smaller version of the tree it is drawing (70% of the size). Finally, the Turtle moves backward so that it is at the bottom of the trunk, where it was initially, and facing in the same direction it was initially.

Listing 1.11 A program that draws a fractal image

```
public class FractalApp
{
   // Draw a Pythogorean tree.

   public static void main (String[ ] args)
   {  FractalTurtle pythagoras;
      pythagoras = new FractalTurtle();
      pythagoras.move (90, -240);
      pythagoras.drawTree (80);
   }  //======================
}
//##############################################################


public class FractalTurtle extends Turtle
{
   public void drawTree (double trunk)
   {  paint (0, trunk);                // go to top of trunk
      move (30, 0);                    // face to the left
      if (trunk > 1)
         drawTree (trunk * 0.7);       // make branches on the left
      move (-60, 0);                   // face to the right
      if (trunk > 1)
         drawTree (trunk * 0.7);       // make branches on the right
      move (30, -trunk);               // go to bottom of trunk
   }  //======================
}
```

The process just described goes on forever, which is too long for a computer program, even with a very fast chip. So the Turtle cheats a little. As soon as the tree it is drawing is so small that its trunk is less than a single pixel in length, the Turtle does not draw any of that tree's branches -- it figures you cannot see them on the screen anyway. Figure 1.7 shows the result. It is in the category of fractals which are primarily made up of smaller versions of themselves. The choices of 30 degrees and 70% are arbitrary, chosen after some experimentation to find a nice balance. You could try making different choices for the numbers.

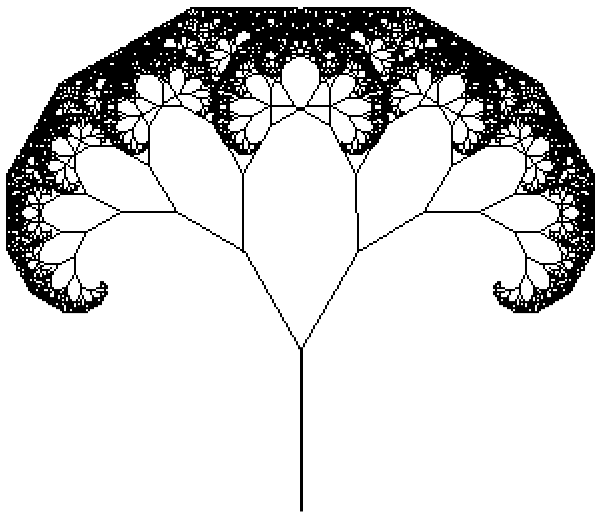
Figure 1.7 Result of executing FractalApp

Exercise 1.33** In the execution of the FractalApp program, how many points are there where the Turtle draws the trunk of a tiny tree that has no branches?

1.11 Review Of Chapter One

Listing 1.3 and Listing 1.4 illustrate all Java language features introduced in this chapter that you need to remember. `http://java.sun.com/docs` has much useful reference material and `http://java.sun.com/products` has the free compiler. Section 1.5 has details on using this free compiler.

About the Java language:

- `SomeClass sam` creates a variable of type SomeClass and declares `sam` as the name of that variable. The phrase `SomeClass sam` is a **variable declaration**. For instance, `Turtle sue` declares a variable for storing a Turtle object.
- Execution of a phrase of the form `sam = new SomeClass()` creates an **instance** (object) of SomeClass and puts a reference to that object in `sam`.
- Anything on a line after `//` is a **comment**; the compiler ignores it.
- A class named SomeClass is to be stored in a plain text file (the **source code**) named `SomeClass.java`. The **compiler** translates this file into a form the **runtime system** can use (the **object code**), stored in a file named `SomeClass.class`. The object code is expressed in **bytecode** which runs on a **Java Virtual Machine** (**JVM**).
- If a class is an **application program**, i.e., with a method whose heading is `public static void main (String[] args)`, then that **main method** can be executed by the runtime system with a command in the **terminal window** of the form `java SomeClass`. An **object class** (with instance methods and no main method) cannot be executed this way.
- When a **method call** has a variable before the dot (period), as in `sam.move(0,10)`, the variable refers to the **executor** of that call. If the statements within some method M include a method call whose executor is not stated, then the executor of the method called is by **default** M's executor.
- A class defined with the phrase `extends Turtle` in the heading is a **subclass** of the Turtle class, and Turtle is its **superclass** (and similarly for other classes besides Turtle). The subclass **inherits** each public method defined in the superclass, i.e., each instance of the subclass can use those methods as if they were defined in the subclass.
- See Figure 1.8 and Figure 1.9 for the remaining new language features. In Figure 1.8, `ArgumentList` stands for whatever values are required by the particular method, with commas separating them if you have more than one of them. Phrases in italics indicate optional parts -- sometimes they are present and sometimes not. A `DeclarationGroup` is zero or more declarations, and a `StatementGroup` is zero or more statements.

`ClassName VariableName;`	**statement** that creates a reference variable
`VariableName = new ClassName();`	**statement** that creates an object and assigns its reference to a reference variable
`VariableName.MethodName ( `*`ArgumentList`*` );`	**statement** that sends a message to the object referred to by the variable named
`MethodName ( `*`ArgumentList`*` );`	**statement** that sends a message to the executor of the method declaration it is in

Figure 1.8 Statements introduced in Chapter One

`public class ClassName`
`                  `*`extends SuperClassName`*
`{  DeclarationGroup`
`}`
	declaration of a class; "extends SuperClassName" is optional

`public static void main (String[ ] args)`
`{  StatementGroup`
`}`
	declaration of a main method of an application

`public void MethodName()`
`{  StatementGroup`
`}`
	declaration of an instance method that is called as a stand-alone statement

Figure 1.9 Declarations introduced in Chapter One

Other vocabulary to remember:

- The programs you run and all the classes they use are **software**. The physical components of the computer (chip, RAM, disk, monitor, etc.) are **hardware**.
- The **method heading** is the first part of the method up through the parentheses; the **method body** is the following matched pair of **braces** `{ }` and its contents. The method body is a sequence of **statements**, most of which are a **command** followed by a semicolon.
- The absence of `static` in the method heading signals you must have an executor in order to call the method. Such a method is an **instance method**.
- A **declaration of a variable name** declares the type of value to be stored there. It is usually followed directly by a **definition of the variable**. You may define (give a value to) a variable many times but declare its name only once.
- The heading of a method **declares** its name to be the name of the method; the body **defines** what the method does. That is, the heading says how it is used and the body says what happens when you use it.
- The **Sun standard library** comes with the installation of Java that you obtain from Sun Microsystems Inc. It includes the String class and hundreds of others.
- The **keywords** that occur in this chapter are all the words in Figure 1.8 and Figure 1.9 that begin with a small letter, except `args` is not. All non-keywords in a Java program that begin with a letter are **identifiers** (names) of classes, methods, or variables, except for three which you will see later (`true`, `false`, and `null`).
- When the runtime system executes a program, it **links in** all other class definitions required by the program, either directly or indirectly.
- A **prototype** is an executable program that does only part of what the final product is intended to do. The purpose of creating a prototype is to test parts of the system.

About the nine Turtle methods (developed for this book):

- `new Turtle()` creates a Turtle object in the center of a drawing surface 760 pixels wide and 600 tall. The Turtle initially faces east and carries a black paintbrush. In the following, `sam` is a Turtle variable to which you have assigned a Turtle object.
- `sam.paint(angle, dist)` tells `sam` to turn counterclockwise by `angle` degrees and then go forward by `dist` pixels, leaving a trail of the current drawing color.
- `sam.move(angle, dist)` tells `sam` to turn counterclockwise by `angle` degrees and then go forward by `dist` pixels, without leaving any marks.
- `sam.swingAround(dist)` tells `sam` to draw a circle of radius `dist` pixels with `sam` at the center.
- `sam.fillCircle(dist)` tells `sam` to draw a circle of radius `dist` pixels with `sam` at the center, and fill its interior with the current drawing color.

- `sam.fillBox(width, height)` tells `sam` to draw a rectangle `width` pixels wide and `height` pixels tall, with `sam` at the center, and fill its interior with the current drawing color.
- `sam.switchTo(col)` tells `sam` to change the current drawing color to `col`, which can be any of BLACK, GRAY, BLUE, GREEN, RED, YELLOW, ORANGE, PINK, MAGENTA, and WHITE. Put "Turtle." in front of the name of a color you use in a program, unless you use it in a subclass of the Turtle class.
- `sam.say("whatever")` tells `sam` to print whatever is within the quotes.
- `sam.sleep(milli)` tells `sam` to suspend action for `milli` milliseconds.

Answers to Selected Exercises

1.1
```
sue.paint (90, 60);
sue.paint (90, 120);
sue.paint (90, 60);
sue.paint (90, 120);
```
1.2
```
sam.paint (90, 12);
sam.move (180, 2);
sam.paint (135, 3);
sam.paint (-45, 6);
```
1.3
```
public class LetterB
{     public static void main (String[ ] args)
      {     Turtle cat;
            cat = new Turtle();
            cat.paint (0, 5);  // the top of the bottom part of the 'b'
            cat.paint (-90, 5);
            cat.paint (-90, 6);  // the bottom of the bottom part of the 'b'
            cat.paint (-90, 12);  // the stem of the 'b'
      }
}
```
1.4
```
public class LetterM
{     public static void main (String[ ] args)
      {     Turtle cat;
            cat = new Turtle();
            cat.paint (90, 6);  // the left side of the 'm'
            cat.paint (-90, 4);
            cat.paint (-90, 5);  // the center part of the 'm'
            cat.move (180, 5);
            cat.paint (-90, 4);
            cat.paint (-90, 7);  // the right side of the 'm'
      }
}
```
1.5
```
public class Hexagon
{     public static void main (String[ ] args)
      {     Turtle cat;
            cat = new Turtle();
            cat.paint (60, 50);
            cat.paint (60, 50);
            cat.paint (60, 50);
            cat.paint (60, 50);
            cat.paint (60, 50);
            cat.paint (60, 50);
      }
}
```
1.8
```
public class SquaresInSquares
{     public static void main (String[ ] args)
      {     SmartTurtle cat;
            cat = new SmartTurtle();
            cat.makeBigSquare ();
            cat.move (0, 50);   // this is arbitrary; anything over 40 would do
            cat.makeBigSquare();
            cat.move (90, 25);
            cat.move (90, 25);  // to upper-left corner of inner square
            cat.makeSmallSquare();
            cat.move (0, 50);   //  to upper-left corner of the other inner square
            cat.makeSmallSquare();
      }
}
```

1.9
```
public void goAndSquare()
{       move (90, 70);
        makeSmallSquare();
}
```
Then replace each of the last three pairs of statements in Listing 1.4 by:
```
sue.goAndSquare();
```

1.10
```
public void drawHexagon()
{       paint (60, 30);
        paint (60, 30);
        paint (60, 30);
        paint (60, 30);
        paint (60, 30);
        paint (60, 30);
}
```

1.11
```
public class ThreeHexagons
{       public static void main (String[ ] args)
        {       SmartTurtle cat;
                cat = new SmartTurtle();
                cat.drawHexagon();
                cat.move (120, 0);
                cat.drawHexagon();
                cat.move (120, 0);
                cat.drawHexagon();
        }
}
```

1.15 Put move (0, -20); right after the first drawFlower();.
Put move (0,20); right after the second drawFlower();.

1.16
```
public void drawSmallFlower()
{       paint (90, 7);
        switchTo (RED);
        fillCircle (3);
        switchTo (BLACK);
        move (180, 7);
}
```

1.17 First put the 8 statements of drawLeaf in drawFlower twice, so that it has
13+8+8 = 29 statements. So drawTwoFlowers will have 2+29+29 = 60 statements.

1.18
```
public class Target
{       public static void main (String[ ] args)
        {       Turtle cat;
                cat = new Turtle();
                cat.switchTo (Turtle.RED);
                cat.fillCircle (80);    // the 80 and other numbers are arbitrary
                cat.switchTo (Turtle.YELLOW);
                cat.fillCircle (60);
                cat.switchTo (Turtle.BLUE);
                cat.fillCircle (40);
                cat.switchTo (Turtle.BLACK);
                cat.fillCircle (20);
        }
}
```

1.26 Put the following eight statements in place of the last two statements of Listing 1.7:
```
wealth.waitForYears (5);
wealth.displayCurrentValues();
wealth.waitForYears (5);
wealth.displayCurrentValues();
wealth.waitForYears (5);
wealth.displayCurrentValues();
wealth.waitForYears (5);
wealth.displayCurrentValues();
```

1.27 Add the following three statements at the end of the main method:
```
Worker lowestPaid;
lowestPaid = company.getWorkerWithLowestPay();
System.out.println (lowestPaid);
```

1.31 8+4=12 and 16+4+2=22 and 32+8+2=42.

1.32 3*256+1*16+6=790 and 8*16*256+8*16+8 = 32768+128+8 = 32904.