2

Java Programming Basics

OBJECTIVES
After you have read and studied this chapter, you should be able to

• Identify the basic components of Java programs.
• Distinguish two types of Java programs—applications and applets.
• Write simple Java applications and applets.
• Describe the difference between object declaration and object creation.
• Describe the process of creating and running Java programs.
• Use MainWindow and MessageBox classes from the javabook package to write Java applications.
• Use the Graphics class from the standard Java package.


**Introduction**

We will introduce both Java applications and applets in this chapter and describe the basic structure of simple Java programs. We will also describe the steps you follow to run programs. We expect you to actually run these sample programs to verify that your computer (either your own or the one at the school’s computer center) is set up properly to run the sample programs presented in the book. It is important to verify this now. Otherwise, if you encounter a problem later, you won’t be able to determine whether the problem is the result of a bad program or a bad setup.

We will develop a sample application program in Section 2.4 following the design, coding, and testing phases of the software life cycle. We will develop all of our sample programs in this manner. We stress here again that our objective in this book is to teach object-oriented programming and how to apply object-oriented thinking in program development. The Java language is merely a means to implement a design into an executable program. We chose Java for this book because Java is a much easier language than other object-oriented programming languages to translate a design into an actual code. Beginning students often get lost in the language details and forget the main objective of learning how to design, but the use of Java should minimize this from happening.

To help you concentrate on program design and programming fundamentals instead of language details, we provide predefined classes for you to use in writing Java programs. We will introduce two of these predefined classes in this chapter. As nice and logical as Java can be, beginners still must learn quite a few details before becoming proficient in Java. We provide these predefined classes so that you can start writing meaningful programs immediately without knowing too much about Java language specifics. Please read the Preface for more reasons for using predefined classes in this book.

### 2.1 The First Java Application

Our first Java application program displays a window on the screen, as shown in Figure 2.1. The size of the window is slightly smaller than the screen, and the window is positioned at the center of the screen. Also, the window has a title Sample Java Application. Although this program is very simple, it still illustrates the fundamental structure of an object-oriented program, which is

*An object-oriented program uses objects.*
It may sound too obvious, but let’s begin our study of object-oriented programming with this obvious notion. Here’s the program code:

```java
/*
 * Program MyFirstApplication
 * This program displays a window on the screen. The window is positioned at the center of the screen, and the size of the window is almost as big as the screen.
 */
import javabook.*;
class MyFirstApplication {
    public static void main(String[] args) {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible(true);
    }
}
```

This program declares one class called MyFirstApplication, and the class includes one method called main. From this main method, the MyFirstApplication class creates and uses a MainWindow object named mainWindow by sending a
message setVisible to the object. Expressing this program as an object diagram results in the one shown in Figure 2.2.

FIGURE 2.2 The object diagram for the MyFirstApplication program.

Instead of going through the program line by line from the top, we will start examining this program by concentrating on the following three lines of code:

```java
MainWindow mainWindow;
mainWindow = new MainWindow();
mainWindow.setVisible(true);
```

We will explain the rest of the program in the next section. These three lines of code represent the crux of the program, namely, an object-oriented program uses objects. The rule to remember in using objects is as follows:

*To use an object in a program, we first declare and create an object, and then we send messages to it.*

In the remainder of this section, we will describe how to declare an object, create an object, and use an object by sending messages to the object.

**Object Declaration**

Every object we use in a program must be declared. An object declaration designates the name of an object and the class to which the object belongs. Its syntax is

```
<class name> <object names> ;
```

object declaration syntax
where `<object names>` is a sequence of object names separated by commas and `<class name>` is the name of a class to which these objects belong. Here’s how the general syntax is matched to the object declaration of the program:

```
MainWindow    mainWindow;
```

Here are more examples:

```
Account    checking;
Customer    john, jack, jill;
```

The first declaration declares an `Account` object named `checking`, and the second declaration declares three `Customer` objects.

To declare an object as an instance of some class, the class must be defined already. We will first study how to use objects from predefined classes. Later in the book, we will show you how to define your own classes, from which you can create instances.

When we declare an object, we must give it a name. Any valid identifier that is not reserved for other uses can be used as an object name. An `identifier` is a sequence of letters and digits with the first one being a letter. We use an identifier to name a class, object, method, and others. The following words are all valid identifiers:

```
MyFirstApplication
FunTime
ComputeArea
```

Upper- and lowercase letters are distinguished, so the following four identifiers are distinct:

```
mainwindow    m AinWindow
MAINWindow    mainWINDOW
```
No spaces are allowed in an identifier, and therefore, the following three lines

```
Sample Program
My First Application
Program FunTime
```

are all invalid identifiers.

Since upper- and lowercase letters are distinguished, you can use `mainWindow` as the name for an object of the class `MainWindow`. We name objects in this manner whenever possible in this book so we can easily tell to which class the object belongs. We follow the Java naming convention of using an uppercase letter for the first letter of the class names and a lowercase letter for the first letter of the object names in this book. It is important to follow the standard naming convention so others who read your program can easily distinguish the purposes of identifiers. Programs that follow the standard naming convention are easier to read than those that do not follow the standard. And remember that software maintenance is easier with easy-to-understand programs.

When an identifier consists of multiple words, the first letter from every word, except the first word, will be capitalized, e.g., `myMainWindow`, not `my-mainwindow`.

---

Follow the standard naming convention in writing your Java programs to make them easier to read.

**Object Creation**

No objects are actually created by the declaration. An object declaration simply declares the name (identifier) that we use to refer to an object. For example, the declaration

```
Account account;
```

designates that the name `account` is used to refer to an `Account` object, but the actual `Account` object is not yet created. We create an object by invoking the `new` operation. The syntax for `new` is

```
<object name> = new <class name> ( <arguments> ) ;
```

where `<object name>` is the name of a declared object, `<class name>` is the name of the class to which the object belongs, and `<arguments>` is a sequence
of values passed to the `new` operation. Let’s match the syntax to the actual statement in the sample program:

```java
mainWindow = new MainWindow ( );
```

Although this example does not pass any argument, it is possible to do so in creating a new instance of `MainWindow`. If we replace the preceding statement with

```java
mainWindow = new MainWindow( "This is my first window" );
```

and run the program, the window shown in Figure 2.3 will appear on the screen. Notice the change in the window title. The double quotes (" ) are used to specify a string data value. If you don’t use double quotes, then the system will treat them as five identifiers, and this will result in an error.

**FIGURE 2.3**

Result of running the program `MyFirstApplication` when the argument to the `new` operation is the string argument "This is my first window".
Chapter 2 Java Programming Basics

Figure 2.4 shows the distinction between object declaration and creation. The object diagram notation we introduced in Chapter 1 is a simplified version of the state-of-memory diagram, as shown in Figure 2.5.

**FIGURE 2.4** Distinction between object declaration and object creation.

---

**State of Memory**

- **A**
  
  ```java
  Account account;
  account = new Account();
  ```

  After **A** is executed, the identifier `account` is declared and space is allocated in memory.

- **B**
  
  ```java
  Account account;
  account = new Account();
  ```

  After **B** is executed, an `Account` object is created and the identifier `account` is set to refer to it.

---

Now, consider the following object declaration and two statements of object creation:

```java
Customer customer;
customer = new Customer();
customer = new Customer();
```

What do you think will happen? An error? No. It is permissible to use the same name to refer to different objects of the same class. Figure 2.6 shows the state-
of-memory diagram after the second `new` is executed. Since there is no reference to the first `Customer` object anymore, it will eventually be erased and returned to the system. Remember that when an object is created, a certain amount of memory space is allocated for storing this object. If this allocated but unused space is not returned to the system for other uses, the space gets wasted. This returning of space to the system is called `deallocation`, and the mechanism to deallocate unused space is called `garbage collection`.

**Message Sending**

After the object is created, we can start sending messages to it. The syntax for sending a message to an object is

```
<object name> . <method name> ( <arguments> ) ;
```

where `<object name>` is an object name, `<method name>` is the name of a method of the object, and `<arguments>` is a sequence of values passed to the method. In the sample program, we send the `setVisible` message with the argument `true`
to the `mainWindow` object to make it appear on the screen. Once again, let’s match the components in the general syntax to the actual statement:

```java
Customer customer;
customer = new Customer();
customer = new Customer();
```

Figure 2.7 shows the correspondence between message sending as represented in the object diagram and in the Java statement. Because the object that receives a message must possess a corresponding method, we often substitute the expression “sending a message” with “calling a method.” We will use these expressions interchangeably.

*The expression "calling object O's method M" is synonymous to "sending message M to object O."*
Notice the argument for the `setVisible` message does not include double quotes as the one for the `new` operation in the example shown on page 43. The argument `true` is one of the two possible logical values (the other is `false`) used in Java programs. We will study more about the use of logical values later in the book, starting from Chapter 6. For now, it suffices to remember that there are two logical values—`true` and `false`—used for certain specific purposes.

Passing `true` in the `setVisible` message makes the receiving object appear on the screen. Passing `false` makes the object disappear from the screen. So, for example, if we write

```java
mainWindow.setVisible(true);
mainWindow.setVisible(false);
mainWindow.setVisible(true);
```

then `mainWindow` will appear once, disappear, and then appear on the screen again. (Note: Because the computer will execute these statements so quickly, you may not notice any difference from the original program. See exercise 26 on page 81.)

The word `true` (and `false`) is called a `reserved word`. It is an identifier that is used for a specific purpose and cannot be used for any other purpose such as for the name of an object.
1. Which of the following are invalid identifiers?
   a. one          e. hello
   b. my Window    f. JAVA
   c. 1234         g. hello,there
   d. acct122      h. DecafLattePlease

2. What’s wrong with the following code?
   ```java
   MainWindow mainWindow();
   mainWindow.show();
   ```

3. Is there anything wrong with the following declarations?
   ```java
   MainWindow mainWindow;
   Account, Customer account, customer;
   ```

4. Which of the following statements is valid?
   a. mainWindow.setVisible( "true" );
   b. mainWindow.setVisible( true );

2.2 Program Components

Now that we have covered the crux of the first sample program, let’s examine the rest of the program. The first sample application program `MyFirstApplication` is composed of three parts: comment, import statement, and class declaration. These three parts are included universally in Java programs.

_A Java program is composed of comments, import statements, and class declarations._

You can write a Java program that includes only a single class declaration, but that is not a norm. In any nontrivial program, you will see these three components. We will explain the three components and their subparts in this section.

Comments

In addition to the instructions for computers to follow, programs contain comments in which we state the purpose of the program, explain the meaning of code, and provide any other descriptions to help programmers understand the program. Here’s the comment in the sample `MyFirstApplication` program:
A comment is any sequence of text that begins with the marker `/*` and terminates with another marker `*/`. The beginning and ending markers are matched in pairs; that is, every beginning marker must have a matching ending marker. A beginning marker is matched with the next ending marker that appears. Any beginning markers that appear between the beginning marker and its matching ending marker are treated as part of the comment. In other words, you cannot put a comment inside another comment. The examples in Figure 2.8 illustrate how the matching is done.

Another marker for a comment is double slashes `//`. This marker is used for a single-line comment. Any text between the double-slash marker and the end of a line is a comment. The following example shows the difference between multiline and single-line comments:

```java
/*
   This program displays a window on the screen. The window is positioned at the center of the screen, and the size of the window is almost as big as the screen.
*/

import javabook.*;

class MyFirstApplication
{
    public static void main(String[] args)
    {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible( true );
    }
}
```

```java
import javabook.*;

class MyFirstApplication
{
    public static void main(String[] args)
    {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible( true );
    }
}
```

The examples in Figure 2.8 illustrate how the matching is done.
Although not required to run the program, comments are indispensable in writing easy-to-understand code.

The third type of comment is called a javadoc comment. It is a specialized comment that can appear before the class declaration and other program elements yet to be described in the book. Javadoc comments begin with the /** marker and end with the */ marker. Here’s an example of a javadoc comment:

```
/**
 * This class provides basic clock functions. In addition
 * to reading the current time and today’s date, you can
 * use this class for stopwatch functions.
 */
```

The asterisks on the second, third, and fourth lines are not necessary, but useful in marking the comments clearly. It is an accepted convention to place asterisks
in this manner. Although not required, aligning the end marker as shown in the example is also an accepted convention. In this book, we will be using this convention for non-javadoc comments also.

One major benefit of using javadoc comments is the availability of a tool that generates a web page automatically from the javadoc comments. Javadoc comments are the standard way of documenting Java programs, and we will be using them extensively in documenting the sample programs in this book. However, the use of javadoc comments is more restricted and difficult than the other two types of comments, so we will have to introduce elements of javadoc comments gradually. We will start using javadoc comments from Chapter 4.

Information on javadoc and sample web documentation pages that are generated from javadoc comments can be found at www.drcaffeine.com/supportinfo/javadoc.html. You can get a full documentation of the standard Java classes generated from javadoc comments at www.java.sun.com/j2ee/j2sdkee/techdocs/api/.

Comments are intended for the programmers only and are ignored by the computer. Therefore, comments are really not necessary in making a program executable, but they are an important aspect of documenting the program. It is not enough to write a program that executes correctly. We need to document the program, and commenting the program is an important part of program documentation. Other parts of program documentation include object diagrams, programmers’ work log, design documents, and user manuals. If you can write a program once and use it forever without ever modifying it, then writing a program with no comments may be tolerable. However, in the real world, using programs without ever making any changes almost never happens. For example, you may decide to add new features and capabilities or modify the way the user interacts with the program. Even if you don’t improve the program, you still have to modify the program when you detect some errors in it. Also, for commercial programs, those who change the programs are most often not the ones who developed them. When the time comes for a programmer to modify his or someone else’s program, he must first understand the program, and program documentation is an indispensable aid to understanding the program.

There are several different uses of comments. The first is the header comment. At the beginning of a program, we place a comment to describe the program. We characterize such a comment as a header comment. We also may include header comments at the beginning of methods to describe their purposes. Depending on the length and complexity of programs, the description may range from short and simple to long and very detailed. A typical header comment for a beginning programming class may look something like this:
Chapter 2 Java Programming Basics

For your own programs, you should write header comments following the guideline provided by your instructor. For listing the sample programs in the book, we will include only the program name and a short description in the header comment, mainly for reference purposes. The header comment in the actual programs, available from our website, includes additional information such as compiler used, copyright notices, and others. The header comment is written in the javadoc format.

Another use of comments is to explain code whose purpose may not be obvious. Your aim is always to write easily understandable, self-explanatory program code. But there are times this is not possible, and you should attach comment to code that is not so easy to understand. There also are times when the original code may not work as intended, and as a temporary measure, you modify the code slightly so the program will continue to work. You should clearly mark such modification with a comment, so you remember what you have done. If you did not put in an appropriate comment and later read your code without remembering about the modification, you would have no idea why you wrote such code. If you cannot understand your own code, imagine the frustration of other programmers (or your T.A. or instructor) trying to understand your modified code.

/*
 * Program: TextEditor
 * Author: C. Thomas Wu
 * ctwu@cs.nps.navy.mil
 * Written: January 1, 2000
 * Course: Comp Sci 101
 * Spring 2000
 * Program Assignment No. 7
 * Compiler: JDK 1.2.2
 * Platform: Windows 98
 * Description:
 * This is a simple text editor. The editor allows the user
 * to save text to a file and read text from a file. The
 * editor displays text using Courier font only and does not
 * allow formatting (e.g., bold, italic, etc.). The editor
 * supports standard editing functions Cut, Copy, and
 * Paste, but does not support Undo. For more details,
 * please refer to the TxEditReadme file.
 */

Note: The use of the asterisks is in the style of javadoc, but this is not a javadoc comment.
Yet another use of comments is to identify or summarize a block of code. Suppose a program is divided into three major parts: getting input values from the user, performing computation using the input values, and displaying the computation results. You can place comments at the top of each part to delineate the three major parts clearly.

Remember that adding comments to a poorly designed program will not make it a better program. Your foremost goal is to develop a well-designed program that runs efficiently and is easy to understand. Commenting a program is only a means toward that goal, not a goal itself. In fact, excessive use of comments makes it harder to follow and understand a program.

Import Statement
We develop object-oriented programs by using predefined classes whenever possible and defining our own classes when no suitable predefined classes are available. In Java, classes are grouped into packages. The Java compiler comes with many packages, and we supply one package called javabook with this textbook. We will explain the motivation behind using the javabook package in the next section. You also can put your own classes into a package so they can be used in other programs. Here’s the import statement in the same MyFirstApplication program:

```java
package javabook;

import javabook.*;

class MyFirstApplication {
    public static void main(String[] args) {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible(true);
    }
}
```

Comment markers are useful in disabling a portion of a program. Let’s say you find a portion that may be causing the program to crash and you want to try out different code for the problem portion. Instead of replacing the whole problem portion with new code, you can leave the questionable code in the program by converting it into a "comment" with comment markers. You can remove the comment markers if you need this code later.
To use a class from a package, you may refer to the class in your program using the following format:

\[ \text{<package name> . <class name>} \]

For example, to use the `MainWindow` class in the `javabook` package, we refer to it as

\[ \text{javabook.MainWindow} \]

which we read as “javabook dot MainWindow.” This notation is called \textit{dot notation}.

A package can include subpackages, forming a hierarchy of packages. In referring to a class in a deeply nested package, we use multiple dots. For example, we write

\[ \text{java.awt.image.ColorModel} \]

to refer to the class `ColorModel` in the `java.awt.image` package; that is, the image package is inside the `awt` package, which in turn is inside the `java` package. Dot notation with the names of all packages to which a class belongs is called the class’s \textit{fully qualified name}. Using the fully qualified name of a class is frequently too cumbersome, especially when you have to refer to the same class many times in a program. You can use the import statement to avoid this problem.

An \textit{import} statement at the beginning of your program eliminates the need for fully qualified names. Instead of using the expression `java.awt.image.ColorModel` to refer to the class, you can refer to it simply as

\[ \text{ColorModel} \]

by including the \textit{import} statement

\[ \text{import java.awt.image.ColorModel;} \]

at the beginning of the program. Notice that the import statement is terminated by a semicolon. If you need to import more than one class from the same package, then instead of using an import statement for every class, you can import them all using asterisk notation:
import <package name> . * 

For example, if we state

import java.awt.image.*;

then we are importing all classes from the java.awt.image package. We use this asterisk notation in our sample program, although we use only one of the many classes available in the javabook package. We could have used

import javabook.MainWindow;

but it is more conventional to use asterisk notation.

Notice that the package names are all in lowercase letters. This is another standard Java naming convention. When you create your own packages, which you will in Chapter 11, make sure to follow this naming convention.

**Class Declaration**

A Java program is composed of one or more classes, some of them are pre-defined classes, while others are defined by ourselves. In this sample program, there are two classes—MainWindow and MyFirstApplication. The MainWindow class is from the javabook package and the MyFirstApplication class is the class we define ourselves. To define a new class, we must *declare* it in the program. The syntax for declaring the class is

```
class <class name>
{
    <class member declarations>
}
```

where `<class name>` is the name of the class and `<class member declarations>` is a sequence of class member declarations. The word *class* is a reserved word used to mark the beginning of a class declaration. A class member is either a
data value or a method. We can use any valid identifier to name the class. Here’s the class declaration in the sample `MyFirstApplication` program:

```java
import javabook.*;

class MyFirstApplication {
    public static void main(String[] args) {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible(true);
    }
}
```

One of the classes in a program must be designated as the main class. The main class of the sample program is `MyFirstApplication`. Exactly how you designate a class as the main class of the program depends on which Java program development tool you use. We will use the name of a main class to refer to a whole application. For example, we say the `MyFirstApplication` class when we refer to the class itself and say the `MyFirstApplication` application when we refer to the whole application.

If we designate a class as the main class, then we must define a method called `main`, because when a Java program is executed, the `main` method of a main class is executed first. We will explain how to define a method in the next section.

### Method Declaration

The syntax for method declaration is

```
<modifiers> <return type> <method name> ( <parameters> )
{
    <method body>
}
```

where `<modifiers>` is a sequence of terms designating different kinds of methods, `<return type>` is the type of data value returned by a method, `<method name>` is the name of a method, `<parameters>` is a sequence of values passed to
a method, and <method body> is a sequence of instructions. Here’s the method
declaration for the main method:

```java
import javabook.*;
class MyFirstApplication {
    public static void main(String[] args) {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible(true);
    }
}
```

Let’s match these components to the actual method declaration of the sample program:

- **Modifier**: `public`  
- **Modifier**: `static`  
- **Return Type**: `void`  
- **Method Name**: `main`  
- **Parameter**: `(String[] args)`

```java
public static void main(String[] args) {
    MainWindow mainWindow;
    mainWindow = new MainWindow();
    mainWindow.setVisible(true);
}
```

We will not explain the meanings of modifiers, return types, and parameters here. We will explain them in detail gradually as we progress through the
book. For now, we ask you to follow a program template that we will present next.

**A Program Template for Simple Java Applications**

The diagram in Figure 2.9 shows a program template for simple Java applications. You can follow this program template to write very simple Java applications. The structure of the sample program MyFirstApplication follows this template.

**FIGURE 2.9** A program template for simple Java applications.

---

**Quick Check**

1. Name three components of a Java program.
2. Locate three program components in the FunTime program from Chapter 1.

3. Compare FunTime and MyFirstApplication and list the similarities and differences.

2.3 Edit–Compile–Run Cycle

We will walk through the steps involved in executing the first sample program. What we outline here are the overall steps common to any Java development tool you use. You need to get detailed instructions on how to use your chosen development tool to actually run programs. The steps we present in this section should serve as a guideline for more detailed instructions specific to your program development tool.

Step 1

Type in the program using an editor and save the program to a file. Use the name of the main class and the suffix .java for the filename. This file, in which the program is in a human-readable form, is called a source file.

```
import javabook.*;

class MyFirstApplication {
    public static void main (String[] args) {
        MainWindow mainWindow;
        mainWindow = new MainWindow();
        mainWindow.setVisible(true);
    }
}
```

It is critical that you configure the development tool properly so the compiler knows where to locate the javabook package. Each development tool requires you to set the configuration differently, so please consult the relevant materials for more information.
Step 2

Compile the source file. Many compilers require you to create a project file and then place the source file in the project file in order to compile the source file. When the compilation is successful, the compiled version of the source file is created. This compiled version is called bytecode, and the file that contains bytecode is called a bytecode file. The compiler-generated bytecode file will have the same name as the source file with the suffix .class.

When any error occurs in a program, an error message will be displayed. If the sample program contains no errors in syntax, then instead of an error message, you will get a message stating something like “Compiled successfully.” To see what kind of error messages are displayed, try compiling the following program. We purposely introduced three errors—can you find them? Make sure to compile the correct MyFirstApplication again before proceeding to the next step.

class MyFirstApplication //BAD version
{
    public static void main(String[] args)
    {
        mainWindow = new MainWindow();
        mainWindow.setVisible( true );
    }
}

Errors detected by the compiler are called compilation errors. Compilation errors are actually the easiest type of errors to correct. Most compilation errors are due to the violation of syntax rules.
Step 3
Execute the bytecode file. An interpreter will go through the bytecode file and execute the instructions in it. If your program is error free, a window will appear on the screen.

If an error occurs while running the program, the interpreter will catch it and stop its execution. Errors detected by the interpreter are called execution errors. If you did not see the expected results, go back to the previous steps and verify that your program is entered correctly. If you still do not see the expected results, then most likely your development environment is not set up correctly. Please refer to other sources of information for further help.

2.4 The javabook Package

We have used the MainWindow and SketchPad classes from the javabook package in our sample programs. There are many useful classes in the javabook package, and in the next section, we will show you the third class from the package. We decided to provide the javabook package with this book because

To become a good object-oriented programmer, one must first learn how to use predefined classes.

Eventually, you must learn how to define your own classes, the classes you will reuse in writing programs. But before you can become adept in defining your own classes, you must first learn how to use existing classes. For this purpose, we provide the javabook classes in this book. Learning first to use the predefined javabook classes has the following advantages:
Chapter 2  Java Programming Basics

1. It gives you a taste of how real-world programs are developed. In real-world object-oriented programming, you develop programs by reusing existing classes whenever possible. You will get hands-on experience of code reuse by using classes from the javabook package.

2. It minimizes the impact of programming language syntax and semantics. The use of javabook classes lets students concentrate on learning concepts instead of Java language features. Using predefined classes minimizes the impact of programming language because these predefined classes hide the complexity of the underlying programming language. Remember that our objective is to teach object-oriented thinking, not Java language.

3. It allows you to write practical programs without learning too many details of the Java language. Java comes with a number of standard packages, but using the standard classes such as java.awt from the beginning is not practical because these classes require programming sophistication that beginning students do not possess. Easy-to-use classes such as the javabook classes are most appropriate for beginning programmers.

4. It serves as a good example of how to design classes. When the time comes for you to design your own classes, intuitive and easy to use classes from javabook should serve as your model.

Although we begin teaching object-oriented programming by using the javabook classes, you will not be dependent on them. You will become an object-oriented programmer, not a javabook programmer. We use javabook because it is pedagogically sound to do so. Complete documentation of the javabook classes are provided in Appendix A.

2.5 Sample Program: Displaying Messages

Now that you have acquired a basic understanding of Java application programs, let’s write a new application. We will go through the design, coding, and testing phases of the software life cycle to illustrate the development process. Since the program we develop here is very simple, we can write it without really going through the phases. However, it is extremely important for you to get into a habit of developing a program following the software life cycle stages. Small programs can be developed in a haphazard manner, but not large programs. We will teach you the development process with small programs first, so you will be ready to use it to develop large programs later.
Problem Statement
We start our development with a problem statement. The problem statement for our sample programs will be short, ranging from a sentence to a paragraph, but the problem statement for complex and advanced applications may contain many pages. Here’s the problem statement for this sample program:

Write an application that displays the message I Love Java.

Design
In the design stage, we translate the given problem statement into a design document that can be implemented. For object-oriented programs, a design document will include a list of classes. The classes in the design document are either predefined or custom-made. For each class in the list, we identify its purpose, the methods and data values that will be defined, assumptions made in its use, and so forth. We will begin with a very simplistic design document and gradually build more detailed design documents as our sample programs become more complex later in the book.

The problem states that the program is to display a message. It does not specify how, so in the design stage, we will decide how to do this. From what we know so far, we can do this by setting the title of the MainWindow object like this:
MainWindow mainWindow;
mainWindow = new MainWindow("I Love Java");
mainWindow.setVisible( true );

Is this a good design? This is the best we could do from our limited knowledge, but most likely, this is not what the user wants. When the design does not meet the user’s needs, we go back to the drawing board and redesign.

The first thing we do in redesign is to search for a class or classes that will perform the task we want to implement. We may start searching from the standard Java classes. If we can find suitable classes, we will use them. We may have to build a new class from these existing classes. If no suitable classes are found or building one from the existing classes seems too difficult, we will search other packages. We could search for free packages available on the Internet or packages supplied by your school’s computer center. Some software companies may sell a package that contains the classes we want. If the package is within our budget, we may buy it. If no such package can be found, then we will develop the necessary classes ourselves.

In our case, we will search the javabook package first. As you become more proficient in Java language, you may want to search the Java standard libraries first. If you go through the package documentation in Appendix A, you will notice a class called MessageBox. This is the class we are looking for.

We will describe MessageBox very briefly here. More information on MessageBox will be given later as we use more of its features. A MessageBox object is used to display a single line of text. Figure 2.11 shows the MessageBox object with the message I Love Java. When you click the OK button, the object disappears from the screen. The sequence of declaring, creating, and sending the message is

```java
MessageBox messageBox;
messageBox = new MessageBox( mainWindow );
messaging.messageBox.show("I Love Java");
```

A MessageBox object is a special kind of window called a dialog box, or more simply, dialog. Every dialog box requires another kind of window called a frame as its owner. A MainWindow object is a frame window. We will elaborate on their differences later. For now, it suffices to know that a frame is a general-purpose window and a dialog is a limited-purpose window used primarily for displaying simple information such as error messages or getting a simple response such as yes or no.

We call a dialog box a subordinate of the owner frame. One characteristic of this relationship is that a subordinate dialog always appears in front of its owner.
frame. The owner–subordinate relationship is established when a dialog box is created by passing the owner frame window as an argument in `new`, as in

```java
messageBox = new MessageBox( mainWindow );
```

And we display the text by passing it as an argument in the `show` message to a `MessageBox` object:

```java
messageBox.show( "I Love Java" );
```

The object diagram of the program is shown in Figure 2.10 and here’s the design document of the program:

<table>
<thead>
<tr>
<th>Design Document: DisplayMessage</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>Purpose</strong></td>
</tr>
<tr>
<td>DisplayMessage</td>
<td>The main class of the program.</td>
</tr>
<tr>
<td>MainWindow</td>
<td>The main frame window of the program. The title is set to Display Message. This class is from javabook.</td>
</tr>
<tr>
<td>MessageBox</td>
<td>The dialog for displaying the required message. This class is from javabook.</td>
</tr>
</tbody>
</table>

**FIGURE 2.10** The object diagram for the DisplayMessage program.
Chapter 2 Java Programming Basics

Coding
After the design is completed, we translate it into an actual code. We will implement this program using the program template of Figure 2.9.

Here’s the source code for the program DisplayMessage:

```java
import javabook.*;

class DisplayMessage {
    public static void main(String[] args) {
        // declare two objects
        MainWindow mainWindow;
        MessageBox messageBox;

        // create two objects
        mainWindow = new MainWindow("Display Message");
        messageBox = new MessageBox(mainWindow);

        // display two objects: first the frame and then the dialog
        mainWindow.setVisible(true);
        messageBox.show("I Love Java");
    }
}
```

Testing
After the program is written, we test the program to verify that the program runs as intended. Since this program is very simple, there’s not much testing strategy we can employ here. We just run the program and make sure that the main window and the dialog appear on the screen as shown in Figure 2.11. For subsequent sample programs, the testing strategy will be more involved.

2.6 The First Java Applet

Now let’s move on to the second type of Java program—an applet. Here’s a Java applet that displays the message I Love Java and draws a rectangle around the text:

```java
/* Program MyFirstApplet
   An applet that displays the text "I Love Java"
   and a rectangle around the text.
*/
```
Notice a resemblance in the program structure between this applet and the applications we wrote earlier. Applications and applets are both programs, and, therefore, they share the same program structure, composed of three parts: a comment, an import statement, and a class declaration. Figure 2.12 identifies the three components in the applet. We include the header comment to describe the applet and two import statements to import the required `java.applet` and `java.awt` packages. The classes `Applet` and `Graphics` used in the applet are defined in `java.applet` and `java.awt`. The object diagram of `MyFirstApplet` is shown in Figure 2.13.

Notice that `MyFirstApplet` does not have a main method, because the `AppletViewer` class is the main class of the program. For applications, we define our own main class, but for applets, we use the predefined main class `AppletViewer`.

```java
import java.applet.*;
import java.awt.*;

public class MyFirstApplet extends Applet
{
   public void paint( Graphics graphic)
   {
      graphic.drawString("I Love Java",70,70);
      graphic.drawRect(50,50,100,30);
   }
}
```

**FIGURE 2.11** Result of running the DisplayMessage program.
For applets, we define a subclass of `Applet` that the `AppletViewer` uses when executed. A subclass of `Applet` is declared as:

```java
import java.awt.*;

public class MyFirstApplet extends Applet {
    public void paint(Graphics graphic) {
        graphic.drawString("I Love Java", 70, 70);
        graphic.drawRect(50, 50, 100, 30);
    }
}
```
public class `MyFirstApplet` extends `Applet`

We declared one method in the `MyFirstApplet` class:

```java
public void paint(Graphics graphic)
{
    graphic.drawString("I Love Java", 70, 70);
    graphic.drawRect(50, 50, 100, 30);
}
```

The `AppletViewer` class automatically sends the `paint` message with a `Graphics` object as an argument to an applet. We use `Graphics` objects to display text, lines, and other graphics. In the sample applet, we send two messages to the object `graphic`. The first message
The first message

```java
graphic.drawString("I Love Java", 70, 70);
```
displays the text I Love Java at the specified position (70, 70). The position is determined in the manner illustrated in Figure 2.14.

**FIGURE 2.14** The diagram illustrates how the position of text is determined by the `drawString` method.

**Syntax**

```
<text> is displayed at position (<x>,<y>).
```

**Example:**

```java
graphic.drawString("I Love Java", 70, 70);
```

The second message

```java
graphic.drawRect(50, 50, 100, 30);
```
displays a rectangle 100 pixels wide and 30 pixels high at the specified position (50, 50). The position is determined as illustrated in Figure 2.15.

A program template for simple Java applets is shown in Figure 2.16.
2.6 The First Java Applet

The diagram illustrates how the position of a rectangle is determined by the `drawRect` method.

**Syntax**
A rectangle `<width>` wide and `<height>` high is displayed at position `<x>,<y>`.

```
graphic.drawRect(<x>, <y>, <width>, <height>);
```

**Example:**
```
graphic.drawRect(50,50,100,30);
```

**FIGURE 2.16**
A program template for simple Java applets.

```
import java.applet.*;
import java.awt. *

public class extends Applet
{
    public void paint(Graphics graphic)
    {
    }
}
```
Table 2.1 lists some of the available graphic-drawing methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>drawLine(x1,y1,x2,y2)</td>
<td>Draws a line between (x1,y1) and (x2,y2).</td>
</tr>
<tr>
<td>drawRect(x,y,w,h)</td>
<td>Draws a rectangle with width w and height h at (x,y).</td>
</tr>
<tr>
<td>drawRoundRect(x,y,w,h,aw,ah)</td>
<td>Draws a rounded-corner rectangle with width w and height h at (x,y). Parameters aw and ah determine the angle for the rounded corners.</td>
</tr>
<tr>
<td>drawOval(x,y,w,h)</td>
<td>Draws an oval with width w and height h at (x,y).</td>
</tr>
<tr>
<td>drawString(&quot;text&quot;,x,y)</td>
<td>Draws the string text at (x,y).</td>
</tr>
</tbody>
</table>
The steps you take to run the applet are essentially the same as those you take to run the application. The only difference is the creation of an html file and the use of an applet viewer (or a web browser) to run the applet. The next diagram illustrates the steps you take to run the applet:

Executing the applet MyFirstApplet displays a window called an applet viewer on the screen as shown in Figure 2.17. The AppletViewer class is responsible for displaying an applet viewer window and running the applet within the applet viewer window. The applet viewer provides a quick and easy way to run and test applets without using any web browser. When you use a web browser to run applets, then the browser assumes the role of the main class.

We mentioned that an applet is a mini-application that is intended to be executed within a web browser. A web browser reads a formatted document called a Web document or Web page. A web document written in HyperText Markup Language (HTML) is called an HTML document, and a file that contains an HTML document is an HTML file. We use the suffix .html or .htm to name an HTML file. HTML is the standard language used for writing web documents and consists of text and formatting tags called markup elements that specify the
format for headers, paragraphs, hyperlinks, and other components of a document. One of the markup elements specifies the execution of an applet, giving the name of an applet to be executed, the area in the browser in which the applet is executed, and so forth.

We need an HTML file that refers to an applet’s bytecode file to run the applet. We use the html file MyFirstApplet.html to run the MyFirstApplet applet.

```
<APPLET CODE="MyFirstApplet.class" WIDTH=300 HEIGHT=190>
</APPLET>
```

MyFirstApplet.html

The details of HTML are beyond the scope of this book. However, we will describe the portion necessary to run our sample applets. You can use this HTML file as a template to use with other applets by changing the APPLET tag (the
third line in the MyFirstApplet.html file). This tag indicates the applet’s bytecode filename and the size of the rectangular area reserved for the applet:

\[ <APPLET CODE="MyFirstApplet.class" WIDTH=300 HEIGHT=190> \]

For example, to run MyFirstApplet in a smaller area, we can change the tag to

\[ <APPLET CODE="MyFirstApplet.class" WIDTH=200 HEIGHT=100> \]

Although the AppletViewer class can read any web page, it processes only the APPLET tag and ignores other tags. So if you want to run an applet in a web page that processes all tags, you need to use a Java-aware web browser. Figure 2.18 shows MyFirstApplet executed in a popular web browser, Netscape Navigator (version 3.0). Notice that Navigator is running on the Macintosh. The applet was written on the Windows 95 platform but executed in a browser on a different platform. We use the HTML file JavaBook.html that contains more tags (such as specifying the title Intro to Programming with Java and the background spiral notebook image).

### Quick Check

1. How are an applet viewer and a web browser different?
2. Which class is the main class of an applet?
3. Which method of the Graphics class do you use to draw a rectangle?
4. What is the purpose of the APPLET tag in an HTML file?

---

1. Some web browsers do not understand the APPLET tag. You need Java-aware web browsers, such as Netscape Navigator 4.0 or later and MS Internet Explorer 3.0 or later that understand the APPLET tag to run Java applets.
FIGURE 2.18  
Result of running the applet **MyFirstApplet** in Netscape Navigator.
2.7 Exercises

1. Identify all errors in the following program:

```java
/*
   Program Exercise1
   A program with many errors.
   //
   import javabook.mainwindow;

   class Exercise 1{
       public void Main() {
           MainWindow mainWindow;
           mainWindow.show();
       }
   }
```

2. Identify all errors in the following program:

```java
//
   Program Exercise2
   A program with many errors.
   //
   import JavaBook.*;

   class TWO{
       public static void main method() {
           mainWindow mainWindow;
           MessageBox mybox1, mybox2;
           mainWindow = new MainWindow();
           mybox2 = new MessageBox();
           mybox2.show;}
   }
```

3. Identify all of the errors in the following program:
4. Describe the purpose of comments. Name the types of comments available. Can you include comment markers inside a comment?

5. What is the purpose of the import statement? Does a Java program always have to include an import statement?

6. Show the syntax for importing one class and all classes in a package.

7. Describe the class that must be included in any Java application.

8. What is a reserved word? List all the Java reserved words mentioned in this chapter.

9. Which of the following are invalid identifiers?
   a. R2D2  
   b. Whatchamacallit  
   c. HowAboutThis?  
   d. Java  
   e. GoodChoice  
   f. 12345  
   g. 3CPO  
   h. This is okay.  
   i. thisIsReallyOkay  
   j. aPPlEt  
   k. Bad-Choice  
   l. A12345

10. Describe the steps you take to run a Java application and the tools you use in each step. What are source files and bytecode files? What different types of errors are detected at each step?

11. Describe the difference between object declaration and object creation. Use a state-of-memory diagram to illustrate the difference.

12. Show a state-of-memory diagram after each of the following statements is executed:
13. Show a state-of-memory diagram after each of the following statements is executed:

```java
Person person1, person2;
person1 = new Person();
person2 = new Person();
```

14. Which of the following identifiers violate the naming convention for class names?

a. r2D2 e. CPO
b. whatchamacallit f. ThisIsReallyOkay
c. Java g. java
d. GoodName h. aPPlet

15. Which of the following identifiers violate the naming convention for object names?

a. R2D2 e. 3CPO
b. isthisokay? f. ThisIsReallyOkay
c. Java g. java
d. goodName h. anotherbadone

16. How are Java applets different from Java applications?

17. What is a web page? What tag is required to run an applet in a web browser?

18. Write a Java application that displays a MainWindow window 300 pixels wide and 200 pixels high with the title This is My first application. You
change the size of a window by sending the `setSize` message to the window. The syntax for `setSize` is

```java
setSize( <width>, <height> )
```

19. In exercise 18, you used the `setSize` method to set the dimension of a `MainWindow` window. This method will let you set the size of a window, but not the location where this window appears on the screen. To position a window at a specified location, you use the `setLocation` method as in

```java
// assume mainWindow is declared and created
mainWindow.setLocation( 50, 50 );
mainWindow.setVisible( true );
```

Through experimentation, determine how the two arguments in the `setLocation` method affect the positioning of the window.

20. Write a Java application that displays two separate messages I Can Design and And I Can Program.

21. Write a Java application that displays a very long message. Try a message that is wider than the display of your computer screen and see what happens.

22. Write a Java applet that displays the text `Hello` in an applet viewer as illustrated below:

```
Applet Viewer
applet started
   Hello   Hello
       Hello
   Hello   Hello
applet
```
23. Write a Java applet that draws a house in an applet viewer. You might want to draw a house that is much more interesting than the one shown here. Use the methods listed in Table 2.1 as needed.

![Applet Viewer](image)

24. Add the moon and a tree to the house you drew in exercise 23.

25. If you know how to open an HTML file from a Web browser, run the Java applets from exercises 22 through 24 in a web browser.

26. Because today’s computers are very fast, you will probably not notice any discernible difference on the screen between the code

```java
MainWindow myWindow;
myWindow = new MainWindow( );
myWindow.setVisible( true );
```

and

```java
MainWindow myWindow;
myWindow = new MainWindow( );
myWindow.setVisible( true );
myWindow.setVisible( false );
myWindow.setVisible( true );
```

One way to see the disappearance and reappearance of the window is to put a delay between the successive `setVisible` messages. To put a delay, you can use a `Clock` object from the `javabook` package. Here’s a simple usage of the `Clock` class:

```java
Clock myClock;
myClock = new Clock( );

//put statement X here
```
myClock.pause( 2 );

//put statement Y here

The unit for the argument you pass in the pause message is seconds. If you want a half-second delay, for example, then you pass 0.5 as an argument. Using the Clock class, write a program that makes a MainWindow object appear, disappear, and appear again. The window remains visible for 5 seconds when it appears for the first time, and once it disappears, it won’t reappear for 3 seconds.

27. At the Dr. Caffeine’s website, you will find a Java package called galapagos (www.drcaffeine.com/packages). The galapagos package includes a Turtle class that is modeled after Seymour Papert’s Logo. This Turtle has a pen, and when you move the Turtle, its pen will trace the movement. So by moving a Turtle object, you can draw many different kinds of geometrical shapes. For example, the following program commands a Turtle to draw a square:

```java
import galapagos.*;

class Square {
    public static void main( String[] arg ) {
        Turtle turtle;
        turtle = new Turtle( );

        turtle.move( 50 );
        turtle.turn( 90 ); //make it turn 90 degrees
                        //counterclockwise
        turtle.move( 50 );
        turtle.turn( 90 );

        turtle.move( 50 );
        turtle.turn( 90 );

        turtle.move( 50 );

        turtle.move( 50 );
    }
}
```

Write a program to draw a triangle. Read the documentation and see if you can find a way to draw the square in a different color and line thickness.

28. Write another program to draw a star using a Turtle from exercise 27.