2.2 Your First Program in Java: Printing a Line of Text

A Java **application** is a computer program that executes when you use the **java command** to launch the Java Virtual Machine (JVM). Sections 2.2.1–2.2.2 discuss how to compile and run a Java application. First we consider a simple application that displays a line of text. Figure 2.1 shows the program followed by a box that displays its output.

```
// Fig. 2.1: Welcome1.java
// Text-printing program.

public class Welcome1 {
   // main method begins execution of Java applic
   public static void main(String[] args) {
      System.out.println("Welcome to Java Program
   } // end method main
} // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. 2.1

Text-printing program.

We use line numbers for instructional purposes—they’re *not* part of a Java program. This example illustrates several important Java features. We’ll see that line 7 does the work—displaying the phrase `"Welcome to Java Programming!"` on the screen.

Commenting Your Programs

We insert **comments** to document programs and improve their readability. The Java compiler *ignores* comments, so they do *not* cause the computer to perform any action when the program is run.

By convention, we begin every program with a comment indicating the figure number and the program’s filename. The comment in line 1

```
// Fig. 2.1: Welcome1.java
```

begins with `//`, indicating that it’s an **end-of-line comment**—it terminates at the end of the line on which the `//` appears. An end-of-line comment need not begin a line; it also can begin in the middle of a line and continue until the end (as in lines 5, 8 and 9). Line 2,

```
// Text-printing program.
```

by our convention, is a comment that describes the purpose of the program.

Java also has **traditional comments**, which can be spread over several lines as in

```
/* This is a traditional comment. It
   can be split over multiple lines */
```

These begin with the delimiter **/*** and end with ***/**. The compiler ignores all text between the delimiters. Java incorporated traditional comments and end-of-line comments from the C and C++ programming languages, respectively.

Java provides comments of a third type—**Javadoc comments**. These are delimited by **/**** and ***/**. The compiler ignores all text between the delimiters. Javadoc comments enable you to embed program documentation directly in your programs. Such comments are the preferred Java documenting format in industry. The `javadoc` **utility program** (part of the JDK) reads Javadoc comments and uses them to prepare program documentation in HTML5 web-page format. We use // comments throughout our code, rather than traditional or Javadoc comments, to save space. We demonstrate Javadoc comments and the `javadoc` utility in online Appendix G, Creating Documentation with `javadoc`.

Common Programming Error 2.1

Forgetting one of the delimiters of a traditional or Javadoc comment is a syntax error. A ***syntax error*** *occurs when the compiler encounters code that violates Java’s language rules (i.e., its syntax). These rules are similar to natural-language grammar rules specifying sentence structure, such as those in English, French, Spanish, etc. Syntax errors are also called* ***compiler errors****,* ***compile-time errors*** *or* ***compilation errors****, because the compiler detects them when compiling the program. When a syntax error is encountered, the compiler issues an error message. You must eliminate all compilation errors before your program will compile properly.*

Good Programming Practice 2.1

Some organizations require that every program begin with a comment that states the purpose of the program and the author, date and time when the program was last modified.

Using Blank Lines

Blank lines (like line 3), space characters and tabs can make programs easier to read. Together, they’re known as **white space**. The compiler ignores white space.

Good Programming Practice 2.2

Use white space to enhance program readability.

Declaring a Class

Line 4

```
public class Welcome1 {
```

begins a **class declaration** for class `Welcome1`. Every Java program consists of at least one class that you define. The `class` **keyword** introduces a class declaration and is immediately followed by the **class name** (`Welcome1`). **Keywords** are reserved for use by Java and are spelled with all lowercase letters. The complete list of keywords is shown in Appendix C.

In Chapters 2–7, every class we define begins with the `public` keyword. For now, we simply require it. You’ll learn more about `public` and non-`public` classes in Chapter 8.

Filename for a **`public`** Class

A `public` class *must* be placed in a file that has a filename of the form *ClassName*`.java`, so class `Welcome1` is stored in the file `Welcome1.java`.

Common Programming Error 2.2

A compilation error occurs if a `public` *class's filename is not exactly the same name as the class (in terms of both spelling and capitalization) followed by the* `.java` *extension.*

Class Names and Identifiers

By convention, class names begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`). A class name is an **identifier**—a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does *not* begin with a digit and does *not* contain spaces. Some valid identifiers are `Welcome1`, `$value`, `_value`, `m_inputField1` and `button7`. The name `7button` is *not* a valid identifier because it begins with a digit, and the name `input field` is *not* a valid identifier because it contains a space. Normally, an identifier that does not begin with a capital letter is not a class name. Java is **case sensitive**—uppercase and lowercase letters are distinct—so `value` and `Value` are different (but both valid) identifiers.

Good Programming Practice 2.3

By convention, every word in a class-name identifier begins with an uppercase letter. For example, the class-name identifier `DollarAmount` *starts its first word,* `Dollar`*, with an uppercase* `D` *and its second word,* `Amount`*, with an uppercase A. This naming convention is known as **camel case**, because the uppercase letters stand out like a camel's humps.*

Underscore (_) in Java 9

As of Java 9, you can no longer use an underscore (_) by itself as an identifier.

Class Body

A **left brace** (at the end of line 4), **{**, begins the **body** of every class declaration. A corresponding **right brace** (at line 9), **}**, must end each class declaration. Lines 5–8 are indented.

Good Programming Practice 2.4

Indent the entire body of each class declaration one “level” between the braces that delimit the class’s. This format emphasizes the class declaration’s structure and makes it easier to read. We use three spaces to form a level of indent—many programmers prefer two or four spaces. Whatever you choose, use it consistently.

Good Programming Practice 2.5

IDEs typically indent code for you. The Tab *key may also be used to indent code. You can configure each IDE to specify the number of spaces inserted when you press* Tab.

Common Programming Error 2.3

It’s a syntax error if braces do not occur in matching pairs.

Error-Prevention Tip 2.1

When you type an opening left brace, `{`*, immediately type the closing right brace,* `}`*, then reposition the cursor between the braces and indent to begin typing the body. This practice helps prevent errors due to missing braces. Many IDEs do this for you.*

Declaring a Method

Line 5

```
// main method begins execution of Java application
```

is a comment indicating the purpose of lines 6–8 of the program. Line 6

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

is the starting point of every Java application. The **parentheses** after the identifier `main` indicate that it’s a program building block called a **method**. Java class declarations normally contain one or more methods. For a Java application, one of the methods *must* be called `main` and must be defined as in line 6; otherwise, the program will not execute.

Methods perform tasks and can return information when they complete their tasks. We'll explain the purpose of keyword `static` in Section 3.2.5. Keyword **`void`** indicates that this method will *not* return any information. Later, we'll see how a method can return information. For now, simply mimic `main`'s first line in your programs. The `String[] args` in parentheses is a required part of `main`'s declaration—we discuss this in Chapter 7.

The left brace at the end of line 6 begins the **body of the method declaration**. A corresponding right brace ends it (line 8). Line 7 is indented between the braces.

Good Programming Practice 2.6

Indent the entire body of each method declaration one "level" between the braces that define the method's body. This emphasizes the method's structure and makes it easier to read.

Performing Output with `System.out.println`

Line 7

```
System.out.println("Welcome to Java Programming!");
```

instructs the computer to perform an action—namely, to display the characters between the double quotation marks. The quotation marks themselves are *not* displayed. Together, the quotation marks and the characters between them are a **string**—also known as a **character string** or a **string literal**. White-space characters in strings are *not* ignored by the compiler. Strings *cannot* span multiple lines of code—later we'll show how to conveniently deal with long strings.

The `System.out` object—which is predefined for you—is known as the **standard output object**. It allows a program to display information in the **command window** from which the program executes. In Microsoft Windows, the command window is the **Command Prompt**. In UNIX/Linux/macOS, the command window is called a **terminal** or a **shell**. Many programmers call it simply the **command line**.

Method `System.out.println` displays (or prints) a *line* of text in the command window. The string in the parentheses in line 7 is the method's **argument**. When `System.out.println` completes its task, it positions the output cursor (the location where the next character will be displayed) at the beginning of the next line in the command window. This is similar to what happens when you press the *Enter* key while typing in a text editor—the cursor appears at the beginning of the next line in the document.

The entire line 7, including `System.out.println`, the argument `"Welcome to Java Programming!"` in the parentheses and the **semicolon** (**;**), is called a **statement**. A method typically contains statements that perform its task. Most statements end with a semicolon.

Using End-of-Line Comments on Right Braces for Readability

As an aid to programming novices, we include an end-of-line comment after a closing brace that ends a method declaration and after a closing brace that ends a class declaration. For example, line 8

```
      } // end method main
```

indicates the closing brace of method `main`, and line 9

```
   } // end class Welcome1
```

indicates the closing brace of class `Welcome1`. Each comment indicates the method or class that the right brace terminates. We’ll omit such ending comments after this chapter.

2.2.1 Compiling the Application

We’re now ready to compile and execute the program. We assume you’re using the Java Development Kit’s command-line tools, not an IDE. The following instructions assume that the book’s examples are located in `c:\examples` on Windows or in your user account’s `Documents/examples` folder on Linux/macOS.

To prepare to compile the program, open a command window and change to the directory where the program is stored. Many operating systems use the command `cd` to change directories (or folders). On Windows, for example,

```
cd c:\examples\ch02\fig02_01
```

changes to the `fig02_01` directory. On UNIX/Linux/macOS, the command

```
cd ~/Documents/examples/ch02/fig02_01
```

changes to the `fig02_01` directory. To compile the program, type

```
javac Welcome1.java
```

If the program does not contain compilation errors, this command creates the file called `Welcome1.class` (known as `Welcome1`’s **class file**) containing the platform-independent Java bytecodes that represent our application. When we use the `java` command to execute the application on a given platform, the JVM will translate these bytecodes into instructions that are understood by the underlying operating system and hardware.

Common Programming Error 2.4

The compiler error message “`class Welcome1 is public, should be declared in a file named Welcome1.java`” *indicates that the filename does not match the name of the* `public` *class in the file or that you typed the class name incorrectly when compiling the class.*

When learning how to program, sometimes it’s helpful to “break” a working program to get familiar with the compiler’s error messages. These messages do not always state the exact problem in the code. When you encounter an error, it will give you an idea of what caused it. Try removing a semicolon or brace from the program of Fig. 2.1, then recompiling to see the error messages generated by the omission.

Error-Prevention Tip 2.2

When the compiler reports a syntax error, it may not be on the line that the error message indicates. First, check the line for which the error was reported. If you don't find an error on that line, check several preceding lines.

Each compilation-error message contains the filename and line number where the error occurred. For example, `Welcome1.java:6` indicates that an error occurred at line 6 in `Welcome1.java`. The rest of the message provides information about the syntax error.

2.2.2 Executing the Application

Now that you've compiled the program, type the following command and press *Enter*:

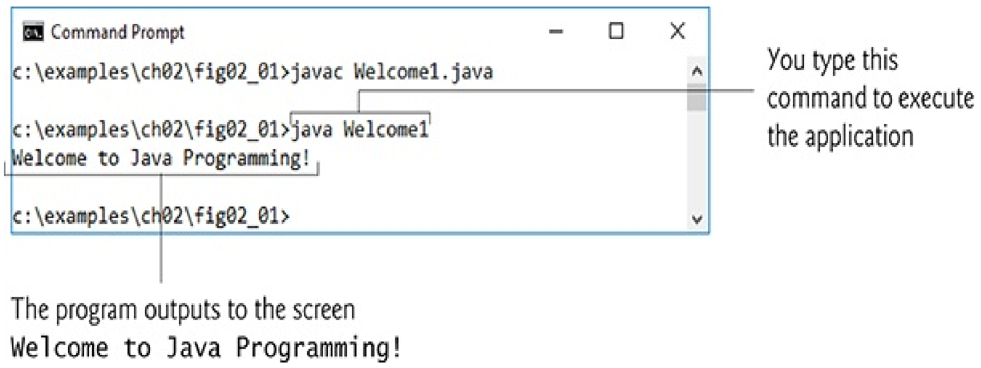
```
java Welcome1
```

to launch the JVM and load the `Welcome1.class` file. The command *omits* the `.class` file-name extension; otherwise, the JVM will *not* execute the program. The JVM calls `Welcome1`'s `main` method. Next, line 7 of `main` displays `"Welcome to Java Programming!"`. Figure 2.2 shows the program executing in a Microsoft Windows

Command Prompt window. [*Note:* Many environments show command windows with black backgrounds and white text. We adjusted these settings to make our screen captures more readable.]

Error-Prevention Tip 2.3

When attempting to run a Java program, if you receive a message such as "`Exception in thread "main" java.lang.NoClassDefFoundError: Welcome1`*," your* `CLASSPATH` *environment variable has not been set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, you may need to reboot your computer or open a new command window after configuring the* `CLASSPATH`.



```
c:\examples\ch02\fig02_01>javac Welcome1.java

c:\examples\ch02\fig02_01>java Welcome1
Welcome to Java Programming!

c:\examples\ch02\fig02_01>
```

Command Prompt
You type this command to execute the application
The program outputs to the screen Welcome to Java Programming!

Fig. 2.2

Executing `Welcome1` from the **Command Prompt**.

Description