1.9 A Typical Java Development Environment

We now explain the steps to create and execute a Java application. Normally there are five phases—edit, compile, load, verify and execute. We discuss them in the context of the Java SE 8 Development Kit (JDK). See the *Before You Begin* section *for information on downloading and installing the JDK on Windows, Linux and macOS.*

Phase 1: Creating a Program

Phase 1 consists of editing a file with an *editor program*, normally known simply as an *editor* (Fig. 1.6). Using the editor, you type a Java program (typically referred to as **source code**), make any necessary corrections and save it on a secondary storage device, such as your hard drive. Java source code files are given a name ending with the **`.java` extension**, indicating that the file contains Java source code.

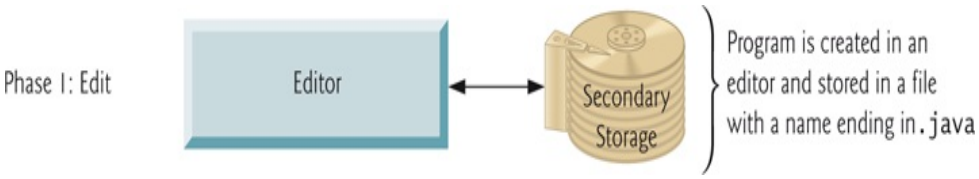

Fig. 1.6

Typical Java development environment—editing phase.

Two editors widely used on Linux systems are `vi` and `emacs` (). Windows provides **Notepad**. macOS provides **TextEdit**. Many freeware and shareware editors are also available online, including Notepad++ (`http://notepad-plus-plus.org`), EditPlus (`http://www.editplus.com`), TextPad (`http://www.textpad.com`), jEdit (`http://www.jedit.org`) and more.

Integrated development environments (IDEs) provide tools that support the software development process, such as editors, debuggers for locating **logic errors** that cause programs to execute incorrectly and more. The most popular Java IDEs are:

- Eclipse (`http://www.eclipse.org`)
- IntelliJ IDEA (`http://www.jetbrains.com`)
- NetBeans (`http://www.netbeans.org`)

On the book's website at

```
http://www.deitel.com/books/jhtp11
```

we provide videos that show you how to execute this book's Java applications and how to develop new Java applications with Eclipse, NetBeans and IntelliJ IDEA.

Phase 2: Compiling a Java Program into Bytecodes

In Phase 2, you use the command **`javac`** (the **Java compiler**) to **compile** a program (Fig. 1.7). For example, to compile a program called `Welcome.java`, you'd type

```
javac Welcome.java
```

in your system's command window (i.e., the **Command Prompt** in Windows, the **Terminal** application in macOS) or a Linux shell (also called **Terminal** in some Linux versions). If the program compiles, the compiler produces a `.class` file called `Welcome.class`. IDEs typically provide a menu item, such as **Build** or **Make**, that invokes the `javac` command for you. If the compiler detects errors, you'll need to go back to Phase 1 and correct them. In Chapter 2, we'll say more about the kinds of errors the compiler can detect.

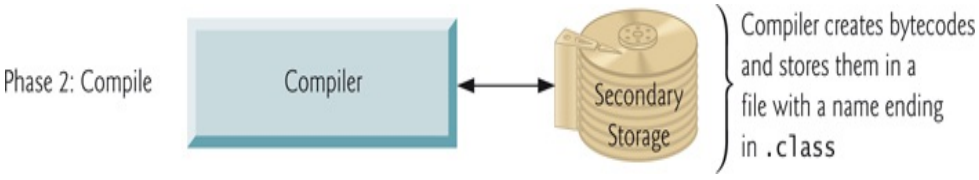

Fig. 1.7

Typical Java development environment—compilation phase.

Common Programming Error 1.1

When using `javac`*, if you receive a message such as* "`bad command or filename`," "`javac: command not found`" *or* "`'javac' is not recognized as an internal or external command, operable program or batch file`," *then your Java software installation was not completed properly. This indicates that the system's* `PATH` *environment variable was not set properly. Carefully review the installation instructions in the Before You Begin section of this book. On some systems, after correcting the* `PATH`*, you may need to reboot your computer or open a new command window for these settings to take effect.*

The Java compiler translates Java source code into **bytecodes** that represent the tasks to execute in the execution phase (Phase 5). The **Java Virtual Machine** (**JVM**)—a part of the JDK and the foundation of the Java platform—executes bytecodes. A **virtual machine** (**VM**) is a software application that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it. If the same VM is implemented on many computer platforms, applications written for that type of VM can be used on all those platforms. The JVM is one of the most widely used virtual machines. Microsoft's .NET uses a similar virtual-machine architecture.

Unlike machine-language instructions, which are *platform*

dependent (that is, dependent on specific computer hardware), bytecode instructions are *platform independent*. So, Java's bytecodes are **portable**—without recompiling the source code, the same bytecode instructions can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled. The JVM is invoked by the **`java`** command. For example, to execute a Java application called `Welcome`, you'd type the command

```
java Welcome
```

in a command window to invoke the JVM, which would then initiate the steps necessary to execute the application. This begins Phase 3. IDEs typically provide a menu item, such as **Run**, that invokes the `java` command for you.

Phase 3: Loading a Program into Memory

In Phase 3, the JVM places the program in memory to execute it—this is known as **loading** (Fig. 1.8). The JVM's **class loader** takes the `.class` files containing the program's bytecodes and transfers them to primary memory. It also loads any of the `.class` files provided by Java that your program uses. The `.class` files can be loaded from a disk on your system or over a network (e.g., your local college or company network, or the Internet).

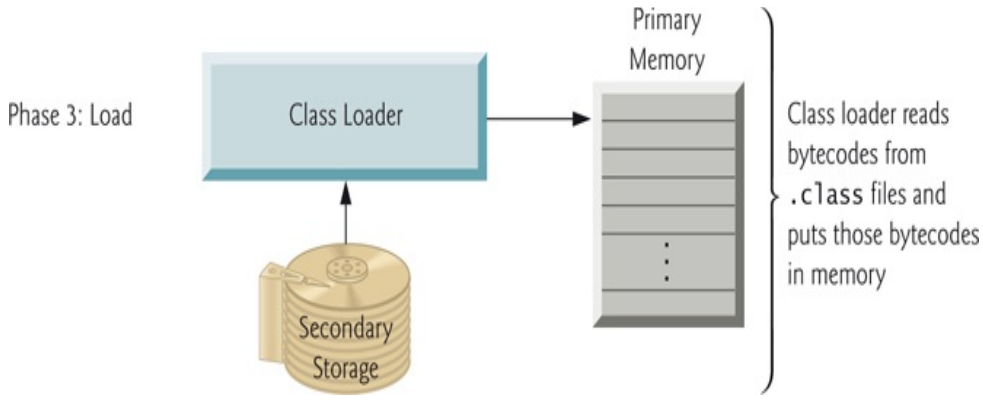

Fig. 1.8

Typical Java development environment—loading phase.

Phase 4: Bytecode Verification

In Phase 4, as the classes are loaded, the **bytecode verifier** examines their bytecodes to ensure that they’re valid and do not violate Java’s security restrictions (Fig. 1.9). Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might).

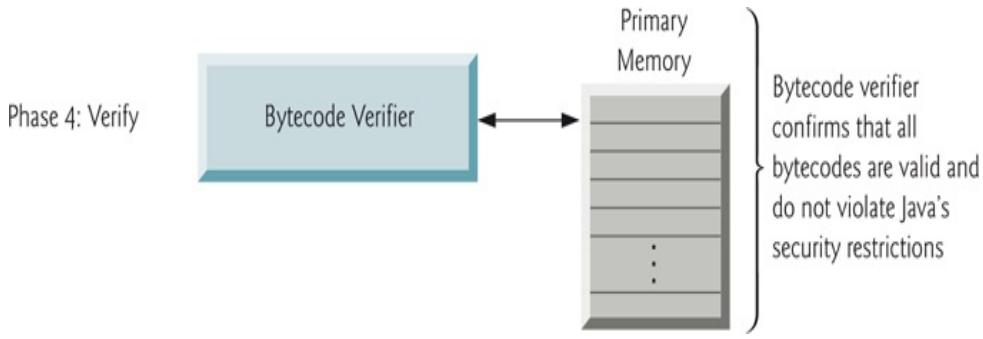

Fig. 1.9

Typical Java development environment—verification phase.

Phase 5: Execution

In Phase 5, the JVM **executes** the bytecodes to perform the program's specified actions (Fig. 1.10). In early Java versions, the JVM was simply a Java-bytecode *interpreter*. Most programs would execute slowly, because the JVM would interpret and execute one bytecode at a time. Some modern computer architectures can execute several instructions in parallel. Today's JVMs typically execute bytecodes using a combination of interpretation and **just-intime (JIT) compilation**. In this process, the JVM analyzes the bytecodes as they're interpreted, searching for *hot spots*—bytecodes that execute frequently. For these parts, a **just-in-time** (**JIT**) **compiler**, such as Oracle's **Java HotSpot™ compiler**, translates the bytecodes into the computer's machine language. When the JVM encounters these compiled parts again, the faster machine-language code executes. Thus programs

actually go through *two* compilation phases—one in which Java code is translated into bytecodes (for portability across JVMs on different computer platforms) and a second in which, during execution, the bytecodes are translated into *machine language* for the computer on which the program executes.

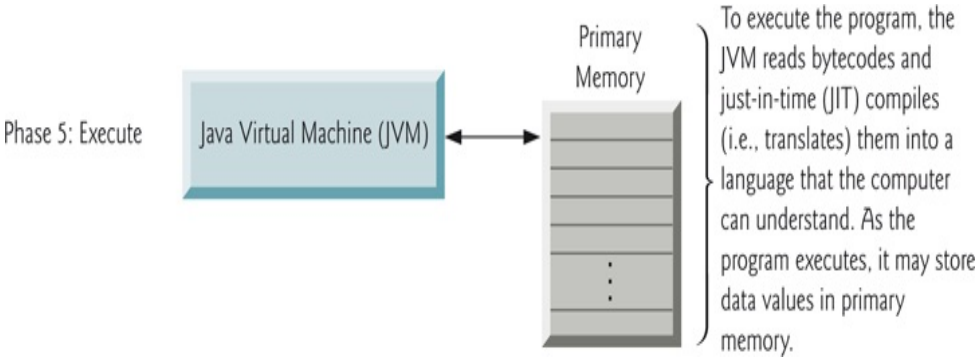

Fig. 1.10

Typical Java development environment—execution phase.

Description

Problems That May Occur at Execution Time

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we'll discuss throughout this book. For example, an executing program might try to divide by zero (an illegal operation for whole-number arithmetic in Java). This would cause the Java

program to display an error message. If this occurred, you'd return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine whether the corrections fixed the problem(s). [*Note:* Most programs in Java input or output data. When we say that a program displays a message, we normally mean that it displays that message on your computer's screen.]

Common Programming Error 1.2

Errors such as division by zero occur as a program runs, so they're called ***runtime errors*** *or* ***execution-time errors****.* ***Fatal runtime errors*** *cause programs to terminate immediately without having successfully performed their jobs.* ***Nonfatal runtime errors*** *allow programs to run to completion, often producing incorrect results.*