Chapter 1

The Context of Software Development

A computer program, from one perspective, is a sequence of instructions that dictate the flow of electrical impulses within a computer system. These impulses affect the computer's memory and interact with the peripheral devices like the display screen in such a way as to produce the "magic" that permits humans to perform useful tasks and solve high-level problems. One program allows a personal computer to assume the role of a financial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

- at the lower, more concrete level electrical impulses alter the internal state of the computer, while
- at the higher, more abstract level computer users accomplish real-world work or derive actual pleasure.

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract, level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Sophisticated tools hide the lower-level details from the programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and efficiency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language. While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as *software engineers* develop software to drive particular systems. These systems are defined by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and profilers. This chapter examines the context of software development, including computer systems and tools.

1.1 Software

A computer program is an example of computer *software*. One can refer to a program as a *piece* of software as if it were a tangible object, but software is actually quite intangible. It is stored on a *medium*. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used software must be stored in the computer's memory. Typically computer programs are loaded into memory from a medium like the computer's hard disk. The electromagetic

pattern stored on the hard drive is transferred to the computer's memory where the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system:

```
...10001011011000010001000001001110...
```

To the underlying computer hardware, specifically the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that a paricular point on the screen will be colored red. Unfortunately, only a miniscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word for an Intel-based computer running Microsoft Windows. Further, almost none of those who could produce the binary sequence would claim to enjoy the task. The Word program for Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Pentium processor in the Windows machine accepts a completely different binary language than the PowerPC processor in the Mac. We say the processors have their own *machine language*.

1.2 Development Tools

If very few humans can (or want) to speak the machine language of the computers' processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that for humans are easier to manage than binary sequences. Tools exist that convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like Java , C++, C#, FORTRAN, COBOL, and Perl allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Most programmers today, especially those concerned with high-level applications, do not usually worry about the details of underlying hardware platform and its machine language.

One might think that ideally a tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called *compilers* that readily translate one computer language into another have been around for over 50 years, but natural language recognition is still an active area of artificial intelligence research. Natural languages, as they are used by most humans, are inherently ambiguous. To properly understand all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today's software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any program that can be solved by a computer.

Consider the following program fragment, written in the C programming language:

```
subtotal = 25;
tax = 3;
total = subtotal + tax;
```

(The Java language was derived from C.) The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, `subtotal`, `tax`, and `total`, called *variables*, are used

to hold information (§ 2.2). [1] Variables have been used in mathematics for hundreds of years before the first digital computer was built. In programming, a variable represents a value stored in the computer's memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the C language, not machine language, it cannot be executed directly on any processor. A program called a *compiler* translates the C code to machine code. The typical development cycle for higher-level programming languages is shown in Figure 1.1.

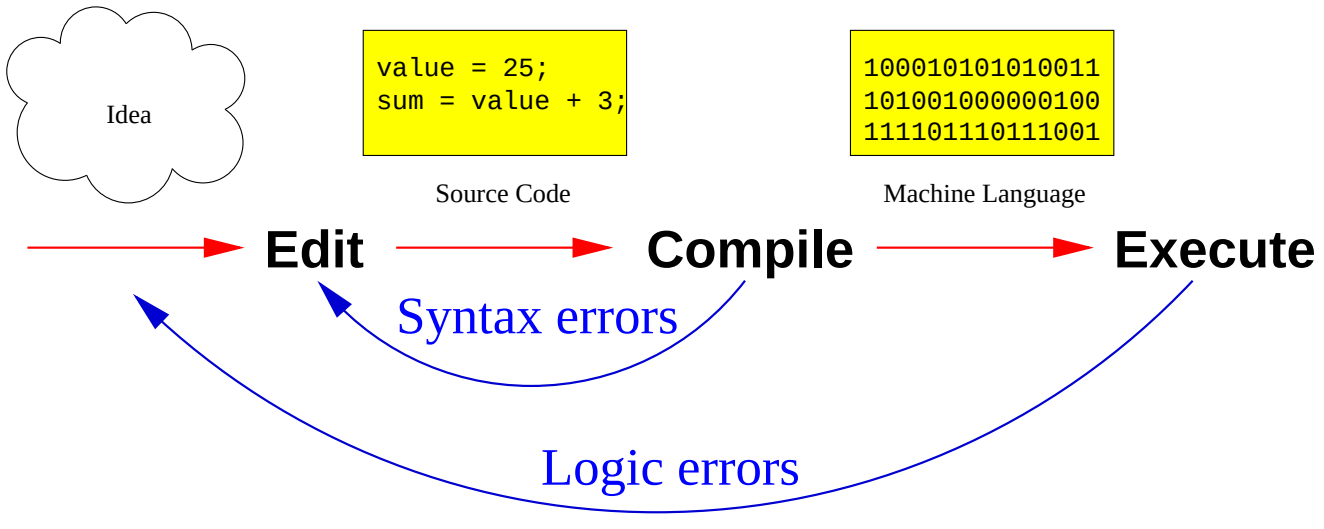

Figure 1.1: Program development cycle for typical higher-level languages. Errors are discussed in Section 8.5.

The higher-level language code is called *source code*. The compiled machine language code is called the *target code*.

The beauty of higher-level languages is that the same C source code can be compiled to different target platforms and execute identically. The only requirement is that the target platform has a C compiler available. So, a program can be developed on one system and then recompiled for another system. The compiled code will be very different on the various platforms since the machine architecture varies. Importantly, the human programmer thinks about writing the solution to a problem in C, not in a specific machine language.

For true compatibility, the program to be transported to multiple platforms should not exploit any feature unique to any target platform; for example, graphical windows behave differently and have different capabilities under different window managers (for example, the Windows XP graphical environment is different from the Unix/Linux X windows environment and Mac OS X environment). In practice, programs are often decomposed into platform *dependent* parts and platform *independent* parts. The platform independent parts capture the core functionality or purpose of the software (sometimes called the *business logic*). For example, all spreadsheets have some things in common (cells, formatting, calculation, etc.), regardless of the platform upon which they run. The platform dependent parts of the program take care of the unique user interface or file structure of the target platform. Porting the application to another platform requires modifying only the platform dependent parts of the software.

The ability to develop a program once and deploy it easily on other platforms is very attractive to developers since software development is an expensive and time-consuming task. § 1.3 shows how Java provides a new dimension to software portability.

As mentioned above, computer programs can be written in the absence of actual computers on which to run them. They can be expressed in abstract mathematical notation and be subjected to mathematical proofs and analysis that verify their correctness and execution efficiency. Some advantages of this mathematical approach include:

[1] The section symbol (§) is used throughout the text to direct you to sections that provide more information about the current topic. The listing symbol () refers to a source code listing such as a Java class definition.

- **Platform independence**. Since the program is not expressed in a "real" programming language for execution on a "real" computer system, the programmer is not constrained by limitations imposed by the OS or idiosyncrasies of the programming language.

- **Avoids the limitations of testing**. Testing cannot prove the absence of errors; it can only demonstrate the presence of errors. Mathematical proofs, when done correctly, are not misled by testing traps such as:

  – Coincidental correctness—as a simple example, consider the expression $x+y$. If the programmer accidentally uses multiplication instead of addition and values $x = 2$ and $y = 2$ are used to check, then the program appears to evaluate this expression correctly, since both $2+2 = 4$ and $2 \times 2 = 4$.

  – Poor coverage—in complex programs, certain parts of a program may be executed only under unusual conditions. Often these unusual conditions elude testing, and lurking errors appear only after the software is deployed.

Unlike pure mathematicians, most programmers do not often use formal mathematical reasoning techniques as they write programs. Some argue, with a strong case, that this lack of formality leads to poor-quality software. However, capable programmers employ techniques that are often far less formal but highly effective in producing high-quality software. These informal techniques and notions often map directly to their more abstract mathematical counterparts. Some circumstances, such as the development of critical software systems concerning safety, privacy, or financial affairs, may warrant a more formal approach.

Programmers have a variety of concrete tools available to enhance the software development process. Some common tools include:

- **Editors**. An *editor* allows the user to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors before the program is compiled.

- **Compilers**. A *compiler* translates the source code to target code. The target code may be machine language for a particular platform, another source language, or, as in the case of Java, instructions for a *virtual machine* (§ 1.3).

- **Debuggers**. A *debugger* allows programmers to simultaneously run a program and see which source code line is currently being executed. The values of variables and other program elements can be watched to see if their values change as expected. Debuggers are valuable for locating errors (also called *bugs*) and repairing programs that contain errors. (See § 8.5 for more information about programming errors.)

- **Profilers**. A *profiler* is used to evaluate a program's performance. It indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also are used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. (This is known as *coverage*. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing.)

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of IDEs include *Eclipse*, Microsoft's *Visual Studio*, *NetBeans*, and *DrJava*.

Despite the plethora of tools (and tool vendors' claims), the programming process for all but trivial programs is not automatic. Good tools are valuable and certainly increase the productivity of developers, but they cannot write software. There are no substitutes for sound logical thinking, creativity, common sense, and, of course, programming experience.

1.3 The Java Development Environment

Java was conceived in the early 1990s to address some outstanding issues in software development:

- **Computing platforms were becoming more diverse**. Microprocessors were becoming ubiquitous for controlling electronic devices like televisions, cell phones, and home bread makers. These computers, called *embedded computers*, were driven, of course, by software. Meeting the software needs of this multitude of embedded computers had become quite a task. A uniform platform for software development is desirable from one point of view, but forcing all manufacturers to use the same hardware is impractical.
- **Embedded computers had to be robust.** Consumers should not have to reboot their television or toaster because its control software has crashed. These appliances must be as reliable as their noncomputerized predecessors.
- **Computing platforms were becoming more distributed**. In order to accomplish a task, a system may need to download software from a remote system. For example, the functionality of a web browser can be extended by downloading special software called a *plug-in*. Cell phone service may be enhanced by upgrading some or all of its existing software. In general, though, downloading arbitrary software from a remote machine is risky. The software may do damage to the local machine either intentionally (like a virus) or accidentally (due to an error in the software). A framework was needed that would provide a safer environment in which untrusted code could run.
- **Object-oriented techniques had revolutionized software development.** Object-oriented (OO) programming languages and development environments, in the hands of knowledgeable developers, facilitate the construction of complex and large software systems. OO development had been embraced by many applications developers, but non-OO languages like C and assembly language still dominated systems programming.

No existing development environment at the time adequately addressed all these issues. Engineers at Sun Microsystems developed Java to address these issues. Around the same time as Java's inception the World Wide Web was emerging. Like the software appliances described above, Web software needed to be distributed, robust, and secure. The design of the Java programming language matched well with the needs of Web software developers.

Roughly speaking, web-based software can be classified into two types:

- **Client side software**. This software executes on the *client's* computer. A web browser is an example of a client; usually many clients can access a single website.
- **Server side software**. This software executes on the *server* computer. The computer providing the web page that clients (browsers) read is the server. Usually one server sends information to many clients.

Java was designed to address the issues of distributed computing, including security. Ordinarily, a program is loaded and executed by the operating system (OS) and hardware. Java programs, however, do not execute *directly* on the client OS and hardware. Java code is executed in a special environment called the *Java Runtime Environment* (JRE). The JRE provides several features that ensure secure computing on the target platform:

- **The Java Virtual Machine (JVM)**. The JVM is an executing program that emulates a computer that understands a special machine language called Java *bytecode*. Compiled Java programs run on the JVM, not directly on the native processor of the computer, so the JVM can be designed to prohibit any insecure behavior of executing programs.
- **Run time error detection**. The JRE detects many types of errors that an executing program may exhibit. For example, an arithmetic calculation may attempt to divide a number by zero.

- **Memory management**. A memory manager controls memory access, allocation, and deallocation. The memory manager ensures only memory assigned to the JVM is affected.

- **Security**. A security manager enforces policies that limit the capabilities of certain Java programs. The spectrum ranges from untrusted applets downloaded from the web that are not allowed to do any damage to a client's system to Java applications that have the same abilities as any other software.

The JRE includes hundreds of predefined Java components that programmers can incorporate into their programs. These components consist of compiled Java bytecode contained in units called *classes*. These classes, called the standard library classes, provide basic functionality including file manipulation, graphics, networking, mathematics, sound, and database access.

The virtual machine concept is so attractive for some applications that compilers are available that translate other higher-level programming languages such as Python, Lisp, and SmallTalk into bytecode that executes on the Java Virtual Machine.

The JRE is itself software. Reconsider the power of software. Software can turn a computer into a flight simulator; that is, a computer can emulate a plane in flight. Simulated controls allow a virtual pilot (the user) to adjust the virtual altitude, airspeed, and attitude. Visual feedback from instruments and a simulated viewport mimic the plane's response to the pilot's input. The laws of physics in this virtual world so closely match those in the real world that the pilot is given the illusion of real flight. If software can so closely model this complex physical activity as well as it does, it should not be surprising that software can cause the same computer to behave as if it were a different kind of computer; that is, it can execute machine language instructions meant for an entirely different processor. This process of one computer behaving as if it were another is called *emulation*. This concept of emulation is not new, but it is central to the Java execution environment. Java's approach to software development and deployment is slightly different from the standard approach:

- **Traditional approach**
  1. Develop the program in a higher-level language.
  2. For each deployment platform, recompile the source code using a compiler for that target platform.
- **Java's approach**
  1. Develop the program in a higher-level language (Java).
  2. Compile the source code to a special machine language called *bytecode*.
  3. For each deployment platform, execute the compiled bytecode using a special emulator built for that target platform. This emulator is called the Java Virtual Machine, or JVM.

Figure 1.2 illustrates the Java technology architecture.

As Figure 1.2 illustrates, the Java development environment works as follows:

1. Applications and applet programmers develop Java source code and store this code in files.
2. The Java compiler translates these source files into bytecode class files.
3. The JVM:
   - loads and executes the bytecode instructions in these programmer-developed class files as needed by the running program,
   - loads any standard library classes required by the program, and

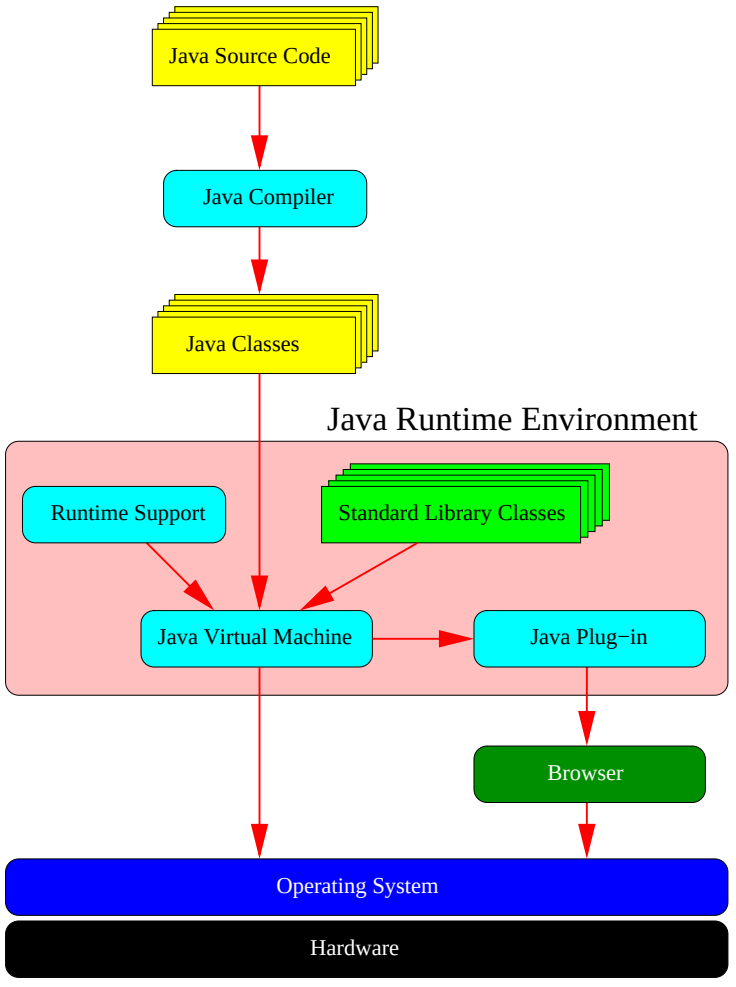

Figure 1.2: Java technology architecture.

   - executes runtime support code that handles assorted tasks like memory management, thread scheduling and run time error checking.

4. The JVM interprets the bytecodes and interacts with the OS using the native machine language of the platform so that the effects of the executing Java program can appear on the user's computer system.

5. The OS, as dictated by the JVM, directs the hardware through device drivers. For example,

   - input received from the keyboard or mouse is directed to the JVM
   - the graphics requested by the JVM appear on the screen
   - files created by the Java program are written to a disk drive.

A *Java plug-in* provides the same JRE to a browser so Java programs called *applets* can be downloaded from the Web and safely executed within the context of the browser. For most web users this process is transparent—most users will visit a web page that displays some interesting effects and be unaware that a Java program has been downloaded and began executing within their browser window.

The Java approach has several advantages:

- **Write once, run anywhere**. A program can be written once, compiled once, and run on any platform for which a Java Virtual Machine exists. In the traditional approach (outlined in § 1.2) this degree of portability is rare.

- **Security**. The implementation of the JVM can provide security features that make it safer to execute code downloaded from remote machines. For example, applets downloaded from a network are subject to security checks and are not allowed to read from or write to clients' disk drives. The JVM includes a security manager that enforces such policies.

One disadvantage of the virtual machine approach is that a Java bytecode program generally will run slower than an equivalent program compiled for the target platform. This is because the bytecode is not executed directly by the target platform. It is the JVM that is executing directly on the target platform; the JVM executes the bytecode. The JVM, therefore, adds some overhead to program execution. The HotSpot™ performance engine JVM addresses this issue by optimizing the bytecode as the program executes; parts of the code that execute frequently are translated into native machine language and executed directly by the processor. Other optimizations are also performed. The HotSpot JVM has significantly reduced the performance disadvantage in recent years.

If a JVM is unavailable for a particular platform, the Java bytecode cannot be executed on that platform. Equivalently, from the traditional approach, if a compiler for a given language is unavailable for a particular platform, the code cannot be ported directly to and executed on that platform.

While Java's special appeal is web-based programming, Java can be used for developing traditional applications as well. Like applets, Java applications typically are executed by the JVM. The Java program development cycle, shown in Figure 1.3, is similar to the development cycle for traditional compiled languages, as shown in Figure 1.1.

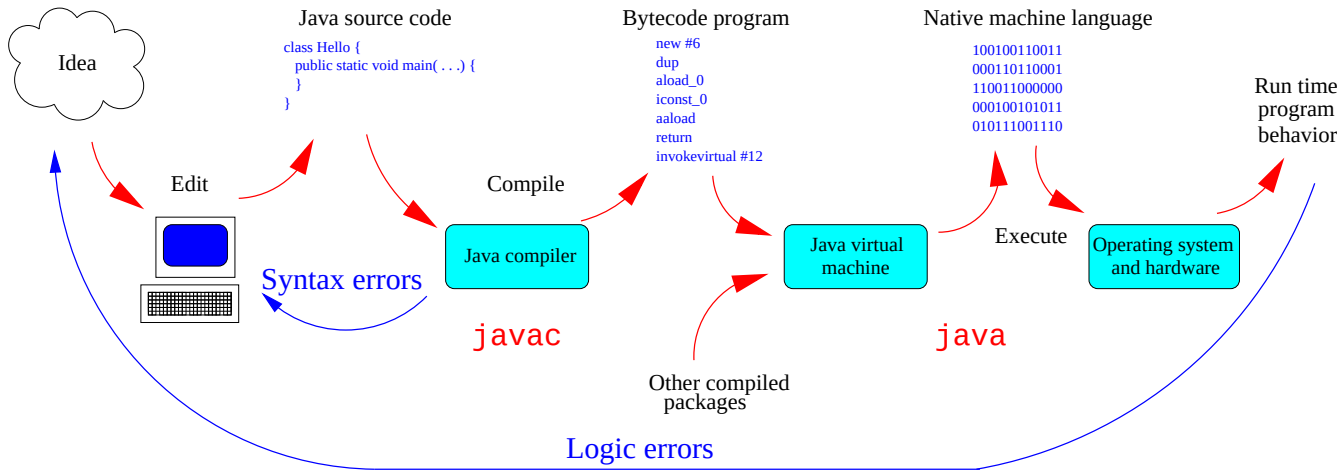

Figure 1.3: Java program development cycle

Sun Microsystems provides a set of tools for Java development, called the Java Software Developers Kit (Java SDK). These tools are freely available for noncommercial use from `http://java.sun.com`. The SDK includes a JRE, but the JRE is available separately for nondevelopers wishing only to run Java programs. Microsoft Windows, Linux, and Solaris platforms are supported. The Apple Mac OS X platform has the Java tools available from Apple. The code in this book is based on features available in the Java 5.0 (also called Java 2 SDK 1.5.0) SDK.

Sun Microsystems owns the intellectual property rights to Java and Java-related technology and defines strict standards for its implementation. Vendors' Java tools must pass stringent compatibility tests before they may be distributed. As examples:

- A JRE implementation must provide the complete set of standard library classes. If this were not guaranteed, then a Java program expecting to use a standard class would not be able to execute.
- A Java compiler must
  1. accept any correct Java program and translate it into bytecode executable on a JVM

2. reject all incorrect Java programs

This prevents vendors from adding features to the language producing an incompatible version of Java.

Such standards ensure that Java programs truly can be "write once, run anywhere."

For more on the history of Java see

- `http://www.java.com/en/about/`
- `http://www.java.com/en/javahistory/`

For more information about Java technology see

`http://java.sun.com/docs/books/tutorial/getStarted/intro/index.html`.

1.4 The *DrJava* Development Environment

Unlike some programming languages used in education, Java is widely used in industry for commerical software development. It is an industrial strength programming language used for developing complex systems in business, science, and engineering. Java drives the games on cell phones and is used to program smart cards. Java is used to manage database transactions on mainframe computers and is used to enhance the user experience on the Major League Baseball's website. NASA uses Java in its command and control software for the Mars Rover. Java is used for financial analysis in Wall Street investment firms.

In order to accomplish all that it does, Java itself is complex. While experienced programmers can accomplish great things with Java, beginners sometimes have a difficult time with it. Programmers must be aware of a number of language features unrelated to the task at hand in order to write a relatively simple program. These features intimidate novices because they do not have the background to understand what the features mean and why the features are required for such simple programs. Professional software developers enjoy the flexible design options that Java permits. For beginners, however, more structure and fewer options are better so they can master simpler concepts before moving on to more complex ones.

Despite the fact that novice programmers struggle with its complexity, Java is widely used in first-year programming courses in high schools, colleges, and universities. Many instructors believe it to be a cleaner, safer language for beginners than other popular options such as C++, another object-oriented language widely used in industry.

DrJava (`http://drjava.sourceforge.net`) is a Java development environment created especially for students. Its development by the faculty and students at Rice University is ongoing. DrJava presents a simple user interface containing a multipane editor, a simple file manager, an interactive area, console display window, and compiler message area. It also has a built in debugger. DrJava lacks many of the ammenities provided by other Java IDEs, but these other popular IDEs provide a vast array of features that overwhelm beginning programmers. While an introductory programming student may be able to use only 10% of the features provided by an industrial strength IDE, most of DrJava can be mastered quickly. Figure 1.4 is a screenshot of DrJava.

It is important to realize the difference between Java and DrJava. Java is a programming language and associated runtime environment; DrJava is a tool for exploring Java's capabilities and developing Java software. DrJava itself was written in Java.

Java is used widely for commercial software development. In order to support this wide variety of applications development, the Java language is inherently complex. In fact, one cannot write the simplest Java program possible without using several advanced language constructs. For example, suppose we want to experiment with the Java

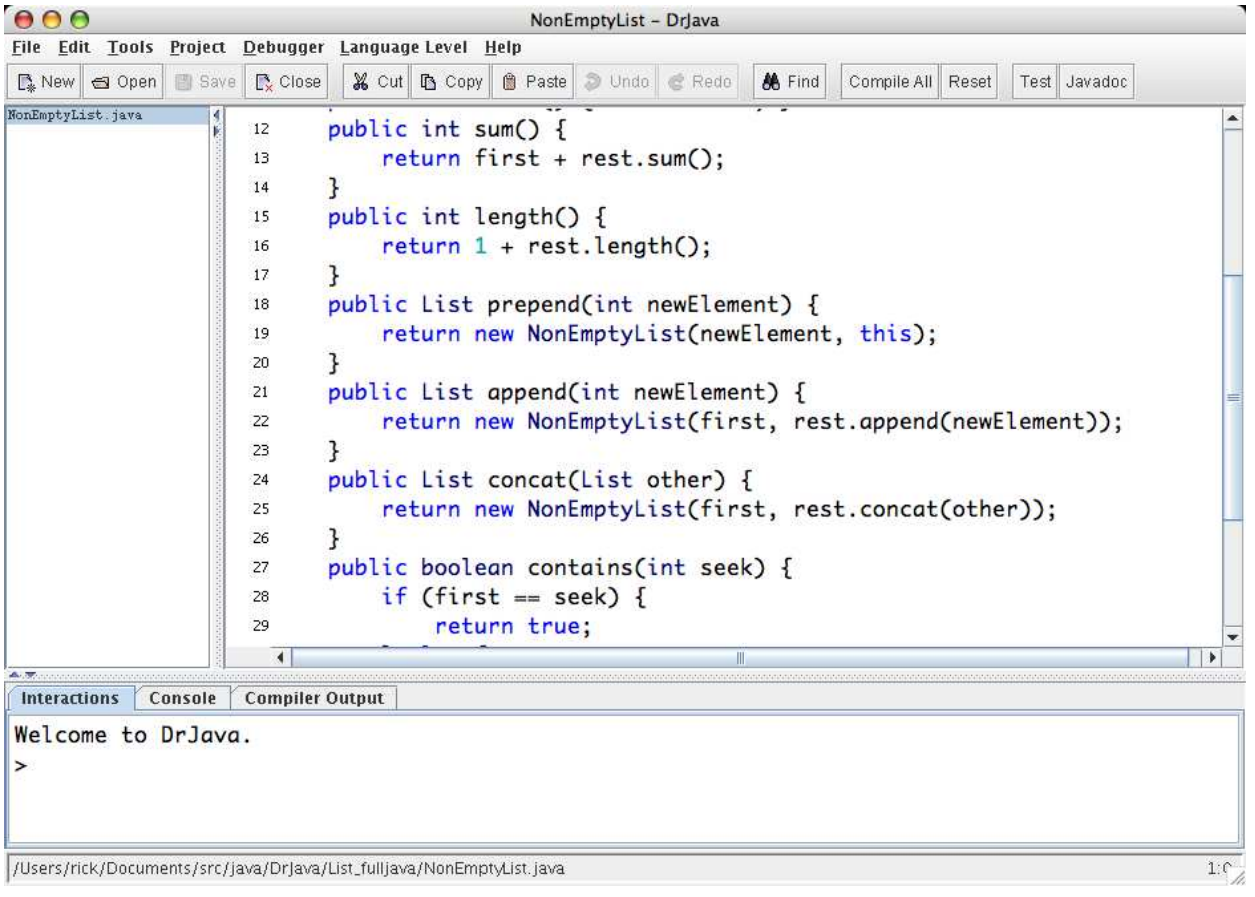

Figure 1.4: DrJava. Shown here is a screenshot of DrJava.

modulus (`%`) operator. Applied to integer values, the modulus operator computes the remainder of a division operation. `RemainderTest` (1.1) is one of the simplest Java programs that computes and displays the remainder of $15 \div 6$.

```
public class RemainderTest {
    public static void main(String[] args) {
        System.out.println(15 % 6); //  Compute remainder
    }
}
```

Listing 1.1: `RemainderTest`—Prints the remainder of $15 \div 6$

We must type `RemainderTest` (1.1) within an editor, compile it, and then execute it and examine the results. This is a lot of work just to satisfy our curiosity about a simple calculation. Besides, what does the word `static` mean? Why is `void` there? These words are required for a complete, legal Java program, but considerable programming experience is necessary to fully understand and appreciate their presence.

To try out the operation on a different pair of values (for example, what if one or both of the numbers are negative?) we must edit the file, recompile it, and rerun it.

For beginning programmers learning the Java language this need to use advanced constructs for even the simplest of programs presents a problem with only two less-than-perfect solutions:

1. attempt to explain briefly these advanced features in terms that a novice could not understand and fully appreciate, or

2. announce that you do it this way "just because," and the secrets will be revealed later as more knowledge and experience is gained.

Neither of these solutions is very satisfying. The first can unnecessarily confuse students with too much, to soon. The second can lead students to believe that they really cannot understand fully what is going on and, therefore, some kind of knowledge known only to gurus is needed to program computers.

DrJava solves this dilemma by providing an interactive environment where simpler concepts can be examined without the clutter of more advanced language features required by standalone programs. This actually allows us to investigate some more advanced features of Java earlier than we otherwise might since we can avoid becoming bogged down in the required (for standalone programs) but distracting details.

How could you see the results of Java's interpretation of $15 \div 6$ under DrJava? Within DrJava's Interactions pane type:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> 15 % 6
3
```

(The indicated working directory will be specific to your computer's environment.) The Interactions pane prints `>`, and you type `15 % 6`. The DrJava interpreter then computes and displays the result, `3`.

Running in the Interactions pane is a Java interpreter that evaluates Java statements as you type them. This interactive environment is an ideal place to learn the basics of the Java language that we will later use to build more powerful Java programs. All the distracting details required to write "real" Java programs will be revealed eventually when the time comes.

Some students may find DrJava's IDE to be a bit too simplistic. The Eclipse IDE (`http://www.eclipse.org`) is used by professional software developers all over the world. It provides a rich set of features that increase the productivity of experienced developers. Some examples include code completion, refactoring tools, advanced code analysis, and the ability to correct simple coding mistakes. Furthermore, Eclipse is designed to allow tool developers to build additional features that can be plugged into its framework.

The DrJava team has constructed a DrJava plug in for Eclipse. This allows students to use a more powerful IDE and still enjoy the interactive environment provided by DrJava. Figure 1.5 is a screenshot of Eclipse. Timid students may find Eclipse to be overwhelming and wish to stick with DrJava's IDE.

Our explorations in this book, especially the earlier sections, assume you have access to the latest versions of DrJava (or Eclipse with the DrJava plug in) and Sun's Java 5.0 development tools (also known as JDK 1.5.0).

1.5 Summary

- Computers require both hardware and software to operate. Software consists of instructions that control the hardware.

- Application software can be written largely without regard to the underlying hardware. A tool called a compiler translates the higher-level, abstract language into the machine language required by the hardware.

- Programmers develop software using tools such as editors, compilers, debuggers, and profilers.

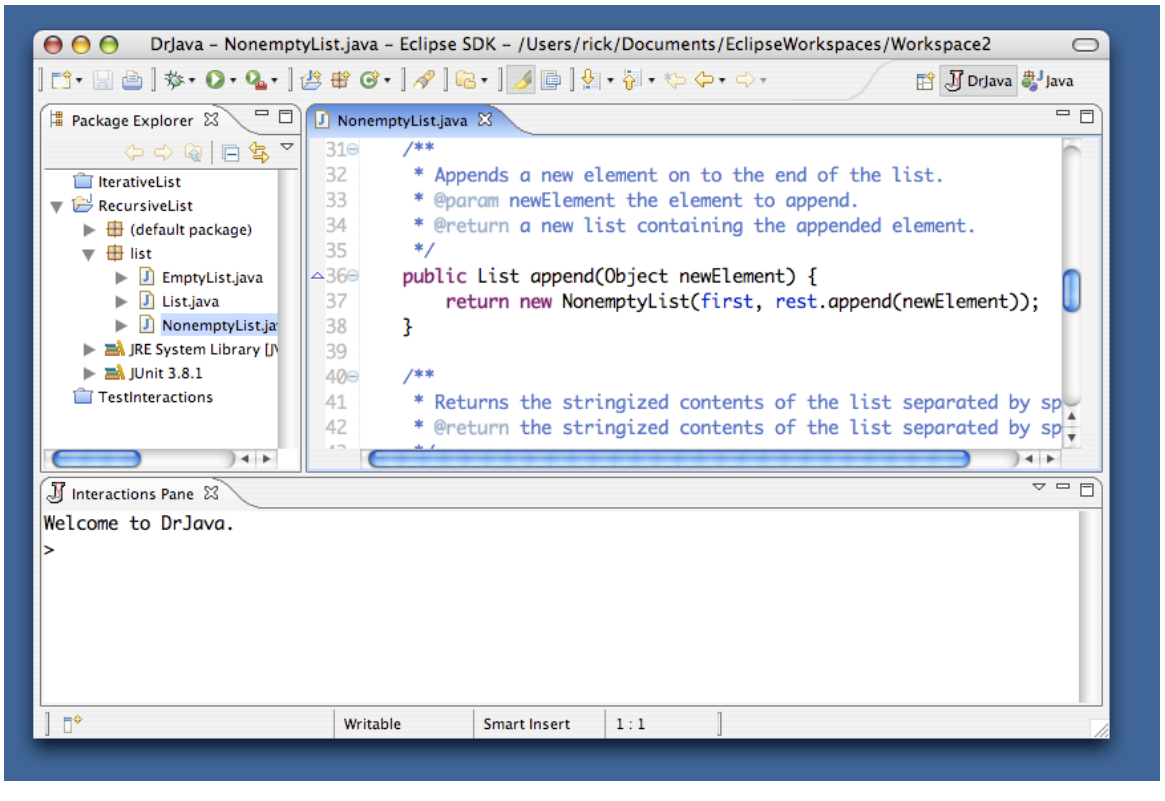

Figure 1.5: Eclipse. Shown here is a screenshot of the Eclipse integrated development environment with the DrJava plug in.

- Software can direct a processor to emulate another processor creating a virtual machine (VM).
- The Java platform uses virtual machine technology. A Java compiler translates Java source code into machine language for a Java virtual machine (JVM).
- Java is a modern language with features that support the demands of today's software such as security, networking, web deployment. Java is widely used commercially.
- An IDE is an integrated development environment—one program that provides all the tools that developers need to write software.
- DrJava is a simple IDE useful for beginning Java programmers.
- The DrJava interactive environment is available in both the DrJava IDE and Eclipse IDE.

1.6 Exercises

1. What advantages does Java's virtual machine concept provide compared to the traditional approach of compiling code for a particular platform? What are the disadvantages?
2. What is a compiler?
3. What is bytecode?
4. How is Java source code related to bytecode?
5. In DrJava, create a new file containing `RemainderTest` (1.1).

6. Compile the program and correct any typographical errors that may be present.

7. Run the program to verify that it correctly computes the remainder of $15 \div 6$.

8. Use the Interactions pane to directly evaluate the expression `15 / 6`.

9. Experiment more with the Interactions pane using numbers and the operators `+`, `-`, `*`, `/`, and `%`. See if the use of parentheses makes any difference in the results. Reflect on the results of your activities and form some conclusions about how these operators work with numbers.

10. Using the web, research the similarities and differences between Java and Microsoft's C# programming language.

11. Using the web, research the similarities and differences between Java's runtime environment and Microsoft's .NET environment.

12. Visit DrJava's website (`http://www.drjava.org`) and begin becoming familiar with some of DrJava's features. How do its capabilities compare to other software applications you have used, such as a wordprocessor?

