Chapter1

What Java Is

The basics: Java is an object-oriented programming language developed by Sun Microsystems that plays to the strengths of the Internet.

Object-oriented programming (OOP) is an unusual but powerful way to develop software. In OOP, a computer program is considered to be a group of objects that interact with each other. Consider an embezzlement program implemented with Java: A Worker object skims some Money objects from the CompanyFunds object and puts them in its own BankAccount object. If another Worker object uses the DoubleCheckFunds object, a Police object will be called.

The feature that is best known about Java is that it can be used to create programs that execute from World Wide Web pages. These programs are called applets.

Java is a general-purpose language that can be used to develop all kinds of programs.

A Java program is created as a text file with the file extension `.java`. It is compiled into one or more files of bytecodes with the extension `.class`. Bytecodes are a set of instructions similar to the machine code instructions created when a computer program is compiled. The difference is that machine code must run on the computer system it was compiled for; bytecodes can run on any computer system equipped to handle Java programs.

Java has quickly become a popular choice for computer programming--both on and off the Internet.

Java is just a small, simple, safe, object-oriented, interpreted or dynamically optimized, byte-coded, architecture-neutral, garbage-collected, multithreaded programming language with a strongly typed exception-handling mechanism for writing distributed, dynamically extensible programs.

Java Is Object Oriented

Object-oriented programming (OOP) is a powerful way of organizing and developing software. The short-form description of OOP is that it organizes a program as a set of components called objects. These objects exist independently of each other, and they have rules for communicating with other objects and for telling those objects to do things. Think back to how Star7 devices were developed as a group of independent devices with methods for communicating with each other. Object-oriented programming is highly compatible with what the Green project was created to do and, by extension, for Java as well.

Java inherits its object-oriented concepts from C++ and other languages such as Smalltalk. The fact that a programming language is object oriented may not seem like

a benefit to some. Object-oriented programming can be an intimidating subject to tackle, even if you have some experience programming with other languages. However, object-oriented programs are more adaptable for use in other projects, easier to understand, and more bugproof.

The Java language includes a set of class libraries that provide basic variable types, system input and output capabilities, and other functions. It also includes classes to support networking, Internet protocols, and graphical user interface functions.

There's a lot of excitement in the programming community because Java provides a new opportunity to use object-oriented techniques on the job. Smalltalk, the language that pioneered object-oriented programming in the 1970s, is well-respected but has never been widely adopted as a software-development choice. As a result, getting the go-ahead to develop a project using Smalltalk can be an uphill struggle. C++ is object-oriented, but concerns about its use have already been described. Java is overcoming the hurdle in terms of usage, especially in regard to Internet programming and the development of distributed applications.

Java Is Safe

Another thing essential to Java's success is that it is safe.A Java program that executes from a Web page is called an applet. All other Java programs are called applications. When an applet is encountered on a Web page (if the user's browser can handle Java), the browser downloads the applet along with the text and images on the page. The applet then runs on the user's computer. This act should raise a red flag--danger! danger!--because a lot of harmful things can occur when programs are executed: viruses, Trojan horses, the Microsoft Network, and so on.

Java provides security on several different levels. First, the language was designed to make it extremely difficult to execute damaging code. The elimination of pointers is a big step in this regard. Pointers are a powerful feature, as the programmers of C-like languages can attest, but pointers can be used to forge access to parts of a program where access is not allowed, and to access areas in memory that are supposed to be unalterable. By eliminating all pointers except for a limited form of references to objects, Java is a much more secure language.

Another level of security is the bytecode verifier. As described earlier, Java programs are compiled into a set of instructions called bytecodes. Before a Java program is run, a verifier checks each bytecode to make sure that nothing suspicious is going on.

In addition to these measures, Java has several safeguards that apply to applets. To prevent a program from committing random acts of violence against a user's disk drives, an applet cannot, by default, open, read, or write files on the user's system. Also, because Java applets can open new windows, these windows have a Java logo and text that identifies them. This arrangement prevents one of these pop-up windows from pretending to be something such as a user name and password dialog box.

Java Is Platform Independent

Most computer software is developed for a specific operating system. If you wanted to run software on Windows and Mac systems, it had to develop two versions of the software at a significant effort and expense. Platform independence is the capability of the same program to work on different operating systems; Java is completely platform independent.

Java's variable types have the same size across all Java development platforms--so an integer is always the same size, no matter which system a Java program was written and compiled on. Also, as shown by the use of applets on the Web, a Java `.class` file of bytecode instructions can execute on any platform without alteration.

Java Is that Other Stuff, Too

One adjective that has been left out thus far is that Java is multithreaded. Threads represent a way for a computer program to do more than one task at the same time. Many operating systems are multitasking. Windows 95, for example, enables a person to write a book chapter with Word in one window while using Netscape Navigator to download every known picture of E! host Eleanor Mondale in the other. (Speaking hypothetically, of course.)

Java provides the tools to write multithreaded programs and to make these programs reliable in execution.

The Java Development Kit

The Java Development Kit (JDK) is a set of command-line tools that can be used to create Java programs.

The Java Development Kit contains a variety of tools and Java development information. Following is a list of the main components of the JDK:

- Runtime interpreter
- Compiler
- Applet viewer
- Debugger
- Class file disassembler
- Header and stub file generator
- Documentation generator
- Archiver
- Digital signer
- Remote Method Invocation tools
- Sample demos and source code
- API source code

The runtime interpreter is the core runtime module for the Java system. The compiler, applet viewer, debugger, class file disassembler, header and stub file generator, and documentation generator are the primary tools used by Java developers. The demos

are interesting examples of Java applets, which all come with complete source code. And finally, if you are interested in looking under the hood of Java, the complete source code for the Java API (Application Programming Interface) classes is provided.

The Runtime Interpreter

The Java runtime interpreter (`java`) is used to run standalone Java executable programs in compiled, bytecode format. The runtime interpreter acts as a command-line tool for running Java programs that are either nongraphical or that manage their own window frame (applications); graphical programs requiring the display support of a Web browser (applets) are executed entirely within a browser. The syntax for using the runtime interpreter follows:

```
java Options ClassName Arguments
```

The ClassName argument specifies the name of the class you want to execute. If the class resides in a package, you must fully qualify the name. You learn about classes and packages in Chap- ter 5, "Classes, Packages, and Interfaces." For example, if you want to run a class called `Roids` that is located in a package called `ActionGames`, you execute it in the interpreter like this:

```
java ActionGames.Roids
```

When the Java interpreter executes a class, what it is really doing is executing the `main()` method of the class. The interpreter exits when the `main()` method and any threads created by it are finished executing.

The `main()` method accepts a list of arguments that can be used to control the program. The Arguments argument to the interpreter specifies the arguments passed into the `main()` method. For example, if you have a Java class called `TextFilter` that performs some kind of filtering on a text file, you would likely pass the name of the file as an argument, like this:

```
java TextFilter SomeFile.txt
```

The Options argument specifies options related to how the runtime interpreter executes the Java program. Please refer to the JDK documentation for more information about the options supported in the runtime interpreter.

The Compiler

The Java compiler (`javac`) is used to compile Java source code files into executable Java bytecode classes. In Java, source code files have the extension `.java`. The Java compiler takes files with this extension and generates executable class files with the `.class` extension. The compiler creates one class file for each class defined in a source file. This means that it is possible for a single Java source code file to compile

into multiple executable class files. When this happens, it means that the source file contains multiple class definitions.

The Java compiler is a command-line utility that works in a manner similar to the Java runtime interpreter. The syntax for the Java compiler follows:

```
javac Options Filename
```

The Filename argument specifies the name of the source code file you want to compile. The Options argument specifies options related to how the compiler creates the executable Java classes. Please refer to the JDK documentation for more information about the options supported by the compiler.

The Applet Viewer

The applet viewer is a tool that serves as a minimal test bed for final release Java applets. You can use the applet viewer to test your programs instead of using a full-blown Web browser. You invoke the applet viewer from a command line like this:

```
appletviewer Options URL
```

```
appletviewer example1.html
```

The URL argument specifies a document URL containing an HTML page with an embedded Java applet. The Options argument specifies how to run the Java applet. There is only one option supported by the applet viewer: `-debug`. The `-debug` option starts the applet viewer in the Java debugger, which enables you to debug the applet.

The Debugger

The Java debugger (`jdb`) is a command-line utility that enables you to debug Java applications. The Java debugger uses the Java Debugger API to provide debugging support within the Java runtime interpreter. The syntax for using the Java debugger follows:

```
jdb Options
```

The Options argument is used to specify different settings within a debugging session.

The Class File Disassembler

The Java class file disassembler (`javap`) is used to disassemble executable Java class files. Its default output consists of the public data and methods for a class. The class file disassembler is useful in cases where you don't have the source code for a class, but you want to know a little more about how it is implemented. The syntax for the disassembler follows:

```
javap Options ClassNames
```

The ClassNames argument specifies the names of one or more classes to be disassembled. The Options argument specifies how the classes are to be disassembled. Refer to the JDK documentation for more information about the options supported in the class file disassembler.

The Header and Stub File Generator

The Java header and stub file generator (`javah`) is used to generate C header and source files for implementing Java methods in C. The files generated can be used to access member variables of an object from C code. The header and stub file generator accomplishes this by generating a C structure whose layout matches that of the corresponding Java class. The syntax for using the header and stub file generator follows:

```
javah Options ClassName
```

The ClassName argument is the name of the class from which to generate C source files. The Options argument specifies how the source files are to be generated. Refer to the JDK documentation for more information about the options supported in the header and stub file generator.

The Documentation Generator

The Java documentation generator (`javadoc`) is a useful tool for generating API documentation directly from Java source code. The documentation generator parses through Java source files and generates HTML pages based on the declarations and comments. The syntax for using the documentation generator follows:

```
javadoc Options FileName
```

The FileName argument specifies either a package or a Java source code file. In the case of a package, the documentation generator creates documentation for all the classes contained in the package. The Options argument enables you to change the default behavior of `javadoc`.

Because the Java documentation generator is covered in detail in Chapter 29, "Documenting Your Code," you'll have to settle for this brief introduction for now. Or you could jump ahead to Chapter 29 to learn more now.

The Archiver

The Java archiver (`jar`) is a tool used to combine and compress multiple files (usually applets or applications) into a single archive file, which is also commonly referred to as a JAR file. Combining the components of an applet or application into a single archive allows them to be downloaded by a browser in a single HTTP transaction instead of requiring a new connection for each individual file. This approach, coupled with the data compression provided by the archiver, dramatically improves download times. Additionally, the archiver can be used with the digital signer (described in the following section) to sign applets and applications so that they can be authenticated at runtime. The syntax for using the archiver follows:

```
jar Options ManifestFileName OuputFileName InputFileNames
```

The ManifestFileName argument specifies a manifest file used to describe the contents of the archive being created. The OutputFileName argument specifies the name of the archive to be created. The InputFileNames argument specifies the files to be added to the archive. The Options argument enables you to change the default behavior of `jar`. You learn all about manifest files, code signing, and the ins and outs of the `jar` tool in Part VIII of this book, "Java Archives and JavaBeans."

The Digital Signer

The Java digital signer (`javakey`), also known as the Java security tool, is an interesting tool that generates digital signatures for archive files. Signatures are used to verify that a file came from a specified entity, or signer. To generate a signature for a particular file, the signer must first be associated with a public/private key pair and must also have one or more certificates authenticating the signer's public key. The digital signer is responsible for managing this database of entities, along with their keys and certificates. The syntax for using the digital signer follows:

```
javakey Options
```

The Options argument is used to control the operation of `javakey`. You learn all about digital code signing, public and private keys, and the specifics of how to use the `javakey` tool in Chapter 37, "Code Signing and JAR Security."

The Remote Method Invocation Tools

The JDK includes three different tools for working with and managing remote method invocation (RMI). These tools consist of an RMI stub compiler, a remote object registry tool, and a serial version tool.

The Java API

The Java Application Programming Interface (API) is a set of classes used to develop Java programs. These classes are organized into groups called packages. There are packages for the following tasks:

- Numeric variable and string manipulation
- Image creation and manipulation
- File input and output
- Networking
- Windowing and graphical user interface design
- Applet programming
- Error handling
- Security
- Database access

- Distributed application communication
- JavaBeans components

The API includes enough functionality to create sophisticated applets and applications. The Java API must be supported by all operating systems and Web software equipped to execute Java programs, so you can count on the existence of Java API class files when developing programs.

Example 1:-

Hello, World!

The best way to learn a programming language is to jump right in and see how a real program works. In keeping with a traditional introductory programming example, your first program is a Java version of the classic "Hello, World!" program. List- ing 3.1 contains the source code for the `HelloWorld` class, which also is located on the CD-ROM that accompanies this book in the file `HelloWorld.java`.

The HelloWorld class.

```
class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello, World!");
  }

}
```

After compiling the program with the Java compiler (`javac`), you are ready to run it in the Java interpreter. The Java compiler places the executable output in a file called `HelloWorld.class`. This naming convention might seem strange considering the fact that most programming languages use the `.EXE` file extension for executables. Not so in Java! Following the object- oriented nature of Java, all Java programs are stored as Java classes that are created and executed as objects in the Java runtime environment. To run the `HelloWorld` program, type java HelloWorld at the command prompt. As you may have guessed, the program responds by displaying `Hello, World!` on your screen. Congratulations! You just wrote and tested your first Java program!

Obviously, `HelloWorld` is a very minimal Java program. Even so, there's still a lot happening in those few lines of code. To fully understand what is happening, let's examine the program line by line. First, you must understand that Java relies heavily on classes. In fact, the first statement of `HelloWorld` reminds you that `HelloWorld` is a class, not just a program. Furthermore, by looking at the `class` statement in its entirety, you can see that the name of the class is defined as `HelloWorld`. This name is used by the Java compiler as the name of the executable output class. The Java compiler creates an executable class file for each class defined in a Java source file. If there is more than one class defined in a `.java` file, the Java compiler stores each class in a separate `.class` file.

The `HelloWorld` class contains one method, or member function. For now, you can think of this function as a normal procedural function that happens to be linked to the class. The details of methods are covered in Chapter 5, "Classes, Packages, and Interfaces." The single method in the `HelloWorld` class is called `main()` and should be familiar if you have used C or C++. The `main()` method is where execution begins when the class is executed in the Java interpreter. The `main()` method is defined as being `public static` with a `void` return type. `public` means that the method can be called from anywhere inside or outside the class. `static` means that the method is the same for all instances of the class. The `void` return type means that `main()` does not return a value.

The `main()` method is defined as taking a single parameter, `String args[]`. `args` is an array of `String` objects that represent command-line arguments passed to the class at execution. Because `HelloWorld` doesn't use any command-line arguments, you can ignore the `args` parameter. You learn a little more about strings later in this chapter.

The `main()` method is called when the `HelloWorld` class is executed. `main()` consists of a single statement that prints the message `Hello, World!` to the standard output stream, as follows:

```
System.out.println("Hello, World!");
```

This statement may look a little confusing at first because of the nested objects. To help make things clearer, examine the statement from right to left. First, notice that the statement ends in a semicolon, which is standard Java syntax borrowed from C/C++. Moving to the left, you see that the `"Hello, World!"` string is in parentheses, which means that it is a parameter to a function call. The method being called is actually the `println()` method of the `out` object. The `println()` method is similar to the `printf()` method in C, except that it automatically appends a newline character (`\n`) at the end of the string. The `out` object is a member variable of the `System` object that represents the standard output stream. Finally, the `System` object is a global object in the Java environment that encapsulates system functionality.

That pretty well covers the `HelloWorld` class--your first Java program. If you got a little lost in the explanation of the `HelloWorld` class, don't be too concerned. `HelloWorld` was presented with no previous explanation of the Java language and was only meant to get your feet wet with Java code. The rest of this chapter focuses on a more structured discussion of the fundamentals of the Java language.