# *Contents*


**1. Introduction to Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10**

1.1 Beginnings of the Java Language Project. . . . . . . . . . . . . . 12

1.2 Design Goals of Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

1.2.1 Simple, Object Oriented, and Familiar . . . . . . . . . . . 13

1.2.2 Robust and Secure. . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

1.2.3 Architecture Neutral and Portable . . . . . . . . . . . . . . 14

1.2.4 High Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

1.2.5 Interpreted, Threaded, and Dynamic . . . . . . . . . . . . 15

1.3 The Java Base System . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

1.4 The Java Environment—a New Approach to Distributed Computing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**2. Java—Simple and Familiar . . . . . . . . . . . . . . . . . . . . . . . . . . . 18**

2.1 Main Features of the Java Language . . . . . . . . . . . . . . . . 20

2.1.1 Primitive Data Types . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.1.2 Arithmetic and Relational Operators . . . . . . . . . . . . 21

2.1.3 Arrays. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.1.4 Strings . . . 22

2.1.5 Multi-Level Break . . . 23

2.1.6 Memory Management and Garbage Collection . . . 24

2.1.7 The Background Garbage Collector . . . 25

2.1.8 Integrated Thread Synchronization . . . 25

2.2 Features Removed from C and C++ . . . 26

2.2.1 No More Typedefs, Defines, or Preprocessor . . . 26

2.2.2 No More Structures or Unions . . . 27

2.2.3 No More Functions . . . 27

2.2.4 No More Multiple Inheritance . . . 28

2.2.5 No More Goto Statements . . . 29

2.2.6 No More Operator Overloading . . . 29

2.2.7 No More Automatic Coercions . . . 29

2.2.8 No More Pointers . . . 30

2.3 Summary . . . 30

**3. Java is Object Oriented . . . 32**

3.1 Object Technology in Java . . . 33

3.2 What Are Objects? . . . 33

3.3 Basics of Objects . . . 34

3.3.1 Classes . . . 35

3.3.2 Instantiating an Object from its Class . . . 35

3.3.3 Constructors . . . 36

3.3.4 Methods and Messaging . . . 37

3.3.5 Finalizers . . . 39

3.3.6 Subclassing . . . 39
3.3.7 Access Control . . . 41
3.3.8 Class Variables and Class Methods . . . 42
3.3.9 Abstract Methods . . . 42
3.4 Summary . . . 44
**4. Architecture Neutral, Portable, and Robust . . . 46**
4.1 Architecture Neutral . . . 47
4.1.1 Byte Codes . . . 47
4.2 Portable . . . 48
4.3 Robust . . . 49
4.3.1 Strict Compile-Time and Run-Time Checking . . . 49
4.4 Summary . . . 50
**5. Interpreted and Dynamic . . . 52**
5.1 Dynamic Loading and Binding . . . 53
5.1.1 The Fragile Superclass Problem . . . 53
5.1.2 Solving the Fragile Superclass Problem . . . 54
5.1.3 Java Language Interfaces . . . 54
5.1.4 Run-Time Representations . . . 55
5.2 Summary . . . 55
**6. Security in Java . . . 56**
6.1 *Memory Allocation and Layout* . . . 56
6.2 The Byte Code Verification Process . . . 57
6.2.1 The Byte Code Verifier . . . 58

6.3 Security Checks in the Bytecode Loader . . . . . . . . . . . . . . 59
6.4 Security in the Java Networking Package . . . . . . . . . . . . . 59
6.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
**7. Multithreading in Java. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62**
7.1 Threads at the Java Language Level . . . . . . . . . . . . . . . . . 62
7.2 Integrated Thread Synchronization . . . . . . . . . . . . . . . . . . 63
7.3 Multithreading Support—Conclusion . . . . . . . . . . . . . . . . 64
**8. Performance and Comparisons. . . . . . . . . . . . . . . . . . . . . . . . . 66**
8.1 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
8.2 The Java Language Compared . . . . . . . . . . . . . . . . . . . . . 67
8.3 A Major Benefit of Java: Fast and Fearless Prototyping. . 70
8.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
**9. The HotJava World-Wide Web Browser . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72**
9.1 The Evolution of Cyberspace . . . . . . . . . . . . . . . . . . . . . . 73
9.1.1 First Generation Browsers . . . . . . . . . . . . . . . . . . . . 74
9.1.2 The HotJava Browser—A New Concept in Web Browsers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
9.1.3 The Essential Difference. . . . . . . . . . . . . . . . . . . . . . 75
9.1.4 Dynamic Content . . . . . . . . . . . . . . . . . . . . . . . . . . 76
9.1.5 Dynamic Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
9.1.6 Dynamic Protocols . . . . . . . . . . . . . . . . . . . . . . . . . 78
9.2 Freedom to Innovate . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
9.3 Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . . . 80
9.4 Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

9.4.1 The First Layer—the Java Language Interpreter . . . 81
9.4.2 The Next Layer—the Higher Level Protocols . . . . . 82
9.5 HotJava—the Promise . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
**10. Further Reading. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84**

# Introduction to Java 1

The Next Stage of the Known,
Or a Completely New Paradigm?

Taiichi Sakaiya—*The Knowledge-Value Revolution*

### *The Software Developer's Burden*

Imagine you're a software application developer. Your programming language of choice (or the language that's been foisted on you) is C or C++ . You've been at this for quite a while and your job doesn't seem to be getting any easier. These past few years you've seen the growth of multiple incompatible hardware architectures, each supporting multiple incompatible operating systems, with each platform operating with one or more incompatible graphical user interfaces. Now you're supposed to cope with all this and make your applications work in a distributed client-server environment. The growth of the Internet, the World-Wide Web, and "electronic commerce" have introduced new dimensions of complexity into the development process.

The tools you use to develop applications don't seem to help you much. You're still coping with the same old problems; the fashionable new object-oriented techniques seem to have added new problems without solving the old ones. You say to yourself and your friends, "There *has* to be a better way"!

### *The Better Way is Here Now*

Now there *is* a better way—it's the ***Java***™ *programming language environment* ("Java" for short) from Sun Microsystems. Imagine, if you will, this development world…

- Your programming language is *object oriented,* yet it's still dead *simple.*
- Your development cycle is much *faster* because Java is *interpreted*. The compile-link-load-test-crash-debug cycle is obsolete—now you just compile and run.
- Your applications are *portable* across multiple platforms. Write your applications once, and you never need to port them—they will run without modification on multiple operating systems and hardware architectures.
- Your applications are *robust* because the Java run-time system manages memory for you.
- Your interactive graphical applications have *high performance* because multiple concurrent threads of activity in your application are supported by the *multithreading* built into Java environment.
- Your applications are *adaptable* to changing environments because you can dynamically download code modules from anywhere on the network.
- Your end users can trust that your applications are *secure*, even though they're downloading code from all over the Internet; the Java run-time system has built-in protection against viruses and tampering.

You don't need to dream about these features. They're here now. The Java Programming Language Environment provides a *portable*, *interpreted*, *high-performance*, *simple*, *object-oriented* programming language and supporting run-time environment. This introductory chapter provides you with a brief look at the main design goals of the Java system; the remainder of this paper examines the features of Java in more detail.

At the end of this paper you'll find a chapter that describes the ***HotJava***™ *Browser* ("HotJava" for short). HotJava is an innovative World-Wide Web browser, and the first major applications written using the Java environment. HotJava is the first browser to dynamically download and execute Java code fragments from anywhere on the Internet, and to so so in a secure manner.

## 1.1 Beginnings of the Java Language Project

Java is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume the minimum of system resources, can run on any hardware and software platform, and can be extended dynamically.

Java originated as part of a research project to develop advanced software for a wide variety of networked devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating environment. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language environment. Design and architecture decisions drew from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result is a language environment that has proven ideal for developing secure, distributed, network-based end-user applications in environments ranging from networked-embedded devices to the World-Wide Web and the desktop.

## 1.2 Design Goals of Java

The design requirements of Java are driven by the nature of the computing environments in which software must be deployed.

The massive growth of the Internet and the World-Wide Web leads us to a completely new way of looking at development and distribution of software. To live in the world of electronic commerce and distribution, Java must enable the development of *secure, high performance,* and highly *robust* applications on *multiple platforms* in *heterogeneous*, *distributed networks*.

Operating on multiple platforms in heterogeneous networks invalidates the traditional schemes of binary distribution, release, upgrade, patch, and so on. To survive in this jungle, Java must be *architecture neutral*, *portable*, and *dynamically adaptable*.

The Java system that emerged to meet these needs is *simple*, so it can be easily programmed by most developers; *familiar*, so that current developers can easily learn Java; *object oriented*, to take advantage of modern software development methodologies and to fit into distributed client-server applications; *multithreaded*, for high performance in applications that need to perform multiple concurrent activities, such as multimedia; and *interpreted*, for maximum portability and dynamic capabilities.

Together, the above requirements comprise quite a collection of buzzwords, so let's examine some of them and their respective benefits before going on.

### 1.2.1 Simple, Object Oriented, and Familiar

Primary characteristics of Java include a *simple* language that can be programmed without extensive programmer training while being attuned to current software practices. The fundamental concepts of Java are grasped quickly; programmers can be productive from the very beginning.

Java is designed to be *object oriented* from the ground up. Object technology has finally found its way into the programming mainstream after a gestation period of thirty years. The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. Java provides a clean and efficient object-based development environment.

Programmers using Java can access existing libraries of tested objects that provide functionality ranging from basic data types through I/O and network interfaces to graphical user interface toolkits. These libraries can be extended to provide new behavior.

Even though C++ was rejected as an implementation language, keeping Java looking like C++ as far as possible results in Java being a *familiar* language, while removing the unnecessary complexities of C++. Having Java retain many of the object-oriented features and the "look and feel" of C++ means that programmers can migrate easily to Java and be productive quickly.

### 1.2.2 Robust and Secure

Java is designed for creating highly *reliable* software. It provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmers towards reliable programming habits. The memory management model—no pointers or pointer arithmetic—eliminates entire classes of programming errors that bedevil C and C++ programmers. You can develop Java language code with confidence that the system will find many errors quickly and that major problems won't lay dormant until after your production code has shipped.

Java is designed to operate in distributed environments, which means that *security* is of paramount importance. With security features designed into the language and run-time system, Java lets you construct applications that can't be invaded from outside. In the networked environment, applications written in Java are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

### 1.2.3 Architecture Neutral and Portable

Java is designed to support applications that will be deployed into heterogeneous networked environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces. To accommodate the diversity of operating environments, the Java compiler generates *bytecodes*—an *architecture neutral* intermediate format designed to transport code efficiently to multiple hardware and software platforms. The interpreted nature of Java solves both the binary distribution problem and the version problem; the same Java language byte codes will run on any platform.

Architecture neutrality is just one part of a truly *portable* system. Java takes portability a stage further by being strict in its definition of the basic language. Java puts a stake in the ground and specifies the sizes of its basic data types and the behavior of its arithmetic operators. Your programs are the same on every platform—there are no data type incompatibilities across hardware and software architectures.

The architecture-neutral and portable language environment of Java is known as the *Java Virtual Machine*. It's the specification of an abstract machine for which Java language compilers can generate code. Specific implementations of the Java Virtual Machine for specific hardware and software platforms then provide the concrete realization of the virtual machine. The Java Virtual Machine is based primarily on the POSIX interface specification—an industry-standard definition of a portable system interface. Implementing the Java Virtual Machine on new architectures is a relatively straightforward task as long as the target platform meets basic requirements such as support for multithreading.

### 1.2.4 High Performance

*Performance* is always a consideration. Java achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The *automatic garbage collector* runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance. Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java environment. In general, users perceive that interactive applications respond quickly even though they're interpreted.

### 1.2.5 Interpreted, Threaded, and Dynamic

The *Java interpreter* can execute Java bytecodes directly on any machine to which the interpreter and run-time system have been ported. In an interpreted environment such as Java system, the link phase of a program is simple, incremental, and lightweight. You benefit from much faster development cycles—prototyping, experimentation, and rapid development are the normal case, versus the traditional heavyweight compile, link, and test cycles.

Modern network-based applications, such as the HotJava World-Wide Web browser, typically need to do several things at the same time. A user working with HotJava can run several animations concurrently while downloading an image and scrolling the page. Java's *multithreading* capability provides the means to build applications with many concurrent threads of activity. Multithreading thus results in a high degree of interactivity for the end user.

Java supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the `Thread` class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java's high-level system libraries have been written to be *thread safe*: the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

While the Java compiler is strict in its compile-time static checking, the language and run-time system are *dynamic* in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network. In the case of the HotJava browser and similar applications, interactive executable code can be loaded from anywhere, which enables transparent updating of applications. The result is on-line services that constantly evolve; they can remain innovative and fresh, draw more customers, and spur the growth of electronic commerce on the Internet.

## 1.3 The Java Base System

The complete Java system includes a number of libraries of utility classes and methods of use to developers in creating multi-platform applications. Very briefly, these libraries are:

`java.lang`—the collection of base types (language types) that are always imported into any given compilation unit. This where you'll find the declarations of `Object` (the root of the class hierarchy) and `Class`, plus threads, exceptions, wrappers for the primitive data types, and a variety of other fundamental classes.

`java.io`—streams and random-access files. This is where you find the rough equivalent of the Standard I/O Library you're familiar with on most UNIX systems. A further library is called `java.net`, and provides support for sockets, telnet interfaces, and URLs.

`java.util`—container and utility classes. Here you'll find classes such as `Dictionary`, `HashTable`, and `Stack`, among others, plus encoder and decoder techniques, and `Date` and `Time` classes.

`java.awt`—an Abstract Windowing Toolkit that provides an abstract layer enabling you to port Java applications easily from one window system to another. This library contains classes for basic interface components such as events, colors, fonts, and controls such as buttons and scrollbars.

## 1.4 The Java Environment—a New Approach to Distributed Computing

Taken individually, the characteristics discussed above can be found in a variety of software development environments. What's completely new is the manner in which Java and its run-time system have combined them to produce a flexible and powerful programming system.

Developing your applications using Java results in software that is *portable* across multiple machine architectures, operating systems, and graphical user interfaces, *secure*, and *high performance*. With Java, your job as a software developer is much easier—you focus your full attention on the end goal of shipping innovative products on time, based on the solid foundation of Java. The *better way* to develop software is here, now, brought to you by the Java language environment.

# Java—Simple and Familiar 2

You know you've achieved perfection in design,
Not when you have nothing more to add,
But when you have nothing more to take away.

Antoine de Saint Exupery.

In his science-fiction novel, *The Rolling Stones*, Robert A. Heinlein comments:

> *Every technology goes through three stages: first a crudely simple and quite unsatisfactory gadget; second, an enormously complicated group of gadgets designed to overcome the shortcomings of the original and achieving thereby somewhat satisfactory performance through extremely complex compromise; third, a final proper design therefrom.*

Heinlein's comment could well describe the evolution of many programming languages. Java presents a new viewpoint in the evolution of programming languages—creation of a small and simple language that's still sufficiently comprehensive to address a wide variety of software application development. While Java superficially like C and C++, Java gained its simplicity from the systematic removal of features from its predecessors. This chapter discusses two of the primary design features of Java, namely, it's *simple* (from removing features) and *familiar* (because it looks like C and C++). The next chapter discusses Java's *object-oriented* features in more detail. At the end of this chapter you'll find a discussion on features eliminated from C and C++ in the evolution of Java.

### *Design Goals*

*Simplicity* is one of Java's overriding design goals. Simplicity and removal of many "features" of dubious worth from its C and C++-based ancestors keep Java relatively small and reduce the programmer's burden in producing reliable applications. To this end, Java design team examined many aspects of the "modern" C and C++ languages* to determine features that could be eliminated in the context of modern object-oriented programming.

Another major design goal is that Java look *familiar* to a majority of programmers in the personal computer and workstation arenas, where a large fraction of system programmers and application programmers are familiar with C and C++. Thus, Java "looks like" C++. Programmers familiar with C, Objective C, C++, Eiffel, Ada, and related languages should find their Java language learning curve quite short—on the order of a couple of weeks.

To illustrate the simple and familiar aspects of Java, we follow the tradition of a long line of illustrious programming books by showing you the `HelloWorld` program. It's about the simplest program you can write that actually does something. Here's `HelloWorld` implemented in Java.

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

This example declares a *class* named `HelloWorld`. Classes are discussed in the next chapter on object-oriented programming, but in general we assume the reader is familiar with object technology and understands the basics of classes, objects, instance variables, and methods.

Within the `HelloWorld` class, we declare a single *method* called `main()` which in turn contains a single *method invocation* to display the string "Hello world!" on the standard output. The statement that prints "Hello world!" does so by

* Now enjoying their silver anniversaries

invoking the `println` method of the `out` object. The `out` object is a class variable in the `System` class that performs output operations on files. That's all there is to `HelloWorld`.

## 2.1 Main Features of the Java Language

Java follows C++ to some degree, which carries the benefit of it being familiar to many programmers. This section describes the essential features of Java and points out where the language diverges from its ancestors C and C++.

### 2.1.1 Primitive Data Types

Other than the primitive data types discussed here, everything in Java is an object. Even the primitive data types can be encapsulated inside library-supplied objects if required. Java follows C and C++ fairly closely in its set of basic data types, with a couple of minor exceptions. There are only three groups of primitive data types, namely, *numeric* types, *Boolean* types, and *arrays*.

#### Numeric Data Types

*Integer* numeric types are 8-bit `byte`, 16-bit `short`, 32-bit `int`, and 64-bit `long`. The 8-bit `byte` data type in Java has replaced the old C and C++ `char` data type. Java places a different interpretation on the `char` data type, as discussed below.

There is no `unsigned` type specifier for integer data types in Java.

*Real* numeric types are 32-bit `float` and 64-bit `double`. Real numeric types and their arithmetic operations are as defined by the IEEE 754 specification. A floating point *literal* value, like `23.79`, is considered `double` by default; you must explicitly cast it to `float` if you wish to assign it to a `float` variable.

#### Character Data Types

Java language *character* data is a departure from traditional C. Java's `char` data type defines a sixteen-bit *Unicode* character. Unicode characters are unsigned 16-bit values that define character codes in the range 0 through 65,535. If you write a declaration such as

```
char  myChar = ‘Q’;
```

you get a Unicode (16-bit unsigned value) type that's initialized to the Unicode value of the character Q. By adopting the Unicode character set standard for its character data type, Java language applications are amenable to internationalization and localization, greatly expanding the market for world-wide applications.

#### *Boolean Data Types*

Java has added a `boolean` data type as a primitive type, tacitly ratifying existing C and C++ programming practice, where developers define keywords for TRUE and FALSE or YES and NO or similar constructs. A Java `boolean` variable assumes the value `true` or `false`. A Java `boolean` is a distinct data type; unlike common C practice, a Java `boolean` type can't be converted to any numeric type.

### *2.1.2 Arithmetic and Relational Operators*

All the familiar C and C++ operators apply. Because Java lacks `unsigned` data types, the **>>>** operator has been added to the language to indicate an unsigned (logical) right shift. Java also uses the **+** operator for string concatenation; concatenation is covered below in the discussion on strings.

### *2.1.3 Arrays*

In contrast to C and C++, Java language *arrays* are first-class language objects. An array in Java is a real object with a run-time representation. You can declare and allocate arrays of any type, and you can allocate arrays of arrays to obtain multi-dimensional arrays.

You declare an array of, say, `Points` (a class you've declared elsewhere) with a declaration like this:

```
Point  myPoints[];
```

This code states that `myPoints` is an uninitialized array of `Points`. At this time, the only storage allocated for myPoints is a reference handle. At some future time you must allocate the amount of storage you need, as in:

```
myPoints = new Point[10];
```

to allocate an array of ten references to `Points` that are initialized to the null reference. Notice that this allocation of an array doesn't actually allocate any objects of the Point class for you; you will have to also allocate the Point objects, something like this:

```
int  i;

for (i = 0;  i < 10;  i++) {
    myPoints[i] = new Point();
}
```

Access to elements of `myPoints` can be performed via the normal C-style indexing, but all array accesses are checked to ensure that their indices are within the range of the array. An *exception* is generated if the index is outside the bounds of the array.

To get the length of an array, use the `length()` *accessor method* on the array object whose length you wish to know: `myPoints.length()` returns the number of elements in `myPoints`. For instance, the code fragment:

```
howMany = myPoints.length();
```

would assign the value 10 to the `howMany` variable.

The C notion of a pointer to an array of memory elements is gone, and with it, the arbitrary pointer arithmetic that leads to unreliable code in C. No longer can you walk off the end of an array, possibly trashing memory and leading to the famous "delayed-crash" syndrome, where a memory-access violation today manifests itself hours or days later. Programmers can be confident that array checking in Java will lead to more robust and reliable code.

### *2.1.4 Strings*

*Strings* are Java language objects, not pseudo-arrays of characters as in C. There are actually two kinds of string objects: the `String` class is for read-only (immutable) objects. The `StringBuffer` class is for string objects you wish to modify (mutable string objects).

Although strings are Java language objects, Java compiler follows the C tradition of providing a syntactic convenience that C programmers have enjoyed with C-style strings, namely, the Java compiler understands that a string of characters enclosed in double quote signs is to be instantiated as a `String` object. Thus, the declaration:

```
String hello = "Hello world!";
```

instantiates an object of the String class behind the scenes and initializes it with a character string containing the Unicode character representation of `"Hello world!"`.

Java has extended the meaning of the + operator to indicate *string concatenation*. Thus you can write statements like:

```
System.out.println("There are " + num + " characters in the file.");
```

This code fragment concatenates the string `"There are "` with the result of converting the numeric value `num` to a string, and concatenates that with the string `" characters in the file."`. Then it prints the result of those concatenations on the standard output.

Just as with array objects, `String` objects provide a `length()` accessor method to obtain the number of characters in the string.

### 2.1.5 Multi-Level Break

Java has no `goto` statement. To `break` or `continue` multiple-nested loop or switch constructs, you can place labels on loop and `switch` constructs, and then `break` out of or `continue` to the block named by the label. Here's a small fragment of code from Java's built-in `String` class:

```
test:  for (int i = fromIndex; i + max1 <= max2; i++) {
           if (charAt(i) == c0) {
               for (int k = 1; k<max1; k++) {
                   if (charAt(i+k) != str.charAt(k)) {
                       continue test;
                   }
               }     /*  end of inner for loop  */
           }
       }             /*  end of outer for loop  */
```

The `continue test` statement is inside a `for` loop nested inside another `for` loop. By referencing the label `test`, the `continue` statement passes control to the outer `for` statement. In traditional C, `continue` statements can only continue the immediately enclosing block; to continue or exit outer blocks, programmers have traditionally either used auxiliary Boolean variables whose only purpose is to determine if the outer block is to be continued or exited;

alternatively, programmers have (mis)used the `goto` statement to exit out of nested blocks. Use of labelled blocks in Java leads to considerable simplification in programming effort and a major reduction in maintenance.

The notion of labelled blocks dates back to the mid-1970s, but it hasn't caught on to any large extent in modern programming languages. Perl is another modern programming language that implements the concept of labelled blocks. Perl's `next` *label* and `last` *label* are equivalent to `continue` *label* and `break` *label* statements in Java.

### 2.1.6 Memory Management and Garbage Collection

C and C++ programmers are by now accustomed to the problems of explicitly managing memory: allocating memory, freeing memory, and keeping track of what memory can be freed when. Explicit memory management has proved to be a fruitful source of bugs, crashes, memory leaks, and poor performance.

Java completely removes the memory management load from the programmer. C-style pointers, pointer arithmetic, `malloc,` and `free` do not exist. *Automatic garbage collection* is an integral part of Java and its run-time system. While Java has a `new` operator to allocate memory for objects, there is no explicit `free` function. Once you have allocated an object, the run-time system keeps track of the object's status and automatically reclaims memory when objects are no longer in use, freeing memory for future use.

Java's memory management model is based on *objects* and *references* to objects. Because Java has no pointers, all references to allocated storage, which in practice means all references to an object, are through symbolic "handles". The Java memory manager keeps track of references to objects. When an object has no more references, the object is a candidate for garbage collection.

Java's memory allocation model and automatic garbage collection make your programming task easier, eliminate entire classes of bugs, and in general provide better performance than you'd obtain through explicit memory management. Here's a code fragment that illustrates when garbage collection happens. It's an example from the on-line Java language programmer's guide:

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
```

```
                dest.appendChar(source.charAt(i));
            }
            return dest.toString();
        }
    }
```

The variable `dest` is used as a temporary object reference during the execution of the `reverseIt` method. When `dest` goes out of scope (the `reverseIt` method returns), the reference to that object has gone away and it's then a candidate for garbage collection.

### 2.1.7 The Background Garbage Collector

The Java garbage collector achieves high performance by taking advantage of the nature of a user's behavior when interacting with software applications such as the HotJava browser. The typical user of the typical interactive application has many natural pauses where they're contemplating the scene in front of them or thinking of what to do next. The Java run-time system takes advantage of these idle periods and runs the garbage collector in a low priority thread when no other threads are competing for CPU cycles. The garbage collector gathers and compacts unused memory, increasing the probability that adequate memory resources are available when needed during periods of heavy interactive use.

This use of a thread to run the garbage collector is just one of many examples of the synergy one obtains from Java's integrated multithreading capabilities—an otherwise intractable problem is solved in a simple and elegant fashion.

### 2.1.8 Integrated Thread Synchronization

Java supports multithreading, both at the language (syntactic) level and via support from its run-time system and thread objects. While other systems have provided facilities for multithreading (usually via "lightweight process" libraries), building multithreading support into the language itself provides the programmer with a much easier and more powerful tool for easily creating thread-safe multithreaded classes. Multithreading is discussed in more detail in Chapter 5.

## 2.2 Features Removed from C and C++

The earlier part of this chapter concentrated on the principal features of Java. This section discusses features removed from C and C++ in the evolution of Java.

The first step was to *eliminate redundancy* from C and C++. In many ways, the C language evolved into a collection of overlapping features, providing too many ways to say the same thing, while in many cases not providing needed features. C++, in an attempt to add "classes in C", merely added more redundancy while retaining many of the inherent problems of C.

### 2.2.1 No More Typedefs, Defines, or Preprocessor

Source code written in Java is *simple*. There is no *preprocessor*, no `#define` and related capabilities, no `typedef`, and absent those features, no longer any need for *header files*. Instead of header files, Java language source files provide the definitions of other classes and their methods.

A major problem with C and C++ is the amount of context you need to understand another programmer's code: you have to read all related header files, all related `#defines`, and all related `typedefs` before you can even begin to analyze a program. In essence, programming with `#defines` and `typedefs` results in every programmer inventing a new programming language that's incomprehensible to anybody other than its creator, thus defeating the goals of good programming practices.

In Java, you obtain the effects of `#define` by using constants. You obtain the effects of `typedef` by declaring classes—after all, a class effectively declares a new type. You don't need header files because the Java compiler compiles class definitions into a binary form that retains all the type information through to link time.

By removing all this baggage, Java becomes remarkably *context-free*. Programmers can read and understand code and, more importantly, modify and reuse code much faster and easier.

### 2.2.2 No More Structures or Unions

Java has no structures or unions as complex data types. You don't need structures and unions when you have classes; you can achieve the same effect simply by declaring a class with the appropriate instance variables.

The code fragment below declares a class called `Point`.

```
class Point extends Object {
    double  x;
    double  y;
      methods to access the instance variables
}
```

The following code fragment declares a class called `Rectangle`, that uses objects of the `Point` class as instance variables.

```
class Rectangle extends Object {
    Point  lowerLeft;
    Point  upperRight;
      methods to access the instance variables
}
```

In C you'd define these classes as structures. In Java, you simply declare classes. You can make the instance variables as private or as public as you wish, depending on how much you wish to hide the details of the implementation from other objects.

### 2.2.3 No More Functions

Java has no *functions*. Object-oriented programming supersedes functional and procedural styles. Mixing the two styles just leads to confusion and dilutes the purity of an object-oriented language. Anything you can do with a function you can do just as well by defining a class and creating methods for that class. Consider the `Point` class from above. We've added public methods to set and access the instance variables:

```
class Point extends Object {
    double  x;
    double  y;
```

```
        public void setX(double x) {
            this.x = x;
        }
        public void setY(double y) {
            this.y = y;
        }
        public double x() {
            return x;
        }
        public double y() {
            return x;
        }
    }
```

If the `x` and `y` instance variables are private to this class, the only means to access them is via the public methods of the class. Here's how you'd use objects of the `Point` class from within, say, an object of the `Rectangle` class:

```
    class Rectangle extends Object {
        Point  lowerLeft;
        Point  upperRight;

        public void setEmptyRect() {
            lowerLeft.setX(0.0);
            lowerLeft.setY(0.0);
            upperRight.setX(0.0);
            upperRight.setY(0.0);
        }
    }
```

It's not to say that functions and procedures are inherently wrong. But given classes and methods, we're now down to only one way to express a given task. By eliminating functions, your job as a programmer is immensely simplified: you work *only* with classes and their methods.

### 2.2.4 No More Multiple Inheritance

*Multiple inheritance*—and all the problems it generates—has beed discarded from Java. The desirable features of multiple inheritance are provided by *interfaces*—conceptually similar to Objective C protocols.

An interface is not a definition of an object. Rather, it's a definition of a set of methods that one or more objects will implement. An important issue of interfaces is that they declare only methods and constants. No variables may be defined in interfaces.

### 2.2.5 No More Goto Statements

Java has no `goto` statement*. Studies illustrated that `goto` is (mis)used more often than not simply "because it's there". Eliminating `goto` led to a simplification of the language—there are no rules about the effects of a `goto` into the middle of a `for` statement, for example. Studies on approximately 100,000 lines of C code determined that roughly 90 percent of the `goto` statements were used purely to obtain the effect of breaking out of nested loops. As mentioned above, multi-level `break` and `continue` remove most of the need for `goto` statements.

### 2.2.6 No More Operator Overloading

There are no means provided by which programmers can overload the standard arithmetic operators. Once again, the effects of operator overloading can be just as easily achieved by declaring a class, appropriate instance variables, and appropriate methods to manipulate those variables.

### 2.2.7 No More Automatic Coercions

Java prohibits C and C++ style *automatic coercions.* If you wish to coerce a data element of one type to a data type that would result in loss of precision, you must do so explicitly by using a cast. Consider this code fragment:

```
int  myInt;
double  myFloat = 3.14159;
myInt = myFloat;
```

The assignment of `myFloat` to `myInt` would result in a compiler error indicating a possible loss of precision and that you must use an explicit cast. Thus, you should re-write the code fragments as:

* However, `goto` is still a reserved word.

```
int  myInt;
double  myFloat = 3.14159;
myInt = (int)myFloat;
```

### 2.2.8 No More Pointers

Most studies agree that *pointers* are one of the primary features that enable programmers to inject bugs into their code. Given that structures are gone, and arrays and strings are objects, the need for pointers to these constructs goes away. Thus, Java has no pointers. Any task that would require arrays, structures, and pointers in C can be more easily and reliably performed by declaring objects and arrays of objects. Instead of complex pointer manipulation on array pointers, you access arrays by their arithmetic indices. The Java run-time system checks all array indexing to ensure indices are within the bounds of the array.

You no longer have dangling pointers and trashing of memory because of incorrect pointers, because there are no pointers in Java.

## 2.3 Summary

To sum up this chapter, Java is:

- *Simple*—the number of language constructs you need to understand to get your job done is minimal.
- *Familiar*—Java looks like C and C++ while discarding the overwhelming complexities of those languages.

Now that you've seen how Java was simplified by removal of features from its predecessors, read the next chapter for a discussion on the *object-oriented* features of Java.

# *Java is Object Oriented* 3

My Object All Sublime
I Will Achieve in Time

Gilbert and Sullivan—*The Mikado*

To stay abreast of modern software development practices, Java is *object oriented* from the ground up. The point of designing an object-oriented language is not simply to jump on the latest programming fad. The object-oriented paradigm meshes well with the needs of client-server and distributed software. Benefits of object technology are rapidly becoming realized as more organizations move their applications to the distributed client-server model.

Unfortunately, "object oriented" remains misunderstood, over-marketed as the silver bullet that will solve all our software ills, or takes on the trappings of a religion. The cynic's view of object-oriented programming is that it's just a new way to organize your source code. While there may be some merit to this view, it doesn't tell the whole story, because you can achieve results with object-oriented programming techniques that you can't with procedural techniques.

An important characteristic that distinguishes objects from ordinary procedures or functions is that an object can have a lifetime greater than that of the object that created it. This aspect of objects is subtle and mostly overlooked.In the distributed client-server world, this creates the potential for objects to be created in one place, passed around networks, and stored elsewhere, possibly in databases, to be retrieved for future work.

As an object-oriented language, Java draws on the best concepts and features of previous object-oriented languages, primarily Eiffel, SmallTalk, Objective C, and C++. Java goes beyond C++ in both extending the object model and removing the major complexities of C++. With the exception of its primitive data types, everything in Java is an object, and even the primitive types can be encapsulated within objects if the need arises.

## 3.1 Object Technology in Java

To be truly considered “object oriented”, a programming language should support at a minimum four characteristics:

- *Encapsulation*—implements information hiding and modularity (abstraction)
- *Polymorphism*—the same message sent to different objects results in behavior that’s dependent on the nature of the object receiving the message
- *Inheritance*—you define new classes and behavior based on existing classes to obtain code re-use and code organization
- *Dynamic binding*—objects could come from anywhere, possibly across the network. You need to be able to send messages to objects without having to know their specific type at the time you write your code. Dynamic binding provides maximum flexibility while a program is executing

Java meets these requirements nicely, and adds considerable run-time support to make your software development job easier.

## 3.2 What Are Objects?

At its simplest, object technology is a collection of analysis, design, and programming methodologies that focuses design on *modelling* the characteristics and behavior of objects in the real world. True, this definition appears to be somewhat circular, so let’s try to break out into clear air.

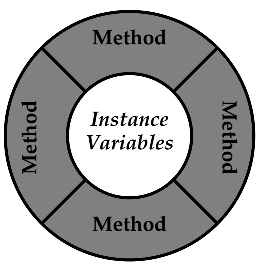
What are objects? They’re software programming *models*. In your everyday life, you’re surrounded by objects: cars, coffee machines, ducks, trees, and so on. Software applications contain objects: buttons on user interfaces, spreadsheets and spreadsheet cells, property lists, menus, and so on. These objects have *state* and *behavior*. You can represent all these things with software constructs called objects, which can also be defined by their *state* and their *behavior*.

In your everyday transportation needs, a *car* can be modelled by an object. A car has *state* (how fast it's going, in which direction, its fuel consumption, and so on) and *behavior* (starts, stops, turns, slides, and runs into trees).

You drive your car to your office, where you track your stock portfolio. In your daily interactions with the stock markets, a *stock* can be modelled by an object. A stock has *state* (daily high, daily low, open price, close price, earnings per share, relative strength), and *behavior* (changes value, performs splits, has dividends).

After watching your stock decline in price, you repair to the cafe to console yourself with a cup of good hot coffee. The *espresso machine* can be modelled as an object. It has *state* (water temperature, amount of coffee in the hopper) and it has *behavior* (emits steam, makes noise, and brews a perfect cup of *java*).

## 3.3 Basics of Objects




In the programming implementation of an object, its *state* is defined by its *instance variables*. Instance variables are private to the object. Unless explicitly made public or made available to other "friendly" classes, an object's instance variables are inaccessible from outside the object.

An object's *behavior* is defined by its *methods*. Methods manipulate the instance variables to create new state; an object's methods can also create new objects.

The small picture to the left is a commonly used graphical representation of an object. The diagram illustrates the conceptual structure of a software object—it's kind of like a cell, with an outer membrane that's its interface to the world, and an inner nucleus that's protected by the outer membrane.

An object's *instance variables* (data) are packaged, or encapsulated, within the object. The instance variables are surrounded by the object's *methods*. With certain well-defined exceptions, the object's methods are the only means by which other objects can access or alter its instance variables. In Java, classes can declare their instance variables to be `public`, in which cases the instance variables are globally accessible to other objects. Declarations of accessibility are covered in later in *Access Specifiers*.

### 3.3.1 Classes

A *class* is a software construct that defines the *instance variables* and *methods* of an object. A class in and of itself is not an object. A class is a *template* that defines how an object will look and behave when the object is created or *instantiated* from the specification declared by the class. You obtain concrete objects by instantiating a previously defined class. You can instantiate many objects from one class definition, just as you can construct many houses that area all the same from a single architect's drawing. Here's the basic declaration of a very simple class called Point

```
class Point extends Object {
    public double  x;    /*  instance variable   */
    public double  y;    /*  instance variable   */
}
```

As mentioned, this declaration merely defines a template from which real objects can be instantiated, as described next.

### 3.3.2 Instantiating an Object from its Class

Having declared the size and shape of the Point *class* above, any other object can now create a `Point` *object*—an instance of the `Point` class—with a fragment of code like this:

```
Point   myPoint;            //  declares a variable to refer to a  Point  object

myPoint = new Point();      //  allocates an instance of  a  Point  object
```

Now, you can access the variables of this Point object by referring to the names of the variables, qualified with the name of the object:

```
myPoint.x = 10.0;
myPoint.y = 25.7;
```

This referencing scheme, similar to a C structure reference, works because the instance variables of `Point` were declared `public` in the class declaration. Had the instance variables not been declared `public`, objects outside of the package within which `Point` was declared could not access its instance variables in this direct manner. The `Point` class declaration would then need to provide *accessor methods* to set and get its variables. This topic is discussed in a little more detail after the discussion on constructors.

### 3.3.3 Constructors

When you declare a class in Java, you can declare optional *constructors* that perform initialization when you instantiate objects from that class. You can also declare an optional *finalizer*, discussed later. Let's go back to our `Point` class from before:

```
class Point extends Object {
    public double  x;     /*  instance variable   */
    public double  y;     /*  instance variable   */

    Point() {          /*   constructor to initialize to default zero value   */
        x = 0.0;
        y = 0.0;
    }
                      /*   constructor to initialize to specific value   */
    Point(double x, double y) {
        this.x = x;      /*   set instance variables to passed parameters   */
        this.y = y;
    }
}
```

Methods with the same name as the class as in the code fragment are called *constructors*. When you create (instantiate) an object of the `Point` class, the constructor method is invoked to perform any initialization that's needed—in this case, to set the instance variables to an initial state.

This example is a variation on the `Point` class from before. Now, when you wish to create and initialize `Point` objects, you can get them initialized to their default values, or you can initialize them to specific values:

```
Point  lowerLeft;
Point  upperRight;

lowerLeft = new Point();          /*  initialize to default zero value   */
upperRight = new Point(100.0, 200.0);   /*  initialize to non- zero    */
```

The specific constructor that's used when creating a new `Point` object is determined from the type and number of parameters in the `new` invocation.

### *The* **`this`** *Variable*

What's the `this` variable in the examples above? `this` refers to the object you're "in" right now. In other words, `this` refers to the receiving object. You use `this` to clarify which variable you're referring to. In the two-parameter `Point` method, `this.x` means the `x` instance variable of this object, rather than the `x` parameter to the `Point` method.

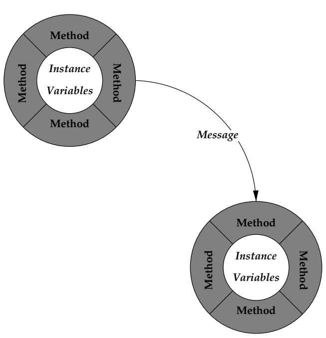
In the example above, the constructors are simply conveniences for the `Point` class. There are situations, however, where constructors are necessary, especially in cases where the object being instantiated must itself instantiate other objects. Let's illustrate this by declaring a `Rectangle` class that uses two `Point` objects to define its bounds:

```
class Rectangle extends Object {
   private Point lowerLeft;
    private Point upperRight;

    Rectangle() {
        lowerLeft = new Point();
        upperRight = new Point();
    }
        . . .
     instance methods appear in here
        . . .
}
```

In this example, the `Rectangle()` constructor is vitally necessary to ensure that the two `Point` objects are instantiated at the time a `Rectangle` object is instantiated, otherwise, the `Rectangle` object would subsequently try to reference points that have not yet been allocated, and would fail.

## 3.3.4 Methods and Messaging




If an object wants another object to do some work on its behalf, then in the parlance of object-oriented programming, the first object sends a *message* to the second object. In response, the second object selects the appropriate *method* to invoke. Java method invocations look similar to functions in C and C++.

Using the message passing paradigms of object-oriented programming, you can build entire networks and webs of objects that pass messages between them to change state. This programming technique is one of the best ways to create models and simulations of complex real-world systems. Let's redefine the declaration of the `Point` class from above such that its instance variables are `private`, and supply it with *accessor methods* to access those variables.

```
class Point extends Object {
    private double  x;    /*  instance variable   */
    private double  y;    /*  instance variable   */

    Point() {     /*   constructor to initialize to zero    */
        x = 0.0;
        y = 0.0;
    }
                                  /*   constructor to initialize to specific value    */
    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void setX(double x) {     /*   accessor method    */
        this.x = x;
    }
    public void setY(double y) {     /*   accessor method    */
        this.y = y;
    }
    public double getX() {     /*   accessor method    */
        return x;
    }
    public double getY() {     /*   accessor method    */
        return y;
    }
}
```

These method declarations provides the flavor of how the `Point` class provides access to its variables from the outside world. Another object that wants to manipulate the instance variables of `Point` objects must now do so via the accessor methods:

```
Point  myPoint;            //   declares a variable to refer to a  Point  object

myPoint = new Point();     //   allocates an instance of  a  Point  object
myPoint.setX(10.0);        //   sets the x variable via the accessor method
myPoint.setY(25.7);
```

Making instance variables public or private is a design tradeoff the designer makes when declaring the classes. By making instance variables public, you are exposing some of the details of the implementation of the class, thereby providing higher efficiency and conciseness of expression at the possible expense of hindering future maintenance efforts. By hiding details of the internal implementation of a class, you have the potential to change the implementation of the class in the future without breaking any code that uses that class.

### 3.3.5 Finalizers

You can also declare an optional *finalizer* that will perform necessary tear-down actions when the garbage collector is about to free an object. This code fragment illustrates a `finalize` method in a class.

```
/**
  * Close the stream when garbage is collected.
  */
protected void finalize() {
    try {
        file.close();
    } catch (Exception e) {
    }
}
```

This `finalize` method will be invoked when the object is about to be garbage collected, which means that the object must shut itself down in an orderly fashion. In the particular code fragment above, the `finalize` method merely closes an I/O file stream that was used by the object, to ensure that the file descriptor for the stream is closed.

### 3.3.6 Subclassing

*Subclassing* is the mechanism by which new and enhanced objects can be defined in terms of existing objects. One example: a zebra is a horse with stripes. If you wish to create a zebra object, you notice that a zebra is kind of like a horse, only with stripes. In object-oriented terms, you'd create a new class called Zebra, which is a *subclass* of the Horse class. In Java language terms, you'd do something like this:

```
class Zebra extends Horse {
    Your new instance variables and new methods go here
}
```

The definition of `Horse`, wherever it is, would define all the methods to describe the *behavior* of a horse: eat, neigh, trot, gallop, buck, and so on. The only method you need to override is the method for drawing the hide. You gain the benefit of already written code that does all the work—you don't have to re-invent the wheel, or in this case, the hoof. The `extends` keyword tells the Java compiler that Zebra is a subclass of Horse. Zebra is said to be a *derived class*—it's derived from Horse, which is called a *base class*.

Here's an example of subclassing a variant of our `Point` class from previous examples to create a new three-dimensional point called `ThreePoint`:

```
class Point extends Object {
    protected double  x;     /*  instance variable   */
    protected double  y;     /*  instance variable   */

    Point() {      /*  constructor to initialize to zero   */
        x = 0.0;
        y = 0.0;
    }
}
class ThreePoint extends Point {
    protected double z;     /*  the z coordinate of the point   */

    ThreePoint() {         /*  default constructor   */
        x = 0.0;           /*  initialize the coordinates   */
        y = 0.0;
        z = 0.0;
    }
    ThreePoint(double x, double y, double z) {/* specific constructor  */
        this.x = x;         /*  initialize the coordinates   */
        this.y = y;
        this.z = z;
    }
}
```

Notice that `ThreePoint` adds a new instance variable for the *z* coordinate of the point. The *x* and *y* instance variables are *inherited* from the original `Point` class, so there's no need to declare them in `ThreePoint`. However, notice we had to make `Point`'s instance variables `protected` instead of `private` as in the previous examples. Had we left `Point`'s instance variables `private`, even its subclasses would be unable to access them, and the compilation would fail.

Subclassing enables you to use existing code that's already been developed and, much more important, tested, for a more generic case. You override the parts of the class you need for your specific behavior. Thus, subclassing gains you reuse of existing code—you save on design, development, and testing. The Java run-time system provides several libraries of utility functions that are tested and are also *thread safe*.

### 3.3.7 Access Control

When you declare a new class in Java, you can indicate the level of access permitted to its instance variables and methods. Java provides four levels of access specifiers. Three of the levels must be explicitly specified if you wish to use them. They are `public`, `protected`, and `private`.

The fourth level doesn't have a name—it's often called "friendly" and is the access level you obtain if you don't specify otherwise. The "friendly" access level indicates that your instance variables and methods are accessible to all objects within the same package, but inaccessible to objects outside the package.

The friendly access level comes in handy if you're creating *packages* of classes that are related to each other and can access each other's instance variables directly. A geometry package consisting of `Point` and `Rectangle` classes, for instance, might well be easier and cleaner to implement, as well as more efficient, if the `Point`'s instance variables were directly available to the `Rectangle` class. Outside of the geometry package, however, the details of implementations are hidden from the rest of the world, giving you the freedom to changed implementation details without worrying you'll break code that uses those classes. Packages are a Java language construct that gather collections of related classes into a single container. For example, all Java I/O system code is collected into a single package. The primary benefit of packages is organizing many class definitions into a single unit. The secondary benefit from the programmer's viewpoint is that the "friendly" instance variables and methods are available to all classes within the same package, but not to classes defined outside the package.

`public` methods and instance variables are available to any other class anywhere.

`protected` means that instance variables and methods so designated are accessible only to subclasses of that class, and nowhere else.

`private` methods and instance variables are accessible only from within the class in which they're declared—they're not available even to their subclasses.

### 3.3.8 Class Variables and Class Methods

Java follows conventions from other object-oriented languages in providing *class methods* and *class variables*. Normally, variables you declare in a class definition are *instance variables*—there is one of those variables in every separate object that's created (instantiated) from the class. A class variable, on the other hand, is local to the class itself—there's only a single copy of the variable and it's shared by every object you instantiate from the class.

To declare class variables and class methods, you declare them as `static`. This short code fragment illustrates the declaration of class variables:

```
class Rectangle extends Object {
    static  final int version = 2;
    static  final int revision = 0;
}
```

The `Rectangle` class declares two `static` variables to define the version and revision level of this class. Now, every instance of Rectangle that you create from this class will share these same variables. Notice they're also defined as `final` because you want them to be constants.

Class methods are methods that are common to an entire class. When would you use class methods? Usually, when you have behavior that's common to every object of a class. For example, suppose you have a `Window` class. A useful item of information you can ask the class is the width of the border around the window. There's no point in having an instance method to obtain this information that's shared by every instance of `Window`—it makes more sense to have just one class method to return the border width.

Class methods can operate *only* on class variables. Class methods can't access instance variables, nor can they invoke instance methods. Like class variables, you declare class methods by defining them as `static`.

### 3.3.9 Abstract Methods

Abstract methods are a powerful construct in the object-oriented paradigm. To understand abstract methods, we look at the notion of an *abstract superclass*. An abstract superclass is a class in which you define methods that aren't actually implemented by that class—they only provide place-holders such that subsequent subclasses must override those methods and supply their actual implementation.

This all sounds wonderfully, well, *abstract*, so why would you need an abstract superclass? Let's look at a *concrete* example, no pun intended. Let's suppose you're going to a restaurant for dinner, and you decide that tonight you want to eat fish. Well, *fish* is somewhat abstract—you generally wouldn't just order fish; the waiter is highly likely to ask you what *specific* kind of fish you want. When you actually get to the restaurant, you will find out what kind of fish they have, and order a specific fish, say, sturgeon, or salmon, or opakapaka.

In the world of objects, an abstract class is like generic fish—the abstract class defines generic state and generic behavior, but you'll never see a real live implementation of an abstract class. What you will see is a *concrete subclass* of the abstract class, just as opakapaka is a specific (concrete) kind of fish.

Suppose you are creating a drawing application. The initial cut of your application can draw rectangles, lines, circles, polygons, and so on. Furthermore, you have a series of operations you can perform on the shapes—move, reshape, rotate, fill color, and so on. You *could* make each of these graphic shapes a separate class—you'd have a Rectangle class, a Line class, and so on. Each class needs instance variables to define its position, size, color, rotation and so on, which in turn dictates methods to set and get at those variables.

At this point, you realize you can collect all the instance variables into a single abstract superclass called `Graphical`, and implement most of the methods to manipulate the variables in that abstract superclass. The skeleton of your abstract superclass might look something like this:

```
abstract class Graphic extends Object {
    protected Point lowerLeft;      //  lower left of bounding box
    protected Point upperRight;     //  upper right of bounding box
                     . . .
                more instance variables
                     . . .
    public void setPosition(Point ll, Point ur) {
        lowerLeft = ll;
        upperRight = ur;
    }
    abstract void drawMyself();    //  abstract method
    }

}
```

Now, you can't instantiate the `Graphical` class, because it's declared `abstract`. You can only instantiate a *subclass* of it. When you implement the `Rectangle` class or the `Circle` class, you'd extend (subclass) `Graphical`. Within `Rectangle`, you'd provide a *concrete* implementation of the `drawMySelf()` method that draws a rectangle, because the definition of `drawMySelf()` must by necessity be unique to each shape inherited from the `Graphical` class. Let's see a small fragment of the `Rectangle` class declaration, where its `drawMySelf()` method operates in a somewhat PostScript'y fashion:

```
abstract class Rectangle extends Graphical {
    void drawMySelf() {          //   really does the drawing
        moveTo(lowerLeft.x, lowerLeft.y);
        lineTo(upperRight.x, lowerLeft.y);
        lineTo(upperRight.x, upperRight.y)
        lineTo(lowerLeft.x, upperRight.y);
                    . . .
              and so on and so on
                    . . .
    }
}
```

Notice, however, that in the declaration of the `Graphical` class, the `setPosition()` method was declared as a regular (`public void`) method. All methods that *can* be implemented by the abstract superclass can be declared there and their implementations defined at that time. Then, every class that inherits from the abstract superclass will also inherit those methods.

You can continue in this way adding new shapes that are subclasses of `Graphical`, and most of the time, all you ever need to implement is the methods that are unique to the specific shape. You gain the benefit of re-using all the code that was defined inside the abstract superclass.

## 3.4 Summary

This chapter has conveyed the essential aspects of Java as an *object-oriented* language. To sum up:

- *Classes* define templates from which you *instantiate* (create) distinct concrete *objects*.
- *Instance variables* hold the *state* of a specific object.

- Objects communicate by sending *messages* to each other. Objects respond to messages by selecting a *method* to execute.
- *Methods* define the *behavior* of objects instantiated from a class. It is an object's methods that manipulate its instance variables. Unlike regular procedural languages, classes in an object-oriented language may have methods with the same names as other classes. A given object responds to a message in ways determined by the nature of that object, providing *polymorphic* behavior.
- *Subclassing* provides the means by which a new class can *inherit* instance variables and methods from any already defined class. The newly declared class can add new instance variables (extra state), can add new methods (new behavior), or can *override* the methods of its superclass (different behavior). Subclassing provides code reuse.

Taken together, the concepts of object-oriented programming create a powerful and simple paradigm for software developers to share and re-use code and build on the work of others.

# *Architecture Neutral, Portable, and Robust* 4

With the phenomenal growth of networks, today's developers must "think distributed". Applications—even parts of applications—must be able to migrate easily to a wide variety of computer systems, a wide variety of hardware architectures, and a wide variety of operating system architectures. They must operate with a plethora of graphical user interfaces.

Clearly, applications must be able to execute anywhere on the network without prior knowledge of the target hardware and software platform. If application developers are forced to develop for specific target platforms, the binary distribution problem quickly becomes unmanageable. Various and sundry methods have been employed to overcome the problem, such as creating "fat" binaries that adapt to the specific hardware architecture, but such methods are not only clumsy but are still geared to a specific operating system. To solve the binary-distribution problem, software applications and fragments of applications must be *architecture neutral* and *portable*.

R*eliability* is also at a high premium in the distributed world. Code from anywhere on the network should work *robustly* with low probabilities of creating "crashes" in applications that import fragments of code.

This chapter describes the ways in which Java has addressed the issues of architecture neutrality, portability, and reliability.

## 4.1 Architecture Neutral

The solution that the Java system adopts to solve the binary-distribution problem is a "binary code format" that's independent of hardware architectures, operating system interfaces, and window systems. The format of this system-independent binary code is *architecture neutral*. If the Java run-time system is made available on a given hardware and software platform, an application written in Java can then execute on that platform without the need to perform any special porting work for that application.

### 4.1.1 Byte Codes

The Java compiler doesn't generate "machine code" in the sense of native hardware instructions—rather, it generates *bytecodes*: a high-level, machine-independent code for a hypothetical machine that is implemented by the Java interpreter and run-time system.

One of the early examples of the bytecode approach was the UCSD P-System, which was ported to a variety of eight-bit architectures in the middle 1970s and early 1980s and enjoyed widespread popularity during the heyday of eight-bit machines. Coming up to the present day, current architectures have the power to support the bytecode approach for distributed software. Java bytecodes are designed to be easy to interpret on any machine, or to dynamically translate into native machine code if required by performance demands.

The architecture neutral approach is useful not only for network-based applications, but also for single-system software distribution. In today's software market, application developers have to produce versions of their applications that are compatible with the IBM PC, Apple Macintosh, and fifty-seven flavors of workstation and operating system architectures in the fragmented UNIX marketplace.

With the PC market (through Windows 95 and Windows NT) diversifying onto many CPU architectures, and Apple moving full steam from the 68000 to the PowerPC, production of software to run on all platforms becomes almost impossible until now. Using Java, coupled with the Abstract Window Toolkit, the same version of your application can run on all platforms.

## 4.2 Portable

The primary benefit of the interpreted byte code approach is that compiled Java language programs are *portable* to any system on which the Java interpreter and run-time system have been implemented.

The architecture-neutral aspect discussed above is one major step towards being portable, but there's more to it than that. C and C++ both suffer from the defect of designating many fundamental data types as "implementation dependent". Programmers labor to ensure that programs are portable across architectures by programming to a lowest common denominator.

Java eliminates this issue by defining standard behavior that will apply to the data types across all platforms. Java specifies the sizes of all its primitive data types and the behavior of arithmetic on them. Here are the data types:

| | |
|---|---|
| `byte` | 8-bit two's complement |
| `short` | 16-bit two's complement |
| `int` | 32-bit two's complement |
| `long` | 64-bit two's complement |
| `float` | 32-bit IEEE 754 floating point |
| `double` | 64-bit IEEE 754 floating point |
| `char` | 16-bit Unicode character |

The data types and sizes described above are standard across all implementations of Java. These choices are reasonable given current microprocessor architectures because essentially all central processor architectures in use today share these characteristics. That is, most modern processors can support two's-complement arithmetic in 8-bit to 64-bit integer formats, and most modern processors support single- and double-precision floating point.

The Java environment itself is readily portable to new architectures and operating systems. The Java compiler is written in Java. The Java run-time system is written in ANSI C with a clean portability boundary which is essentially POSIX-compliant. There are no "implementation-dependent" notes in the Java language specification.

## 4.3 Robust

Java is intended for developing software that must be *robust,* highly *reliable*, and *secure,* in a variety of ways. There's strong emphasis on early checking for possible problems, as well as later dynamic (run-time) checking, to eliminate error-prone situations.

### 4.3.1 Strict Compile-Time and Run-Time Checking

The Java compiler employs extensive and stringent compile-time checking so that syntax-related errors can be detected early, before a program is deployed.

One of the advantages of a strongly typed language (like C++) is that it allows extensive compile-time checking, so bugs can be found early. Unfortunately, C++ inherits a number of loopholes in its compile-time checking from C. Unfortunately, C++ and C are relatively lax, most notably in the area of method or function declarations. Java imposes much more stringent requirements on the developer: Java *requires* explicit declarations and does not support C-style implicit declarations.

Many of the stringent compile-time checks at the Java compiler level are carried over to the run time, both to check consistency at run time, and to provide greater flexibility. The linker understands the type system and repeats many of the type checks done by the compiler, to guard against version mismatch problems.

The single biggest difference between Java and C or C++ is that Java's memory model eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays and strings, which means that the interpreter can check array and string indexes. In addition, a programmer can't write code that turns an arbitrary integer into an object reference by casting.

While Java doesn't pretend to completely remove the software quality assurance problem, removal of entire classes of programming errors considerably eases the job of testing and quality assurance.

## 4.4 Summary

Java—an *architecture-neutral* and *portable* programming language—provides an attractive and simple solution to the problem of distributing your applications across heterogeneous network-based computing platforms. In addition, the simplicity and *robustness* of the underlying Java language results in higher quality, reliable applications in which users can have a high level of confidence. The next chapter contains a brief discussion of Java's *interpreted* implementation.

# Interpreted and Dynamic 5

Programmers using "traditional" software development tools have become resigned to the artificial edit-compile-link-load-throw-the-application-off-the-cliff-let-it-crash-and-start-all-over-again style of current development practice.

Additionally, keeping track of what must be recompiled when a declaration changes somewhere else strains the capabilities of development tools—even fancy "make"-style tools such as found on UNIX systems. This development approach bogs down as the code bases of applications grow to hundreds of thousands of lines.

Better methods of fast and fearless prototyping and development are needed. The Java language environment is one of those better ways, because it's *interpreted* and *dynamic*.

As discussed in the previous chapter on architecture-neutrality, the Java compiler generates *byte codes* for the Java Virtual Machine[*], which was introduced briefly in Chapter 4. The notion of a virtual interpreted machine is not new. But the Java language brings the concepts into the realm of secure, distributed, network-based systems.

The Java language virtual machine is a strictly defined virtual machine for which an *interpreter* must be available for each hardware architecture and operating system on which you wish to run Java language applications. Once

* One of the ancestors of the virtual machine concept was the UCSD P System, developed by Kenneth Bowles at the University of California at San Diego in the late 1970s.

you have the Java language interpreter and run-time support available on a given hardware and operating system platform, you can run any Java language application from anywhere, always assuming the specific Java language application is written in a portable manner.

The notion of a separate "link" phase after compilation is pretty well absent from the Java environment. Linking, which is actually the process of loading new classes by the *Class Loader*, is a more incremental and lightweight process. The concomitant speedup in your development cycle means that your development process can be much more rapid and exploratory, and because of the robust nature of the Java language and run-time system, you will catch bugs at a much earlier phase of the cycle.

## 5.1 Dynamic Loading and Binding

The Java language's portable and interpreted nature produces a highly *dynamic* and *dynamically-extensible* system. The Java language was designed to adapt to evolving environments. Classes are linked in as required and can be downloaded from across networks. Incoming code is verified before being passed to the interpreter for execution.

Object-oriented programming has become accepted as a means to solve at least a part of the "software crisis", by assisting encapsulation of data and corresponding procedures, and encouraging reuse of code. Most programmers doing object-oriented development today have adopted C++ as their language of choice. But C++ suffers from a serious problem that impedes its widespread use in the production and distribution of "software ICs". This defect is known as the *fragile superclass problem*.

### 5.1.1 The Fragile Superclass Problem

This problem arises as a side-effect of the way that C++ is usually implemented. Any time you add a new method or a new instance variable to a class, any and all classes that reference that class will require a recompilation, or they'll break. Keeping track of the dependencies between class definitions and their clients has proved to be a fruitful source of programming error in C++, even with the help of "make"-like utilities.The fragile superclass issue is sometimes also referred to as the "constant recompilation problem." You *can* avoid these problems in C++, but with extraordinary difficulty, and doing so effectively means not using any of the language's object-oriented features directly. By avoiding the object-oriented features of C++, developers defeat the goal of re-usable "software ICs".

### 5.1.2 Solving the Fragile Superclass Problem

The Java language solves the fragile superclass problem in several stages. The Java compiler doesn't compile references down to numeric values—instead, it passes symbolic reference information through to the byte code verifier and the interpreter. The Java interpreter performs final name resolution once, when classes are being linked. Once the name is resolved, the reference is rewritten as a numeric offset, enabling the Java interpreter to run at full speed.

Finally, the storage layout of objects is not determined by the compiler. The layout of objects in memory is deferred to run time and determined by the interpreter. Updated classes with new instance variables or methods can be linked in without affecting existing code.

At the small expense of a name lookup the first time any name is encountered, the Java language eliminates the fragile superclass problem. Java programmers can use object-oriented programming techniques in a much more straightforward fashion without the constant recompilation burden engendered by C++. Libraries can freely add new methods and instance variables without any effect on their clients. Your life as a programmer is simpler.

### 5.1.3 Java Language Interfaces

An *interface* in the Java language is simply a specification of methods that an object implements. The concept of an interface in the Java language was borrowed from the Objective-C concept of a *protocol*. An interface does not include instance variables or implementation code. You can import and use multiple interfaces in a flexible manner, providing the benefits of multiple inheritance without the inherent difficulties created by the usual rigid class inheritance structure.

### 5.1.4 Run-Time Representations

Classes in the Java language have a run-time representation. There is a class named `Class`, instances of which contain run-time class definitions. If you're handed an object, you can find out what class it belongs to. In a C or C++ program, you may be handed a pointer to an object, but if you don't know what type of object it is, you have no way to find out. In the Java language, finding out based on the run-time type information is straightforward.

It is also possible to look up the definition of a class given a string containing its name. This means that you can compute a data type name and easily have it dynamically-linked into the running system.

## 5.2 Summary

The interpreted and dynamic nature of Java provides several benefits:

- The interpreted environment enables fast prototyping without waiting for the traditional compile and link cycle,
- The environment is dynamically extensible, whereby classes are loaded on the fly as required,
- The fragile superclass problem that plagues C++ developers is eliminated because of deferral of memory layout decisions to run time.

# Security in Java 6

*Security* commands a high premium in the growing use of the Internet for products and services ranging from electronic distribution of software and multimedia content, to "digital cash". The area of security with which we're concerned here is how the Java compiler and run-time system restrict application programmers from creating subversive code.

The Java language compiler and run-time system implement several layers of defense against potentially incorrect code. The environment starts with the assumption that nothing is to be trusted, and proceeds accordingly. The next few sections discuss the Java security models in greater detail.

## 6.1 Memory Allocation and Layout

One of the Java compiler's primary lines of defense is its memory allocation and reference model. First of all, *memory layout* decisions are not made by the Java language compiler, as they are in C and C++. Rather, memory layout is deferred to run time, and will potentially differ depending on the characteristics of the hardware and software platforms on which the Java system executes.

Secondly, Java does not have "pointers" in the traditional C and C++ sense of memory cells that contain the addresses of other memory cells.The Java compiled code references memory via symbolic "handles" that are resolved to real memory addresses at run time by the Java interpreter. Java programmers can't forge pointers to memory, because the memory allocation and referencing model is completely opaque to the programmer and controlled entirely by the underlying run-time system.

Very late binding of structures to memory means that programmers can't infer the physical memory layout of a class by looking at its declaration. By removing the C and C++ memory layout and pointer models, the Java language has eliminated the programmer's ability to get behind the scenes and manufacture pointers to memory. These features must be viewed as positive benefits rather than a restriction on the programmer, because they ultimately lead to more reliable and secure applications.

## 6.2 The Byte Code Verification Process

What about the concept of a "hostile compiler"? Although the Java compiler ensures that Java source code doesn't violate the safety rules, when an application such as the HotJava Browser imports a code fragment from anywhere, it doesn't actually know if code fragments follow Java language rules for safety: the code may not have been produced by a known-to-be trustworthy Java compiler. In such a case, how is the Java run-time system on your machine to trust the incoming bytecode stream? The answer is simple: the Java run-time system doesn't trust the incoming code, but subjects it to *bytecode verification*.

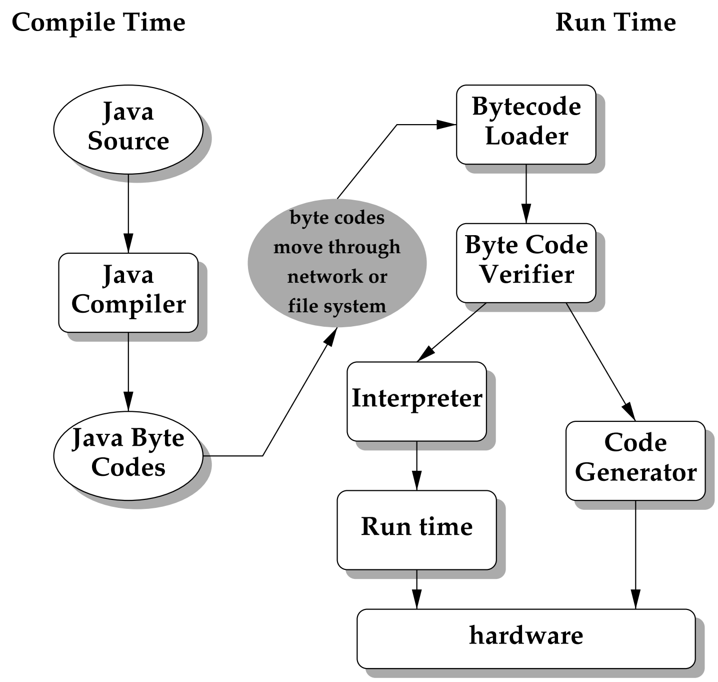
The tests range from simple verification that the format of a code fragment is correct, to passing each code fragment through a simple theorem prover to establish that it plays by the rules:

- it doesn't forge pointers,
- it doesn't violate access restrictions,
- it accesses objects as what they are (for example, `InputStream` objects are always used as `InputStreams` and never as anything else).

A language that is safe, plus run-time verification of generated code, establishes a base set of guarantees that interfaces cannot be violated.

### 6.2.1 The Byte Code Verifier

The *bytecode verifier* traverses the bytecodes, constructs the type state information, and verifies the types of the parameters to all the bytecode instructions.




The illustration shows the flow of data and control from Java language source code through the Java compiler, to the bytecode verifier and hence on to the Java interpreter. The important issue is that the Java bytecode loader and the bytecode verifier make no assumptions about the primary source of the bytecode stream—the code may have come from the local system, or it may have travelled halfway around the planet. The bytecode verifier acts as a sort of gatekeeper: it ensures that code passed to the Java interpreter is in a fit state to be executed and can run without fear of breaking the Java interpreter. Imported code is not allowed to execute by any means until after it has passed the verifier's tests. Once the verifier is done, a number of important properties are known:

- There are no operand stack overflows or underflows

- The types of the parameters of all bytecode instructions are known to always be correct
- Object field accesses are known to be legal—private, public, or protected

While all this checking appears excruciatingly detailed, by the time the bytecode verifier has done its work, the Java interpreter can proceed, knowing that the code will run securely. Knowing these properties makes the Java interpreter much faster, because it doesn't have to check anything. There are no operand type checks and no stack overflow checks. The interpreter can thus function at full speed without compromising reliability.

## 6.3 Security Checks in the Bytecode Loader

While a Java program is executing, it may in its turn request that a particular class or set of classes be loaded, possibly from across the network. After incoming code has been vetted and determined clean by the bytecode verifier, the next line of defense is the Java bytecode loader. The environment seen by a thread of execution running Java bytecodes can be visualized as a set of classes partitioned into separate *name spaces*. There is one name space for classes that come from the local file system, and a separate name space for each network source.

When a class is imported from across the network it is placed into the private name space associated with its origin. When a class references another class, it is first looked for in the name space for the local system (built-in classes), then in the name space of the referencing class. There is no way that an imported class can "spoof" a built-in class. Built-in classes can never accidentally reference classes in imported name spaces—they can only reference such classes explicitly. Similarly, classes imported from different places are separated from each other.

## 6.4 Security in the Java Networking Package

Java's networking package provides the interfaces to handle the various network protocols (FTP, HTTP, Telnet, and so on). This is your front line of defense at the network interface level. The networking package can be set up with configurable levels of paranoia. You can

- Disallow all network accesses
- Allow network accesses to only the hosts from which the code was imported

- Allow network accesses only outside the firewall if the code came from outside
- Allow all network accesses

## 6.5 Summary

Java is *secure* to survive in the network-based environment. The *architecture-neutral* and *portable* aspects of the Java language make it the ideal development language to meet the challenges of distributing *dynamically extensible* software across networks.

# *Multithreading in Java* 7

Sophisticated computer users become impatient with the do-one-thing-at-a-time mindset of the average personal computer. Users perceive that their world is full of multiple events all happening at once, and they like to have their computers work the same way.

Unfortunately, writing programs that deal with many things happening at once can be *much* more difficult than writing in the conventional single-threaded C and C++ style. You *can* write multithreaded applications in languages such as C and C++, but the level of difficulty goes up by orders of magnitude, and even then there are no assurances that vendors' libraries are thread-safe.

The term *thread-safe* means that a given library function is implemented in such a manner that it can be executed by multiple concurrent threads of execution.

The major problem with explicitly programmed thread support is that you can never be quite sure you have acquired the locks you need and released them again at the right time. If you return from a method prematurely, for instance, or if an exception is raised, for another instance, your lock has not been released; deadlock is the usual result.

## *7.1 Threads at the Java Language Level*

Built-in support for *threads* provides Java programmers with a powerful tool to improve interactive performance of graphical applications. If your application needs to run animations and play music while scrolling the page and downloading a text file from a server, *multithreading* is the way to obtain fast, lightweight concurrency within a single process space. Threads are sometimes also called lightweight processes or execution contexts.

Threads are an essential keystone of Java. The Java library provides a `Thread` class that supports a rich collection of methods to start a thread, run a thread, stop a thread, and check on a thread's status.

Java thread support includes a sophisticated set of *synchronization primitives* based on the widely used *monitor* and *condition variable* paradigm introduced twenty years ago by C.A.R. Hoare and implemented in a production setting in Xerox PARC's Cedar/Mesa system. Integrating support for threads into the language makes them much easier to use and more robust. Much of the style of Java's integration of threads was modelled after Cedar and Mesa.

Java's threads are *pre-emptive*, and depending on platform on which the Java interpreter executes, threads can also be *time-sliced*. On systems that don't support time-slicing, once a thread has started, the only way it will relinquish control of the processor is if another thread of a higher priority takes control of the processor. If your applications are likely to be compute-intensive, you might consider how to give up control periodically by using the `yield()` method to give other threads a chance to run; doing so will ensure better interactive response for graphical applications.

## 7.2 Integrated Thread Synchronization

Java supports multithreading at the language (syntactic) level and via support from its run-time system and thread objects. At the language level, methods within a class that are declared `synchronized` do not run concurrently. Such methods run under control of *monitors* to ensure that variables remain in a consistent state. Every class and instantiated object has its own monitor that comes into play if required.

Here are a couple of code fragments from the sorting demonstration in the HotJava web browser. The main points of interest are the two methods `stop` and `startSort`, which share a common variable called `kicker` (it kicks off the sort thread):

```
public synchronized void stop() {
    if (kicker != null) {
        kicker.stop();
```

```
        kicker = null;
    }
}
private synchronized void startSort() {
    if (kicker == null || !kicker.isAlive()) {
        kicker = new Thread(this);
        kicker.start();
    }
}
```

The `stop` and `startSort` methods are declared to be `synchronized`—they can't run concurrently, enabling them to maintain consistent state in the shared `kicker` variable. When a `synchronized` method is entered, it acquires a monitor on the current object. The monitor precludes any other `synchronized` methods in that object from running. When a `synchronized` method returns by any means, its monitor is released. Other `synchronized` methods within the same object are now free to run.

If you're writing Java applications, you should take care to implement your classes and methods so they're thread-safe, in the same way that Java run-time libraries are thread-safe. If you wish your objects to be thread-safe, any methods that may change the values of instance variables should be declared `synchronized`. This ensures that only one method can change the state of an object at any time. Java monitors are *re-entrant*: a method can acquire the same monitor more than once, and everything will still work.

## 7.3 Multithreading Support—Conclusion

While other systems have provided facilities for multithreading (usually via "lightweight process" libraries), building multithreading support into the language as Java has done provides the programmer with a much more powerful tool for easily creating thread-safe multithreaded classes.

Other benefits of multithreading are better interactive responsiveness and real-time behavior. Stand-alone Java run-time environments exhibit good real-time behavior. Java environments running on top of popular operating systems provide the real-time responsiveness available from the underlying platform.

# Performance and Comparisons 8

This chapter addresses two issues of interest to prospective adopters of Java, namely, what is the *performance* of Java, and how does it stack up against other comparable programming languages? Let's first address the performance question and then move on to a brief comparison with other languages.

## 8.1 Performance

Java has been ported to and run on a variety of hardware platforms executing a variety of operating system software. Test measurement of some simple Java programs on current high-end computer systems such as workstations and high-performance personal computers show results roughly as follows:

| | |
|---|---|
| `new Object` | 119,000 per second |
| `new C()` (class with several methods) | 89,000 per second |
| `o.f()` (method `f` invoked on object `o`) | 590,000 per second |
| `o.sf()` (synchronized method `f` invoked on object `o`) | 61,500 per second |

Thus, we see that creating a new object requires approximately 8.4 µsec, creating a new class containing several methods consumes about 11 µsec, and invoking a method on an object requires roughly 1.7 µsec.

While these performance numbers for interpreted bytecodes are usually more than adequate to run interactive graphical end-user applications, situations may arise where higher performance is required. In such cases, the Java bytecodes can be translated on the fly (at run time) into machine code for the particular CPU on which the application is executing. For those accustomed to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader.

The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple. Reasonably good code is produced: it does automatic register allocation and the compiler does some optimization when it produces the bytecodes. Performance of bytecodes converted to machine code is roughly the same as native C or C++.

## 8.2 The Java Language Compared

There are literally hundreds of programming languages available for developers to write programs to solve problems in specific areas. Programming languages cover a spectrum ranging across fully interpreted languages such as UNIX Shells, awk, TCL, Perl, and so on, all the way to "programming to the bare metal" languages like C and C++.

Languages at the level of the Shells and TCL, for example, are fully interpreted high-level languages. They deal with "objects" (in the sense they can be said to deal with objects at all) at the system level, where their objects are files and processes rather than data structures. Some of these languages are suitable for very fast prototyping—you can develop your ideas quickly, try out new approaches, and discard non-working approaches without investing enormous amounts of time in the process. Scripting languages are also highly portable. Their primary drawback is performance; they are generally much slower than either native machine code or interpreted bytecodes. This tradeoff may well be reasonable if the run time of such a program is reasonably short and you use the program infrequently.

In the intermediate ground come languages like Perl, that share many characteristics in common with Java. Perl's ongoing evolution has led to the adoption of object-oriented features, security features, and it exhibits many features in common with Java, such as robustness, dynamic behavior, architecture neutrality, and so on.

At the lowest level are compiled languages such as C and C++, in which you can develop large-scale programming projects that will deliver high performance. The high performance comes at a cost, however. Drawbacks include the high cost of debugging unreliable memory management systems and the use of multithreading capabilities that are difficult to implement and use. And of course when you use C++, you have the perennial fragile superclass issue. Last but definitely not least, the binary distribution problem of compiled code becomes unmanageable in the context of heterogeneous platforms all over the Internet.

The Java language environment creates an extremely attractive middle ground between very high-level and portable but slow scripting languages and very low level and fast but non-portable and unreliable compiled languages. The Java language fits somewhere in the middle of this space. In addition to being extremely simple to program, highly portable and architecture neutral, the Java language provides a level of performance that's entirely adequate for all but the most compute-intensive applications.

Prospective adopters of the Java language need to examine where the Java language fits into the firmament of other languages. Here is a basic comparison chart illustrating the attributes of the Java language—simple, object-oriented, threaded, and so on—as described in the earlier parts of this paper.

● *Feature exists*

◑ *Feature somewhat exists*

○ *Feature doesn't exist*

| | *Java* | *SmallTalk* | *TCL* | *Perl* | *Shells* | *C* | *C++* |
|---|---|---|---|---|---|---|---|
| *Simple* | ● | ● | ● | ◑ | ◑ | ◑ | ○ |
| *Object Oriented* | ● | ● | ○ | ● | ○ | ○ | ◑ |
| *Robust* | ● | ● | ● | ● | ● | ○ | ○ |
| *Secure* | ● | ◑ | ◑ | ● | ◑ | ○ | ○ |
| *Interpreted* | ● | ● | ● | ● | ● | ○ | ○ |
| *Dynamic* | ● | ● | ● | ● | ◑ | ○ | ○ |
| *Portable* | ● | ◑ | ● | ● | ◑ | ◑ | ◑ |
| *Neutral* | ● | ◑ | ◑ | ● | ◑ | ○ | ○ |
| *Threads* | ● | ○ | ○ | ● | ○ | ○ | ○ |
| *Garbage Collection* | ● | ● | ○ | ○ | ○ | ○ | ○ |
| *Exceptions* | ● | ● | ○ | ● | ○ | ○ | ◑ |
| *Performance* | *High* | *Medium* | *Low* | *Medium* | *Low* | *High* | *High* |

From the diagram above, you see that the Java language has a wealth of attributes that can be highly beneficial to a wide variety of developers. You can see that Java, Perl, and SmallTalk are comparable programming environments offering the richest set of capabilities for software application developers.

## 8.3 A Major Benefit of Java: Fast and Fearless Prototyping

Very dynamic languages like Lisp, TCL, and SmallTalk are often used for *prototyping*. One of the reasons for their success at this is that they are very robust—you don't have to worry about freeing or corrupting memory.

Similarly, programmers can be relatively fearless about dealing with memory when programming in Java. The garbage collection system makes the programmer's job vastly easier; with the burden of memory management taken off the programmer's shoulders, storage allocation errors go away.

Another reason commonly given that languages like Lisp, TCL, and SmallTalk are good for prototyping is that they don't require you to pin down decisions early on—these languages are semantically rich.

Java has exactly the opposite property: it forces you to make explicit choices. Along with these choices come a lot of assistance—you can write method invocations and, if you get something wrong, you get told about it at compile time. You don't have to worry about method invocation error.

## 8.4 Summary

From the discussion above, you can see that the Java language provides *high performance*, while its *interpreted* nature makes it the ideal development platform for fast and fearless prototyping. From the previous chapters, you've seen that the Java language is extremely *simple* and *object oriented*. The language is *secure* to survive in the network-based environment. The *architecture-neutral* and *portable* aspects of the Java language make it the ideal development language to meet the challenges of distributing *dynamically extensible* software across networks.

### *Now We Move On to the HotJava World-Wide Web Browser*

These first eight chapters have been your introduction to the Java language environment. You've learned about the capabilities of Java and its clear benefits to develop software for the distributed world. Now it's time to move on to the next chapter and take a look at the HotJava™ World-Wide Web browser—a major end-user application developed to make use of the dynamic features of the Java language environment.

# The HotJava World-Wide Web Browser 9

It’s a *jungle* out there,
So drink your **Java**

T-shirt caption from *Printer’s Inc Cafe*, Palo Alto, California

The **HotJava**™ Browser (“HotJava”) is a new World-Wide Web browser implemented entirely in the Java programming language. HotJava is the first major end-user application created using the Java programming language and its run-time system as a base. HotJava not only showcases the powerful features of the Java environment, it also provides an ideal platform for distributing Java programs across the Internet—the most complex, distributed, heterogeneous network in the world.HotJava and its rapidly growing Web population of Java language programs called *applets* (mini-applications), are the most compelling demonstration of the dynamic capabilities of Java.

HotJava includes many innovative features and capabilities above and beyond the first generation of static Web browsers. HotJava is *extensible*. Its foremost feature is its ability to download Java programs (applets) from anywhere, even across networks, and execute them on the user’s machine. HotJava builds on the network-browsing techniques established by Mosaic and other Web browsers and expands them by adding dynamic behavior that transforms static documents into dynamic applications.

HotJava goes far beyond the current generation of statically-oriented Web browsers and brings a much needed measure of *interactivity* to the concept of the Web browser. It transforms the existing static data display of current generation Web browsers into a new and dynamic viewing system for hypertext, described below. It enables creation and display of animation-oriented applications. World-Wide Web content developers can have their applications distributed across the Internet with the click of a button on the user's client computer.

## 9.1 The Evolution of Cyberspace

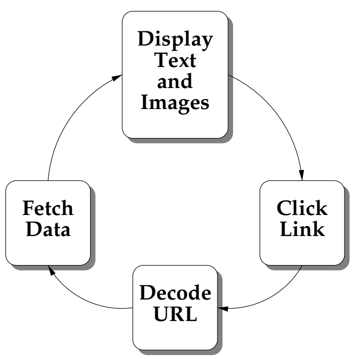
The *Internet* has evolved into an amorphous ocean of data stored in many formats on a multiplicity of network hosts. Over time, various data storage and transmission protocols have evolved to impose some order on this chaos. One of the fastest growing areas of the net—the one we're primarily interested in here—is the World-Wide Web (WWW), which uses a hypertext-based markup system to enable users to navigate their way across the oceans of data.

The concept of *hypertext* is by no means new, but its realization has spanned decades. The idea behind hypertext was described in an essay by Vannevar Bush in 1945, and evangelized by Theodore (Ted) Nelson in the 1960s and 1970s. Although Apple Computer's HyperCard product for Macintosh provided an early if somewhat primitive implementation, the real power of hypertest comes from the ability to create inter-document links across multiple host computers on the network. The first practical if small implementation of a network-based hypertext system was created by Tim Berners-Lee at CERN, using the NEXTSTEP development environment to create what would blossom into HTML (HyperText Markup Language), HTTP (HyperText Transport Protocol), and the WWW (World-Wide Web, or W3).

Web browsers combine the functions of fetching data with figuring out what the data is and displaying it if possible. One of the most prevalent file formats browsers deal with is HyperText Markup Language, or HTML— a markup language that embeds simple text-formatting commands within text to be formatted. The main key to the hypertext concept is HTML's use of *links* to other HTML pages either on the same host or elsewhere on the Internet.

A user in search of gold mining data, for instance, can follow links across the net from Mountain View, California, to the University of the Witwatersrand, South Africa, and arrive back at commercial data providers in Montreal, Canada, all within the context of tracing links in hypertext "pages". For a topic of timely relevance to the World-Wide Web, a user interested in aspects of multimedia law relative to the World-Wide Web can tune in to the home page at `www.oikoumene.com/oikoumene` for links to intellectual property issues.

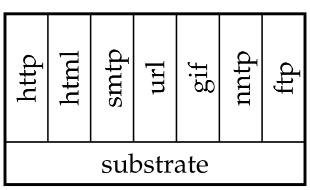
### 9.1.1 First Generation Browsers




What we could call the "first-generation" Web browsers—exemplified by NCSA Mosaic and the first release of Netscape Navigator—provide an illusion of being interactive. By using the (somewhat limited) language of HTML these browsers provide hypertext *links* on which you can click. The browser goes off across the network to fetch the data associated with that link, downloads the data, and displays it on your local screen. As we said, this is an illusion of interactivity.

This illustration depicts roughly the "interactive" flow of control in the first-generation Web browsers. As you see, it's not really interactive—it's just a fancy data fetching and display utility.

HotJava brings a new twist to the concept of client-server computing. The general view of client-server computing is a big centralized server that clients connect to for a long time and from which they access data and applications. It is roughly a star with a big server in the middle and clients arrayed around it. The new model exemplified by the World-Wide Web is a wide-spread collection of independent nodes with short -lived connections between clients and many servers. The controlling intelligence shifts from the server to the client and the answer to "who's in charge?" shifts from the server to the client.




A conventional browser: a monolithic chunk, all bound tightly together.

The primary problem with the first-generation web browsers is that they're built in a monolithic fashion with their awareness of every possible type of data, protocol, and behavior hard wired in order for them to navigate the Web. This means that every time a new data type, protocol, or behavior is invented, these browsers must be upgraded to be cognizant of the new situation. From the viewpoint of end users, this is an untenable position to be in. Users must continually be aware of what protocols exist, which browsers deal with those protocols, and which versions of which browsers are compatible with each other. Given the growth of the Internet, this situation is clearly out of control.

### 9.1.2 The HotJava Browser—A New Concept in Web Browsers

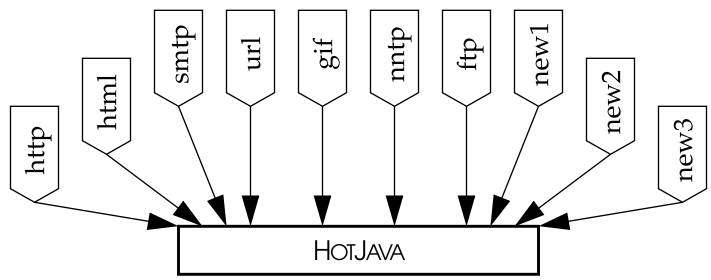
HotJava solves the monolithic approach and moves the focus of interactivity away from the Web server and onto the Web *client*—that is, to the computer on which the user is browsing the Web. Because of its basis in the Java system, a HotJava client can dynamically download segments of code that are executed right there on the client machine. Such Java-based "applets" (mini-applications) can provide full animation, play sound, and generally interact with the user in real time.

HotJava removes the static limitations of the Mosaic generation of Web browsers with its ability to add arbitrary behavior to the browser. Using HotJava, you can add applications that range from interactive science experiments in educational material, to games and specialized shopping applications. You can implement interactive advertising, customized newspapers, and a host of application areas that haven't even been thought of yet. The capabilities of a Web browser whose behavior can be dynamically updated are open-ended.

Furthermore, HotJava provides the means for users to access these applications in a new way. Software migrates transparently across the network as it's needed. You don't have to "install" software—it comes across the network as you need it—perhaps after asking you to pay for it. Content developers for the World-Wide Web don't have to worry about whether or not some special piece of software is installed in a user's system—it just gets there automatically. This transparent acquiring of applications frees content developers from the boundaries of the fixed media types such as images and text and lets them do whatever they'd like.

### 9.1.3 The Essential Difference

The central difference between HotJava and other browsers is that while these other browsers have knowledge of the Internet protocols hard-wired into them, HotJava understands essentially none of them. What it *does* understand is how to find out about things it doesn't understand. The result of this lack of understanding is great flexibility and the ability to add new capabilities very easily.

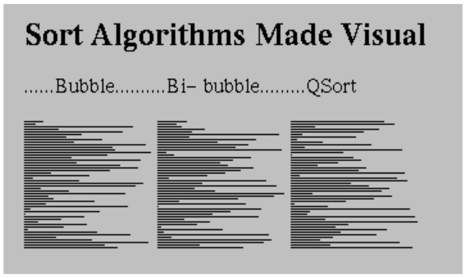



HotJava is the coordinator of a federation of pieces, each with individual responsibility. New pieces can be added at any time. Pieces can be added from across the network, without needing to be concerned with what CPU architecture they were designed for and with reasonable confidence that they won't compromise the integrity of a user's system.

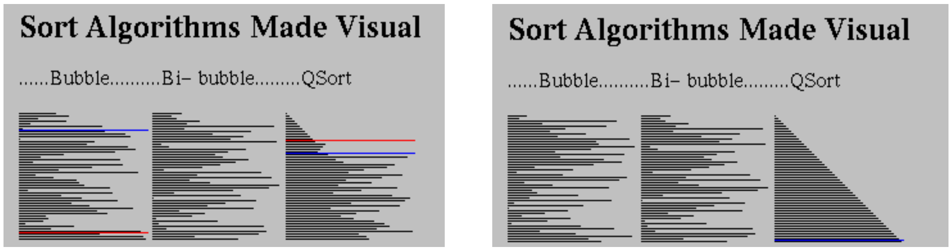
### 9.1.4 Dynamic Content

One of the most visible uses of HotJava's ability to dynamically add to its capabilities is something we call *dynamic content*. For example, someone could write a Java program to implement an interactive chemistry simulation, following the rules of the HotJava API. People browsing the net with HotJava could easily get this simulation and interact with it, rather than just having a static picture with some text. They can do this and be assured that the code that brings their chemistry experiment to life doesn't also contain malicious code that damages the system. Code that attempts to be malicious or which has bugs essentially can't breach the walls placed around it by the security and robustness features of Java.

For example, the following is a snapshot of HotJava in use. Each diagram in the document represents a different sort algorithm. Each algorithm sorts an array of integers. Each horizontal line represents an integer: the length of the line corresponds to the value of the integer and the position of the line in the diagram corresponds to the position of the integer in the array.




In a book or HTML document, the author has to be content with these static illustrations. With HotJava the author can enable the reader to click on the illustrations and see the algorithms animate:

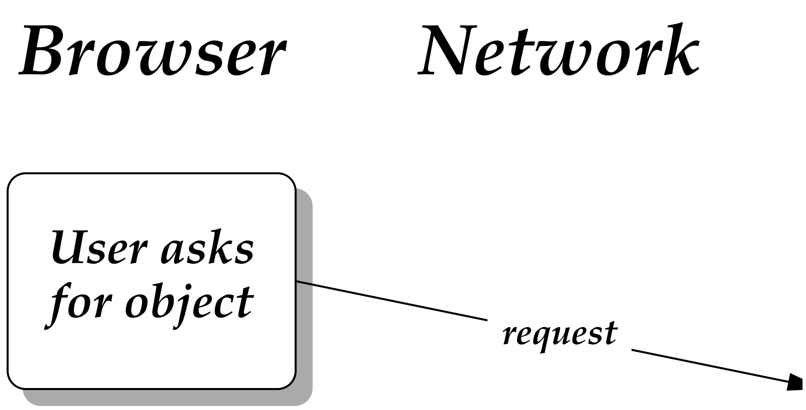



Using these dynamic facilities, content providers can define new types of data and behavior that meet the needs of their specific audiences, rather than being bound by a fixed set of objects.

### 9.1.5 Dynamic Types

HotJava's dynamic behavior is also used for understanding different types of objects. For example, most Web browsers can understand a small set of image formats (typically GIF, X11 pixmap, and X11 bitmap). If they see some other type, they have no way to deal with it. HotJava, on the other hand, can dynamically link the code from the host that has the image allowing it to display the new format. So, if someone invents a new compression algorithm, the inventor just has to make sure that a copy of its Java code is installed on the server that contains the images they want to publish; they don't have to upgrade all the browsers in the world. HotJava essentially upgrades itself on the fly when it sees this new type.

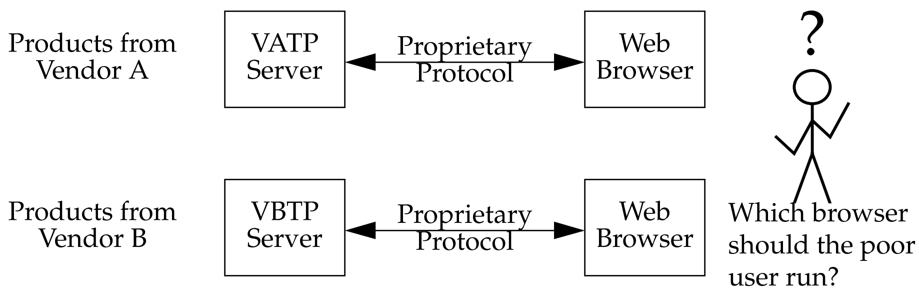
The following is an illustration of how HotJava negotiates with a server when it encounters an object of an unknown type:



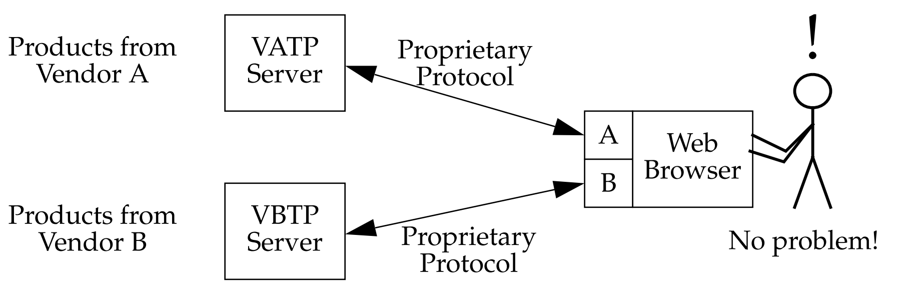

### 9.1.6 Dynamic Protocols

The protocols that Internet hosts use to communicate among themselves are key components of the net. For the World-Wide Web (WWW), HTTP (*HyperText Transfer Protocol*) is the most important of these communication protocols. Within WWW documents, a reference to another document (even to a document on another Internet host computer) is called a URL, meaning a *Uniform Resource Locator*. The URL contains the name of the protocol, HTTP, that is used to find that document. Most of the current generation of Web browsers have the knowledge of HTTP built-in. Rather than having built-in protocol handlers, HotJava uses the protocol name to link in the appropriate handler as required, allowing new protocols to be incorporated dynamically.

The dynamic incorporation of protocols has special significance to how business is done on the Internet. Many vendors are providing new Web browsers and servers with added capabilities, such as billing and security. These capabilities most often take the form of new protocols. So each vendor comes up with their unique style of security (for example) and sells a server and browser that speak this new protocol. If a user wants to access data on multiple servers on which each has proprietary new protocols, the user needs multiple browsers. This is incredibly clumsy and defeats the synergistic cooperation that makes the World-Wide Web work.



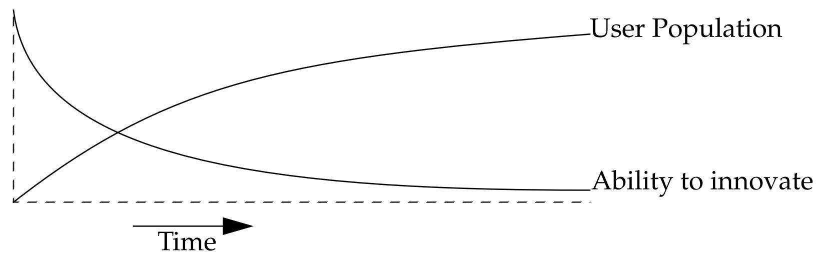

With HotJava as a base, vendors can produce and sell exactly the piece that is their added value, and integrate smoothly with other vendors, creating a final result that is seamless and very convenient for the end user.




Protocol handlers get installed in a sequence similar to how content handlers get installed: The HotJava Browser is given a reference to an object (a URL). If the handler for that protocol is already loaded, it will be used. If not, the HotJava Browser will search first the local system and then the system that is the target of the URL.

## 9.2 Freedom to Innovate

Innovation on the Internet follows a pattern: initially: someone develops a technology. They're free to try all kinds of things since no one else is using the technology and there are no compatibility issues. Slowly, people start using it, and as they do, compatibility and interoperability concerns slow the pace of innovation. The Internet is now in a state where even simple changes that everyone agrees will have significant merit are very hard to make.




Within a community that uses HotJava, individuals can experiment with new facilities while at the same time preserving compatibility and interoperability. Data can be published in new formats and distributed using new protocols and the implementations of these will be automatically and safely installed. There is no upgrade problem.

One need not be inventing new things to need these facilities. Almost all organizations need to be able to adapt to changing requirements. TheHotJava browser's flexibility can greatly aid that. As new protocols and data types become important, they can be transparently incorporated.

## 9.3 Implementation Details

The basic structure of HotJava is instructive. It is easiest understood from the operation of Mosaic.

Mosaic starts with a URL and fetches the object referenced by that URL using the specified protocol. The *host* and *localinfo* fields are passed to the protocol handler. The result of this is a bag of bytes that contains the object that has been fetched. These bytes are inspected to determine the type of the data (HTML document or JPEG image, for example). From this type information, code is invoked to manipulate and view the data.

That's all there is to Mosaic. It's essentially very simple. But despite this, the Mosaic program is actually huge since it must contain specialized handlers for all of these data types. It's bundled together into one big monolithic lump.

In contrast, HotJava is very small, since all of the protocol and data handlers are brought in from the outside. For example, when it calls the protocol handler, instead of having a table that has a fixed list of protocols that it understands, HotJava instead uses this type string to derive a Java language class name. The protocol handler for this type is dynamically linked in if it is missing. They can be linked in from the local system, or they can be linked in from definitions stored on the host where the URL was found, or anywhere else on the net that HotJava suspects might be a good place to look. In a similar fashion, HotJava can dynamically locate and load the code to handle different types of data objects and different ways of viewing them.

## 9.4 Security

*Network security* is of paramount importance to Internet users, especially with the exponential growth of Internet commerce. Network-based applications must be able to defend themselves against a veritable gallimaufry of network viruses, worms, Trojan horses, and other forms of intruders. This section discusses the layers of defense provided by Java, the Java run-time system, and the higher-level protocols of HotJava itself.

One of the most important technical challenges in building a system like HotJava is making it secure. Downloading, installing,and executing fragments of code importedfrom across the network is potentially an open invitation to all sorts of problems. On the one hand, such a facility provides great power that can be used to achieve very valuable ends; on the other hand, the facility could potentially be subverted to become a breeding ground for computer viruses. The topic of safety is a very broad one and doesn't have a single answer. HotJava has a series of facilities that layer and interlock to provide a fairly high degree of safety.

### 9.4.1 The First Layer—the Java Language Interpreter

The first layer of security in Java applications come from the ground rules of Java itself. These features have been described in detail in previous chapters in this paper.

When HotJava imports a code fragment, itdoesn't actually know whether or not the code fragment follows Java language rules for safety. As described earlier, imported code fragments are subjected to a series of checks, starting with straightforward tests that the format of the code is correct and ending with a series of consistency checks by the Bytecode Verifier.

### 9.4.2 The Next Layer—the Higher Level Protocols

Given this base set of guarantees that interfaces cannot be violated, higher level parts of the system implement their own protection mechanisms. For example, the file access primitives implement an access control list that controls read and write access to files by imported code (or code invoked by imported code). The defaults for these access control lists are very restrictive. If an attempt is made by a piece of imported code to access a file to which access has not been granted, a dialog box pops up to allow the user to decide whether or not to allow that specific access.

These access restrictions err on the conservative side, which makes constructing some very useful extensions impossible or awkward. We have a mechanism whereby public keys can be securely attached to code fragments that allows code with trusted public keys to have fewer restrictions. This mechanism isn't in the public release for legal reasons.

## 9.5 HotJava—the Promise

The HotJava Web browser, based upon the foundations of the Java environment, brings a hitherto unrealized *dynamic* and *interactive* capability to the World-Wide Web. Dynamic *content*, dynamic *data types*, and dynamic *protocols* provide content creators with an entirely new tool that facilitates the burgeoning growth of electronic commerce and education.

The advent of the dynamic and interactive capabilities provided by the HotJava Web browser brings the World-Wide Web to life, turning the Web into a new and powerful business and communication tool for all users.

# Further Reading 10

I've got a little list.
I've got a little list.

Gilbert and Sullivan—*The Mikado*

***The Java Programmer's Guide***

Sun Microsystems

`http://java.sun.com/progGuide/index.html`

This is the draft version of the Java/HotJava Programmer's Guide.

***Pitfalls of Object-Oriented Development***, by Bruce F. Webster
Published by M&T Books.

A collection of "traps to avoid" for people adopting object technology. Recommended reading—it alerts you to the problems you're likely to encounter and the solutions for them.

***The Design and Evolution of C++***, by Bjarne Stroustrop
Published by Addison Wesley

A detailed history of how we came to be where we are with C++.

***NEXTSTEP Object-Oriented Programming and the Objective C Language***.
Addison Wesley Publishing Company, Reading, Massachusetts, 1993.

The book on Objective C. A good introduction to object-oriented programming concepts.

***Discovering Smalltalk***. By Wilf Lalonde.
Benjamin Cummings, Redwood City, California, 1994.

An introduction to Smalltalk.

***Eiffel: The Language***. By Bertrand Meyer.
Prentice-Hall, New York, 1992.

An introduction to the Eiffel language, written by its creator.

***An Introduction to Object-Oriented Programming***. By Timothy Budd.
Addison Wesley Publishing Company, Reading, Massachusetts.

An introduction to the topic of object-oriented programming, as well as a comparison of C++, Objective C, SmallTalk, and Object Pascal.

***Monitors: An Operating System Structuring Concept***. By C. A. R. Hoare.
Communications of the ACM, volume 17 number 10, 1974. Pages 549-557.

The original seminal paper on the concept of monitors as a means to synchronizing multiple concurrent tasks.

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000