The Java Language Environment

A White Paper

Sun

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
415 960-1300 FAX 415 969-9131

May 1995

The Java Language Environment

A White Paper

James Gosling
Henry McGilton

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
415 960-1300 FAX 415 969-9131

May 1995

Contents

Introduction 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 C or C++ software application developer. 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 architecture supporting multiple incompatible operating systems, and each platform operating with one or more incompatible graphical user interfaces. And now you're supposed to cope with all this and make the 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 work with to develop applications don't appear to be helping you a whole lot. You're still coping with the same old problems, and the use of object-oriented techniques seem to have added new problems without solving the old ones. You keep saying to yourself and your friends, "There *has* to be a better way!"

There Has To Be A Better Way

Now there *is* a better way—it's the **Java** programming language environment 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 the Java language is *interpreted*. The old compile-link-load-test-and-crash cycle has gone the way of the zumbooruk[*]—now you just compile and run. When you're satisfied with your application, you can obtain maximum performance by using the built in *just-in-time compiler* to compile the Java intermediate code to native machine code.
- 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—you don't have dangling pointers and memory leaks and trashing of memory because of bad pointers, because there are no pointers.
- Your interactive graphical applications have *high performance* because multiple concurrent threads of activity in your application are supported by the *multithreading* built into the language.
- Your applications are *dynamically adaptable* to changing environments because you can dynamically load code modules from anywhere in the network.
- Your end users can trust that your applications are *secure*, even though they're downloading code from all over the Internet, because the Java run-time system has built-in protection against viruses and tampering.

[*] A small cannon designed to be mounted on and fired from the back of a camel—implies obsolescence.

The Better Way Is Here

You don’t need to dream about these features to make developer life easier. They’re here now in the form of the Java Programming Environment, otherwise known as “the Java language”—a *portable*, *interpreted*, *high performance*, *simple*, *object-oriented* programming language and supporting run-time environment developed at Sun Microsystems. The rest of this introductory chapter takes a brief look at the main design goals of the Java language system. The remainder of this paper examines the features of the Java language in more detail. At the end of this paper you’ll find a chapter on the *HotJava browser*—an innovative World-Wide web browser, a major application written using the Java language environment. The HotJava browser is unique in its ability to download and execute Java code fragments on the fly from anywhere on the Internet, and do so in a secure manner.

Beginnings of the Java Language Project

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

The Java language 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 was 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 was a language environment that has proven ideal for development of secure, distributed, network-based end-user applications in environments ranging from networked embedded devices to the World-Wide Web and the desktop.

Design Goals of the Java Language

The design requirements of Java can be summed up as a set of characteristics—to *live*, to *survive*, and to *flourish*:

- 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, the Java language must support *secure, high-performance,* and highly *robust* application development 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, the Java language must be *architecture-neutral*, *portable*, and *dynamically adaptable*.
- To ensure you can *flourish* within your software development environment, the Java language 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 the Java language, *object oriented*, to fit into distributed client-server applications, *multithreaded*, for high performance in applications that need to perform multiple concurrent activities, and *interpreted*, for maximum portability and dynamic capabilities.

All in all, the above requirements comprise quite a collection of buzzwords, so let's briefly examine some of the features and their respective benefits before getting into details in the rest of this paper.

Simple, Object Oriented, and Familiar

A primary goal of the Java language was a *simple* language that could be programmed without extensive programmer training and which would be roughly attuned to current software practice. The fundamental concepts of the Java language can be grasped quickly—programmers can be productive from the start.

The Java language was designed as an *object-oriented* language from the ground up. Object-oriented 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 packaged, message-passing paradigms of object-based software. To function within increasingly complex,

[*] 15 percent per month by some estimates.

network-based environments, programming systems must adopt object-oriented concepts. The Java language provides a clean and efficient object-based development environment.

Programmers starting with the Java language have access to existing libraries of tested objects that provide functionality ranging from basic data types through I/O interfaces and network interfaces to graphical user interface toolkits. Many of these libraries can be extended (subclassed) to provide new behavior.

Even though C++ was rejected as an implementation language, keeping the language *familiar* resulted in the Java language looking as close to C++ as possible, while removing the unnecessary complexities of C++. Making the Java language a minimal subset of C++ while simultaneously retaining its “look and feel” means that programmers can migrate to the Java language easily and be productive quickly—the Java language learning curve can be as low as a couple of days.

Robust and Secure

The Java language is designed for creating highly *reliable* software. Emphasis is on extensive compile-time checking, and a second level of run-time checking. Java language features guide programmers into reliable programming habits. The Java memory management model—basically, no pointers or pointer arithmetic—eliminates entire classes of programming errors that bedevil C and C++ programmers. You can develop Java code with confidence that the system will find errors quickly and that major problems won’t slip out into production code.

The Java language was designed to operate in distributed environments. With *security* features designed into the language and run-time system, the Java language enables construction of tamper-free programs. In the networked environment, Java programs are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

Architecture Neutral and Portable

The Java language was designed to support applications operating in networked environments, operating on a variety of hardware architectures, and running a variety of operating systems and language environments. The Java language compiler generates byte codes—an architecture-neutral intermediate format used to transport code efficiently to multiple hardware and software platforms. Because of the interpreted nature of the Java language, you don't have a binary distribution and versioning problem—the same Java language byte codes will run on any platform.

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

The Java virtual machine is based on a well-defined porting layer, primarily based on the POSIX interface standard—an industry standard definition of a portable system interface. Porting to new architectures is a relatively straightforward task.

High Performance

Performance is always a consideration, and the Java language 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 language environment. In general, users perceive that interactive applications perform adequately even though they're interpreted. When you need the full speed of the host hardware for compute-intensive applications, the Java language system provides an on-the-fly "just in time" compiler that will compile the Java language byte codes to machine code at run time. This innovative feature provides native code performance levels without compromising the portability of applications.

Interpreted, Threaded, and Dynamic

The Java *interpreter* can execute Java byte codes directly on any machine to which the interpreter and run-time system have been ported. In an interpreted environment such as the Java language system, the "link" phase of a program is simple, incremental, and lightweight. You gain a major benefit of much faster development cycles—prototyping, experimenting, and rapid development are the normal case.

Modern network-based applications such as the HotJava World-Wide Web browser typically need to do several things at the same time. A user operating the HotJava browser can run several animations concurrently while downloading an image and scrolling the page. The ability to perform multiple tasks concurrently is provided by the Java language's *multithreading* capability, which 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.

The Java language supports multithreading at the language level with the addition of sophisticated synchronization primitives, at the language library level with the `Thread` class, and at the run-time level with monitor and condition lock primitives. At the library level, the Java language's high-level system libraries have been written to be *thread-safe*—the functionality provided by the libraries is available 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 as needed. New code modules can be linked in on demand from a variety of sources, even across a network. In the case of the HotJava web browser and similar applications, interactive executable code can be loaded from anywhere, enabling transparent updating of applications. The result is on-line services that constantly evolve, remaining innovative and fresh, drawing more customers, and spurring the growth of electronic commerce on the Internet.

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 the Java language and run-time system have combined them to produce an flexible and powerful programming system.

The Java language environment results in software applications that are *portable*, highly *secure*, and *high performance*. With the Java language to develop your software, 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 the Java environment.

Simple, Object Oriented, 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 *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 the C language over the course of its long lifetime. This chapter discusses the three primary design features of the Java language, namely, *simple*, *object-oriented*, and *familiar*. The end of this chapter contains a discussion on those features that were eliminated from C and C++ in the evolution towards the Java language.

Simple

Based on the design methodology known as KISS[*], *simplicity* is one of the Java language's overriding design goals from a programming standpoint. Simplicity and removal (from C++-based languages) of many dubious "features" keep the Java language relatively small and reduce the programmer's burden in producing reliable applications. To this end, the Java language 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.

Object Oriented

To stay abreast of modern software development practices, the Java language was designed to be *object oriented* from the ground up. The benefits of object technology are slowly becoming realized as more organizations move their applications to the distributed client-server model. The Java language goes beyond C++ in both extending the object models and removing the major complexities of C++. With the exception of its primitive data types, everything in the Java language is an object.

Familiar

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

To illustrate the simple, object-oriented, and familiar aspects of the Java language, we follow the tradition of a long line of illustrious (and some not so 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` in the Java language.

* Keep It Small and Simple

† Now enjoying their silver anniversaries

```
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 section on object-oriented programming, but in general we assume the reader is familiar with object technology and understands the basic notions 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 invoking the `println` method of the `out` object, which is a class variable of the `System` class. That's all there is to `HelloWorld`.

2.1 Main Features of the Java Language

Primitive Data Types

Other than the primitive data types discussed here, everything in the Java language is an object. Even the primitive data types can be wrapped inside language-supplied objects as required. The Java language 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—numeric types, Boolean types, and arrays.

Integer numeric types are 8-bit `byte`, 16-bit `short`, 32-bit `int`, and 64-bit `long`. There are no `unsigned` types.

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.

Character data is a minor departure from traditional C. The Java language uses the Unicode character set standard, so the traditional C `char` data type was expanded from eight to sixteen bits. If you write a declaration such as

```
char  myChar = ‘Q’;
```

you get a Unicode (16-bit unsigned character) type that's initialized to the Unicode value of the character Q.

The *Boolean* is a primitive type in the Java language, tacitly ratifying existing C and C++ programming practice. A `boolean` variable assumes the values `true` or `false`. A Java `boolean` is a distinct data type—unlike common C practices, a `boolean` can not be converted to a numeric type by casting.

Arithmetic and Relational Operators

All the familiar C and C++ operators apply in the Java language. Because the Java language lacks `unsigned` data types, the Java language adds the **`>>>`** operator to mean an unsigned (logical) right shift. The Java language also uses the **`+`** operator for string concatenation, discussed below.

Arrays

Arrays in the Java language are first-class language objects. A Java language array is a real object with a run-time representation. You can declare and allocate arrays of any type. To obtain multi-dimensional arrays, you allocate arrays of arrays. You declare an array of, say, `Object` with a declaration like this:

```
Object  myArray[];
```

This code declares that `myArray` is an uninitialized array of `Object`. This variable has no storage as yet. At some future point you must allocate the amount of storage you need, say:

```
myArray = new Object[10];
```

to allocate an array of 10 `Object`s. To get the length of an array, add `.length` to the array name—`myArray.length` would return the number of elements in `myArray`. Access to elements of `myArray` can be performed via the normal C indexing, but all array accesses are checked to ensure that their indexes are within the range of the array. An exception will be generated if the index is outside the bounds of the array.

The C notion of a pointer to an array of memory elements has 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 Java array checking will help lead to more robust and reliable code.

Strings

Strings are objects, not arrays of characters as in C. String objects are instances of the String class. There are actually two kinds of String objects. The String class is for read-only (immutable) objects. The StringBuffer class is for mutable String objects.

Although Strings are objects, the Java language compiler "knows" about Strings as a special syntactic feature to accommodate the syntactic convenience that C programmers have enjoyed with C-style strings[*]. The Java language 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 the character string `"Hello world!"`.

The Java language 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.

String objects supply a `length` accessor method to obtain the number of characters in the string.

Multi Level Break

The Java language has no `goto` statement. To `break` or `continue` multiply nested loop or switch constructs, you use labels to label loop and switch constructs, and then `break` or `continue` to the label. Here's a small fragment of code from the Java language's built-in `String` class.

```
test:  for (int i = fromIndex; i + max1 <= max2; i++) {
    if (charAt(i) == c0) {
```

[*] Which are really just an array of characters.

```
        for (int k = 1; k<max1; k++)
            if (charAt(i+k) != str.charAt(k)) {
                 continue test;
```

The `continue` statement that leads to `test` is nested two levels deep inside `for` statements. This Java language feature leads to considerable simplification of programming time, and major reduction in future maintenance efforts.

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.

The Java language 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 the Java language and run-time system. 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.

The Java language's memory management model is based on *objects* and *references* to objects. Because the Java language has no pointers, all references to allocated storage—which in practice means references to objects—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. While the Java language has a `new` operator to allocate space for objects, there is no explicit `free` function.

The Java language'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.

The Background Garbage Collector

A common criticism of garbage collection schemes is that they tend to fire up at inappropriate times, creating poor interactive response for the user.

The Java language run-time system largely solves this problem by running the garbage collector in a low priority *thread*. This, by the way, is just one of many examples of the synergy one obtains from the Java language's integrated *multithreading* capabilities—an otherwise intractable problem is solved in a simple and elegant fashion.

The typical user of the typical interactive application has many natural pauses where the user is contemplating the scene in front of them or thinking of what to do next. The Java language run-time system can take advantage of these idle periods and run the garbage collector when no other threads are competing for CPU cycles The garbage collector gathers and compacts unused memory, thereby improving the probability that adequate memory resources are available when needed during periods of heavy interactive activity.

Integrated Thread Synchronization

The Java language 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 provides the programmer with a much more powerful tool for easily creating thread-safe multithreaded classes. Multithreading is discussed in more detail in a separate chapter.

2.2 Object Oriented

My Object All Sublime
I Will Achieve in Time

Gilbert and Sullivan—*The Mikado*

The Java language is *object oriented*. It draws on the best concepts and features of previous object-oriented languages, primarily Eiffel, SmallTalk, C++, and Objective-C.

Unfortunately, “object oriented” remains misunderstood, over-marketed, 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 that you can’t with procedural techniques.

Object Technology in the Java Language

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

- *Encapsulation*—to implement information hiding and modularity
- *Inheritance*—code re-use and code organization
- *Dynamic binding*—for maximum flexibility while a program is executing.

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

At its simplest, object technology is a collection of analysis, design, and programming methodologies that focus 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.

* Or not even a new way, depending on your time perspective.

Objects

What are objects? They're software programming *models* of objects of everyday life. In your everyday life, you're surrounded by objects—cars, coffee machines, ducks, trees, and so on. These objects have *state* and *behavior*. You can model everyday objects with software constructs called objects. Software objects can also be defined by their *state* and their *behavior*.

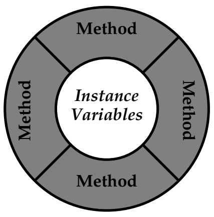
In your everyday transportation needs, a *car* can be modelled by an object. A car has *state* (how fast it's going, in which direction, its fuel consumption, and so on) and *behavior* (starts, stops, turns, slides, and runs into trees).

In your daily interactions with the financial markets, a *call option* can also be modelled by an object. A call option has *state* (current price, strike price, expiration date), and *behavior* (changes value, moves into and out of the money, or expires worthless[*]).

After your call options have expired worthless, you repair to the cafe for 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*).

An important characteristic that distinguishes objects from 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 means that objects can be created in one place, passed around networks, and stored elsewhere to be retrieved at later stages for future work.

The Basics of Objects



In the implementation of an object, its *state* is defined by its *instance variables*. Instance variables are variables local to the object. Unless explicitly made public or made available to other "friendly" classes, they are inaccessible from outside the object. An object's *behavior* is defined by its *methods*. Methods manipulate the instance variables to create new state, and can also create new objects. This picture 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.

* More often than not.

An object’s *instance variables* (data) are packaged, or encapsulated, with the object, hidden safely inside the nucleus, safe from access by other objects. 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—this topic is covered in later in *Access Specifiers*.

Classes

A *class* 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*. You can instantiate many objects from one class definition, just as you can construct many houses all the same from a single architect’s drawing. The only way you can get concrete objects is by instantiating a previously defined class. Here’s the declaration for a very simple class called Point

```
class Point {
    public float  x;
    public float  y;

    Point() {
        x = 0.0;
        y = 0.0;
    }
}
```

Methods with the same name as the class as in the code fragment above are called *constructors*. When you create (instantiate) an object of this 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.

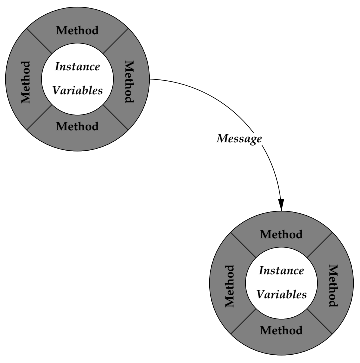
Any other object can now create an instance of Point with a line of code like this:

```
Point  myPoint;

myPoint = new Point();
```

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;.
```



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 must send a *message* to the other object. An object sends a message to another object by invoking one of its *methods*, which look similar to functions in C and C++.

In general, methods operate only on the instance variables of a specific *instance* of the object. However, an object can declare its instance variables to be `public`, in which case other objects can access the instance variables directly without going through method invocations.

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 complex simulations of real-world systems.

Constructors and Finalizers

When you declare a class in the Java language, you can declare optional *contructors* that perform initialization when you instantiate objects from that class, and you can declare an optional *finalizer* that will perform necessary tear down actions when the garbage collector is about to free the object. This minimal code fragment illustrates a *constructor*.

```
public final class Integer extends Number {
    private int value

    public Integer(int value){
        this.value = value;
    }
}
```

This example declares a class called `Integer`. There is in fact an `Integer` class in the Java language foundation—it's an object wrapper when you need to encapsulate an integer value inside an object. The public method `Integer` is the *constructor* for this class. You would access it in code via a declaration of the form

```
Integer  myIntegerObject = new Integer(123);
```

to create a new object of the Integer class initialized to the value 123.

By the way, what is the variable `this` in the above example? It is a special name that refers to this instance of the class—from inside an object, you need a way to refer to the object itself. That's what the `this` variable achieves.

This code fragment illustrates a finalize method in a class.

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

This `finalize` method will be invoked when the object is about to be garbage collected. In the particular code fragment, 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.

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*.

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, sub-classing gains you re-use of existing code—you save on design, development, and testing. The Java supporting run-time system provides several libraries of utility functions that are tested and are also *thread-safe*.

Access Control

When you declare a new class in the Java language, you can indicate the level of access permitted to its instance variables and methods. The Java language 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.

`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.

Static and Final Methods and Variables

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

```
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 you declare that operate only on instance variables of the class. 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`.

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*. This 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 can 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.

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, 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.

```
class Graphic {
    Point p;                      //  position
    float fillcolor;              //  gray shade of interior
    float linecolor;              //  gray shade of outline
    . . .;                        //  more instance variables
                                  //  more methods
    abstract void drawmyself()    //  do nothing method
}
```

Now, you can't instantiate the Graphic class. You can only instantiate a subclass of it. When you implement the Rectangle class, you'd subclass Graphic. Within Rectangle, you'd provide a concrete implementation of the `drawmyself` method that draws a rectangle. You can continue in this way adding new shapes that are subclasses of Graphic, and all you ever need to implement is the method to draw the shape. You gain the benefit of re-using all the code that was defined inside the abstract superclass.

Packages

Packages are a Java language construct that gather collections of related classes into a single container. For example, all the Java language I/O system code is collected into a single package. The primary benefit of packages is organizing many class definitions into a single compilation 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.

As an example, you could create a rectangle geometry package containing the definitions of a point class and a rectangle class, plus all the operations you'd want to perform on points and rectangles. Within the package, you could have rectangles accessing the "friendly" instance variables of the point class directly, thereby improving the overall performance of the objects. But other classes outside the package would be able to access the internal state of point objects and rectangle objects only by invoking the `public` methods of the classes.

2.3 Features Removed from C and C++

The previous sections concentrated on features of the Java language. This section discusses features removed from C and C++ in the design of the Java language. 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++, even in an attempt to add "classes in C"[*] merely added more redundancy while retaining the inherent problems of C.

[*] Or even just add some class to C.

Typedefs, Defines, and the Preprocessor

Source code written in the Java language 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 *interfaces* 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 `#define`s, and all related `typedef`s before you can even begin to analyze a program. In a sense, programming in this style essentially results in every programmer inventing a new programming language that's incomprehensible to anybody other than its creator and defeats the goals of good programming practices.

You can 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 language compiler compiles class definitions into a binary form that retains all the type information through to link time.

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

Structures and Unions

The Java language 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 using instance variables of a class. This code fragment declares Point and Rectangle classes. In C you'd define these as structures. In the Java language, you simply declare a class. 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.

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

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

Functions

The Java language 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. By eliminating functions, the Java language has immensely simplified your job as a programmer—you work *only* with classes and methods.

Multiple Inheritance and Interfaces

The Java language discards *multiple inheritance* and all the problems it generates. 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 methods only—in general, no instance variables other than `final` variables (which are constants) may be defined in interfaces.

Goto Statements Gone

The Java language 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 `goto`ing into the middle of a `for` statement, for example. As mentioned above, multi-level `break` and `continue` remove the need for `goto` statements.

Operator Overloading

The Java language has no C++ style *operator overloading*.

[*] goto is still a reserved word—it just doesn't do anything.

Automatic Coercions

The Java language 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;
float  myFloat = 3.14159;
myInt = myFloat;
```

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

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

Pointers

Most studies agree that *pointers* are one of the primary features that enable programmers to put 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 the Java language has no pointers. Any task that would require arrays, structures, and pointers in C can be much more easily and reliably performed in the Java language by declaring objects and arrays of objects.

2.4 Summary

To sum up this chapter, you've by now gained the understanding that the Java language is:

- *Simple*—the number of language constructs you need to understand to get your job done is minimal.
- *Object-oriented*—you can develop software using up-to-date software development practices.
- *Familiar*—the Java language "looks like" C and C++ while discarding the overwhelming complexities of those languages.

Architecture Neutral, Portable, and Robust 3

With the phenomenal growth of networks, today's developers must "think distributed". Applications—and 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, if applications must be able to execute anywhere on the network without *a priori* knowledge of the hardware and software platform on which it will run, the binary distribution problem quickly becomes unmanageable. To solve this problem, software must become *architecture-neutral* and *portable*. Finally, reliability is at a high premium in the distributed world—code from anywhere on the network should work robustly without low probabilities of creating "crashes" in applications importing code fragments.

3.1 Architecture Neutral

The solution that the Java language system adopts to solve the binary distribution problem is a "binary code format" that's independent of hardware architectures and operating system and window system interfaces. This file format is *architecture-neutral*. If the Java run-time system is made available on a given hardware and software platform, Java language applications can then execute on many different processors and operating system architectures without the need to port them.

Byte Codes

The Java language compiler doesn't generate "machine code" in the sense of native hardware instructions. Rather, the Java language compiler generates *byte codes*—a high-level, machine-independent "machine code" for a hypothetical machine that is implemented by the Java interpreter and run-time system. One of the early examples of this approach was the UCSD P System, which was immensely popular in the middle 1970s and early 1980s.

The architecture-neutral approach is useful not only for network-based applications, but also for single system software distribution. In today's personal computer market, application writers have to produce versions of their applications that are compatible with the IBM PC, Apple Macintosh, and 57 flavors of workstation architectures in the fragmented UNIX marketplace. With the PC market (through Windows/NT) diversifying into 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. Using the Java language, the same version of your application can run on all platforms.

The Java language byte codes are designed to be both easy to interpret on any machine and easy to dynamically translate into native machine code if required.

3.2 Portable

The primary benefit of using 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 programs are portable across architectures by programming to a lowest common denominator.

The Java language eliminates this issue by defining standard behavior that will apply to the data types across all platforms. The sizes of the Java language's primitive data types are specified, as is 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

Note that the Java language has no `unsigned` data types.

The data types and sizes described above are standard across all implementations of the Java language environment. These choices are reasonable given current microprocessor architectures because essentially all of the 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 support single- and double-precision floating point.

The libraries that are part of the Java language run-time system define portable interfaces. For example, there is an abstract Window class and implementations of it for UNIX, Windows, and Macintosh.

The Java language environment itself is readily portable to new architectures and operating systems. The Java compiler is itself written in the Java language. 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.

3.3 Robust

The Java language is intended for developing software that must be *robust,* highly *reliable*, and *secure,* in a variety of ways. The Java language places a lot of emphasis on early checking for possible problems, later dynamic (run-time) checking, and eliminating error prone situations.

Strict Compile 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 into service.

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 this checking from C, which is relatively lax (the major issue is method/procedure declarations). The Java language *requires* declarations and does not support C-style implicit declarations.

Many of the stringent compile-time checks are carried over to the run-time, 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 the Java language and C/C++ is that the Java language's memory model eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, the Java language has true arrays, 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 a pointer by casting.

Garbage collection makes the programmer's job vastly easier. With the burden of memory management taken off the programmer's shoulders, storage allocation errors go away.

While the Java language 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.

A Major Benefit—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.

Programmers can be relatively fearless about dealing with memory when programming in the Java language because they don't have to worry about it getting messed up.

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.

The Java language 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.

3.4 Summary

The Java language—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.

Interpreted, Dynamic, Secure, and Threaded 4

Programmers using “traditional” software development tools have become inured 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 is straining the capabilities of development tools—even fancy “make”-style tools such as are found on UNIX systems. This development approach bogs down as applications grow into the hundreds of thousands of lines of code sizes.

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*. The notion of virtual interpreted machines 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 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.

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.

4.1 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 the re-use of code. Most programmers doing object-oriented development today adopted C++ as their language of choice. But C++ suffers from a somewhat serious problem that impedes its widespread use in the production and distribution of "software ICs." This defect is known as the *fragile superclass problem*.

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 re-compilation, 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 re-compilation problem." You *can* avoid these problems in C++, but with extraordinary difficulty, and doing so effectively means not using any of the language's OO features directly. By avoiding the object-oriented features of C++, developers defeat the goal of re-usable "software ICs."

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.

Interfaces

The Java language implements the concept of an *interface*—a concept borrowed from Objective C[*] and similar to a class. An interface is simply a specification of a collection of methods that an object responds to. An interface does not include any instance variables or implementations. You can import and use multiple interfaces in a flexible manner, thus providing the benefits of multiple inheritance without the inherent difficulties created by the usual rigid class inheritance structure.

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++

* It's similar in concept to Objective C's *protocols*.

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 have it easily dynamically-linked into the running system.

4.2 Security in the Java Environment

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. One of the Java compiler's primary lines of defense is its memory allocation and reference model. Simply put, Java does not have "pointers" in the traditional C and C++ sense—memory cells that contain the addresses of other memory cells.

Memory layout decisions are not made by the 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 language system is executing. The Java interpreter references memory via symbolic "handles" that are resolved to real memory addresses at run time. 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/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.

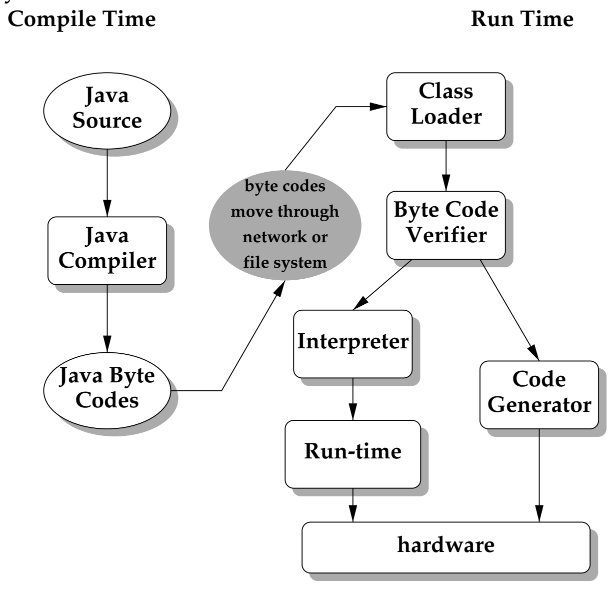
The Byte Code Verification Process

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 web browser imports a code fragment from anywhere, it doesn’t actually know if code fragments follow the 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 byte code stream? The answer is simple—it doesn’t trust the incoming code, but subjects it to *byte code verification*.

The tests range from simple verification that the format of a code fragment is correct, to passing through a simple theorem prover to establish that the code fragment plays by the rules—that it doesn’t forge pointers, it doesn’t violate access restrictions, and it accesses objects as what they are (for example, that “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.

The Byte Code Verifier

The last phase of the byte code loader is the *verifier*. It traverses the byte codes, constructs the type state information, and verifies the types of the parameters to all the byte code instructions.



The illustration shows the flow of data and control from Java language source code through the Java compiler, to the byte code verifier and hence on to the Java interpreter. The important issue is that the Java class loader and the byte code verifier make no assumptions about the primary source of the byte code stream—the code may have come from the local system, or it may have travelled halfway around the planet. The byte code verifier acts as a sort of gatekeeper. The byte code verifier ensures that the 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 byte code instructions are known to always be correct
- No illegal data conversions are done, like converting integers to pointers
- Object field accesses are known to be legal—private or public or protected

While all this checking appears excruciatingly detailed, by the time the byte code 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.

Security Checks in the Class Loader

After incoming code has been vetted and determined clean by the byte code verifier, the next line of defense is the Java *class loader*. The environment seen by a thread of execution running Java byte codes 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.

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 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

4.3 Multithreading

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 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.

Java Supports Threads at the Language Level

Built in support for *threads* provides Java programmers with a powerful tool to improve perceived 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 the Java language. Threads are a class from which you can instantiate new Thread objects. The Thread class provides a rich collection of methods to start the thread, run the thread, stop the thread, and make enquiries about the 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 the Java language's integration of threads was modelled after Cedar and Mesa.

Java's threads are pre-emptive. If your applications are likely to be compute-intensive, you might consider how to give up control periodically to give other threads a chance to run. This will ensure better interactive response for graphical applications.

Integrated Thread Synchronization

The Java language 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. The monitor precludes `synchronized` methods from running while any other such method is running in that thread instance. 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 are writing Java language applications, you should take care to implement your classes and methods so they're thread-safe, in the same way that the Java language 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 language monitors are re-entrant—a method can acquire the same monitor more than once, and everything will still work.

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.

4.4 Java Class Libraries

Java supplies extensive and well developed class libraries so programmers can get started quickly.

Language Foundation Classes

At the lowest level of the Java language, the foundation classes implement wrappers for the primitive types, threads, exceptions, and a variety of other fundamental classes.

I/O Class Library

At the lowest level of the Java language, there are foundation classes that implement

Another Window Toolkit Class Library

The window toolkit library implements the functionality you need to display and interact with graphical user interface components on the screen. This library contains classes for the basic interface components, events, fonts, color, and controls such as buttons and scrollbars.

Utility Class Library

The utility class library implements a variety of encoder and decoder techniques, date and time, hash table, vector, and stack.

Network Interface Class Library

The network interface class library extends the functionality of the I/O class library with socket interfaces and Telnet interfaces.

4.5 Performance

Test measurement of some simple Java programs on current high-end computer systems 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

While these performance numbers for interpreted byte codes are usually more than adequate to run interactive graphical end-user applications, situations may arise where higher performance is required. In such cases, the byte codes 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 byte code 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 byte codes. Performance of byte codes converted to machine code is roughly the same as native C or C++.

4.6 The Java Language Compared

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	●	●	●	◑	◑	◑	○
O-O	●	●	○	○	○	○	◑
Robust	●	●	●	◑	●	○	○
Secure	●	◑	◑	◑	◑	○	○
Interpreted	●	●	●	●	●	○	○
Dynamic	●	●	●	○	◑	○	○
Portable	●	◑	●	◑	◑	◑	◑
Neutral	●	◑	◑	◑	◑	○	○
Threads	●	○	○	○	○	○	○
Exceptions	●	●	○	○	○	○	◑
Performance	*High*	*Medium*	*Low*	*Low*	*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.

There are literally hundreds of programming languages available for developers to write programs to solve problems in specific areas. The 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 Shells and TCL, for example, are fully interpreted high level languages. They deal with "objects" (at least in the sense they can be said to deal with objects at all) at the system level—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 byte codes. This tradeoff may well be reasonable if the run time of such a program is reasonably short and you use the program on an itinerant basis.

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 non-robust memory management systems and difficult to implement and use multithreading capabilities. 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 the very high-level and portable but slow scripting languages and the 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.

4.7 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 four chapters have been your introduction to the Java language environment. You've learned about the capabilities of the Java language environment and its clear benefits to develop software for the distributed world. 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 5

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

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

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

The HotJava browser includes many innovative features and capabilities above and beyond the first generation of static Web browsers. The HotJava browser 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. It 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. The

HotJava browser brings a much needed measure of *interactivity* to the concept of the Web browser, transforming the existing static data display of current generation Web browsers into a new dynamic, animation-oriented, interactive viewing system for hypertext. Content developers candistribute their applications across the Internet with the click of a button.

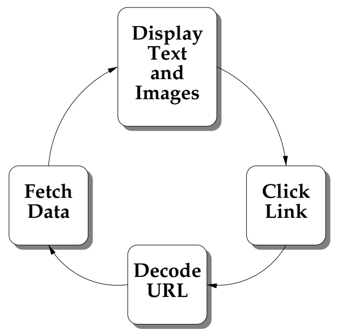
5.1 The Evolution of Cyberspace

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.

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 broswers 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."

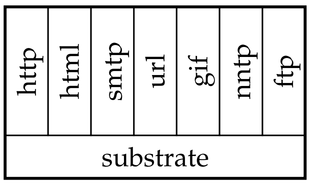
5.2 First Generation Browsers



What we could call the "first-generation" Web browsers—exemplified by NCSA Mosaic and 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.

The HotJava browser 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 Web 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.

5.3 The HotJava Browser—A New Concept in Web Browsers

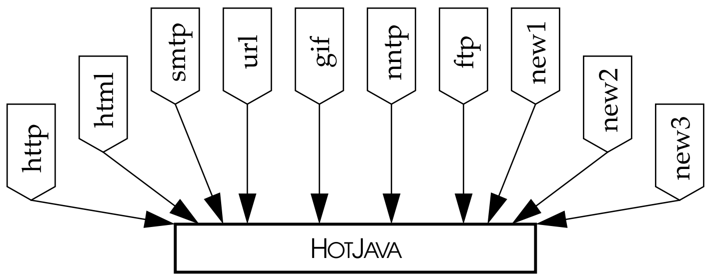
The HotJava browser solves the monolithic approach and moves the focus of interactivity away from the Web server and onto the Web *client*—that is, to the computer on which the user is browsing the Web. Because of its basis in the Java language system, a HotJava browser client can dynamically download 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.

The HotJava browser removes the static limitations of the Mosaic generation of Web browsers with its ability to add arbitrary behavior to the browser. Using the HotJava browser 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, the HotJava browser 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.

5.4 The Essential Difference

The central difference between the HotJava browser and other browsers is that while these other browsers have knowledge of the Internet protocols hard-wired into them, the HotJava browser understands essentially none of them. What it *does* understand is to how 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.

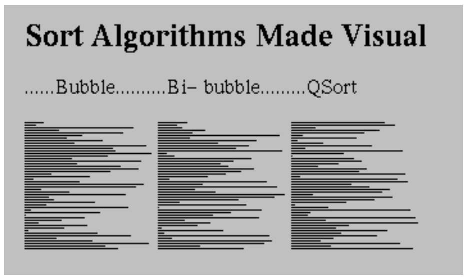


The HotJava browser is the coordinator of a federation of pieces, each with individual responsibility. New pieces can be added at any time. Pieces can be added from across the network, without needing to be concerned with what CPU architecture they were designed for and with reasonable confidence that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

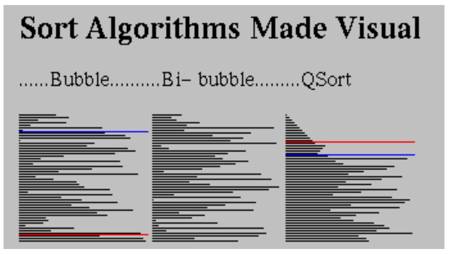
5.5 Dynamic Content

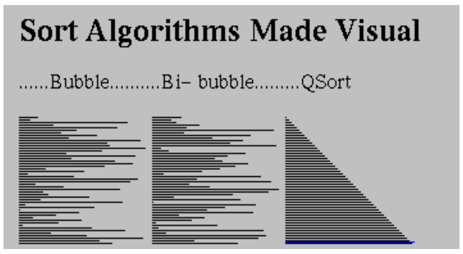
One of the most visible uses of the HotJava browser's ability to dynamically add to its capabilities is something we call *dynamic content*. For example, someone could write a Java language program following the HotJava API that implemented an interactive chemistry simulation. People browsing the net with the HotJava browser 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 the Java language.

For example, the following is a snapshot of the HotJava browser 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.


Sort Algorithms Made Visual
......Bubble..........Bi- bubble.........QSort

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


Sort Algorithms Made Visual
......Bubble..........Bi- bubble.........QSort

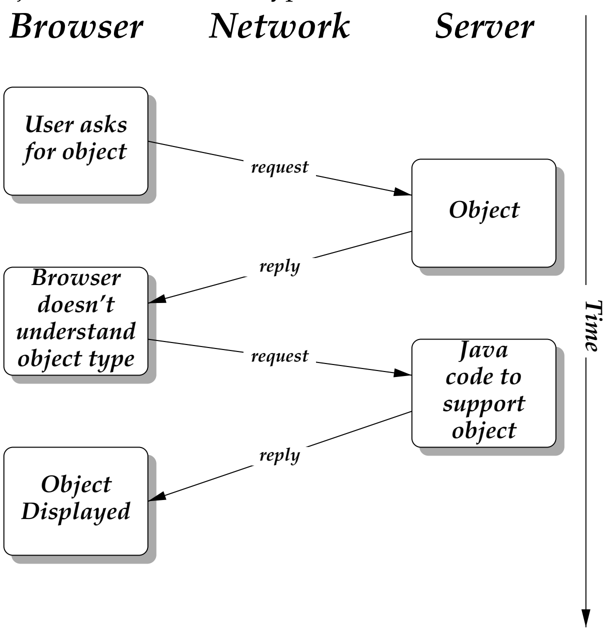

Sort Algorithms Made Visual
......Bubble..........Bi- bubble.........QSort

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.

5.6 Dynamic Types

The HotJava browser'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 directly. The HotJava browser, 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 the Java language 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. The HotJava browser essentially upgrades itself on the fly when it sees this new type.

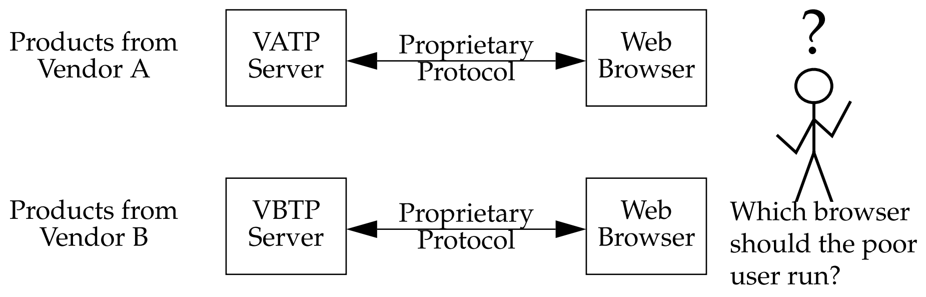
The following is an illustration of how HotJava negotiates with a server when it encounters an object of an unknown type:



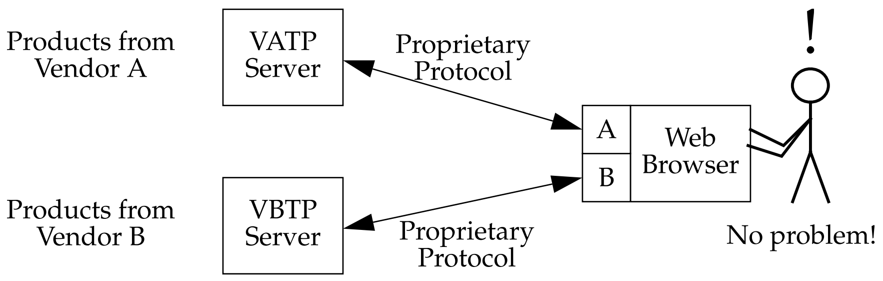
5.7 Dynamic Protocols

The protocols that Internet hosts use to communicate among themselves are key components of the net. For the World-Wide Web (WWW), HTTP is the most important of these communication protocols. In documents on the WWW a reference to a document is called a URL. The URL contains the name of the protocol, HTTP, that is used to find that document. Current 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. This allows 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.



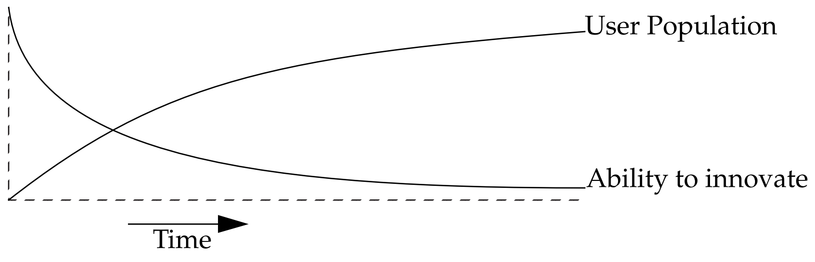
With the HotJava browser as a base, vendors can produce and sell exactly the piece that is their added value, and integrate smoothly with other vendors, creating a final result that is seamless and very convenient for the end user.



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.

5.8 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 the HotJava browser, 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. the HotJava browser's flexibility can greatly aid that. As new protocols and data types become important, they can be transparently incorporated.

5.9 Implementation Details

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

Mosaic starts with a URL and fetches the object 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 (for example, HTML document or JPEG image). From this type information, code to manipulate and view this data is invoked.

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

In contrast, the HotJava browser 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, the HotJava browser instead uses this type string to derive a 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 the browser suspects might be a good place to look. Similarly, the code to handle different types of data objects and different ways of viewing them.

What makes this federation of pieces and dynamic addition of capabilities possible is the Java language, the underlying programming language described in the earlier parts of this document, and the environment on which HotJava is built.

5.10 Security

Network security is of primary importance to Internet users, especially with the burgeoning 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 the Java language, run-time system, and the higher-level protocols of the HotJava browser itself.

One of the most important technical challenges in building a system like the HotJava browser is making it secure. Installing and running imported code fragments from across the network is 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, but on the other hand it can 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. The HotJava browser has a series of facilities that layer and interlock to provide a fairly high degree of safety.

The First Layer—The Java Language Interpreter

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

When a code fragment is imported into HotJava it doesn't actually know whether or not the code fragment follows the Java language rules for safety, since it may not have been produced by the local Java language compiler. 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 Verifier.

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.

5.11 Summary

The HotJava Web browser—based upon the foundations of the Java language 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 6

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

Gilbert and Sullivan—*The Mikado*

The Java/HotJava 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

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) 413 2666
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: 358-0-502 27 00
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 221-7021
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: 861-831-5568
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