



---

# 18

## Java

---

This chapter will be added to the next printing of the book: M. Ben-Ari, Understanding Programming Languages, John Wiley & Sons, 1996.  
© M. Ben-Ari 1998.  
All rights reserved, except that an owner of the book may print a single copy of this chapter.

Java<sup>1</sup> is both a programming language and a model for developing software for networks. We start with a description of the Java model, because it is essential to understand that the model is independent of the language. The language is interesting in itself, but the excitement that Java has generated is due more to the model than to the language design.

### 18.1 The Java model

You can write a compiler for the Java language exactly as you would for any other imperative, object-oriented language. The Java model, however, is based on the concept of an interpreter (see Figure 3.2, reproduced in modified form as Figure 18.1) like the Pascal interpreters we discussed in Section 3.10.

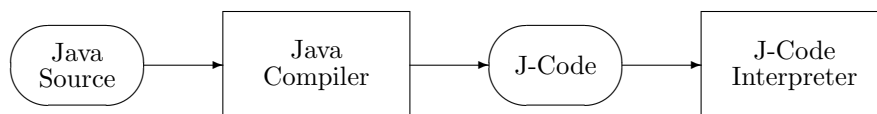


Figure 18.1 An interpreter for Java

---

<sup>1</sup>Java is a trademark of Sun Microsystems, Inc.

In Java, however, the arrow from the J-Code to the J-Code interpreter does not represent simply the flow of data between components of a software development environment. Instead, the J-Code can be packaged in an applet<sup>2</sup> that can be transmitted across the communications system of a network. The receiving computer executes the J-Code using an interpreter called a Java Virtual Machine (JVM). The JVM is usually embedded within a net browser, which is a program for retrieving and displaying information obtained from a network. When the browser identifies that an applet has been received, it invokes the JVM to execute the J-Code. Furthermore, the Java model includes standard libraries for graphical user interfaces, multimedia and network communications, which are mapped by each implementation of the JVM onto the facilities of the underlying operating system. This is a significant extension of the concept of a virtual machine beyond the simple computation of Pascal P-Code.

Note that you can also write an ordinary program in Java which is called an application. An application is not subject to the security limitations discussed below. Unfortunately, there are a few irritating differences between the programming of an applet and an application.

## *Performance*

Too good to be true? Of course. The Java model suffers from the same performance problems that affect any model based on interpretation of an abstract machine code. For relatively simple programs running on powerful personal computers and workstations, this will not be a serious problem, but performance may set a limit on the applicability of the Java model.

One solution is to include, within the browser on the receiving end, a compiler that translates the abstract J-Code into machine code for the receiving computer. In fact, the computer can carry out this translation simultaneously, or nearly so, with the reception of the J-Code; this is called on-the-fly compilation. At worst, performance will improve the second time you execute the applet you have loaded.

However, this solution is only a partial one. It would not be practical or economical to bundle a sophisticated optimizing compiler within each browser for each combination of computer and operating system. Nevertheless, for many types of applications, the Java model will be very successful for developing portable software.

## *Security*

Suppose that you download an applet and execute it on your computer. How do you know that the applet is not going to erase your hard disk? Considering the extensive damage that has been caused by viruses maliciously appended to otherwise valid programs, you will appreciate how dangerous it is to download arbitrary programs

---

<sup>2</sup>Applet is a new term derived from “application”.

from a remote location. The Java model employs several strategies to eliminate (or at least reduce!) the possibility that an applet, downloaded from a remote location, will damage the software and data on the receiving computer:

**Strong type-checking** As in Ada, the idea is to eliminate accidental or malicious damage to data caused by out-of-bound array indices, arbitrary or dangling pointers, and so on. We will elaborate on this in the following sections.

**J-Code verification** The interpreter on the receiving computer verifies that the stream of bytes received from the remote computer actually consists of legal J-Code instructions. This ensures that the secure semantics of the model is actually what is executed, and that the interpreter cannot be “spoofed” into causing damage.

**Applet restrictions** An applet is not allowed to perform certain operations on the receiving computer, such as writing or erasing files. This is the most problematical aspect of the security model, because we would like to write applets that can do anything that an ordinary program can.

Clearly, the success of Java depends on the degree to which the security model is strict enough to prevent malicious use of the JVM, while at the same time retaining enough flexibility to enable useful programs to be built.

### ***Language independence***

The astute reader will have noticed that the previous section has been written without reference to the Java programming language! This is intentional, because the Java model is both valid and useful even if the source code of the applets is written in some other language. For example, there are compilers that translate Ada 95 into J-Code.<sup>3</sup> However, the Java language was designed together with the JVM and the language semantics closely match the capabilities of the model.

## 18.2 The Java Language

The superficial syntax and semantics of the Java language are similar to those of C++. However, while C++ maintains almost complete backward compatibility with C, Java rejects compatibility in favor of “fixing” the problematic constructs of C. Despite the superficial similarities, Java and C++ are quite different languages and a program in C++ cannot be easily ported to Java.

The major similarities between the languages are in the following areas:

- Elementary data types, expressions and control statements.

---

<sup>3</sup>See, for example: <http://www.inmet.com>.

- Functions and parameters.
- Class declarations, class members and accessibility.
- Inheritance and dynamic polymorphism.
- Exceptions.

The following sections discuss five areas where the design of Java is significantly different from C++: reference semantics, polymorphic data structures, encapsulation, concurrency and libraries. In the exercises, we ask you to explore the other differences between the languages.

### 18.3 Reference semantics

Perhaps the worst aspect of C (and C++) is the unrestricted and excessive use of pointers. Not only are pointer manipulations difficult to understand, they are extremely error-prone as described in Chapter 8. The situation in Ada is much better because strong type-checking and access levels ensure that pointers cannot be used to break the type system, but data structures still must be built using pointers.

Java (like Eiffel and Smalltalk) uses reference semantics instead of value semantics.<sup>4</sup> Whenever a variable of non-primitive type<sup>5</sup> is declared, you do not get a block of memory allocated; instead, an implicit pointer is allocated. A second step is needed to actually allocate memory for the variable. We now demonstrate how reference semantics works in Java.

#### *Arrays*

If you declare an array in C, you are allocated the memory you requested (Figure 18.2(a)):

```
int a.c[10]; C
```

while in Java, you only get a pointer that can be used to store an array (Figure 18.2(b)):

```
int[] a.java; Java
```

Allocating the array takes an additional instruction (Figure 18.2(c)):

```
a.java = new int[10]; Java
```

though it is possible to combine the declaration with the allocation:

<sup>4</sup>Do not confuse these terms with the similar terms used to discuss parameter passing in Section 7.2.

<sup>5</sup>Java's primitive types are: float, double, int, long, short, byte, char, boolean. Do not confuse this term with primitive operations in Ada (Section 14.5).

```
int[] a_java = new int[10];
```

Java

If you compare Figure 18.2 with Figure 8.4, you will see that Java arrays are similar to structures defined in C++ as `int *a` rather than as `int a[]`. The difference is that the pointer is implicit so you do not need to concern yourself with pointer manipulation or memory allocation. In fact, in the case of an array, the variable will be a dope

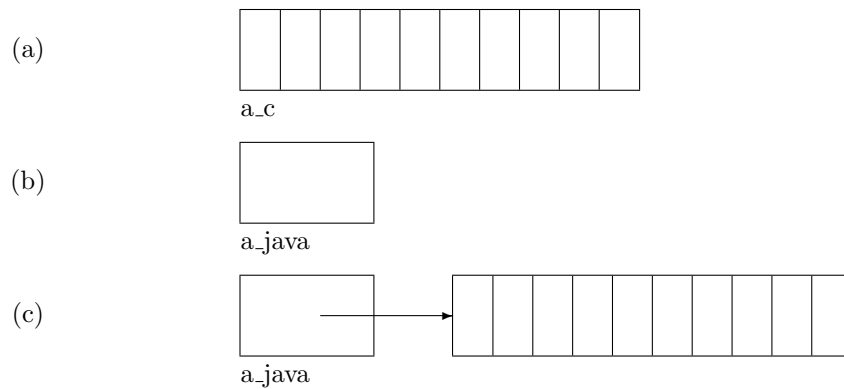


Figure 18.2 Value versus reference semantics

vector (Figure 5.4), enabling bounds checking to be done when accessing the array.

Note that the Java syntax is rather easier to read than the C++ syntax: `a_java` is of type `int[]`, that is “integer array”; in C++, the component type `int` and the array indication `[10]` appear on opposite sides of the variable name.<sup>6</sup>

Under reference semantics, dereferencing of the pointer is implicit, so that once the array is created you access the array as usual:

```
for (i = 1; i ≤ 10; i++)
    a_java[i] = i;
```

Java

Of course, this indirect access can be much less efficient than direct access unless optimized by the compiler.

Note that allocating an object and assigning it to a variable can be done in any statement, resulting in the following possibility:

```
int[] a_java = new int[10];
...
a_java = new int[20];
```

Java

<sup>6</sup>To ease the transition to Java, the language also accepts the C++ syntax.

The variable `a.java`, which pointed to a ten-element array, now points to a twenty-element array, and the original array is now garbage (Figure 8.7). The Java model specifies that a garbage collector must exist within the JVM.

### *Dynamic data structures*

How can you create lists and trees without pointers?! The declarations for linked lists in C++ and Ada given in Section 8.2 would seem to indicate that we need a pointer type to describe the type of the next field:

```
typedef struct node *Ptr;
typedef struct node {
    int    data;
    Ptr   next;
} node;
```

C

But in Java, every object of non-primitive type is automatically a pointer:<sup>7</sup>

```
class Node {
    int data;
    Node next;
}
```

Java

The field `next` is merely a pointer to a `Node`, not a `Node`, so there is no circularity in the declaration. Declaration of a list is simply:

```
Node head;
```

Java

which creates a pointer variable whose value is null (Figure 18.3 (a)). Assuming that there is an appropriate constructor (Section 15.3) for `Node`, the following statement creates a node at the head of the list (Figure 18.3 (b)):

```
head = new Node(10, head);
```

Java

### *Equality and assignment*

The behavior of the assignment statement and the equality operator in languages with reference semantics can be a source of surprises for programmers with experience in a language with value semantics. Consider the Java declarations:

<sup>7</sup>C++ retains the struct construct for compatibility with C. However, the construct is identical with a class all of whose members are declared public. Java does not retain compatibility with C; instead, every type declaration must be a class.

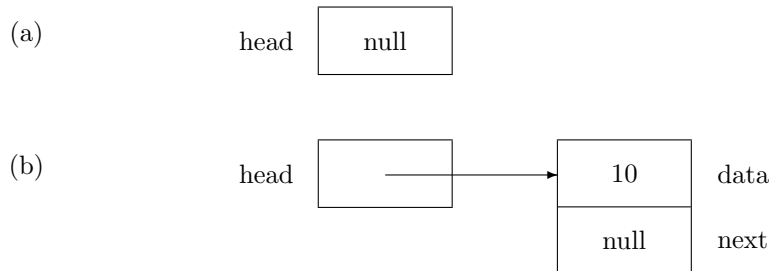


Figure 18.3 Linked list in Java

```
String s1 = new String("Hello");
String s2 = new String("Hello");
```

Java

This results in the data structure shown in Figure 18.4.

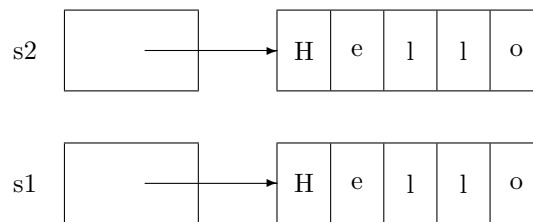


Figure 18.4 Assignment and equality

Now suppose that we compare the string variables:

```
if (s1 == s2) System.out.println("Equal");
else System.out.println("Not equal");
```

Java

The program will print Not equal! The reason is actually clear from Figure 18.4: the variables are pointers with different values, and the fact that they point to equal arrays is irrelevant. Similarly, if we assign one string to the other `s1 = s2`, the pointers will be assigned and no copying of values will be done. In this case, of course, `s1 == s2` will be true.

Java distinguishes between shallow copy and comparison, and deep copy and comparison. The latter are declared in the common ancestor class `Object` and are called `clone` and `equals`. The predefined class `String`, for example, overrides these operations,

so `s1.equals(s2)` is true. You can override the operations to create deep operations for your classes.

To summarize the use of reference semantics in Java:

- Flexible data structures can be safely manipulated.
- Programming is simpler because no explicit pointers need be used.
- There is an overhead associated with indirect access to data structures.

## 18.4 Polymorphic data structures

In Ada and C++, we have two ways of constructing polymorphic data structures: generics in Ada and templates in C++ for compile-time polymorphism, and class-wide types in Ada and pointers/references to classes in C++ for run-time polymorphism. The advantage of generics/templates is that the data structure is fixed when it is instantiated at compile-time; this can improve both the efficiency of code generation and the memory that needs to be allocated for the data structure.

Java chooses to implement only run-time polymorphism. As in Smalltalk and Eiffel, every class in Java is considered to be derived from a root class called `Object`. This means that a value of any non-primitive type<sup>8</sup> can be assigned to an object of type `Object`. (Of course this works because of the reference semantics.)

To create a linked list, a node class would first be defined as containing (a pointer to) an `Object`. The list class would then contain methods to insert and retrieve values of type `Object`:

```
class Node {
    Object data;
    Node next;
}

class List {
    private Node head;
    void Put(Object data) { ... };
    Object Get() { ... };
}
```

Java

If `L` is an object of type `List` and `a` is an object of type `Airplane_Data`, then `L.Put(a)` is valid because `Airplane_Data` is derived from `Object`. When a value is retrieved from the list, it must be cast to the proper descendant of `Object`:

<sup>8</sup>To allow a primitive type to be considered as derived from `Object`, “wrapper” classes such as `Integer` are defined.

```
a = (Airplane_Data) List.Get();
```

Java

Of course, if the value returned is not of type `Airplane_Data` (or descended from the type), an exception will be raised.

The advantage of this paradigm is that it is very easy to write generalized data structures in Java, but compared with generics/templates, there are two disadvantages: (1) the additional overhead of the reference semantics (even for a list of integers!), and (2) the danger that an object placed on the wrong queue will cause a run-time error when retrieved.

## 18.5 Encapsulation

In Section 13.1, we discussed the fact that C has no encapsulation construct, and in Section 13.5 we noted that the scope resolution operator and the namespace construct in C++ improve on C's coarse approach to global name visibility. For compatibility, C++ does not have an encapsulation construct; instead it depends on "h"-files. Ada has the package construct which supports encapsulation of constructs into modules (see Section 13.3), and the specification and implementation (body) of packages can be separately compiled. `with`-clauses enable the software engineer to precisely specify dependencies among the modules, and child packages (briefly mentioned in Section 15.2) can be used to develop module structures with hierarchial accessibility.

Java contains an encapsulation construct called a package, but confusingly, the construct is closer in spirit to the C++ namespace than to the Ada package! A package is a collection of classes:

```
package Airplane_Package;
```

Java

```
public class Airplane_Data
{
    int speed;                // Accessible in package
    private int mach_speed;   // Accessible in class

    public void set_speed(int s) {...}; // Globally accessible
    public int get_speed() {...};
}

public class Airplane_Database
{
    public void new_airplane(Airplane_Data a, int i)
    {
```

```

        if (a.speed < 1000) {           // OK !
            a.speed = a.mach_speed;    // Error !
        }

        private Airplane_Data[] database = new Airplane_Data[1000];
    }

```

A package can be divided into several files, but a file can contain classes from only one package.<sup>9</sup>

The specifiers `public` and `private` are similar to those of C++: `public` means that the member is accessible anywhere outside the class, while `private` limits accessibility to other members of the class. If no specifier is given, then the member is visible within the package. In the example, we see that the member `int speed` of class `Airplane_Data` has no specifier, so a statement within the class `Airplane_Database` is allowed to access it since the two classes are in the same package. The member `mach_speed` is declared `private`, so it is accessible only within the class `Airplane_Data` in which it is declared.

Similarly, classes have accessibility specifiers. In the example, both classes are declared `public`, meaning that other packages can access any (`public`) member of these classes. If a class is declared `private`, it is only accessible within the package. For example, we might want to declare a `private` class `Airplane_File` which would be used within the package to store the database.

Packages are important in Java software development because they allow you to group related classes together, while retaining explicit control over the external interface. The hierarchical library structure simplifies the construction of software development tools.

## *Comparison with other languages*

Java packages serve to control global naming and accessibility in a manner similar to namespaces in C++. Given the declarations in our example, any Java program can contain:

```

Airplane_Package.Airplane_Data a;
a.set_speed(100);

```

Java

because the class and method names are `public`. Without examining the complete source code of a package, you cannot know which classes are imported. There is an `import` statement which opens up the name space of a package enabling direct visibility. This construct is similar to using `using` in C++ and `use` in Ada.

<sup>9</sup>You are not required to explicitly declare a package; if you do not, all classes in the file are considered to belong to a common anonymous package.

A key difference between Java and Ada is that in Ada the package specification and the package body are separate. This is not just a convenience for reducing the size of compilations, but a significant factor in managing, developing and maintaining large software systems. The package specification can be frozen, allowing both its implementation and development of other units to proceed in parallel. In Java, the “interface” of a package is simply the collection of all public declarations. The development of large systems in Java will require software tools to extract package specifications,<sup>10</sup> and to ensure the consistency of the specification and the implementation.

The package construct does give Java one major advantage over C++. The packages themselves use a hierarchical naming convention that enables the compiler and interpreter to automatically locate classes. For example, the standard library contains a function called `java.lang.String.toUpperCase`. This is interpreted just as an operating system interprets an expanded filename: `toUpperCase` is a function in package `java.lang.String`. Java libraries can (but need not) be implemented as hierarchical directories where each function is a file in the directory for its class. Note that the hierarchical names are meaningful only outside the language; a subpackage has no special accessibility privileges to its parent. This is in stark contrast to an Ada child package which can access the private declarations of its parent, subject to rules which prevent it from exporting private declarations.

## 18.6 Concurrency

Ada is one of the few languages that include support for concurrency within the language, rather than delegating the support to the operating system. The Java language follows the Ada philosophy so that concurrent programs can be ported across operating systems. An important application of concurrency in Java is the programming of servers: each client request causes the server to spawn a new process to fulfil that request.

The Java construct for concurrent programming is called threads. There is no essential difference between threads and the standard term processes in concurrency; the distinction is more implementation oriented in that multiple threads are assumed to execute within the same address space. For purposes of developing and analyzing concurrent algorithms, the model is the same as that described in Chapter 12—interleaved execution of atomic instructions of processes.

A Java class that inherits from the class `Thread` declares a new thread type. To actually create the thread, an allocator must be called, followed by a call to `start`. This causes the `run` method in the thread to begin execution:<sup>11</sup>

---

<sup>10</sup>In Eiffel, this is called the short form of a class (Section 15.4).

<sup>11</sup>Java uses the keyword `extends` in place of the colon used by C++.

```
class My_Thread extends Thread
{
    public void run() {...};           // Called by start
}

My_Thread t = new My_Thread();       // Create thread
t.start();                            // Activate thread
```

Java

One thread can explicitly create, destroy, block and release another thread.

These constructs are similar to the Ada constructs which allow a task type to be defined followed by dynamic creation of tasks.

## ***Synchronization***

Java supports a form of synchronization similar to monitors (see Section 12.4). A class may contain methods specified as synchronized. A lock is associated with each object of the type of this class; the lock ensures that only one thread at a time can execute a synchronized method on the object. The following example shows how to declare a monitor to protect a shared resource from simultaneous access by several threads:<sup>12</sup>

```
class Monitor
{
    synchronized public void seize() throws InterruptedException
    {
        while (busy) wait();
        busy = true;
    }

    synchronized public void release()
    {
        busy = false;
        notify();
    }

    private boolean busy = false;
}
```

Java

The monitor protects the boolean variable which indicates the state of the resource. If two threads try to execute the seize method of a monitor, only one will successfully

<sup>12</sup>Since wait can raise an exception, the exception must either be handled within the method, or the method must declare that it throws the exception. In C++, this clause would be optional.

acquire the lock and execute. This thread will set `busy` to true and proceed to use the resource. Upon leaving the method, the lock will be released and the other thread will be allowed to execute `seize`. Now, however, the value of `busy` is false. Rather than waste CPU time continuously checking the variable, the thread chooses to release the CPU by calling `wait`. When the first thread completes the use of the shared resource, it calls `notify`, which will allow the waiting thread to resume execution of the synchronized method.

The Java constructs for concurrency are quite low-level. There is nothing to compare with the sophisticated Ada rendezvous for direct process-to-process communication. Even when compared with Ada 95 protected objects, Java's constructs are relatively weak:

- A barrier of a protected object is automatically re-evaluated whenever its value could change; in Java, you must explicitly program loops.
- Java maintains simple locks for each object; in Ada, a queue is maintained for each entry. Thus if several Java threads are waiting for a synchronized object, you cannot know which one will be released by `notify`, so starvation-free algorithms are difficult to program.

## 18.7 The Java Libraries

The trend in programming language design is to limit the scope of the language by providing functionality in standard libraries. For example, `write` is a statement in the Pascal language with a special syntax, while in Ada there are no I/O statements; instead, I/O is supported through packages in the standard library.

The standard Ada libraries supply computational facilities such as I/O, character and string processing, mathematical functions, and system interfaces. C++ also supports container classes such as stacks and queues. Similarly, Java contains basic libraries called `java.lang`, `java.util` and `java.io`, which are part of the language specification.

In addition to the language specification, there is a specification for the Application Programming Interface (API) that all Java implementations are supposed to support. The API consists of three libraries: `java.applet`, `java.awt` and `java.net`.

`java.applet` contains support for creating and executing applets and for creating multimedia applications. The Abstract Window Toolkit (AWT) is a library for creating graphical user interfaces (GUI): windows, dialog boxes and bit-mapped graphics. The library for network communications provides the essential interface for locating and transferring data on the net.

To conclude:

- Java is a portable object-oriented language with reference semantics.
- The Application Programming Interface supplies portable libraries support software development on networks.
- Security and safety are designed into the language and the model.
- Many of the important concepts of Java are really part of the underlying language-independent JVM.

## 18.8 Exercises

1. Given an arithmetic expression such as:

$$(a + b) * (c + d)$$

Java specifies that it be evaluated from left to right, while C++ and Ada allow the compiler to evaluate the subexpression in any order. Why is Java more strict in its specification?

2. Compare the final construct in Java with constants in C++ and Ada.
3. What is the relationship between the friend specifier in C++ and the package construct in Java.
4. C++ uses the specifier protected to enable visibility of members in derived classes. How does the package construct affect the concept of protected in Java?
5. Compare interface in Java with multiple inheritance in C++.
6. Analyze the differences between namespace in C++ and package in Java, especially with respect to the rules concerning files and nesting.
7. The exception construct in Java are quite similar to the exception construct in C++. One important difference is that a Java method must declare all exceptions that it can possibly throw. Justify this design decision and discuss its implications.
8. Compare Java monitors with the classic monitor construct.<sup>13</sup>
9. Compare the string processing capabilities of Ada 95, C++ and Java.
10. Compare clone and equals in Java with these operations in Eiffel.

## References

The official specification of the language is given in:

James Gosling, Bill Joy and Guy Steele. The Java Language Specification. Addison-Wesley, 1997.

Sun Microsystems, Inc., where Java was designed, maintains a Web site containing documents and software: <http://java.sun.com>.

---

<sup>13</sup>See: M. Ben-Ari, Principles of Concurrent and Distributed Programming, Prentice-Hall International, 1990, Chapter 5.