Preface

The Java language began as a program for embedded devices, specifically consumer electronics. The driving concepts behind the language included portability, enabling reuse as the underlying processors were changed in new versions of the device; and simplicity, to keep the best aspects of related languages and to throw out the fluff.

As the World Wide Web blasted onto the scene, the Java development team realized that with some additional functionality (specifically a GUI interface API) a language such as Java could readily be used to enable the specification of executable content on web pages. The inclusion of Java support in the Netscape 2.0 browser provided enough publicity and support for the language that it immediately became the de facto standard language for programming executable content.

Although Java has its roots in embedded systems and the web, it is important to realize that it is a fully functional high-level programming language that can provide users with a wide range of functionality and versatility. In "The Java Language: A White Paper," Sun Microsystems developers describe Java as:

> Java: A simple, object-oriented, distributed interpreted, robust, secure, architecture neutral, aportable, high-performance, multithreaded, and dynamic language.

This list of terms describes Sun's design goals for the Java language and portrays some of the most important features of the language. Simplicity refers to a small language learning curve, similarity to C and C++, and removal of some standard (but dangerous) features of languages such as C++. For example, Java does not use pointers as in C and C++, but rather references as in Pascal. This avoids potential pointer manipulation errors by the programmer. In addition, Java provides no header files as in C and C++, thus enabling more automated bookkeeping (although compilers are not currently utilizing the full potential of such a system, resulting in sometimes awkward coding constructs).

As an object-oriented language Java supports the concepts of abstract data typing as encapsulated in many object-oriented programming languages. The user defines a class which specifies a collection of data items and methods to operate on those items. Data and methods of the class can exist with the class (one instance per program execution) or with specific instances (objects) of the class. Classes are arranged in a hierarchy to provide mechanisms for inheritance and reuse. Java classes are further arranged into packages that provide some additional protection and bookkeeping for Java programmers. To assist the programmer, there exists a set of predefined classes that are provided with Java development/execution environments. Some of these classes are essential as they provide a bridge between the portable Java code and the underlying native operating system.

As a distributed language, the Java API provides functionality for inter process communication and remote data access. It is important to understand that

this distributed nature of Java is solely the benefit of good API classes and not any inherent distributed or parallel programming capabilities.

As an interpreted language, Sun means that the user creates an intermediate file containing the "bytecode" implementation of the program. It is the byte code that is subsequently interpreted and not the raw Java source code. The Java interpreter and the supporting run-time system implement what is called the Java Virtual Machine.

As a robust language, Java is strongly typed and does not use pointers. These two features greatly reduce the possibility of very common software flaws. In addition, Java has both built-in automatic garbage collection routine to prevent memory leakage and exception handling. The exception handling allows almost all errors to be caught and managed by the software.

As a secure language, Java provides for access control restrictions on class or object methods and data items. These may be implemented as part of the basic protection attribute of the items or through a run-time security monitor.

As architecture neutral and portable, Java functionality does not rely on any underlying architecture specifics, thus allowing the code (or even the byte code) to be executed on any machine with a virtual machine implementation.

As a high-performance language, Java is meant to execute well with respect to similar high-level languages. The use of special "just in time" compilers or other features may improve performance even more.

As a multithreaded language, Java permits the development of user specified concurrent threads of control, as well as synchronization mechanisms to establish consistency between the users.

As a dynamic language, Java is intended to be able to dynamically load code from the network and execute the new version of the code in the current virtual machine as opposed to recompiling the whole project.

The above list of features describes Sun's view of the Java language, a view that is shared by many users. We will assume that these features represent the base capabilities of the language. In the rest of this part we describe various features of Java, highlighting their challenges for formal method specifications of the language.

Java Basic Data Types

The Java language includes several built-in basic data types. These include boolean, char and numeric types: byte, short, integer, long for (8, 16, 32 and 64-bit integer calculations) and float and double for (32 and 64-bit floating point operations). Java provides standard operations for these types with the few special features discussed below. In addition Java has a reference data type for use of objects and a special reference data type, the array. Like Pascal and unlike C, the Java reference data type can only be copied; no increment or other operations can be performed on it.

Java is strongly typed and only permits limited type casting or automatic conversions. This strengthens the reliability of the language. The only problems is that the explicit casting of integer values (with either 32 or 64-bits) to smaller

integers such as byte results in truncation of the high-order bits, resulting in information loss and even potential change in sign.

Java Classes

All user-defined Java data types are specified using a class definition. A class defines the fields and methods of the object and their appropriate access modifiers. With Java 1.1 and later, users have the ability to define subclasses and anonymous classes within their own classes.

Java Files and Packages

A Java program consists of one or more packages, each of which consist of a collection of Java classes. A class within a single package has a stronger trust relationship with other classes in that package than with those outside of the package. In addition, the package relationship provides the ability to utilize a hierarchical naming convention for Java classes.

Each Java class is defined within a single file. Although a file may contain more than one class definition, only one file in that class may be declared public (and must be named the same as the source file). Classes within the same file are implicitly within the same package.

Exception Handling

Java provides a flexible exception handling capability. Any time an exception occurs the violating routine can throw a named exception, abruptly terminating the statement. All Java statements can be encapsulated within a try-catch statement. If the enclosed statement throws an exception that is specified within the catch clause, the violating statement is terminated and the code in the catch clause is executed. Otherwise, the thrown exception is propagated up the call hierarchy.

The Java Virtual Machine

All Java programs are compiled into an intermediate form, the Java Byte Code. The Java Virtual Machine (JVM) reads and executes the byte code. In addition the JVM is responsible for downloading and verifying byte code from local and remote sources. The virtual machine checks access rights to class fields and methods, provides links to native code libraries and even implements security monitors for further limited access.

Formal Methods

"Formal methods" is a term that refers to the application of formal mathematical models to computer systems and subsystems. The intent of this book is

to provide a forum for the presentation of a variety of approaches to formal specifications, execution models and analysis of Java programs.

There are several styles of formal methods, a few of which are used in this book. The most common approach to specifying the meaning of a program currently in use is operational semantics. The purpose behind an operational semantics is to provide an abstract model of the internal state of the computer (as referenced by the program) and to specify the modifications of that state with respect to program statements and expressions. A typical semantic clause could be of the form:

$$\frac{< c, \sigma > \rightarrow < c', \sigma' >}{< c; c_1, \sigma > \rightarrow < c'; c_1, \sigma' >}$$

This clause states that if partial execution of command c while in state ?, will result the remaining command c' to be executed in state ?', then the semantics of the sequential composition of c and c1 will behave similarly.

Another type of semantics used in this book is denotational semantics. In this form of semantics, each statement, expressions and other programming language constructs are mapped into functions. These functions are defined as mapping semantic domains to semantic domains. These domains may represent anything from the basic data values stored in variables to the effects of complex recursive functions on the state of the system.

Acknowledgements

I would like to thank all of the contributors to this volume, who patiently waited for me to assemble this document. They also helped edit and review each other's work as well as their own. In addition, I would like to thank the following reviewers for their help: B. Auernheimer, J. Buffenbarger, P. Ciancarini, C. Pusch, P. Sestoft, E.T. Schubert, A. Sobel, and A. Wabenhorst.

Jim Alves-Foss