A Practical Crash Course in Java 1.1+ Programming and Technology: Part I

by George Crawford III

Introduction

Overview

In this column and subsequent Objective Viewpoint columns we will explore several features of the Java environment. Our primary concern will be the most recent Sun Java release, which is presently the Java Development Kit (JDK) 1.1.3. Specifically, we will investigate current topics such as how to construct software components with JavaBeans, create interfaces with the Java Foundation Classes, and learn how the Java environment enables software developers to readily implement sophisticated applications for heterogeneous distributed computing using tools and utilities such as Remote Method Invocation (RMI), Java Interface Definition Language (JavaIDL), JavaSpaces, and Servlets.

Java language support has experienced rapid incremental growth since its alpha release in May, 1995. The first iteration of the Java platform, JDK 1.0, contained only eight standard packages and two hundred eleven classes and interfaces. In contrast, the current Application Programming Interface (API) consists of twenty-two standard packages and four hundred seventy-seven classes and interfaces, and more are under development. It is hoped that the Java articles presented here will establish a practical foundation that will prepare you for further investigation into the ever-growing Java knowledge base.

In this particular column, we will review Java history, its most prominent features, the Java runtime environment, and explore the language and some of its constructs. No prior experience with Java is expected; however, a sufficient grasp of object-oriented concepts and some experience with an alternate object-oriented language such as C++, Eiffel, or Smalltalk is helpful. The tutorials assume that you have already installed the JDK 1.1.3 available at http://www.javasoft.com/products/jdk/1.1/.

History

Java evolved from a research project conducted by a small group of software engineers at Sun

Microsystems, Inc. This team of six, nicknamed Green, convened in 1991 to discover how to enable disparate consumer devices, such as TV set-top boxes, VCRs, wristwatches, vacuum cleaners, roasters, toasters, and other ordinarily boring electronic equipment to communicate with one another. The team also wanted to create a software development and runtime platform that could seamlessly migrate across heterogeneous architectures. At first, C++ was the language of choice. However, the developers soon realized that, even with extensive compiler augmentation, the language remained inadequate to implement their vision. James Gosling, tired of beating a dead horse, sat down to design a new programming language and platform that would provide a common software development environment for all computing.

What emerged from the efforts of Gosling and other Green members was a simple programming language and interpreter called Oak (named after a tree outside Gosling's office), the Green OS, and a standard Graphical User Interface (GUI), all integrated into a small hand held remote control device dubbed "*7" (pronounced "star-seven"). The Green team, incorporated as FirstPerson in 1992, set out to discover markets for their invention. In 1993, the Sun subsidiary submitted a bid in response to a Time-Warner Request For Proposal (RFP) for interactive cable television technology. Although Sun was positioned as the clear winner, SGI, due to what was characterized by Gosling and Naughton as political maneuvering, won the deal.

The company continued to pursue the set-top box arena until 1994. At this time, the World-Wide Web was just beginning to receive global acclaim. The Oak language and platform, which later collectively became known as Java because of trademark infringement concerns, comprised many features, such as security and platform-independence, that were desirable in a heterogeneous distributed system such as the Internet. As a proof of concept, Jonathan Payne and Patrick Naughton developed WebRunner, a web browser with the advanced capability of securely downloading and executing compiled Java code from across a network. At SunWorld '95, Sun formally presented HotJava, the WebRunner descendent, and Java [3]. Only a few months later, Netscape Communications, then a newcomer to the web arena, incorporated Java into version 2.0 of its web browser, planting the seeds for an Internet revolution.

Features

Object-Oriented

Unlike its predecessor, C++, which is hybrid (i.e., has support for both the functional and object-oriented programming paradigm), Java is a pure object-oriented language. Except for primitive data types such as characters, integers, and floats, everything in Java is either a class or an instance of a class. A programming language is defined as object-oriented *only* if it provides support for data types with local hidden state (objects), consists of objects which have types (classes), and includes inheritance [2]. Java meets all of these criteria, and includes some extra object-oriented features that we'll explore when we discuss the Java language.

Platform-Independent

Java ensures this promise of complete executable code portability through the use of machine-independent *bytecodes*, fixed-sized primitive types (integer, float, etc.), big-endian byte ordering, and a standard class library.

Java programs are compiled to an intermediate file that consists of concise language constructs called bytecodes. The bytecodes resemble traditional assembly language instructions, except that these instructions are targeted for a single platform, the Java Virtual Machine (JVM). The JVM interprets and executes compiled Java code, just as an Intel or Motorola processor interprets and executes compiled C code.

Additionally, Java uses fixed-size primitive datatypes which are the same bit-length across computer architectures. Single and double precision numbers in Java comply with IEEE standards. Characters are 16 bits in length for UNICODE support which permits developers to write Java code in their native tongue or write applications that span multiple foreign languages.

The creators of the Java platform have decreed, "The first shall be last, and the last shall be first." Java stores all data in big-endian (network order) style, meaning that the Most Significant Byte (MSB) comes first. The alternative approach is to store the MSB last. Which is the correct way? There are numerous arguments for both approaches and the battle wages on. The Java developers sided with the big-endians, and rightly so. Java is a network-oriented language. TCP/IP and other communication protocols specify a big-endian *network byte order* [4].

Finally, Java is packaged with a class library that has a consistent facade across platforms. The same classes and their respective methods used on Windows can also be used on MacOS, Sun Solaris, OS/2, and other small or large operating systems. Similar efforts have been underway for other languages such as STL (Standard Template Library) for C++ and POSIX (Portable Operating System Interface), but, while still very useful in their domains, they have never achieved the level of sophistication and support attained by the Java class library.

The primary benefit for developers and end users alike is that the JVM might reside on a Personal Digital Assistant (PDA), cellular web phone, or a desktop computer, but, given that the JVM is implemented consistently on each platform, the runtime behavior of the code is the same.

Pointless

No, this doesn't mean you should discard your Java books, stop reading this article (although I'm amazed you made it this far), and start rehearsing C++ idioms. By pointless I mean that Java has removed the most prominent Achilles' heel of students and professional programmers throughout the history of C/C++ sovereignty - pointers.

This is not to say that pointers are a bad thing. Pointers are an invaluable mechanism for communicating

with hardware. Unfortunately, some programmers aren't very good at it, and even masters of C/C++ programming science have been thwarted by boggled machines because of it. Numerous problems are associated with pointer manipulation, especially with memory management. Large applications require extensive resources. Explicit manipulation of these resources with pointers can be a daunting task. Even with very cautious programming efforts, memory leaks and dangling pointers can subtly exist in a software product and appear at the most inopportune time, usually at some point after the release date (when users do all those crazy things programmers would never think of).

Java developers are free from the majority of memory management worries. They do not have to concern themsleves with the deallocation of allocated objects (unlike in C++ where memory consumed by an object is freed inside the destructor method). This is because the JVM executes in parallel with an automatic storage management system, commonly referred to as a *garbage collector*. Sometime after a reference to an object disappears, the garbage collector performs a *mark-sweep* scan of the dynamic memory areas, whereby referenced objects are marked. At the end of the scan, any memory space held by an unmarked object is reclaimed for future allocated objects [3].

Performance

Compiled Java code is structured for compactness, not performance. Java was designed from the start as a network-oriented language (imagine Java agents migrating from one computer to another), so certain concessions were made to make it as concise as possible. Even so, for an interpreted language, Java applications and applets perform reasonably well. This is because the JVM interprets assembly-like bytecodes rather than the actual English-like source code. While program interpretation at this level is still slower than program execution at the machine level, applet and application performance is usually tolerable.

With only direct JVM interpretation, however, Java can only be used to create simple, non-mission critical software products. To overcome this limitation, many vendors have produced Java interpreters equipped with Just-In-Time (JIT) compilers. These compilers translate bytecodes to native machine code on the fly. While some JIT featured JVMs can deliver outstanding performance relative to traditional JVMs, the resultant code is still incapable of matching machine-dependent program execution speeds, primarily since there is no time to perform crucial code optimizations.

A new version of the JVM under production by Sun, codenamed HotSpot, is slated for release this winter. This evolutionary Java interpreter technology is reported by Sun to deliver performance on par with platform-specific program by using adaptive optimization techniques. We will have to wait and see if HotSpot can deliver on its promises.

There are many factors which can affect Java performance; in particular, performance can be evaluated in four elements:

- Software design and programming.

- JVM implementation.
- Garbage collection technology.
- Just-In-Time (JIT) compiler and optimizing compiler technology.

A very well designed software product can improve program efficiency. Java developers who detect poor execution speeds in performance-critical sections of their applications should profile their code to help determine exactly where performance penalties are most prominent and remedy them with better programming technique. Garbage collection research and development has produced better automatic memory cleanup methods that will improve with time. JIT compilers translate Java bytecodes to native machine code on the fly. While this is only partial solution, such tools do provide a better alternative to straight interpretation. The next generation of JITs, such as HotSpot, may greatly accelerate code execution speeds.

Secure

Java is marketed as a solution for distributed application development. In the Java distibuted computing model, bytecodes are transported across a network to a target machine for execution. The migration of executable code across networks raises many security concerns. One does not want arbitrary code to migrate from across the network and execute wildly on one's computer. This is one of the primary areas where Java excels, compared with other distributed application solutions, such as ActiveX.

In the past, Java security has been very rigid. Applet activity was strictly prohibited from performing probable malicious tasks by the JVM resident Security Manager, such as writing to the host's hard disk space. The new Java procurs a flexible security model that allows for more fine-grained access controls as prescribed by the user. Applets that are signed by trusted sources can now be granted higher resource manipulation on a host machine.

Multithreaded

A thread is a single, linear control flow within in a program. Back in the old days (even 4 years ago is a long time in computing terms), nearly all programs were single-threaded. A program would perform its duties one instruction at a time and then (hopefully) return control back to the user. A *multithreaded* application executes multiple program instruction paths concurrently. As an example of concurrency, consider your web browser. Assuming you are using either MS Internet Explorer or Netscape Communicator, you may have downloaded some files to your machine. When a request for a file is made, a pop-up window appears that displays the progress of the file transfer. Meanwhile, you are still able to visit other sites on the web. You did not have to wait for the entire file to download before using the browser again.

The Java language and class library provide standard, cross-platform support for thread creation, manipulation, and *synchronization*. Thread programming in Java is made much easier than in C or C++. Thread synchronization, perhaps the most difficult aspect of a multithreaded application, is essentially a

no-brainer in Java.

Applications and Applets

A Java program can be written as an *application*, an *applet*, or both. Java applications are similar to traditional software programs such as MS Word™ or DOOM™, except that, as previously discussed, the Java executables are completely portable among platforms (MS Windows, MacOS, Sun Solaris, Linux, etc.). Also, as with conventional software, Java programs developed as applications are granted unrestricted access to a host machine.

Java applets are specially designed programs intended to be downloaded and executed on an arbitrary Java-equipped platform. You may already be familiar with applets that are transferred from across a network to a web browser. Most sophisticated web browsers, such as MS Internet Explorer, Netscape Communicator, or HotJava fully support Java applets. However, a web browser is not the only medium through which an applet can be presented. Other possibilities exists, such as Internet-enabled *kiosks* or network-oriented game machines.

Java programs can also be designed as both an application and an applet. Java software developed this way can then be run as either a standalone program or distributed across an Internet or *Intranet* for execution on JVM-readied computing devices. A template for writing a Java applet/application is presented later in this column.

Using the JDK

Before we delve into Java language fundamentals, you need to have a basic understanding of the standard Java development environment. The JDK consists of a variety of tools. These tools enable you to compile, execute, and debug Java programs, as well as generate HTML-based documentation from Java source files. The next two sections demonstrate only Java source code compilation and program execution. For information on the other JDK tools, refer to the *JDK Tool Documentation* at http://www.javasoft.com/products/jdk/1.1/docs/index.html#tools

Echo

Java Application

To illustrate the use of the JDK tools, I have included a short Java program. Copy the program below to a file called *Echo.java*. Do not concern yourself with syntax or semantics at this time.

```
//Echo.java
public class Echo {
   public static final void main(String[] args) {
```

```
      for (int i=0; i < args.length; i++)
         System.out.print(args[i], " ");
      System.out.println();
   }
}
```

Compiling Java Programs

The *javac* compiler translates Java source code to platform-independent bytecodes and stores them in a file with a .class extension. The format for running *javac* is as follows:

javac [Options] <SourceCodeFilename>.java

The Java compiler accepts a variety of options; however, they are not required to compile a Java program. As a beginner, you need not concern yourself too much with the available compiler options. We'll use these in later columns. The final argument to *javac* is a legal Java source code file. The Java compiler only accepts files that end with a .java extension and will complain otherwise. To compile the Echo.java program, type the following at your OS command prompt and press Enter:

javac Echo.java

If you copied the program correctly and have the Java environment set up appropriately, then the *javac* compiler should return with no error messages.

Running Java Programs

You are now ready to run the executable bytecodes produced by the Java compiler. After compiling the *Echo.java* program, you should have another file in the same directory called *Echo.class*. This file contains the bytecode version of the *Echo* source file. You can execute the program by using the *java* runtime interpreter, an implementation of the JVM. The format for running *java* is nearly the same as for *javac*:

java [Options] <BytecodeFileName> [ApplicationParameters]

Like *javac*, the *java* runtime interpreter accepts many options. The second argument, *BytecodeFileName* is the name of the file that contains the Java application bytecodes, the .class file produced by *javac*. Notice that the final argument, *BytecodeFileName* does *not* include an extension. This is because the JVM automagically appends a .class extension to the filename. The optional *ApplicationParameters* argument is a list of parameters that are to be passed to the Java application. The use of command line arguments in a Java application is demonstrated below.

To run the application, type the following at your OS command prompt and press Enter:

java Echo Do I hear an Echo?

Echo is the name of the file produced by the java compiler. As shown, there is no need to include the extension. The series of words following the application name words is a list of arguments to be passed to the *Echo* application. All Java applications accept command line arguments, although the arguments are not used unless the application is explicitly programmed to do so.

When you execute the program, the following output should appear:

Do I hear an Echo?

The *Echo* application was programmed to iterate through the command line arguments and print them to the standard output. We'll learn more about how this is done in a future column.

Java Language

Comments

Program documentation is an important element in effective software maintenance. Even a short program should be properly documented, as the intended behavior of the application may not be immediately obvious at a later date. Java provides three types of comments:

Single-line: // *Comments go here*

Multi-line: /* *Comments go here* */

Documentation: /** *Comments go here* */

Single-line comments are commonly used to document variables or short sections of code. The text follows the two forward slashes and the comment terminates at the end of the line. Multi-line comments span multiple lines. This is typically applied to program or class method documentation. Documentation comments are used to later generate HTML descriptions of class and method declarations in a Java program. The *javadoc* tool is supplied with the JDK for this purpose. [1]

Primitive Datatypes

Although Java is based on the notion of objects, the language does include a set of *primitive datatypes* that are analogous to the basic types found in other programming languages such as C, C++, and Pascal. However, unlike the basic types in C or C++ which may have be of varying bit-length from one platform

to another, Java datatypes have the same size regardless of the underlying computer architecture. Table 2 lists the Java primitive datatypes with a brief description of each.

Table 1: Primitive Datatypes

Type	Description	Type	Description
boolean	**true/false**	**int**	**32-bit integer**
long	**64-bit integer**	**byte**	**8-bit integer**
char	**16-bit unsigned integer**	**float**	**32-bit IEEE 754 single-precision number**
short	**16-bit integer**	**double**	**64-bit IEEE 754 double-precision number**

The *char* type in Java has a sixteen bit length to support the *Unicode* character set. Unicode can represent up to 34,168 characters. This is beneficial for programmers in non-English speaking countries (programs can be written using their native language). It is also useful for developing cross-cultural applications.

Naming Conventions

There are certain rules and conventions used when developing Java applets and applications. Some are strictly enforced by the Java compiler, others, such as coding style, are *de-facto* standards adopted by the majority of the Java programming community.

There are three restrictions for declaring identifiers:

1. Must begin with an uppercase or lowercase letter, an underscore (_), or dollar sign.
2. May contain numbers after the first character.
3. Cannot be a Java keyword.

Additionally, there are certain stylistic rules which are commonly practiced by Java programmers. These conventions are listed below with examples:

- Class names should begin with an uppercase letter, with subsequent words also beginning with a capital letter:

  class MyClass ...

- All variables should begin with a lowercase letter, with subsequent words beginning with an uppercase letter:

MyClass myClassInstance;
int dataCount;

- Package names are typically declared with all lowercase letters:

  package msu.util;

- Use underscores to seperate two distinct words where the first word ends and the second word begins with an uppercase letter:

  Instead of: JVMImplementation ...
  Use: JVM_Implementation ...

- Code blocks begin on the same line as the statement that starts the block and end just after the last block statement:

  Example 1:

```
for (int i = 0; i < count; i++) {
    ...  // code in loop block here
}
```

  Example 2:

```
public final void send(Object[] data) {
    ...  // code in method block here
}
```

- Indent the lines following the start of a code block at least four spaces:

```
public static void main(String[] args) {
    int count = Integer.parseInt(args[0]);
    for (int i = 0; i < count; i++)
        System.out.println(i);
    }
}
```

- Public members appear before protected members, before private members (discussed in detail later):

```
class ProcessGroup {
```

```
    public ProcessGroup() {}
    public void addProcess(Process process) {}
    ....
    protected Hashtable processes;
    ....
    private int id;
}
```

Conclusion

This ends part one of a two part introduction to Java. In the next issue, we'll look at classes and objects, packages, interfaces, and more!

References

1

Arnold, Ken and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, Mass., 1996.

2

Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Co. Redwood City, CA. 1994.

3

Lindholm, Tim and Frank Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, Reading, Mass., 1996.

3

O'Connel, Michael. Java: The Inside Story, Web Publishing, Inc. July, 1995.

4

Stevens, W. Richard. *UNIX Network Programming*, Englewood Cliffs, New Jersey, 1990.