What's Perking? An Introduction to Java

John Jay King
King Training Resources, USA

Abstract

Java has quickly become a standard tool for developers. Java's promise of write once, run anywhere is extremely attractive, if somewhat elusive so far. Java can support back-end processing on the server or GUI applications delivered to the client using most browsers. This session will discuss various development environments such as Sun's SDK, Oracle's JDeveloper, and IBM's Visual Age. The session will also discuss various types of Java programs including: Applications, Applets, Servlets, and Java Server Pages. Attendees will learn the vocabulary and concepts necessary to take the next step in learning about Java and how to begin creating applications.

Introduction

Java is a programming language created and owned by Sun Microsystems. The computing industry has adopted Java quickly, unfortunately, no ANSI or ISO standard exists since Java is still a proprietary product of Sun's. So far, the different versions of Java are denoted using Sun's criteria.

Java promises that code can be "write once, run anywhere" and generally delivers on the promise. Compiling to machine-specific code is the best way to maximize performance of computer programs, unfortunately, machine-specific code cannot be shared easily between environments. The Java compiler creates an object file called **bytecode** that is portable to any platform that supports a Java Virtual Machine (JVM). Using an interpreter specially constructed for each execution environment means that Java code can be written without regard to execution-time specifics. Java's native datatypes and methods are the same in all environments so code does not require change when ported. Java's interpretive approach does not yield the best performance solution, but, it maximizes portability. Some environments are beginning to provide Just In Time (JIT) compilers that create optimized machine-specific code on demand. Just In Time compile still not the best possible performer, but, offers significant improvement if code repeats.

Unfortunately, Java is changing so rapidly that true independence is limited to execution environments that support version-specific features. Today, the two most heavily used browsers (Netscape Navigator and Microsoft Internet Explorer) support Java as does Sun's Hot Java, but, only the latest versions support the newest Java features limiting portability.

The bytecode created by the Java compiler is generic and transportable to any Java environment that supports a Java Virtual Machine (JVM). The JVM specific to the environment compiles the bytecode and generates machine-specific execution code. JVMs have been available for every major execution environment for several years. Java enabled web-browsers include a JVM. A JVM may also be installed on a computer using software included in Sun's Software Developer's Kit (SDK) or in earlier releases the Java Developer's Toolkit (JDK). The JVM also supports the runtime environment including the construction of class objects, garbage collection, optimization, and space management. A document describing the specification for the JVM is available from the Sun Microsystems Java site (http://java.sun.com). This URL is also where one should seek official documentation about Java, white papers, sample code, and downloadable versions of Java software. Two products that should be downloaded by those interested in experimenting with Java are the Software Developer's Kit (SDK) and Java Enterprise Edition (J2EE). The documentation should be downloaded as well. Fortunately, many excellent books have been written about Java and its use. Seek your favorite supplier of technical books and look at a few.

Java Concepts

The following code represents the basic "Hello World" program as written in Java.

```
/**
 * Java application version of "Hello World" program
 *
 * @version 1.01 04/01/2000
 * @author John Jay King
 */
public class Welcome
{
  public static void main(String[] args)
  {
    System.out.println("Hello SUPER STUDENT, welcome to Java!");
  } // end of main method
} // end of Welcome class
```

The sample program above may be read as follows:

```
/** ... */                                  Comments (not required)
class Welcome                               Class name
{                                           Begin class
public static void main (String[] args)     Method head
{                                           Begin main method
System.out.println("Hello SUPER STUDENT, welcome to Java!!");
                                            Print text
}                                           End main method
// end main method                          Comments (not required)
}                                           End class
// end class Welcome                        Comments (not required)
```

In Java a class name is REQUIRED to match the source file's name. In the code above the class name is Welcome, the source code file name MUST be Welcome.java (note: Java names are case-sensitive). The bytecode file created by the compile will be named Welcome.class (again, names are case-sensitive).

Java programs come in several variations, four of particular interest: Application, Applet, Servlet, and Java Server Page. Applications (like Welcome.java) are stand-alone Java programs that execute without a browser (but still require a JVM). Java applications are not subject to applet security restrictions and may perform file I/O and other activities. Java applications require a method named **main** since there is no browser; stand-alone applications must have a starting point (**main** is ignored when a program is executed as an applet). Applets are Java programs attached to web pages that run in browsers. Unlike stand-alone applications, applets require a Java-enabled browser in to run. Applets do not require (or use) a **main** method. Java applets must conform to a rigid security model that forbids or limits access to system resources (more discussion on security later). Caution! Only the latest browsers support Java 1.2 features and some Java 1.1 features might not work the same way on different browser implementations. A Java-enabled browser recognizes a special HTML tag **applet** as shown below:

```
<applet code=WelcomeAppl width=600 height=200></applet>
```

In the line above, **applet** is the HTML tag, **code** provides the name of the applet class file, **width** and **height** control the size of the applet on the browser display. Java Servlets are programs that reside on a web server and are most frequently used to dynamically create web page HTML. Servlets do not include a **main** method, they begin with a function named **Init** instead. Java Server Pages are specialized Servlets that simplify and provide a model for creating dynamically created web pages.

A Java **class** represents a collection of Properties (called Variables) that describe an object and the Actions (called Methods) that may be performed upon that object. Each class represents a program unit that may be used again and again.
A Java program typically uses several classes to accomplish its job. Classes usually correspond to some object in the real world for instance: Invoice, Customer, Person, Student, or Purchase Order. A class includes definitions of all variables (properties) that are necessary to work with an object and the methods (actions) that control all activity with the object's variables. A Class may be thought of as a blueprint, providing a model from which one or many actual objects (instances) may be created (instantiated). For example you might think of a Cookie Cutter as a Class, and an actual Cookie would be referred to as an Object, Instance, or Instantiation. Class Properties (variables) describe the data structure of the class, Class Actions (methods) provide the software routines needed to manipulate the class and its properties. Methods are the program code that make classes work; they are used to initialize variables, get variable values, and set variable values. Variables and Methods may be described as public, private, or protected. **public** members (Variables or Methods) are available to any bit of code that accesses the class. **private** members may only be used by Methods that are part of the same class. **protected** members are same as **private** except that they may also be **inherited** by subclasses that extend the class. The collection of Properties and Actions into one unit is often referred to as Encapsulation.

Java allows the use of one class declaration, known as a **superclass** or **base clas**s, as the basis for the declaration of a second class, known as a **subclass** or **derived class.** For example: an Employee class that describes a company's employees and a Sales class, based on the Employee class describing employees in the sales department. This supports the Object-Oriented concept known as **Class Derivation** or **Inheritance**. Just like the cherished heirlooms you hope to inherit from your granduncle Marty, a subclass/derived class inherits all of the **non-private** data members and member methods from its superclass or base class. The code looks something like this:

```
class Derived extends Base {  //<- Base and Derived are not reserved
                              // words, use any valid class names here ...
```

Derivation chains work simply. Suppose three classes are defined as A, B, and C, where class B extends A and C

extends B. Class A has Variables and Methods. Class B extends A so the Variables and Methods it defines are added to the inherited Members and Variables from Class A. Class C extends B so Variables and Methods it defines are added to those inherited from Class B and those that Class B inherited from Class A. Remember, only non-private members (public and protected) are inherited. Typically, Variables will be defined as private (cannot be inherited) or private (may be inherited) and Methods are usually defined as public.

Java was designed specifically to prevent programmers from causing problems in computers other than the one where the code is executing including: No direct manipulation of memory or pointers is allowed; All code is interpreted by the JVM and not executed directly; Java's security manager checks all actions against the Java run-time library to ensure that no compiler alteration has occurred; Applets execute within a security "Sandbox" ensuring that local code is trusted to have full to system resources while downloaded code is not trusted and has limited access as provided inside the sandbox. Applet security in particular must exist within a "Sandbox" here are the original rules: Applets cannot execute local programs; Applets may not read or write files on the local computer; Applets have access only to Java-specific information from the local computer, specifically, they cannot access user name, email address, or other personal data of the computer user's; New windows "popped-up" by the applet display a warning message; Applets cannot communicate with any server other than the one where the applet was downloaded from ("originating host"). In recent versions Applets may be "signed" for added security and flexibility.

Syntax Overview

Java syntax is based upon C++ syntax. If you have a background in C++ (and to some extent C) then you will be familiar with the basics of Java programming syntax with only minor differences to consider. If you are not familiar with C++ you should look for Java training/books that offer the basics of the syntax in addition to the programming concepts of Java itself. C++ is not a prerequisite to learning Java, but, the syntax should be learned as part of your Java experience too. Those accomplished with C++ will be able to quickly pick up Java's syntax.

Java allows comments in two ways: Multi-line comments starting with a slash followed by an asterisk (/*) and ending with an asterisk followed by a slash (*/), and Single-line comments using two slashes (//). Single-line comments may be nested within /* */ comments.

Java has eight simple (or primitive) types of data: Six are numeric data types, Two are character data types. Java allows complex data to be defined using classes, these are also called "reference" data types. Perhaps the most common reference data type is the String class used to hold character string data. The simple (primitive) data types in Java are: Integer (**byte**, **short**, **int**, **long**), Floating-point (**float**, **double**), Character (**char**), and Boolean (**boolean**). Java simple data types are the same size across all platforms, so, there is no reason to have code that changes depending upon local size limitations. The char data type defines a single character, though the character is represented as a Unicode value (two bytes) internally. Unicode is an international standard for multi-byte character data that incorporates virtually all of the world's major languages. Traditional European languages use a limited single-byte character set that fits quite nicely into the ASCII/ISCII encoding scheme. Many languages (Japanese for instance) use a more complex set of characters and a single byte (256 possible values) is simply is not large enough to encode the variety of characters needed. The Unicode Worldwide Character Standard is designed to accommodate all of the world's primary languages by supporting a multi-byte encoding scheme that contains 38,885 distinct characters from 25 languages. Boolean values are two-valued variables that are either **true** or **false** (true and false are reserved words in Java).

Objects and arrays provide complex (non-primitive) data types in Java. Primitive datatypes are often called "non-reference" types since data is passed "by value" to methods. "by value" means that a variable is copied in memory and the actual value of the copy is made available to the method when passed as an argument. Objects and arrays are often called "reference types" because they are handled "by reference." Objects are also called "abstract data types" (ADTs) or "user-defined data types." Classes are used to create objects (instances of a class). One of the greatest strengths of Java is the set of built-in classes included in the Java API, to see a list: 1) Look in the Java SDK Documentation; 2) Look under API & Language; 3) Select Java 2 Platform API Specification; 4) Look at the list of "All Classes."

In many languages, programmers who allocate memory dynamically are responsible for tracking the memory and releasing it properly when no longer used. Failure to properly release memory causes the "memory leaks" that so often plague certain programming environments. Java is specifically designed to avoid memory leaks by cleaning up after itself, this mechanism is called **Garbage Collection** (GC). The Garbage Collector periodically "wakes up" and begins looking for unused memory to free. Objects are considered for Garbage Collection when they are no longer referenced by any variables, nor in any object variables, nor in any array element. Java's Garbage Collection may cause a brief performance slow-down when executing but the advantages of automatic Garbage Collection outweigh any performance issues involved.

Strings are perhaps the most frequently used of Java's built-in classes. Quoted strings are instances of the String class.

```
String abc; // uninitialized string
String def = "Cookie"; // string initialized to "Cookie"
String ghi = ""; // empty string
abc = "Monster"; // assign abc
```

Like all classes, String has many methods (see Java documentation for complete information). Java supports the concatenation of strings two ways: using the plus (+) sign operator and concat method.

Java provides an assignment operator to place values into variables. Assignment uses the equal (=) sign and places a copy of the value to the right of the equal sign into the variable to the left of the equal sign:

```
abc = 1; // assigns the value of 1 to the variable abc
def = abc; // assigns the contents of abc to def
firstName = "Moon Unit"; // assigns "Moon Unit" to variable
```

Assignment statements must be consistent with data types. The assignment operator may also be used at the end of a data declaration:

```
int xyz = 1; // Sets value to 1
char letterA = 'a'; // Assigns letter a to char variable
String lastName = "Zappa";
```

Java provides five mathematical operators for use in calculations: + Addition a =b +c; - Subtraction a =b -c; * Multiplication a =b *c; / Division a =b /c; and % Remainder a =b %c; (the result of a =10 %3;is 1). Java also provides brief variations for the most common operations: **ctrVar = ctrVar + 1** may be rewritten as **++ctrVar** (Increments ctrVar by one before using it in the current statement); or **ctrVar++** (Increments ctrVar by one after using it in the current statement). Other operators include: **- - ctrVar** (Decrements ctrVar by one before using it in the current statement); **ctrVar- -** (Decrements ctrVar by one after using it in the current statement; **varabc += 10** (varabc = varabc + 10); **vardef -= varghi** (vardef = vardef - varghi); **varjkl *= 0.5** (varjkl = varjkl * 0.5); and **varmno /= 2** (varmno = varmno / 2).

Relational operators return a boolean value of **true** when the comparison is True, and **false** when the comparison is False (true and false are reserved words). Relational operators are processed AFTER arithmetic takes place. The relational operators supported by Java are: > Greater than, < Less than, >= Greater than or equal to, <= Less than or equal to, == Equal, **!=** Not equal, and **!** Unary NOT (used for negation). Relational operators used for combined tests: || Logical OR and **&&** Logical AND.

Compound statements are one or more simple statements surrounded by braces **{ }**. Compound statement is another term for code block. Compound statements can go anywhere a single line is allowed. The example below illustrates a compound statement (code block):

```
//...
double  hourlyRate;
int     regHours;
int     otHours;
double  payBeforeTaxes;
char    exempt;
//...
if (exempt == 'N')
{
  payBeforeTaxes = (hourlyRate * regHours) +
                   ((hourlyRate * 1.5) * otHours);
  System.out.println( "Pay=" + payBeforeTaxes);
}
//...
```

Compound statements are often used in conditional statements like **if**:

```
if (comparison expression)
    statement1;
else
    statement2;
if (comparison expression)
{
    statement1;
    statement2;
}
else
```

```
{
    statement3;
    statement4;
}
```

If a repetitive test is necessary against a byte, char, short, or int variable, the **switch** statement can make the coding a lot easier:

```
switch( integer_expression )
{
    case constant:
        statements or code blocks;
        break;
    case constant:
    case constant:
        statements or code blocks;
        break;
    default:
        statements or code blocks;
        break;
}
```

Sometimes it is useful to leave a loop or switch without completing the rest of the code (see previous example). The **break** statement is used to exit from a switch statement or a loop.

Java provides three methods to cause program loops: **while** (Loop while condition is true, might never execute); **do while** (Loop while condition is true, always executes at least once); and **for** (Loop using automatic increment/decrement until limit is reached, might never execute).
while causes the conditional execution of a statement or a group of statements as long as a stated condition is true. The ***do...while*** construct is similar to the ***while***, it just puts the test last rather than first guaranteeing that the loop's code is executed at least once. ***for*** loops offer automatic incrementing (or decrementing) of a control value while processing the loop.

```
for (initial_value; true_test; incrementor)
{
    /* Java statements go here */
}
```

The initial value is set before the loop begins. The true_test is a conditional expression, like those in ***if*** or ***switch*** (without parentheses); the test is performed each time the loop iterates. If the test is true the loop is executed, if the test is false the loop stops executing (or does not execute if the first time). The incrementor “bumps” after the loop body is executed.

An array is a group of field values of the same type stored one after another in the computer. Java allows the processing of an array as a unit, or the processing of individual items in the array. Arrays are often called tables. Data values in an array may be individually addressed using an ***index*** or ***subscript***. The first item in an array uses the index/subscript zero [0], the largest useable subscript is the array size - 1. Indexes/subscripts may be literal values, expressions, or integer variables. Index/subscript values/variables must be enclosed in brackets “[]”when used. Once an array's size has been established, it may not be changed. To have a variable-length array, create an object of the Vector class. Vectors are included in the standard Java library and provide more flexibility than arrays. Unlike conventional arrays, vectors do not have a fixed size, vector size may shrink and grow as needed. Entries may also be easily inserted into the middle of a vector or deleted from the middle of a vector (or either end). Check the Java SDK API documentation for more information about the Vector class, you'll find that it includes various options for handling arrays.

One of Java's greatest strengths is the sophistication of the exception handling that is available. Java's developers learned from earlier languages and came up with a mechanism that is easy to use, comprehensive, and complete. Applications and Applets that are properly designed make use of Java's exception handling to reduce errors and ease debugging. Exception handling is made up of a set of statements and code blocks: **try** (The try block identifies the block of code to be "checked"); **throw** (Statement purposely causes an exception, many of Java's built-in methods throw exceptions as part of their design); **catch** The catch block(s) contain code to handle or at least react to the exceptions that have been thrown); **finally** (The finally block contains code that is so important that it should be executed even if a try block does not complete normally due to throwing an exception). Java's exception handling allows us to notify the user an error has occurred, attempt to save work, and exit gracefully. Programs surround potential-problem code with a try block, this allows the program to "catch" any error thrown within the try block. To reiterate: **throw** is how an exception occurs, some program, somewhere, throws it; **catch**

blocks are where exceptions are handled in a program. **catch** blocks must immediately follow the try block that will throw the exceptions the catch block should handle. Exception handling code is simply whatever code is inside your catch block associated with a particular exception. A method can deal with an exception in one of three ways: Handle the exception in a catch routine and do not list the exception; List the exception in the throws portion of the method header and do not attempt to handle the exception; Handle the exception in a catch routine, but throw the same exception as part of the handling (perform local handling, then, pass exception to higher level handlers), this requires that the exception be listed in the throws portion of the method header also.

As mentioned before, classes may be extended and they may extend other classes. By design, Java classes may only extend (inherit from/be derived from) one class. Sometimes, it is convenient to create a special kind of class that is designed to be derived from (extended) and will never be directly instantiated; these special classes are called Abstract Classes, or **Interfaces**. An abstract class requires that classes that extend it overload and replace at least one of the abstract class's methods. To use the code of an interface, a class **implement**s the interface. In this fashion, multiple inheritance (something specifically avoided by Java's developers) can be used in a limited fashion.

One of the reasons Java caught on so quickly is that the Graphical User Interface (GUI) has pretty much the same "look and feel" regardless of what platform you use it on. Originally, Java offered the Abstract Window Toolkit (AWT) that supports basic GUI programming. Unfortunately, in order to be as portable as possible (Write Once Run Everywhere) the AWT adopts a least-common denominator approach and applications are not as rich as those that are native to the platform. Java 1.1 and Java 2 support a Java Foundation Class (JFC) library usually called Swing. Swing is actually a light subset of the JFC. Swing applications are less intertwined with the host GUI and therefore are smaller, faster, more-portable, and richer. Unfortunately, Swing may not be available in down-level browsers so many applets today still use AWT.

Java Database Overview

Sun released the first version of JDBC (Java DataBase Connectivity) in the summer of 1996. JDBC allows programmers to use SQL to **Connect** to a database, **Query** a database, and **Update** a database. JDBC programs are database vendor and platform independent (mostly). JDBC2 was released in 1998 and is not fully supported in all environments yet. JDBC allows connections to all of the major database vendors: Oracle, IBM DB2, Microsoft, and others. Some vendors (e.g. Oracle and IBM) now build a JVM into the database engine to support Java Stored Procedures and Java-based access. JDBC is generic. Even though JDBC is generic, you may find it is best to obtain JDBC drivers from the database vendor rather than using the ones from Sun. Your application uses the JDBC driver, and the JDBC driver communicates with the database.

The Open DataBase Connectivity (ODBC) standard has been in place much longer than JDBC. ODBC was originally developed by Microsoft and has been supported by all of the major database vendors. Again, you are probably best served by obtaining drivers from the database vendor. Sun offers a JDBC-ODBC "bridge" that allows Java JDBC drivers to connect via ODBC to any ODBC-compliant product. This greatly opens up the available products for application use.

Oracle Java Overview

Oracle8i includes a Java Virtual Machine (JVM) specifically engineered by Oracle to provide multi-threaded support of Java applications instead of having separate JVMs for each bit of Java. This means greatly improved performance for Java applications running with database code. Oracle also supports the creation of stored procedures using Java in addition to those created with PL/SQL.

Java support for programming includes: Java stored procedures (functions and packages, no triggers), Enterprise Java Bean 1.0 support, and support for CORBA 2.0 standards. Java support for SQL includes: JDBC and SQLJ. SQLJ statements are translated by an SQLJ Preprocessor before Java code is submitted to JDBC. Direct JDBC support is more complex, but, yields more control.

Oracle8i Release 2 (8.1.6) dramatically improves Java support including: Java 2 (JDK 1.2) support, JDBC 2.0, multi-byte character support, debugging support, performance improvements, and support for Advanced Queueing. The new release also supports an XML parser implemented with Java.

Oracle9i announced in September 2000, but, not yet production promises expanded Java support.

Is Java replacing PL/SQL? Not any time soon. Today, code that requires database interaction will run faster using PL/SQL than using Java. However, code that requires no database interaction such as text manipulation will probably run faster in Java than in PL/SQL. As time goes by, Oracle will steadily improve the efficiency of Java. But, for the foreseeable future PL/SQL will be the language of choice for many Oracle database stored program units.

Conclusion

This paper has presented an introduction to the Java language, its concepts, and its syntax. It has also introduced the essential facts about Oracle's support for Java in the current database product. Java may be a relatively new language, but, its rapid expansion throughout the information industry and specifically inside Oracle's product mix makes it very important. Each person involved in the creation and deployment of Oracle tools should become familiar with Java and learn what parts of Java will directly impact the applications they work with.

About the Author

John King is a Partner in King Training Resources, a firm providing instructor-led training since 1988 across the United States and Internationally. John has worked with Oracle products and the database since Version 4 and has been providing training to application developers since Oracle Version 5. He has presented papers at various industry events including: IOUG-A Live!, UKOUG Conference, EOUG Conference, ECO, SEOUC, RMOUG Training Days, and the ODTUG conference.

John Jay King
King Training Resources
6341 South Williams Street
Littleton, CO 80121-2627
U.S.A.
Phone: 1.303.798.5727 1.800.252.0652 (within the U.S.)
Fax: 1.303.730.8542
Email: john@kingtraining.com

If you have any questions or comments, please contact me in the fashion most convenient to you. Copies of this paper are available for download from King Training Resources upon request (www.kingtraining.com).

Bibliography

Oracle8i JDBC Developer's Guide, Oracle Corporation

Oracle8i Java Stored Procedures Guide, Oracle Corporation