An Introduction to the Java Programming Language

History of Java

In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices (cable boxes, *etc.*). They wanted the language to be small and use efficient code since these devices do not have powerful CPUs. They also wanted the language to be hardware independent since different manufacturers would use different CPUs. The project was code-named *Green*.

These conditions led them to decide to compile the code to an intermediate machine-like code for an imaginary CPU called a *virtual machine*. (Actually, there is a real CPU that implements this virtual CPU now.) This intermediate code (called *bytecode*) is completely hardware independent. Programs are run by an interpreter that converts the bytecode to the appropriate native machine code. Thus, once the interpreter has been ported to a computer, it can run any bytecoded program.

Sun uses UNIX for their computers, so the developers based their new language on C++. They picked C++ and not C because they wanted the language to be *object-oriented*. The original name of the language was *Oak*. However, they soon discovered that there was already a programming language called Oak, so they changed the name to *Java*.

The Green project had a lot of trouble getting others interested in Java for smart devices. It was not until they decided to shift gears and market Java as a language for web applications that interest in Java took off. Many of the advantages that Java has for smart devices are even bigger advantages on the web.

Currently, there are two versions of Java. The original version of Java is 1.0. At the moment (Nov. 1997), most browsers only support this version. The newer version is 1.1 (in addition 1.2 is in beta). Only *MS Internet Explorer 4.0* and Sun's *HotJava* browsers currently support it. The biggest differences in the two versions are in the massive Java class libraries. Unfortunately, Java 1.1 applets will not run on web browsers that do not support 1.1. (However, it is still possible to create 1.0 applets with Java 1.1 development systems.)

Basics of Java

Applications

There are 2 basic types of Java applications:

standalone
: These run as a normal program on the computer. They may be a simple console application or a windowed application. These programs have the same capabilities of any program on the system. For example, they may read and write files. Just as for other languages, it is easily to write a Java console program than a windowed program. So despite the leanings of the majority of Java books, the place to start Java programming is a standalone console program, *not* an applet!

applets
: These run inside a web browser. They must be windowed and have limited power. They run in a restricted JVM (Java Virtual Machine) called the *sandbox* from which file I/O and printing are impossible. (There are ways for applets to be given more power.)

Development Tools

There are many development tools for Java:

Sun's JDK
: Sun's Java Development Kit has two big advantages:

1. It is the most up to date.
2. It is *free!*

Its main disadvantage is that it only includes command line tools, no IDE. Many Java books include this on a CD-ROM.

Borland's JBuilder
: It contains an IDE and supports Java 1.1

MS Visual J++
: It contains an IDE, but to my knowledge does not yet support Java 1.1

Symantec's Visual Cafe
: It contains an IDE, but no Java 1.1 support yet.

Compiling and Running a Java Standalone Application

Hello, Java program

Here is a short Java program:

```
/* A simple Java program */
```

```
public class Hello {
  public static void main( String args[] )
  {
    System.out.println("Hello, Java");
  }
}
```

Creating a Java source file

Java source files must end in an **`.java`** extension. The root name *must* be the same as the name of the one public class in the source file. In the program above, the class is named **`Hello`** and thus, the file must be named **`Hello.java`** (Yes, case is important!).

Just as for other languages, any text editor can be used to save the text of the program into a text file.

Compiling a Java source file

Sun's JDK includes a Java compiler named **`javac`**. To compile the above Java program one would type:

```
javac Hello.java
```

If successful, this creates a file named **`Hello.class`** If not successful, it prints out error messages like other compilers.

Running a Java program

To run a standalone program, Sun's JDK provides a Java interpreter called **`java`**. To run the **`.class`** file created above, type:

```
java Hello
```

Note that **`.class`** is *not* specified!

The output of running the above program is simply:

```
Hello, Java
```

Java Programming

Classes and objects

Java is an object-oriented language (like C++). An *object* is an abstract *thing* in a program. Generally, objects are not completely different from each other and can be classified into like groups. The group an object belongs to is called a *class*. Objects in the same class share two attributes:

1. A state space (*i.e.*, the set of all possible states of the object)

speak).

Consider the **`Vector`** class that the Java class library provides. A Vector object is basically an array that grow or shrink in size dynamically. It's state is defined by:

1. The elements stored in their given order
2. The number of elements stored

It's methods include:

`addElement()`
: Adds an element to end of the vector increasing its size by 1

`insertElementAt()`
: Adds an element at a specified position in the vector. The elements above are shifted up one position and the size is increased by 1.

`setElementAt()`
: Stores an element at a specified position in the vector. The previous element value is lost.

`removeElementAt()`
: Removes an element at a specified position. The elements above are shifted down and the size is decreased by 1.

`elementAt()`
: Returns the element at a specified position.

`size()`
: Returns the current size of the vector.

If **`v`** is an object of type **`Vector`**, then a method is invoked on **`v`** by the following syntax:

```
v.method-name(method-arguments);
```

For example, to add another object **`x`** to the end of **`v`** use:

```
v.addElement(x);
```

In an object-oriented language, one looks at the statement above as a request for the **`v`** object to add the object **`x`** to the end. That is, invoking a method sends a *message* to the object being acted on. The message asks the object to perform some operation.

Primitive types

Java also supports some primitive types that are not classes. These types are similar to the primitive types of C:

`byte`
: a single byte (8-bit) signed integer

`char`
: a two-byte (16-bit) Unicode character

`short`
: a two-byte (16-bit) signed integer

int
: a four-byte (32-bit) signed integer

long
: an eight-byte (64-bit) signed integer

float
: a four-byte floating point number

double
: an eight-byte floating point number

boolean
: a type of variable that may be either *true* or *false*

Defining a class

The general form of a class definition is:

```
public class class-name {
  /* class state definitions */
  /* class method definitions */
}
```

Here's an actual example that creates a Queue class:

```
public class Queue {
  /*
   * the state of the Queue is represented by an internal vector instance
   * (The private indicates that this part of the Queue is only
   * accessible by the methods of the Queue class. This implementation
   * hiding is known as encapsulation.)
   */

  private Vector v;

  /*
   * This is a constructor - it is used to create a new instance of
   * a Queue. The public indicates that this method is available for
   * use by any class.
   */
  public Queue( )
  {
    v = new Vector();      // construct the internal vector object
  }

  /*
   * This method adds an element to the queue
   */
  public void enque( Object obj )
  {
    v.addElement(obj);     // add to end of internal vector
```

```
  }

  /*
   * This method removes and returns an element from the queue
   */
  public Object deque( )
  {
    if ( v.size() > 0 ) {
      Object obj = v.elementAt(0); // read object from front of vector
      v.removeElementAt(0);        // remove object from front of vector
      return obj;                  // return object
    }
    else
      return null;                 // if queue empty, return special null va
  }

  /*
   * This method returns the number of elements in the queue
   */
  public int size()
  {
    return v.size();               // return size of internal vector
  }

}
```

Using classes

To create an instance of a class, the instance must be constructed by a special method called a *constructor*. Constructor methods always have the same name as the class. To create a queue, one would type:

```
Queue q = new Queue();
```

The **new** keyword says to create a new object. The instance variable **q** can be used to refer to this object. Thus, the statement:

```
q.enque("one");
```

says to enque the string "one" in the Queue referred to by **q**.

It is often stated that Java does not have *pointers* and technically this is true. However, class reference variables act like pointers (or really more like reference variables in C++). Assigning class instance variables do not create new instances. For example:

```
Queue q1 = new Queue();     // q1 refers to created Queue object
Queue q2 = q1;              // both q1 and q2 refer to *same* object!

q1.enque("one");
q2.enque("two");
```

```
  System.out.println((String) q2.deque() );   // prints out "one", not "two'
```
Non-class instance variables (like **int** variables) work just as in C, assignment does copy values of these.

These rules have important consequences for parameters of methods. For example, consider the following method call and code:
```
  // method call
  int x = 5;
  Queue q = new Queue();
  method(x,q);
  ...
  // method code
  void method( int xp, Queue qp )
  {
    xp++;                // changes local var xp, not x!
    qp.enque("word");    // adds "word" to the single object referenced by
                         // q and qp
  }
```
Thus, primitive type instances are *always* passed by value and class instances are *always* passed by reference.

Static methods and state

Normally a method acts on a single object (or instance) of a class. However, it is possible to define a method that acts on a class itself instead of a particular instance of the class. These classes are declared **static**.

The **static** keyword can also be used on the state variables of the class with a similar result. The variable becomes a property of the class itself and not any particular instance.

Static methods can only access the static variables of its class, not the non-static variables. Here's a very simple example:
```
public class StaticTest {

  public int ns;          // ns is a normal non-static variable
  public static int s;    // s is a static variable

  public void ns_method() // normal non-static method
  {
     ns = 3;              // can access ns
     s  = 5;              // and s
  }

  public static void s_method() // static method
  {
```

```
    s = 7;              // can *only* access static s!
  }
}
```

Since static methods and instances are properties of a class and not an instance of a class, they are invoked on the class itself. For example, to call the static method above, one would type:

```
StaticTest.s_method();
```

Now we are ready to understand our initial Java program:

```
public class Hello {
  public static void main( String args[] )
  {
    System.out.println("Hello, Java");
  }
}
```

`System` is a name of a class in the Java class library. **`System.out`** refers to a static object in the class that represents the console screen. The **`println()`** method is used to tell **`System.out`** to display a string on the console. Note that **`main()`** is a static method.

Object-Oriented Programming and Inheritance

The preceding section looked at classes from the purely Abstract Data Type view. The object-oriented paradigm goes further by looking at relationships between different classes. Often different classes have an *IS-A* relationship. This type of relationship exists when one class is a more specialized version of another.

For example, what if one needed a searchable vector class. The Vector class does not include a method to search for objects inside an instance. However, one could create a new vector class that performed just like the old Vector class, but directly supported searching.

The *wrong* way to do this is to create a new SearchableVector class from scratch that contains a internal Vector (like how the Queue was implemented). This would require one to re-specify each of Vector's methods for SearchableVector (there are many more methods than the one's specified above!).

The *right* way is to use Java's inheritance mechanism. The SearchableVector class is an extension of the Vector class. Or, in other words, a SearchableVector IS-A Vector. A SearchableVector can do anything that a Vector can do, plus more. Java allows one to simply define a new class that extends the features of an existing class. The new class automatically has all the state and methods of the existing class. The syntax is:

```
class subclass extends superclass {
  // new features of subclass
}
```
where *subclass* refers to the new class and *superclass* refers to the existing class being extended. All classes in Java are extensions either directly or indirectly from the Object class. Here is the definition of the SearchableVector class:

```
public class SearchableVector extends Vector {

  /* search returns the index of the object if found,
   * else -1
   */
  public int search( Object obj )
  {
    for( int i = 0; i < size(); i++ )
      if ( elementAt(i).equals(obj) )
        return i;
    return -1;
  }
}
```

In the limit of the `for` loop, the `size()` method for a Vector is called. No object is specified since `search()` itself is a method and so `size()` acts on the same object that `search()` does. The object a method acts on can be referred to by the `this` keyword. The `size()` call in the `for` loop could be replaced with `this.size()` with the same effect.

The `if` compares the element at index `i` with the object searched for. The `elementAt(i)` call returns the object at index `i`. The `equals()` method of this returned object is then called to compare it with the object searched for. The `equals` method of a class compares the values of two instances and returns *true* or *false* based on whether the values are the same. The C-like `==` operator does *not* compare values of classes in Java. It only looks at whether the two variables compared are referring to the *same* object. (Remember object variables are really references!).

Where to Go from Here

Sample code

You can download the code from my examples to try out.

- Hello.java
- Queue.java
- SearchableVector.java
- Vtest.java - Test program that uses the two classes above.

Books on Java

There are lots of Java books available. Most of them are not very good! Here are some of my favorites.

- *Core Java 1.1: Volume 1 - Fundamentals* by Cay S. Horstmann and Gary Cornell, Pentice-Hall. (This will be the textbook for the Spring 1998 Java class.)
- *Java in a Nutshell, 2nd Edition* by David Flanagan. O'Reilly. (A very good reference!)

Send corrections/comments to pcarter@dogbert.comsc.ucok.edu