What is Java?

Java (with a capital J) is a high-level, third generation programming language, like C, Fortran, Smalltalk, Perl, and many others. You can use Java to write computer applications that crunch numbers, process words, play games, store data or do any of the thousands of other things computer software can do.

Compared to other programming languages, Java is most similar to C. However although Java shares much of C's syntax, it is not C. Knowing how to program in C or, better yet, C++, will certainly help you to learn Java more quickly, but you don't need to know C to learn Java. Unlike C++ Java is not a superset of C. A Java compiler won't compile C code, and most large C programs need to be changed substantially before they can become Java programs.

What's most special about Java in relation to other programming languages is that it lets you write special programs called *applets* that can be downloaded from the Internet and played safely within a web browser. Traditional computer programs have far too much access to your system to be downloaded and executed willy-nilly. Although you generally trust the maintainers of various ftp archives and bulletin boards to do basic virus checking and not to post destructive software, a lot still slips through the cracks. Even more dangerous software would be promulgated if any web page you visited could run programs on your system. You have no way of checking these programs for bugs or for out-and-out malicious behavior before downloading and running them.

Java solves this problem by severely restricting what an applet can do. A Java applet cannot write to your hard disk without your permission. It cannot write to arbitrary addresses in memory and thereby introduce a virus into your computer. It should not crash your system.

Java is a Platform

Java (with a capital J) is a platform for application development. A platform is a loosely defined computer industry buzzword that typically means some combination of hardware and system software that will mostly run all the same software. For instance PowerMacs running Mac OS 9.2 would be one platform. DEC Alphas running Windows NT would be another.

There's another problem with distributing executable programs from web pages. Computer programs are very closely tied to the specific hardware and operating system they run. A Windows program will not run on a computer that only runs DOS. A Mac application can't run on a Unix workstation. VMS code can't be executed on an IBM mainframe, and so on. Therefore major commercial applications like Microsoft Word or Netscape have to be written almost independently for all the different platforms they run on. Netscape is one of the most cross-platform of major applications, and it still only runs on a minority of platforms.

Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler would. Instead it produces a special format called *byte code*. Java byte code written in hexadecimal, byte by byte, looks like this:

```
CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08
```

This looks a lot like machine language, but unlike machine language Java byte code is exactly the same on every platform. This byte code fragment means the same thing on a Solaris workstation as it does on a Macintosh PowerBook. Java programs that have been compiled into byte code still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine on the fly. The most common such interpreter is Sun's program java (with a little j). Since the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system. The rest of the runtime environment including the compiler and most of the class libraries are written in Java.

All these pieces, the javac compiler, the java interpreter, the Java programming language, and more are collectively referred to as Java.

Java is Simple

Java was designed to make it much easier to write bug free code. According to Sun's Bill Joy, shipping C code has, on average, one bug per 55 lines of code. The most important part of helping programmers write bug-free code is keeping the language simple.

Java has the bare bones functionality needed to implement its rich feature set. It does not add lots of syntactic sugar or unnecessary features. Despite its simplicity Java has considerably more functionality than C, primarily because of the large class library.

Because Java is simple, it is easy to read and write. Obfuscated Java isn't nearly as common as obfuscated C. There aren't a lot of special cases or tricks that will confuse beginners.

About half of the bugs in C and C++ programs are related to memory allocation and deallocation. Therefore the second important addition Java makes to providing bug-free code is automatic memory allocation and deallocation. The C library memory allocation functions malloc() and free() are gone as are C++'s destructors.

Java is an excellent teaching language, and an excellent choice with which to learn programming. The language is small so it's easy to become fluent. The language is interpreted so the compile-run-link cycle is much shorter. The runtime environment provides automatic memory allocation and garbage collection so there's less for the programmer to think about. Java is object-oriented unlike Basic so the beginning programmer doesn't have to unlearn bad programming habits when moving into real world projects. Finally, it's very difficult (if not quite impossible) to write a Java program that will crash your system, something that you can't say about any other language.

Java is Object-Oriented

Object oriented programming is the catch phrase of computer programming in the 1990's. Although object oriented programming has been around in one form or another since the Simula language was invented in the 1960's, it's really begun to take hold in modern GUI environments like Windows, Motif and the Mac. In object-oriented programs data is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields tell you what an object is. Methods tell you what an object does. These fields and methods are closely tied to the object's real world characteristics and behavior. When a program is run messages are passed back and forth between objects. When an object receives a message it responds accordingly as defined by its methods.

Object oriented programming is alleged to have a number of advantages including:

- Simpler, easier to read programs
- More efficient reuse of code
- Faster time to market
- More robust, error-free code

In practice object-oriented programs have been just as slow, expensive and buggy as traditional non-object-oriented programs. In large part this is because the most popular object-oriented language is C++. C++ is a complex, difficult language that shares all the obfuscation of C while sharing none of C's efficiencies. It is possible in practice to write clean, easy-to-read Java code. In C++ this is almost unheard of outside of programming textbooks.

Java is Platform Independent

Java was designed to not only be cross-platform in source form like C, but also in compiled binary form. Since this is frankly impossible across processor architectures Java is compiled to an intermediate form called byte-code. A Java program never really executes natively on the host machine. Rather a special native program called the Java interpreter reads the byte code and executes the corresponding native machine instructions. Thus to port Java programs to a new platform all that is needed is to port the interpreter and some of the library routines. Even the compiler is written in Java. The byte codes are precisely defined, and remain the same on all platforms.

The second important part of making Java cross-platform is the elimination of undefined or architecture dependent constructs. Integers are always four bytes long, and floating point variables follow the IEEE 754 standard for computer arithmetic exactly. You don't have to worry that the meaning of an integer is going to change if you move from a Pentium to a PowerPC. In Java everything is guaranteed.

However the virtual machine itself and some parts of the class library must be written in native code. These are not always as easy or as quick to port as pure Java programs.

Java is Safe

Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was untrusted and possibly malicious.

This required the elimination of many features of C and C++. Most notably there are no pointers in Java. Java programs cannot access arbitrary addresses in memory. All memory access is handled behind the scenes by the (presumably) trusted runtime environment. Furthermore Java has strong typing. Variables must be declared, and variables do not change types when you aren't looking. Casts are strictly limited to casts between types that make sense. Thus you can cast an int to a long or a byte to a short but not a long to a boolean or an int to a String.

Java implements a robust exception handling mechanism to deal with both expected and unexpected errors. The worst that an applet can do to a host system is bring down the runtime environment. It cannot bring down the entire system.

Most importantly Java applets can be executed in an environment that prohibits them from introducing viruses, deleting or modifying files, or otherwise destroying data and crashing the host computer. A Java enabled web browser checks the byte codes of an applet to verify that it doesn't do anything nasty before it will run the applet.

However the biggest security problem is not hackers. It's not viruses. It's not even insiders erasing their hard drives and quitting your company to go to work for your competitors. No, the biggest security issue in computing today is bugs. Regular, ordinary, non-malicious unintended bugs are responsible for more data loss and lost productivity than all other factors combined. Java, by making it easier to write bug-free code, substantially improves the security of all kinds of programs.

Java is High Performance

Java byte codes can be compiled on the fly to code that rivals C++ in speed using a "just-in-time compiler." Several companies are also working on native-machine-architecture compilers for Java. These will produce executable code that does not require a separate interpreter, and that is indistinguishable in speed from C++.

While you'll never get that last ounce of speed out of a Java program that you might be able to wring from C or Fortran, the results will be suitable for all but the most demanding applications.

It is certainly possible to write large programs in Java. The HotJava browser, the Eclipse integrated development environment, the LimeWire file sharing application, the jEdit text editor, the JBoss application server, the Tomcat servlet container, the Xerces XML parser, the Xalan XSLT processor, and the javac compiler are large programs that are written entirely in Java.

Java is Multi-Threaded

Java is inherently multi-threaded. A single Java program can have many different threads executing independently and continuously. Three Java applets on the same page can run together with each getting equal time from the CPU with very little extra effort on the part of the programmer.

This makes Java very responsive to user input. It also helps to contribute to Java's robustness and provides a mechanism whereby the Java environment can ensure that a malicious applet doesn't steal all of the host's CPU cycles.

Unfortunately multithreading is so tightly integrated with Java, that it makes Java rather difficult to port to architectures like Windows 3.1 or the PowerMac that don't natively support preemptive multi-threading.

There is a cost associated with multi-threading. Multi-threading is to Java what pointer arithmetic is to C, that is, a source of devilishly hard to find bugs. Nonetheless, in simple programs it's possible to leave multi-threading alone and normally be OK.

Java is Dynamic(ly linked)

Java does not have an explicit link phase. Java source code is divided into .java files, roughly one per each class in your program. The compiler compiles these into .class files containing byte code. Each .java file generally produces exactly one .class file.

(There are a few exceptions we'll discuss later in the semester, non-public classes and inner classes).

The compiler searches the current directory and directories specified in the CLASSPATH environment variable to find other classes explicitly referenced by name in each source code file. If the file you're compiling depends on other, non-compiled files the compiler will try to find them and compile them as well. The compiler is quite smart, and can handle circular dependencies as well as methods that are used before they're declared. It also can determine whether a source code file has changed since the last time it was compiled.

More importantly, classes that were unknown to a program when it was compiled can still be loaded into it at runtime. For example, a web browser can load applets of differing classes that it's never seen before without recompilation.

Furthermore, Java .class files tend to be quite small, a few kilobytes at most. It is not necessary to link in large runtime libraries to produce a (non-native) executable. Instead the necessary classes are loaded from the user's CLASSPATH.

Java is Garbage Collected

You do not need to explicitly allocate or deallocate memory in Java. Memory is allocated as needed, both on the stack and the heap, and reclaimed by the *garbage collector* when it is no longer needed. There's no malloc(), free(), or destructor methods.

There are constructors and these do allocate memory on the heap, but this is transparent to the programmer.

The exact algorithm used for garbage collection varies from one virtual machine to the next. The most common approach in modern VMs is generational garbage collection for short-lived objects, followed by mark and sweep for longer lived objects. I have never encountered a Java VM that used reference counting.

The Hello World Application

```
class HelloWorld {

  public static void main (String args[]) {
    System.out.println("Hello World!");
  }

}
```

Similarities and Differences

Java does not support **typedefs**, **defines**, or a **preprocessor**. Without a preprocessor, there are no provisions for *including* **header files**.

Since Java does not have a preprocessor there is no concept of *#define macros* or *manifest constants*. However, the declaration of *named constants* is supported in Java through use of the **final** keyword.

Java does not support **enums** but, as mentioned above, does support *named constants*.

Java supports **classes**, but does not support **structures** or **unions**.

All stand-alone C++ programs require a function named **main** and can have numerous other functions, including both stand-alone functions and functions, which are members of a class. There are no stand-alone functions in Java. Instead, there are only functions that are members of a class, usually called **methods**. Global functions and global data are not allowed in Java.

All classes in Java ultimately inherit from the **Object** class. This is significantly different from C++ where it is possible to create inheritance trees that are completely unrelated to one another.

All function or method definitions in Java are contained within the class definition. To a C++ programmer, they may look like **inline** function definitions, but they aren't. Java doesn't allow the programmer to request that a function be made **inline**, at least not directly.

Both C++ and Java support ***class*** (static) methods or functions that can be called without the requirement to instantiate an object of the class.

The **interface** keyword in Java is used to create the equivalence of an abstract base class containing only method declarations and constants. No variable data members or method definitions are allowed. (True abstract base classes can also be created in Java.) The *interface* concept is not supported by C++.

Java does not support **multiple inheritance**. To some extent, the *interface* feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.

While Java does not support multiple inheritance, single inheritance in Java is similar to C++, but the manner in which you implement inheritance differs significantly, especially with respect to the use of constructors in the inheritance chain.

In addition to the access specifiers applied to individual members of a class, C++ allows you to provide an additional access specifier when inheriting from a class. This latter concept is not supported by Java.

Java does not support the **goto** statement (but **goto** is a reserved word). However, it does support labeled **break** and **continue** statements, a feature not supported by C++. In certain restricted situations, labeled **break** and **continue** statements can be used where a **goto** statement might otherwise be used.

Java does not support **operator overloading**.

Java does not support **automatic type conversions (**except where guaranteed safe).

Unlike C++, Java has a **String** type, and objects of this type are immutable (cannot be modified). Quoted strings are automatically converted into **String** objects. Java also has a **StringBuffer** type. Objects of this type can be modified, and a variety of string manipulation methods are provided.

Unlike C++, Java provides true arrays as first-class objects. There is a **length** member, which tells you how big the array is. An exception is thrown if you attempt to access an array out of bounds. All arrays are instantiated in dynamic memory and assignment of one array to another is allowed. However, when you make such an assignment, you simply have two references to the same array. Changing the value of an element in the array using one of the references changes the value insofar as both references are concerned.

Unlike C++, having two "pointers" or references to the same object in dynamic memory is not necessarily a problem (but it can result in somewhat confusing results). In Java, dynamic memory is reclaimed automatically, but is not reclaimed until all references to that memory become NULL or cease to exist. Therefore, unlike in C++, the allocated dynamic memory cannot become invalid for as long as it is being referenced by any reference variable.

Java does not support **pointers** (at least it does not allow you to modify the address contained in a pointer or to perform pointer arithmetic). Much of the need for pointers was eliminated by providing types for arrays and strings. For example, the oft-used C++ declaration **char* ptr** needed to point to the first character in a C++ null-terminated "string" is not required in Java, because a string is a true object in Java.

A class definition in Java looks similar to a class definition in C++, but there is no closing semicolon. Also forward reference declarations that are sometimes required in C++ are not required in Java.

The scope resolution operator (**::**) required in C++ is not used in Java. The dot is used to construct all fully-qualified references. Also, since there are no pointers, the pointer operator (**->**) used in C++ is not required in Java.

In C++, **static** data members and functions are called using the name of the class and the name of the static member connected by the scope resolution operator. In Java, the dot is used for this purpose.

Like C++, Java has primitive types such as **int, float,** etc. Unlike C++, the size of each primitive type is the same regardless of the platform. There is no unsigned integer type in Java. Type checking and type requirements are much tighter in Java than in C++.

Unlike C++, Java provides a true **boolean** type.

Conditional expressions in Java must evaluate to **boolean** rather than to integer, as is the case in C++. Statements such as **if(x+y)...** are not allowed in Java because the conditional expression doesn't evaluate to a **boolean**.

The **char** type in C++ is an 8-bit type that maps to the ASCII (or extended ASCII) character set. The **char** type in Java is a 16-bit type and uses the Unicode character set (the Unicode values from 0 through 127 match the ASCII character set). For information on the Unicode character set see http://www.stonehand.com/unicode.html.

Unlike C++, the >> operator in Java is a "signed" right bit shift, inserting the sign bit into the vacated bit position. Java adds an operator that inserts zeros into the vacated bit positions.

C++ allows the instantiation of variables or objects of all types either at compile time in static memory or at run time using dynamic memory. However, Java requires all variables of primitive types to be instantiated at compile time, and requires all objects to be instantiated in dynamic memory at runtime. Wrapper classes are provided for all primitive types except **byte** and **short** to allow them to be instantiated as objects in dynamic memory at runtime if needed.

C++ requires that classes and functions be declared before they are used. This is not necessary in Java.

The "namespace" issues prevalent in C++ are handled in Java by including everything in a class, and collecting classes into *packages*.

C++ requires that you re-declare **static** data members outside the class. This is not required in Java.

In C++, unless you specifically initialize variables of primitive types, they will contain garbage. Although local variables of primitive types can be initialized in the declaration, primitive data members of a class cannot be initialized in the class definition in C++.

In Java, you can initialize primitive data members in the class definition. You can also initialize them in the constructor. If you fail to initialize them, they will be initialized to zero (or equivalent) automatically.

Like C++, Java supports constructors that may be overloaded. As in C++, if you fail to provide a constructor, a default constructor will be provided for you. If you provide a constructor, the default constructor is not provided automatically.

All objects in Java are passed by reference, eliminating the need for the *copy constructor* used in C++.

(In reality, all parameters are passed by value in Java. However, passing a copy of a reference variable makes it possible for code in the receiving method to access the object referred to by the variable, and possibly to modify the contents of that object. However, code in the receiving method cannot cause the original reference variable to refer to a different object.)

There are no destructors in Java. Unused memory is returned to the operating system by way of a *garbage collector*, which runs in a different thread from the main program. This leads to a whole host of subtle and extremely important differences between Java and C++.

Like C++, Java allows you to overload functions. However, default arguments are not supported by Java.

Unlike C++, Java does not support *templates*. Thus, there are no *generic* functions or classes.

Unlike C++, several *"data structure"* classes are contained in the "standard" version of Java. More specifically, they are contained in the standard class library that is distributed with the Java Development Kit (JDK). For example, the standard version of Java provides the containers **Vector** and **Hashtable** that can be used to contain any object through recognition that any object is an object of type **Object**. However, to use these containers, you must perform the appropriate *upcasting* and *downcasting*, which may lead to efficiency problems.

Multithreading is a standard feature of the Java language.

Although Java uses the same keywords as C++ for access control: **private, public,** and **protected**, the interpretation of these keywords is significantly different between Java and C++.

There is no **virtual** keyword in Java. All non-static methods always use dynamic binding, so the **virtual** keyword isn't needed for the same purpose that it is used in C++.

Java provides the **final** keyword that can be used to specify that a method cannot be overridden and that it can be statically bound. (The compiler *may* elect to make it *inline* in this case.)

The detailed implementation of the *exception handling* system in Java is significantly different from that in C++.

Unlike C++, Java does not support *operator overloading*. However, the (+) and (+=) operators are automatically overloaded to concatenate strings, and to convert other types to **string** in the process.

As in C++, Java applications can call functions written in another language. This is commonly referred to as *native methods*. However, applets cannot call native methods.

Unlike C++, Java has built-in support for program documentation. Specially written comments can be automatically stripped out using a separate program named **javadoc** to produce program documentation.

Generally Java is more robust than C++ due to the following:

- Object handles (references) are automatically initialized to **null**.
- Handles are checked before accessing, and exceptions are thrown in the event of problems.
- You cannot access an array out of bounds.
- Memory leaks are prevented by automatic *garbage collection*.

Command line arguments

```
class PrintArgs {

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

  }

}
```

The name of the class is **not** included in the argument list.

Command line arguments are passed in an array of Strings. The first array component is the zeroth.

For example, consider this invocation:

```
$ java printArgs Hello There
```

args[0] is the string "Hello". args[1] is the string "There". args.length is 2.

All command line arguments are passed as String values, never as numbers. Later you'll learn how to convert Strings to numbers.

Print statements

```
class PrintArgs {

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

  }

}
$ java PrintArgs Hello there!
Hello
there!
```

System.out.println() prints its arguments followed by a platform dependent line separator (carriage return (ASCII 13, \r) and a linefeed (ASCII 10, \n) on Windows, linefeed on Unix, carriage return on the Mac)

System.err.println() prints on standard err instead.

You can concatenate arguments to println() with a plus sign (+), e.g.

```
System.out.println("There are " + args.length + " command line arguments");
```

Using print() instead of println() does not break the line. For example,

```
System.out.print("There are ");
System.out.print(args.length);
System.out.print(" command line arguments");
System.out.println();
```

System.out.println() breaks the line and flushes the output. In general nothing will actually appear on the screen until there's a line break character.

Variables and Data Types

There are eight primitive data types in Java:

- **boolean**

- byte
- short
- **int**
- *long*
- *float*
- **double**
- **char**

However there are only seven kinds of literals, and one of those is not a primitive data type:

- boolean: true or false
- int: 89, -945, 37865
- long: 89L, -945L, 5123567876L
- float: 89.5f, -32.5f,
- double: 89.5, -32.5, 87.6E45
- char: 'c', '9', 't'
- String: "This is a string literal"

There are no short or byte literals.

Strings are a *reference* or *object* type, not a primitive type. However the Java compiler has special support for strings so this sometimes appears not to be the case.

```
class Variables {

  public static void main (String args[]) {

    boolean b = true;
    int low = 1;
    long high = 76L;
    long middle = 74;
    float pi = 3.1415292f;
    double e = 2.71828;
    String s = "Hello World!";

  }

}
```

Comments

Comments in Java are identical to those in C++. Everything between /* and */ is ignored by the compiler, and everything on a single line after // is also thrown away. Therefore the following program is, as far as the compiler is concerned, identical to the first HelloWorld program.

```
// This is the Hello World program in Java
class HelloWorld {

  public static void main (String args[]) {
    /* Now let's print the line Hello World */
    System.out.println("Hello World!");

  } // main ends here

} // HelloWorld ends here
```

The /* */ style comments can comment out multiple lines so they're useful when you want to remove large blocks of code, perhaps for debugging purposes. // style comments are better for short notes of no more than a line. /* */ can also be used in the middle of a line whereas // can only be used at the end. However putting a comment in the middle of a line makes code harder to read and is generally considered to be bad form.

Comments evaluate to white space, not nothing at all. Thus the following line causes a compiler error:

```
int i = 78/* Split the number in two*/76;
```

Java turns this into the illegal line

```
int i = 78 76;
```

not the legal line

```
int i = 7876;
```