Java 1

Right Language, Right Time

Java came on the scene in 1995 to immediate popularity.

Before that, C and C++ dominated -- compiled, no robust memory model, no garbage collector, no large standard library

Java brings together a great set of "programmer efficient" features -- putting more work on the CPU to make things easier for the programmer.

Java is popular partly because it is well designed and brings together useful features, and also partly IMHO because by 1995, hardware had become powerful enough that, at last, a programmer efficient language made sense.

Java -- Buzzwords

From the original Sun Java whitepaper: "Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multi-threaded, and dynamic language." By law, any introductory Java lecture must mention the original buzzwords.

Here is my take on the original java buzzwords...

Java -- Language + Libraries

Java has two parts...

1. The core language -- ints, arrays, objects
   - The Java Virtual Machine (JVM) runs the core language
   - The core language is of moderate size and moderate complexity.
   - Indeed, the core language is simple enough to run on small devices -- phones, smart cards, PDAs
2. The libraries
   - Java includes a large collection of standard library classes to provide "off the shelf" code.
   - e.g. String, ArrayList, HashMap, StringTokenizer, HTTPConnection, Date, ...
   - The libraries, for the most part, are themselves written in Java, so they have the java features of robustness and portability.

Java programmers are more productive in part because they have access to a large set of standard, well documented library classes.

The core java language is simple enough, but java taken together with its thousands of library classes is not simple.

Simple

The core language is simpler than C++ -- no operator overloading, no memory arithmetic, no multiple inheritance

Mimics C/C++ syntax, operators, etc. where possible

The way a java program deals with memory is much simpler than C or C++

Object-Oriented

Java is fundamentally based on the OOP notions of classes and objects.
Java uses a formal OOP type system that must be obeyed at compile-time and run-time.
This is helpful for larger projects, where the structure helps keep the various parts consistent. Contrast to Perl, which has a more anything-goes feel.

Distributed / Network Oriented

Java is network friendly -- both in its portable, threaded nature, and because common networking operations are built-in to the Java libraries.

Robust / Secure / Safe

Java is very robust -- both vs. unintentional errors and vs. malicious code such as viruses. Java has slightly worse performance since it does all this checking. (Or put the other way, C can be faster since it doesn't check anything.)

1. The JVM "verifier" checks the code when it is loaded to verify that it has the correct structure -- that it does not use an uninitialized pointer, or mix int and pointer types. This is one-time "static" analysis -- checking that the code has the correct structure without running it.
2. The JVM also does "dynamic" checking at runtime for certain operations, such as pointer and array access, to make sure they are touching only the memory they should. You will write code that runs into

As a result, many common bugs and security problems (e.g. "buffer overflow") are not possible in java. The checks also make it easier to find many common bugs easy, since they are caught by the runtime checker.
You will generally never write code that fails the verifier, since your compiler is smart enough to only generate correct code. You will write code that runs into the runtime checks all the time as you debug -- array out of bounds, null pointer.
Java also has a runtime Security Manager can check which operations a particular piece of code is allowed to do. As a result, java can run untrusted code in a "sandbox" where, for example, it can draw to the screen but cannot access the local filesystem.

Portable

Java is designed to "Write Once Run Anywhere", and for the most part this works. Not even a recompile is required -- a Java executable can work, without change, on any Java enabled platform.
Java is unusual for doing portability so well.

High-performance

The first versions of java were pretty slow.
Java performance has gotten a lot better with aggressive just-in-time-compiler (JIT) techniques.
Java performance is now similar to C -- a little slower in some cases, faster in a few cases. However memory use and startup time are both worse than C.

Java performance gets better each year as the JVM gets smarter. This works, because making the JVM smarter does not require any great change to the java language, source code, etc.

Multi-Threaded

Java has a notion of concurrency wired right in to the language itself. This works out more cleanly than languages where concurrency is bolted on after the fact.

Dynamic

Class and type information is kept around at runtime. This enables runtime loading and inspection of code in a very flexible way.

Java Compiler Structure

The source code for each class is in a .java file. Compile each class to produce a .class file.

Sometimes, multiple .class files are packaged together into a .zip or .jar "archive" file.

On unix, the java compiler is called "javac". To compile all the .java files in a directory use "javac *.java".

Bytecode

A compiled class stored in a .class files or .jar file

Represent a computation in a portable way -- as PDF is to an image

Java Virtual Machine

Loads and runs the bytecode for a program, and loads and runs the bytecode for the library classes as needed.

The JVM runs the code with the various robustness/safety checks in place -- when the bytecode is loaded and as it runs.

On unix, the JVM is called "java". Suppose we have a class MyClass with a main() in it. Run that main with "java MyClass".

JITs and Hotspot

Just In Time compiler -- the JVM may compile the bytecode to native code at runtime (with the robustness checks still in). (This is one reason why java programs have slow startup times.)

The "hotspot" project tries to do a sophisticated job of which parts of the program to compile. In some cases, hotspot can do a better job of optimization than a C++ compiler, since hotpsot is playing with the code at runtime and so has more information.

Java performance is now similar to C performance -- faster in some cases, slower in others, although startup time is generally worse for java.

Java: Programmer Efficiency

Faster Development
- Building an application in Java takes about 50% less time than in C or C++
- Faster time to market
- Java is said to be "programmer efficient"

OOP
- Java is thoroughly OOP

Robust memory system
- Memory errors largely disappear because of the safe pointers and garbage collector. I suspect the lack of memory errors accounts for much of the increased programmer productivity.

Libraries
- Code re-use at last -- String, ArrayList, Date, ... available and documented in a standard way

Microsoft vs. Java

Microsoft hates Java, since a Java program (portable) is not tied to any particular operating system. If Java is popular, then programs written in Java might promote non-Microsoft operating systems. For basically the same reason, all the non-Microsoft vendors think Java is a great idea.

Microsoft's C# is very similar to Java, but with some improvements, and some questionable features added in, and it is not portable in the way Java is. I think C# will be successful in the way that Visual Basic is: a nice tool to build Microsoft only software.

Microsoft has used its power to try to derail Java somewhat, but Java remains very popular on its merits.

Java Is For Real

Java has a lot of hype, but much of it is deserved.

Java is very well matched for many modern problem
- Using more memory and CPU time but less programmer time is an increasingly appealing tradeoff.
- Robustness and portability can be very useful features

I suspect we will be using some version of the Java language for the next 10 or 20 years.