Nick Parlante

Java 1

Non-Standard Java History

A language for toasters/set-top-boxes re-purposed for the Internet.

Features

Robust/Safe

Pointers checked at run-time
Arrays checked at run-time
Garbage Collector handles dynamic memory
Security Manager allows/denies operations at run-time
This all pretty much requires an interpreter

Binary Portable

If you're going to use an interpreter, you might as well be binary portable (no re-compile required)

Slow

Results

Robust Code

As a practical matter, Java code feels less fragile once it's built and debugged.

Mobile Code

Send code in a packet from one machine to another. It can execute (portable) and the receiver is not afraid that it is a virus (robust/safe).

Programmer Efficiency

The robust pointers and memory system also happen to prevent a lot of common bugs.
In my experience, around 30% of the development time is saved.

Right Place and Time

Before Java, many interpreted languages had features similar to Java, but none were as popular.
CPUs are getting cheaper / programmers are getting more expensive.
At some point, the curves cross and suddenly having a slow, programmer-efficient language makes sense.
My theory is that Java was on the scene at just the right time

Java Dynasty

Now that Java has filled the one new programmer-efficient niche and gotten so much inertia and network effect, we will be using some form of Java for a long, long time.

Java Oddities

The following present a couple misunderstood areas of the language....

1. Classpath / Import Issues

Fully-Qualified Class Names

For the most part, the compile-time and run-time systems use the fully qualified names for classes, such as java.lang.String.

Classpath

Set of directories and jar files

Sort of a hack; source of problems; makes installs hard

In Java 1.1, the Classpath needed to include the "core" java classes as well, typically in a file called "classes.zip". In Java 1.2, the Java installation is supposed to automatically know where the core java classes are, so they should not be included in the Classpath.

CT -- javac

The Classpath is used at compile-time by javac to verify the name and interface of classes used by the code

Does not link in the code

RT -- java

The Classpath is used by java at run-time to find and load the bytecode for a class

Q: What Does Import Do?

All import does is allow you to use short names in the source code -- "Vector" instead of the fully qualified "java.util.Vector".

This is just a superficial source code change -- the .class files use the fully qualified names.

Import does not link in additional code.

Q: Is there a better way?

Use the -classpath option for javac and java to specify what directories and jar files to use. This is less convenient for development, but it is a better way to install.

We'll have a separate discussion of Jar files later.

2. Superclass Construction Trickyness

Constructors essentially run from the superclass down to the subclass -- this creates an awkward state as the constructors run.

If a superclass constructor sends itself a message which has been overridden, you pop down to the subclass method, however the subclass constructor will not have run yet.

```
class A {
   int a =2;
   A() {
      foo();
      a = 4;
   }
   void foo() {}
}

class B extends A {
   int b = 6;
   B() {
      super();
      b = 8;
   }

   void foo() { } // you would expect b == 6 or b==8 here, but it's 0, a == 2
}
```

The Order

new B();

1) Zero the whole object (a and b)
2) Call B(), which immediately calls A() (the super() in the code is optional, A() happens no matter what)
3) a=2 initializer
4) A() runs
5) Pops down to B.foo(), b==0, a==2
6) A() finishes, sets a=4;
7) b=6 initializer
8) Finally B() runs, b=8

Conclusion

Consider not doing a lot of real computation in constructors, just set things up

Consider setting things in the constructor or initializer, but not both. The old "never have two copies of anything" rule applied to code.

Java GUIs

The AWT/Peer Debacle

Many separate implementations trying to implement from a common set of docs **does not work**. If the implementation are to be truly compatible with each other, they must be built from common source -- so called "bug for bug" compatible.

The new Swing/JFC approach

JFC Ideas...

1. Common Source Code
2. Layout Manager Portability
3. Pluggable Look 'n' Feel
4. Listener Event Model
5. Serialized-GUI-Bean technology (not done yet)