Java Conclusions

Boxing / Unboxing

May or may not make it into Java
Problem: collections can only contain pointers
Solution: translate automatically between the primitive (int) and its object form (Integer)

Boxing / Unboxing Example

```
ArrayList<Integer> ints;

ints.add(12);  / / boxing  12 is converted to new Integer(12)

int val = ints.get(0);  / /  unboxing: the Integer is automatically
        / /  unboxed into int val
```

Boxing / Unboxing Issues

Avoids awkward primitive vs object sections in the source code
Problem: easy to write innocent looking code that actually garbages through a lot of memory
On the whole, I think boxing / unboxing is a good idea.

1. OOP Design

Modularity
    Public interface vs. private implementation and storage
Inheritance
    Subclass of library class
    Abstract superclass / clever factoring

2. Programmer Efficiency

Why do Java programmers get things done so quickly?
    Robust -- memory, arrays, ...
    Modular / OOP -> enables modular coding
    Modular / OOP -> enables good, standard libraries
    Portable -> avoids boring versioning problems
Programmer efficiency is never going out of style.
Modularity + libraries + robust = programmers get things done more quickly (30%?)

This will only become more important as machines get faster relative to programmers.

3. Robust / Secure

Another feature that is never going out of style
Nice that it is designed in from the ground up.
Works well with client-server applications -- where viruses are a concern

4. Portable

Useful to be able to target all OSes
Very Useful for networked computers, palm pilots, etc. to be able to send code around that works everywhere

5. Performance

Performance is not great, but acceptable (e.g. the draw program was functional)
Maybe less of a problem as Moore's law churns along
JIT / Hotspot technology transforms the bytecode in to native code anyway
Java still uses significantly more memory -- maybe ok
Slow startup time is the most noticeable problem -- is Sun being too complacent?

Java Dynasty?

Will we be using a Java derivative in 2020? -- I think the answer is yes.
Features
    Programmer efficiency, Robustness, Portability, Slowness
Short-Term / Long-Term
    The above is a better match in the long term than the short term. Java has made it this far, it will just look better in the future.
High road / low road
    Java will be your "high road" structured / OOP language -- big projects. (C++ is also a contender for structured, but I think Java looks better 90% of the time).
    C# is similar to Java, but windows specific
    You will also know a "low road" language for little projects: Perl, Python, Javascript, Visual Basic, ...

CPU / Programmer Curve

CPU cost vs. programmer cost
Up through 1994, CPU cost was more important -> lots of C and C++ coding
In 1994 , the curves cross. Suddenly interpreted languages like Perl, Python, and Java make sense.
This was a one-time switch from the old C / C++ days, to the dynamic, programmer efficient, memory and CPU wasteful Perl / Java days.
Java was on the scene at the right time, and soaked up the network effect and inertia (just as C did 20 years earlier for the compiled language age).

Steve Jobs Software Inertia Design

When at NeXT, Steve Jobs remarked on a strategy for dealing with high software inertia.

Notice that software systems tended to have high inertia vs. their hardware -- that once DOS or MacOS 1.0 became popular, their design decisions remained in force for a long time, and were hard to displace.

In contrast, hardware evolves pretty quickly -- e.g. think about the hardware that DOS derived OS;s have run on vs. the struggle for the software side to advance.

Therefore, a new software paradigm should be as advanced as possible for current hardware.

Java reflects this idea -- it chooses aggressive long term features (robust, programmer efficient, portable), at the cost of mediocre current performance.

According to Steve Jobs, 5 years in the future, this sort of tradeoff looks brilliant

JVM Performance Curve

Bytecode can remain as it is

Java performance depends more on the implementation of the JVM -- note that the JVM can evolve year-to-year, and that has been happening.

In other words, bytecode is fixed, but this does not limit JVM development. JVM can work differently year to year, and the old byte code keeps working.

This flexibility allows Java to get better without compromising backward compatibility.