Class 2: Lecture: History, Design, Intro to Java

Why do we care about the history of the Java?

Because it helps us understand why the language is designed the way it is, which in turn gives us a shot at being better Java programmers.

Java is sometimes called the first language of the Internet. Why?

A few reasons: it was first introduced around the same time as the Web exploded: 1993, when the first graphical web browser (Mosaic) was also introduced. Java and the Internet have kinda evolved together.

But Java wasn't originally designed for the Internet. What was it for?

"Interactive TV." Which still hasn't really happened—but the requirements for the language—small, safe, platform-independent, object-oriented—were basically the same as what you'd want for a language for the Internet. So, Java's designers focused on making it a powerful language for Internet applications.

In particular, Java's design emphasizes two factors: platform-independence, so that a program can be run on essentially any computer, and safety—both in the sense that it protects programmers from many kinds of stupid mistakes, and it protects users from flawed and/or malicious code.

How is it platform-independent? Because of the way it uses *compilation* and *interpretation*....

To run Java, you need a *Java Virtual Machine*.

How is it safe? This is complex and many-layered. We'll see different pieces of this throughout the semester. Here are a few parts:

- Java is very careful about types: everything has a type, and Java makes it hard for programmers to be sloppy about this.
- Java doesn't let the programmer manipulate pointers directly, so it's very difficult for programmers to manipulate memory they should be.
- Java includes several "safety checks" before and during the execution of a program. The *security manager* system makes sure that programs don't have access to any more information than they need. The *bytecode verifier* makes sure that the code that's about to run is "sane." The *class loader* makes it difficult for malicious code to change core Java functionality.

- Java's approach to object-oriented design makes it much easier to create clean designs, and much harder to create complex, difficult-to-understand designs.

Let's talk about one component of this, then work on a simple program that does some old-school keyboard-and-screen input and output.

Pointers and References

In C/C++, what is a *pointer*?

It's basically a memory address; C++ lets you manipulate these values almost as easily as it lets you play with integers. If you have a pointer to a memory location which holds a `double` value, then you can *dereference* that pointer to get the `double` value. (What's the dereferencing operator?)

But you can make all kinds of mistakes with pointers. Like what?

In a networked environment, where your machine may be running code from somewhere else, you don't want this kind of thing to happen; Java doesn't let it.

In C, one common application of pointers is to pass variables to functions in such a way that the function can alter the value of the variable. What's this called?

C++ introduces a new bit of language which achieves the same purpose (and some others), without some of the dangers of pointers. What is this new thing called?

In Java, instead of *pointers*, the language uses *references*. In fact, almost every value/variable you manipulate in a Java program is a *reference*. Specifically, there are exactly two kinds of values in Java. *Primitive* values are values with type `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`. (See page 87.)

Everything else in Java is an object, but we programmers don't manipulate these objects directly. Instead, the values we manipulate are *references* to objects. Think of these references as "handles" that allow us to grasp objects that would otherwise be to slippier/painful to touch. So, for example, if I write

```
String s;
```

I've declared a new reference, `s`, which I can use as a "handle" for a `String` object. Later, I could write something like

```
s = new String("sandwich");
```

which creates a new `String` object and stores a reference to it in `s`. Then I could say

```
System.out.println(s);
```

which will print the `String` that `s` refers to.

```
s = new String("hamburger");
```

So far this looks a fair amount like C++. In C++, what would I do if I decided I was done with `s`? I'd write something like

```
delete s;
```

which would return the space that String was using to the free memory heap.

But in the same way that Java won't give us pointers to memory locations, it won't let us "delete" memory either (what if we're malicious trying to delete memory that really shoudn't be?). Instead, the *Java Virtual Machine* runs a "garbage collection" process which monitors the program as it runs, and identifies objects which are no longer used. (How does it do that? It sees that the object has no references.)

Some Java code

So, let's switch gears a bit and write a simple program (or maybe two). Let's write the classic (if slightly dull) program that reads two integers from input and prints them out. We can write a couple versions of that, then we can modify it (for example) it prints the result to a file.

(We're not going to do much of this in this class, because Java's graphics and windowing libraries are so easy to use, but it's important to understand how to do it, and it also illustrates some design features of Java.)