Scala (continued)

How does O-O mix with FP?

- In O-O/FP language (like Scala):
  - classes have fields (just like Java, say) but these fields can contain *either* values *or* functions:
    - when they are *function* fields, we call them *methods* (similar to Java) and have a special declaration syntax;
  - syntax for method declarations is a mix of a *function* has 0..N parameter sets—each enclosed in "()"—see next slide for detail;
    - a *method* has, additionally, "this" as a *parameter set*; and, by convention, "this" is invisibly *prefixed* to the method name (other parameter sets *follow* the method name)—("infix" notation);
    - example method *apply* on an indexed sequence (with one parameter set) can be invoked thus (all equivalent):
      - `this apply x`
      - `this.apply(x)`
      - `this(x)`
      - `(x)`

Functions and methods

- Functions follow the form of mathematical functions:
  - *f(x, y)*
  - *f(x)(y)*
- These two forms are essentially the same in terms of the result. But, mathematically, they are different. We will talk about this in more detail later.
- But, recall that *x* and *y* can represent any object, including a "tuple". So, if *x = (i, j)* and *y = (k, l)* then we can write our function as *f((i, j))((k, l))*. Scala allows us to simplify this as *f(i, j)(k, l)*.
- Bottom line is that a method in Scala can have any number of "parameter sets":
  - *def f(i, j)(k, l)…. = …*

What's imperative programming?

- Imperative programming comes from an understanding of the hardware architecture of a computer, whether a Turing machine, a Von-Neumann machine, or whatever:
  - The notion is that you have a block of addressable memory and at least one register (the accumulator).
  - The program, as well as the data, is stored in this memory
    - another register (the program counter—PC) points to the current instruction;
    - the system steps through the program by changing the value of the PC.
  - To *use* a value in your program you must:
    - find its address;
    - issue a fetch (or load) statement to get the value into the accumulator;
    - perform some arithmetic (or store) operation on it.
  - To *store* a value from the accumulator into memory you must:
    - determine an appropriate address.
    - issue a store statement to that address.
- All programs written in imperative style *ultimately* perform these operations.

Other differences

- Pointers:
  - In procedural programming, pointers are used everywhere (although good O-O style masks use of some pointers by *this*, etc.)
    - Unfortunately pointers which are mutable can be null—result: *NullPointerException* (NPE).
    - Scala allows *null* for compatibility with Java, but you should *never* use it! (There are better ways)
- Types:
  - With non-strict typing (e.g. as in Java), generic types are used
    - Unfortunately, sometimes these are wrong—result: *ClassCastException*.
  - Scala has *strict* typing

Scala vs. Java 8

- There are four concepts that are common to both languages:
  - Lambdas (anonymous functions)
  - Methods in traits/interfaces
  - “Stream” operations on collections
  - SAM (single-abstract-method) types (used for all *FunctionN*)
- Otherwise, Scala and Java 8 are completely different: Java 8 lacks most of the other functional programming “goodies” that we will learn about.

Scala, Java & the JVM

- Being on the JVM is one of the **great strengths** of Scala.
  - It opened up the entire Hadoop/BigData/Spark world to Scala:
    - There's no question in my mind that, otherwise, the language would never have "taken off";
- But… The cosy relationship between Java and Scala comes with some "baggage":
  - exceptions, nulls, auto-boxing, mutable arrays, erasure, even O-O in the eyes of purists, etc.
  - "pure" *fp* languages like Haskell do not suffer from these issues so Scala does its best to work around them.

Declarative Programming

- If you go back in history to the dawn of programming* you will find that (almost) all programs** were written in the imperative style***:

```
package edu.neu.coe.scala;

public class NewtonApproximation {
    public static void main(String[] args) {
        // Newton's Approximation to solve cos(x) = x
        double x = 1.0;
        int tries = 200;
        for (; tries > 0; tries--) {
            final double y = Math.cos(x)-x;
            if (Math.abs(y)<1E-7) {
                System.out.println("the solution to cos(x)=x is: "+x);
                System.exit(0);
            }
            x = x + y/(Math.sin(x)+1);
        }
    }
}
```

* actually, I do go back almost that far ;)
** does this language look familiar? It should—it's Java.
*** you could also call this the "Von Neumann" style (search for "Backus Turing Award")

Observations on Newton-Raphson Approximation

- This is written in Java but the style is similar to the original Fortran (Backus et al)—the loop would have been a GOTO originally and tries would have been called “I” or “N”
- If you’re unsure about the method, see:
  - *http://en.wikipedia.org/wiki/Newton's method*
- Everything is static (there is no O-O here, as you’d expect)
- There’s not just one but *two* mutable variables (*x* and *tries*)
- The program tells the system *exactly* how to run the calculation
  - there’s a loop until some terminating condition is met;
  - meanwhile, the best estimate (*x*) is explicitly updated.

So, what's wrong with the imperative/declarative style?

- In the beginning a program was loaded into memory (from magnetic/ paper tape or punched cards) and ran until it was finished (or "abend"ed).
- As computers became more shareable, used disks, had other devices attached, the idea of interrupts was born:
  - An interrupt put the current program on hold and started running a different program for a while—when finished it would go back to the original program;
  - The less tolerant of delay was the device, the higher the priority of the interrupt (a card or paper tape reader was intolerant, a disk a little more tolerant).
- This worked fine until networks and shared databases came along (early 70s)—even then it worked OK until the sheer volume and frequency of network interrupts—and the number of database users got too high.
- This problem was "solved" by inventing "threads" (sub-processes)—and using a programming language that explicitly allows threads to be programmed at a low level, e.g. Java.

The straw that broke the camel's back

- Have you ever tried to develop and, more importantly, *test* a threaded application?
- I have—it can be maddeningly frustrating.
- Typical "solutions":
  - synchronize mutable state—but be careful not to synchronize too much at once lest you run into race conditions and deadlocks
  - custom Executor services and a greater number of processors (make the problem "go away"—for a while)
  - defensive deep copying (Aaargh!)