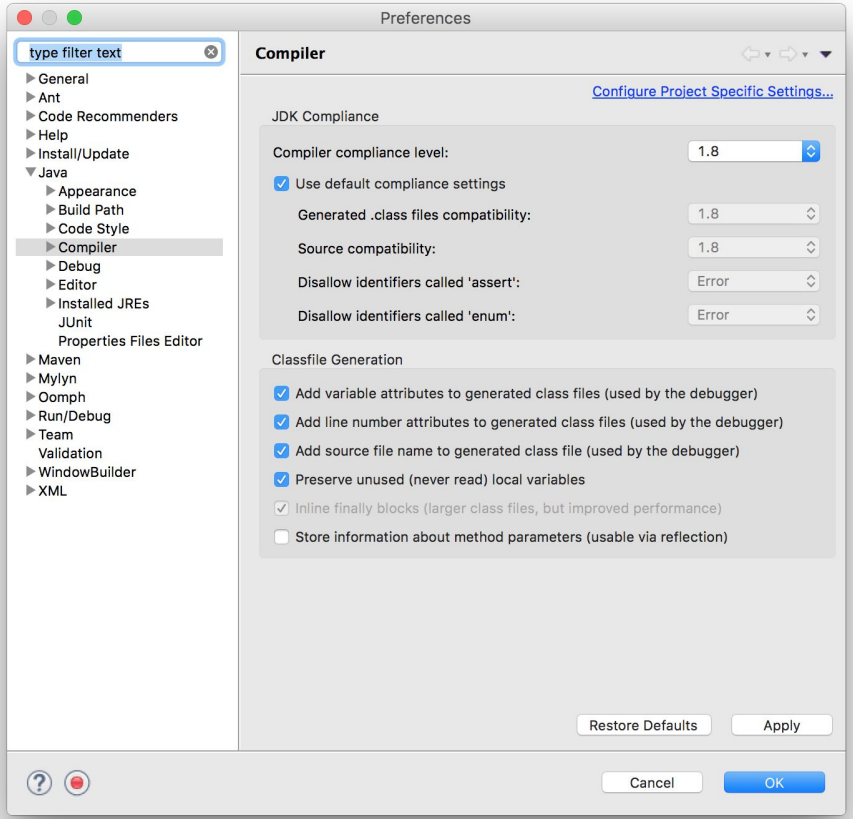
6.178: Introduction to Software Engineering in Java

Lecture 8: Tips & Tricks

Today’s Game Plan

- Lambda Expressions in Java - Andrew
- Map, Filter, Reduce - Graeme
- Recursion - Katy

Check your Eclipse Settings! (Window > Preferences)



Obligatory XKCD

“Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.”

- *XKCD 1270*

What is functional programming? Why do I care?

“In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”

- Wikipedia

Functional programming offers another way of solving problems that makes some solutions cleaner and helps avoid bugs. It can help abstract away iteration and allow more easily see how a system interacts.

Shifting Gears

- So far, we have worked entirely in the realm of Object-Oriented Programming.
  - Which makes sense, because Java is really oriented towards OOP.
- Java 8 includes some things that allow for functional programming
- This is not going to be a lecture on functional programming
- This is going to be a lecture on functional programming tools

Lambda Expressions

- A syntax that represents a function in terms of its inputs and outputs inside of a method body
- Right now, you can only pass around data in Java
  - Your objects return, store, and take in data
- Lambda expressions let you also pass around code, just like you pass around data!
  - Can easily do mathy things like function composition

Let’s write some Lambdas!

Java Functionals

- Make operations on lists a lot easier
- Used to apply methods to each element of a list
  - Often done using lambdas
- Represent a different way to think about programming
  - Indexes and iteration are abstracted away
  - Generally written more compactly than previous code shown in this class
- Most common functionals: Map, Filter, Reduce

Map

- Transform that “maps” elements of a list of one type to different elements of a different (sometimes the same) type
- Can be used for any type of transform
- Will always return a list of the same length
    - Returned list not necessarily same type and returned values may not be related to initial values
- Syntax: list.stream().map(v -> new_value).collect()

Filter

- Filters out elements of a list that don’t “pass” the given “test”
  - Test is given via a boolean expression in filter
  - If the boolean expression is true, the element will remain in the list, otherwise it will be removed
- Will NOT always return a list of the original length, but will not modify the elements otherwise
- Syntax: list.stream().filter(v -> v >= 20).collect()

Reduce

- Goes through the elements of a list and builds a new structure based on the value of the list
- May return any type
- Syntax: list.stream().reduce(0, (a, b) -> a + b)

Reduce

- Goes through the elements of a list and builds a new structure based on the value of the list
- May return any type
- Syntax: list.stream().reduce(0, (a, b) -> a + b)

Recursion

What is it?

Did you mean recursion?

Calling a function within itself to solve a smaller subproblem

ex: Fibonacci sequence

nth fib number = (n-1)th fib number + (n-2)th fib number

Pieces

Base Case(s):

- the smallest subproblem(s)
- a concrete answer

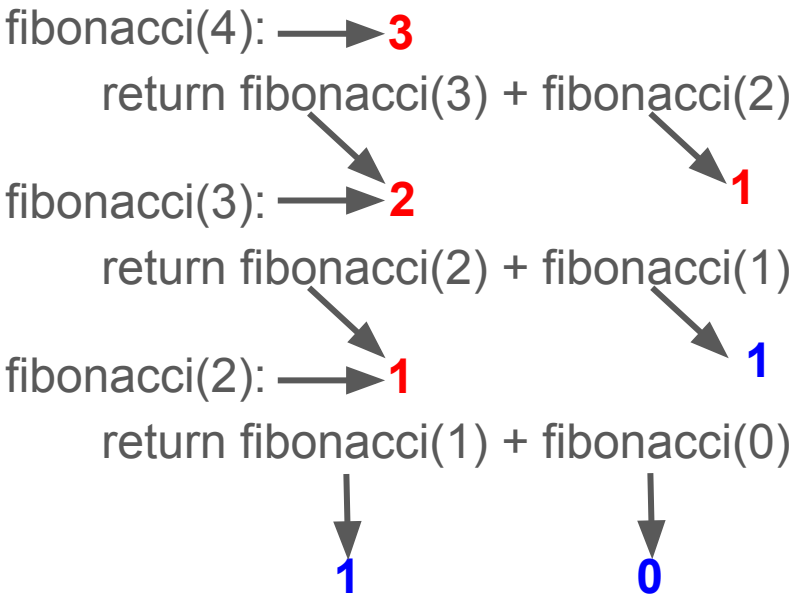
Recursive Step:

- all other subproblems
- calls a ***smaller*** subproblem

Fibonacci

```
int fibonacci(int n) {
    // base cases
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // recursive step
    return fibonacci(n-1) + fibonacci(n-2);
}
```

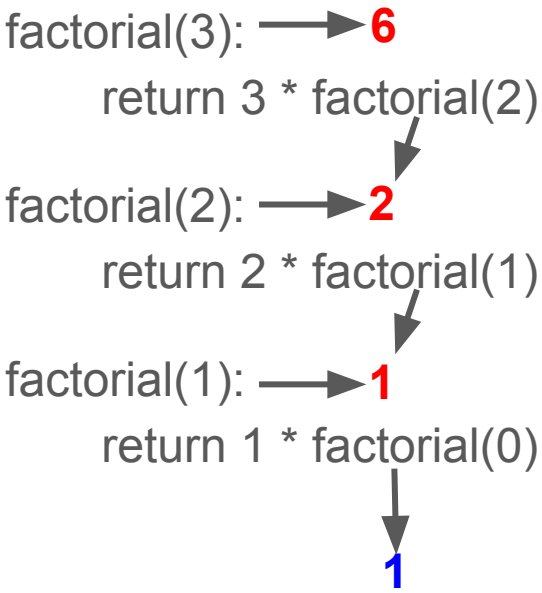
Fibonacci


fibonacci(4):
3
return fibonacci(3) + fibonacci(2)
fibonacci(3):
2
1
return fibonacci(2) + fibonacci(1)
fibonacci(2):
1
1
return fibonacci(1) + fibonacci(0)
1
0

Factorial

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Factorial


factorial(3): → 6
return 3 * factorial(2)
factorial(2): → 2
return 2 * factorial(1)
factorial(1): → 1
return 1 * factorial(0)
1

More practice

http://codingbat.com/java/Recursion-1