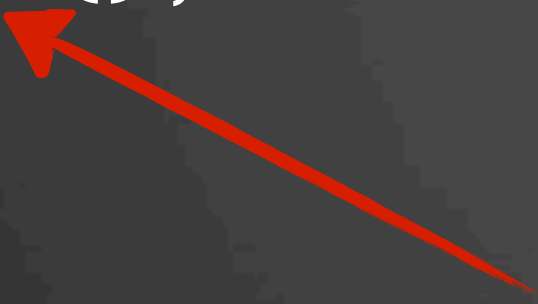

Functional programming

... in Java?

Immutability



```
for (Iterator i = products.iterator(); i.hasNext();) {  
    Product p = (Product) i.next();  
    if (!p.isAvailable(date)) {  
        i.remove();  
    }  
}  
  
return products;
```



```
List<Product> result = new LinkedList<Product>();
```

```
for (Product product : products) {  
    if (product.isAvailable(date)) {  
        result.add(product);  
    }  
}
```

```
return result;
```



```
List<Product> result = Collections.emptyList();
```

```
for (Product product : products) {  
    if (product.isAvailable(date)) {  
        result = ???;  
    }  
}
```

```
return result;
```



Dude, where's my state?

- If we cannot have variables that change their values?
 - Final variables
 - Immutable classes
 - Collections.unmodifiableList, etc.

Input to function calls



Exercise - Product repository

- Exercise 1:
 - Open the file `Exercise_1_Pure_Java_Functional_Test`
 - Implement the function `getAvailableProducts`
 - Use only immutable data structures (final variables / fields, unmodifiable collections)
 - Remove `@Ignore` from tests
 - Make all the tests pass



“Functional Java” solution?



Recursion



Recursion and state

- Functions calling themselves
- New input (state) for the next step in computation



Recursion and state

- Normal to use a helper function
 - Accumulator for results as part of input -> which is really state *thus far* in computation
 - Helper function is called from “main” function with initial state, e.g. the empty list as the accumulator
 - The helper function does the actual recursion



Walk-through of (one possible) solution





Turtles all the way down

- Build higher level abstractions on recursion



"Turtles all the way down"

Valentines Day 2007




Filter



```
List<Person> adults = new LinkedList<Person>();
for (Person p : people) {
    if (p.getAge() > 17) {
        adults.add(p);
    }
}
```

```
List<Integer> even = new LinkedList<Integer>();
for (Integer i : numbers) {
    if (i % 2 == 0) {
        even.add(i);
    }
}
```



```
F<Person, Boolean> isAdult = new F<Person, Boolean>() {
    public Boolean f(Person p) {
        return p.getAge() > 17;
    }
};
```

```
fj.data.List<Person> adults = iterableList(people).filter(isAdult);
```



First-class functions

- Assign functionality, not state, to variables
- Pass as parameters to other functions
- Return from functions

-> Higher order functions




Transform



```
List<String> names = new LinkedList<String>();  
for (Person p : people) {  
    names.add(p.getName());  
}
```

```
List<Integer> lengths = new LinkedList<Integer>();  
for (String s : strings) {  
    lengths.add(s.length());  
}
```



```
F<Person, String> toName = new F<Person, String>() {  
    public String f(Person p) {  
        return p.getName();  
    }  
};
```

```
fj.data.List<String> names = iterableList(people).map(toName)
```




Accumulate



```
int sum = 0;
for (Integer i : numbers) {
    sum = sum + i;
}
```

```
Set<String> names = new HashSet<String>();
for (Person p : people) {
    names.add(p.getName());
}
```



```
F2<Integer, Integer, Integer> add = new F2<Integer, Integer, Integer>() {
    public Integer f(Integer sum, Integer i) {
        return sum + i;
    }
};

int sum = iterableList(numbers).foldLeft(add, 0)
```



Exercices - Filter and map

- Exercise 2:
 - Solve the same problem as in exercise 1 using FunctionalJava
 - Open the file Exercise_2_Filter_Test
 - Implement the method getAvailableProducts using *functions* and *filter*
- Exercise 3:
 - Open the file Exercise_3_Transform_Test
 - Implement the method createOrderAlternatives using *functions* and *map*
 - Bonus: Get the test create_orders_within_timelimit to pass
- Bonus Exercise 4:
 - Solve Euler problem 2 - open Exercise_4_Fibonacci_Test and implement fibSlow and fibFastRecursive



Hint: Check the import statements!

Abstraction



Control flow

- Filter, map, fold etc. abstract away control flow
- Declarative programming (what, not how)
- Easily work on immutable data structures
- Enables different implementation strategies



In particular ...

```
iterableList(itineraries).map(journeyToOrder)
```



```
Strategy<Order> s = Strategy.simpleThreadStrategy();  
s.parMap1(journeyToOrder, iterableList(itineraries));
```



Expressive code

LambdaJ (Java 5+) vs. Java 8 Lambdas



```
List<String> names = new LinkedList<String>();  
for (Person p : people) {  
    names.add(p.getName());  
}
```



LambdaJ:

```
List<String> names = extract(people, on(Person.class).getName());
```

Java 8:

```
Iterable<String> names = people.map(p -> p.getName());
```



Are we there yet?

- No support for immutability in JDK libraries
- JVM doesn't support tail recursion
 - StackOverflowError
 - With tail-call optimization would not consume stack
 - Clojure, Scala work around it in their compilers

