Installation

git clone git://github.com/fredriv/fp-java.git

alternatively, download from https://github.com/fredriv/fp-java/zipball/master

cd fp-java

mvn test

Functional programming

... in Java?

Fredrik Vraalsen - Roots 2012 - 27.04.2012

knowit®

Agenda

- Intro to functional programming
- “Functional programming” in pure Java
- Libraries for functional programming in Java
- Exercises

Fredrik Vraalsen

fvr@knowit.no

Dad,
Java developer,
Scala enthusiast,
sci-fi fan and
Age of Conan assassin

@fredriv

What is functional programming?

Programming with functions

In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

$$f(x) = 2 * x$$

Referential transparency

An expression is said to be referentially transparent
if it can be replaced with its value
without changing the behavior of a program

(in other words, yielding a program that has
the same effects and output on the same input).

http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)

Mutability

Input/Output

Mutability

Side effects

Input/Output

Mutability

In a few languages enforced

- Charity
- Clean
- Curry
- Hope
- Miranda
- Haskell

http://en.wikipedia.org/wiki/List_of_programming_languages_by_category#Pure

In most functional languages a discipline

- Scala
- Clojure
- F#
- ...

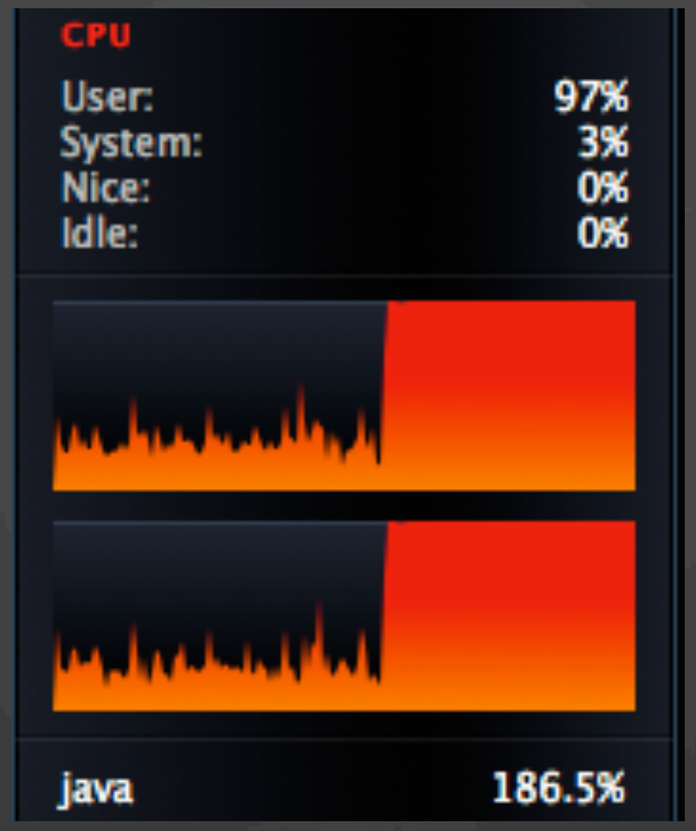
Why should we care?

Philosophical view

Practical view

Benefits of FP

- Concurrency
- Parallelism
- Robustness
- Testability
- Composability
- Better/higher abstractions
- Less code, less bugs!


CPU
User: 97%
System: 3%
Nice: 0%
Idle: 0%
java 186.5%

FP tries to avoid side effects

OO tries to encapsulate side effects

FP in Java

Immutability

```
for (Iterator<Integer> i = numbers.iterator(); i.hasNext();) {
    Integer number = i.next();
    if (number >= 100) {
        i.remove();
    }
}

return numbers;
```

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

for (Integer number : numbers) {
    if (number < 100) {
        result.add(number);
    }
}

return result;
```

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

for (Integer number : numbers) {
    if (number < 100) {
        result = ???;
    }
}

return result;
```

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

for (Integer number : numbers) {
    if (number < 100) {
        result = ???;
    }
}

return result;
```

Dude, where's my state?

- If we cannot have variables that change their values?
  - Final variables
  - Immutable classes ("no setters")
  - Collections.unmodifiableList, etc.

Input to function calls

Exercise 1 - FP in pure Java

- Open the file `Exercise_1_Pure_Java_Functional_Test.java`
- Implement the test `numbers_below_100` so that it passes
  - Uncomment @Ignore to make the test run
  - Use only immutable data structures (final variables / fields, unmodifiable collections)

- Installation:
  - git clone git://github.com/fredriv/fp-java.git
    - alternatively, download from https://github.com/fredriv/fp-java/zipball/master
  - cd fp-java
  - mvn test

Pure Java “Functional” solution?

Recursion

Recursion and state

- Functions calling themselves
- New input (state) for the next step in computation
- Tail recursion
  - Can conceptually be converted to a while loop
  - With tail-call optimization will not consume stack

Recursion and state

- Normal to use a helper function
  - Accumulator for results as part of input
    - which really represents the 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

Turtles all the way down

- Build higher-level abstractions on recursion

Walk-through of (one possible) solution


Aviary

Libraries for FP in Java

Filter

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

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

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

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

Functions

First-class functions

- Assign to variables
- Pass as parameters to other functions
- Return from functions

Functions in Java

- “Fake” it using anonymous inner classes
  - Implement an interface or abstract class
- Example: FunctionalJava

```
F<Integer, Boolean> greaterThanZero = new F<>() {
  public Boolean f(Integer i) {
    return i > 0;
  }
}
```

Filter contd.

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

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

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

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

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

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

```java
Matcher<Person> isAdult = having(on(Person.class).getAge(), greaterThan(17));

List<Person> adults = filter(isAdult, people);
```

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

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

```java
List<Person> adults = filter(having(on(Person.class).getAge(), greaterThan(17)), people);
```

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

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

```java
Matcher<Person> isAdult = having(on(Person.class).getAge(), greaterThan(17));

List<Person> adults = filter(isAdult, people);
```

Exercise 2 - Filter

- Open the file `Exercise_2_Filter_Test.java`
- Implement the tests so that they pass, using FunctionalJava and LambdaJ
  - Uncomment @Ignore to make the test run
  - Hint: Look at imports to see useful classes, static methods

- Documentation:
  - http://code.google.com/p/lambdaj/wiki/LambdajFeatures
  - http://lambdaj.googlecode.com/svn/trunk/html/apidocs/index.html

  - http://code.google.com/p/functionaljava/
  - http://functionaljava.googlecode.com/svn/artifacts/3.0/javadoc/index.html

Transform

```
List<String> titles = new LinkedList<>();
for (Abstract a : abstracts) {
  titles.add(a.getTitle());
}
```

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

```
F<Abstract, String> toTitle = new F<>() {
  public String f(Abstract a) {
    return a.getTitle();
  }
};
```

```
List<String> titles = new LinkedList<>();
for (Abstract a : abstracts) {
  titles.add(a.getTitle());
}
```

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

```
F<Abstract, String> toTitle = fFor(callsTo(Abstract.class).getTitle());

fj.data.List<String> titles = iterableList(abstracts).map(toTitle)
```

```
List<String> titles = new LinkedList<>();
for (Abstract a : abstracts) {
  titles.add(a.getTitle());
}
```

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

```
List<String> titles = extract(abstracts, on(Abstract.class).getTitle())

List<String> titles = with(abstracts).extract(on(Abstract.class).getTitle())
```

Exercise 3 - Transform

- Open the file `Exercise_3_Transform_Test.java`
- Implement the tests so that they pass, using FunctionalJava and LambdaJ
  - Uncomment @Ignore to make the test run
  - Hint: Look at imports to see useful classes, static methods

- Documentation:
  - http://code.google.com/p/lambdaj/wiki/LambdajFeatures
  - http://code.google.com/p/functionaljava/
  - http://code.google.com/p/funcito/

Accumulate

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

```java
Map<Integer, Employee> byId = new HashMap<>();
for (Employee e : employees) {
  byId.put(e.getId(), e);
}
```

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

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

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

```
Map<Integer, Employee> byId = new HashMap<>();
for (Employee e : employees) {
  byId.put(e.getId(), e);
}
```

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

```
[17, 123, 314, 42].foldLeft(add, 0)
```

```
=>  f(f(f(f(0, 17), 123), 314), 42)
```

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

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

```
PairAggregator<Integer> adder = new PairAggregator<> {
  protected Integer aggregate(Integer sum, Integer i) { return sum + i; }
  protected Integer emptyItem() { return 0; }
}

int sum = aggregate(numbers, adder);
```

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

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

```
int sum = sum(numbers).intValue();
```

Exercise 4 - Accumulate

- Open the file `Exercise_4_Accumulate_Test.java`
- Implement the tests so that they pass, using FunctionalJava and LambdaJ
  - Uncomment @Ignore to make the test run
  - Hint: Look at imports to see useful classes, static methods

Closures

Closures

- A closure is a function that is closed over its free variables

Say what!?!?

Closures

- A closure is a function that is closed over its free variables

```
public Collection<Person> getPeopleOfAge(List<Person> people, final int age) {
  return iterableList(people).filter(new F<Person, Boolean> {
    public Boolean f(Person p) { return p.getAge() == age; }
  }).toCollection();
}
```

Closures

- A closure is a function that is closed over its free variables

```
public Collection<Person> getPeopleOfAge(List<Person> people, final int age) {
  F<Person, Boolean> isOfAge = new F<Person, Boolean> {
    public Boolean f(Person p) { return p.getAge() == age; }
  };
  return iterableList(people).filter(isOfAge).toCollection();
}
```

Exercise 5 - Closure

- Open the file `Exercise_5_Closure_Test.java`
- Implement the tests so that they pass, using FunctionalJava
  - Uncomment @Ignore to make the test run
  - Hint: Look at imports to see useful classes, static methods

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(people).map(toName)
```

```
Strategy<Order> s = Strategy.simpleThreadStrategy();
s.parMap1(toName, iterableList(people));
```

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:

```
Mapper<Person, String> toName = p -> p.getName();      // alt: Person#getName
Iterable<String> names = people.map(toName);
```

```
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());
```

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

LambdaJ:

```java
LambdaJMatcher<Person> isAdult = having(on(Person.class).getAge(), greaterThan(17));
List<Person> adults = select(people, isAdult);
```

Java 8:

```java
Predicate<Person> isAdult = p -> p.getAge() > 17;
Iterable<Person> adults = people.filter(isAdult);
```

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

LambdaJ:

```java
LambdaJMatcher<Person> isAdult = having(on(Person.class).getAge(), greaterThan(17));
List<Person> adults = select(people, isAdult);
```

Java 8:

```java
Iterable<Person> adults = people.filter(p -> p.getAge() > 17);
```

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

LambdaJ:

```
int sum = sum(numbers);
```

Java 8:

```
Operator<Integer> sum = (i1, i2) -> i1 + i2;
int sum = numbers.reduce(0, sum);
```

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

LambdaJ:

```
int sum = sum(numbers);
```

Java 8:

```
int sum = numbers.reduce(0, (i1, i2) -> i1 + i2);
```

Benefits of FP

- Concurrency
- Parallelism
- Robustness
- Testability
- Composability
- Better/higher abstractions
- Less code, less bugs!

Questions?

Fredrik Vraalsen

E-mail: fvr@knowit.no

Twitter: @fredriv