The Joy of Functional Programming

NICOLE RAUCH
softwareentwicklung &
entwicklungscoaching

Functional, isn't that a totally esoteric subject?!

Functional, isn't that a totally esoteric subject?!

- ABN AMRO Amsterdam *Risk analysis in investment banking*
- AT&T *Network security: processing of internet abuse complaints*
- Bank of America Merril Lynch
  Backend data transformation and loading
- Barclays Capital Quantitative Analytics Group
  Mathematical modelling of equity derivatives
- Bluespec, Inc. *Modelling & verification of integrated circuits*
- Credit Suisse *Checking, manipulating and transforming spreadsheets*
- Deutsche Bank Equity Proprietary Trading, Directional Credit Trading *All its software infrastructure*
- Facebook *Internal tools*
- Factis Research, Freiburg *Mobile solutions (backend)*
- fortytools gmbh *web-based productivity tools - REST-backend*
- Functor AB, Stockholm *static analysis*

Functional, isn't that a totally esoteric subject?!

- Galois, Inc *Security, information assurance and cryptography*
- Google *Internal projects*
- IMVU, Inc. *Social entertainment*
- JanRain *Network and web software*
- MITRE *Analysis of kryptographic protocols*
- New York Times *Image processing for the New York Fashion Week*
- NVIDIA *In-house tools*
- Parallel Scientific *High-availability cluster management system*
- Sankel Software *CAD/CAM, gaming and computer animation*
- Silk, Amsterdam *Filter and visualize large amounts of information*
- Skedge Me *Online scheduling platform*
- Standard Chartered *Wholesale banking business*
- Starling Software, Tokio *Commercial automated options trading system*
- Suite Solutions *Management of large sets of technical documentation*

(Quelle: `http://www.haskell.org/haskellwiki/Haskell_in_industry`)

Well-known functional languages

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

Lisp

Scala

Well-known functional languages

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

(Java 8)

Lisp

Scala

(JavaScript)

Well-known functional languages

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

(Java 8)

Lisp

Scala

(JavaScript)

Now, what is so special about functional programming?

Now, what is so special about functional programming?

Immutability

Now, what is so special about functional programming?

Immutability

- Each variable can only be assigned to once

Now, what is so special about functional programming?

Immutability

- Each variable can only be assigned to once
- Not directly supported in Java

Now, what is so special about functional programming?

Immutability

- Each variable can only be assigned to once
- Not directly supported in Java
- Can easily be put into effect:

```
class Point {
  private int x, y;
  public Point (int x, int y) {
    this.x = x;
    this.y = y;
  }
  // only read x and y in the remaining code
}
```

Now, what is so special about functional programming?

Absence of side-effects

Now, what is so special about functional programming?

Absence of side-effects

- "everything that changes the execution of a computer program or the outside world without being returned from a function"

Now, what is so special about functional programming?

Absence of side-effects

- "everything that changes the execution of a computer program or the outside world without being returned from a function"
  - Input and output
  - Exceptions
  - Logging
  - Dependency on (external) configurations
  - Change of state
  - Nondeterminism (e.g. use of a random number generator)

Now, what is so special about functional programming?

Absence of side-effects

- "everything that changes the execution of a computer program or the outside world without being returned from a function"
  - Input and output
  - Exceptions
  - Logging
  - Dependency on (external) configurations
  - Change of state
  - Nondeterminism (e.g. use of a random number generator)
- Some languages even indicate side-effects in the type signature

Now, what is so special about functional programming?

Absence of side-effects

not directly supported by Java 8 either - coding rules help

Now, what is so special about functional programming?

Absence of side-effects

not directly supported by Java 8 either - coding rules help

```
class SeparationOfSideEffects {

  public void withSideEffect(){
    String initialValue = System.console().readLine();
    String result = withoutSideEffect(initialValue);
    System.out.println("The Result: " + result);
  }

  public static String withoutSideEffect(String initialValue){
    return /* function result */ ;
  }
}
```

Now, what is so special about functional programming?

Functions are „first order citizens“

Now, what is so special about functional programming?

Functions are „first order citizens“

Functions can be treated in the same way as strings or numbers

Functions are values

Functions are values

Java 8: Static Methods

```
class Examples { static int staticTimes (int x, int y) { return x * y; } }

IntBinaryOperator timesVar = Examples::staticTimes;

timesVar.applyAsInt(3, 5);                         // 15
```

Functions are values

Java 8: Object methods

```
class Examples { int times (int x, int y) { return x * y; } }

Examples examples = new Examples();
IntBinaryOperator timesVar = examples::times;

timesVar.applyAsInt(3, 5);                    // 15
```

Functions are values

Java 8: Lambdas

```
IntBinaryOperator times = (x, y) -> x * y;

times.applyAsInt(3, 5);                          // 15
```

Functions are values

Java 8: Lambdas (with self-defined function interface)

```
interface TimesFunction { int eval(int x, int y); }

TimesFunction times = (x, y) -> x * y;

times.eval(3, 5);                                    // 15
```

Functions are values

Java 8: Lambdas (with self-defined function interface)

```
interface TimesFunction { int eval(int x, int y); }

TimesFunction times = (x, y) -> x * y;

times.eval(3, 5);                                    // 15
```

Haskell:

```
times x y = x * y

timesVar = times

timesVar 3 5                                        -- 15
```

Functions are function parameters

Functions are function parameters

Java 8:

```
class Examples {
    static int apply(IntUnaryOperator func, int arg) {
        return func.applyAsInt(arg);
    }
}

Examples.apply(x -> 3 * x, 5);                 // 15
```

Functions are function parameters

Java 8:

```
class Examples {
    static int apply(IntUnaryOperator func, int arg) {
        return func.applyAsInt(arg);
    }
}

Examples.apply(x -> 3 * x, 5);                    // 15
```

Haskell:

```
apply func arg = func arg

apply (\ x -> 3 * x) 5                            -- 15
```

Functions are return values

Functions are return values

Java 8:

```
interface FunctionFunction { IntUnaryOperator eval(int x); }

FunctionFunction times = x -> { return y -> x * y; };

times.eval(3).applyAsInt(5);                    // 15
```

Functions are return values

Java 8:

```
interface FunctionFunction { IntUnaryOperator eval(int x); }

FunctionFunction times = x -> { return y -> x * y; };

times.eval(3).applyAsInt(5);                    // 15
```

Haskell:

```
times x = (\y -> x * y)

times 3 5                                       -- 15
```

Strange... ?!

Java 8: Two different invocations

```
IntBinaryOperator times = (x, y) -> x * y;
times.applyAsInt(3, 5);                                   // 15

FunctionFunction times = x -> { return y -> x * y; };
times.eval(3).applyAsInt(5);                              // 15
```

Haskell: Two identical invocations

```
times x y = x * y
times 3 5                                                 -- 15

times x = (\y -> x * y)
times 3 5                                                 -- 15
```

Currying! (also known as Schönfinkeling)

Currying! (also known as Schönfinkeling)

In some functional languages we write:

```
times x y = x * y
```

but actually the following happens:

```
times x = (\y -> x * y)
```

Currying! (also known as Schönfinkeling)

In some functional languages we write:

```
times x y = x * y
```

but actually the following happens:

```
times x = (\y -> x * y)
```

Because functions always take exactly one argument

Currying! (also known as Schönfinkeling)

In some functional languages we write:

```
times x y = x * y
```

but actually the following happens:

```
times x = (\y -> x * y)
```

Because functions always take exactly one argument

Useful for partial evaluation:

```
times x y = x * y

times3 = times 3

times3 5                          -- 15
```

Important library functions: filter

- filter or select

Important library functions: filter

- filter or select
- Takes a collection and a function
- Returns those elements of the collection for which the function yields `true`

Important library functions: filter

- filter or select
- Takes a collection and a function
- Returns those elements of the collection for which the function yields `true`

Java 8:

```
Stream<Integer> filteredStream =
        asList(1, 2, 3, 4).stream().filter(x -> x % 2 == 0);

filteredStream.toArray();                             // new Integer[]{2, 4}
filteredStream.collect(Collectors.toList());          // Liste mit 2 und 4
```

Important library functions: filter

- filter or select
- Takes a collection and a function
- Returns those elements of the collection for which the function yields `true`

Java 8:

```
Stream<Integer> filteredStream =
        asList(1, 2, 3, 4).stream().filter(x -> x % 2 == 0);

filteredStream.toArray();                             // new Integer[]{2, 4}
filteredStream.collect(Collectors.toList());          // Liste mit 2 und 4
```

Haskell:

```
filter (\x -> x `mod` 2 == 0) [1,2,3,4]                         -- [2,4]
```

Important library functions: map

- map or collect

Important library functions: map

- map or collect
- Takes a collection and a function
- Yields a collection of the results of applying the function to the elements of the original collection

Important library functions: map

- map or collect
- Takes a collection and a function
- Yields a collection of the results of applying the function to the elements of the original collection

Java 8:

```
Arrays.asList(1, 2, 3, 4).stream().map(x -> x + 5).toArray();
                                                  // new Integer[]{6, 7, 8, 9}
```

Important library functions: map

- map or collect
- Takes a collection and a function
- Yields a collection of the results of applying the function to the elements of the original collection

Java 8:

```
Arrays.asList(1, 2, 3, 4).stream().map(x -> x + 5).toArray();
                                                      // new Integer[]{6, 7, 8, 9}
```

Haskell:

```
map (\x -> x + 5) [1,2,3,4]                                    -- [6,7,8,9]
```

Important library functions: reduce

- reduce or foldl / foldr or inject

Important library functions: reduce

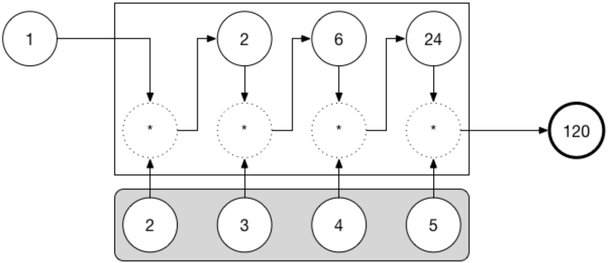
- reduce or foldl / foldr or inject
- Takes a collection, a function and an initial value
- Merges initial value and first collection entry using the function
- Merges the result and the next collection entry
- Continues for all collection entries, yielding a single result

Important library functions: reduce

- reduce or foldl / foldr or inject

Java 8:

```
Arrays.asList(2, 3, 4, 5).stream().reduce(1, (x, y) -> x*y);        // 120
```



Important library functions: reduce

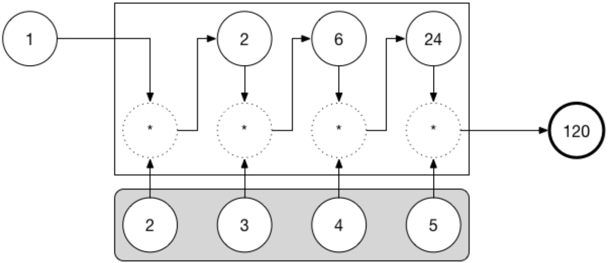
- reduce or foldl / foldr or inject

Java 8:

```
Arrays.asList(2, 3, 4, 5).stream().reduce(1, (x, y) -> x*y);          // 120
```

Haskell:

```
foldl (*) 1 [2,3,4,5]                                                 -- 120
```



Type Inference

- Haskell: strong static type system
- Lightweight use due to type inference
- Derives the most general type

Type Inference

- Haskell: strong static type system
- Lightweight use due to type inference
- Derives the most general type

Example:

```
f x = x
```

Type Inference

- Haskell: strong static type system
- Lightweight use due to type inference
- Derives the most general type

Example:

```
f x = x
```

Type:

```
f :: a -> a
```

a : Type variable (comparable to generics in Java etc.)
-> Function type (argument type to the left, result type to the right)

Type Inference (2)

```
f x = x + 1
```

Type Inference (2)

```
f x = x + 1
```

Type:

```
f :: Num a => a -> a
```

`Num a` : Type class: Restricts the type variable a to numerical types

Type Inference (2)

```
f x = x + 1
```

Type:

```
f :: Num a => a -> a
```

`Num a` : Type class: Restricts the type variable a to numerical types

Recommended: Always annotate type signature! Helps to validate your assumptions.

A simple calculation

$$sum = \sum_{i=1}^{10} i^2$$

A simple calculation

$$sum = \sum_{i=1}^{10} i^2$$

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

Excursion: Clean Code

Single Responsibility Principle

Excursion: Clean Code

Single Responsibility Principle

How many responsibilities does this code have?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

Excursion: Clean Code

Single Responsibility Principle

How many responsibilities does this code have?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Creating the sequence of numbers from 1 to 10

Excursion: Clean Code

Single Responsibility Principle

How many responsibilities does this code have?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Creating the sequence of numbers from 1 to 10
- Calculating the square of a number

Excursion: Clean Code

Single Responsibility Principle

How many responsibilities does this code have?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Creating the sequence of numbers from 1 to 10
- Calculating the square of a number
- Calculating the square of each number in the sequence

Excursion: Clean Code

Single Responsibility Principle

How many responsibilities does this code have?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Creating the sequence of numbers from 1 to 10
- Calculating the square of a number
- Calculating the square of each number in the sequence
- Calculating the sum of two numbers

Excursion: Clean Code

Single Responsibility Principle

How many responsibilities does this code have?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Creating the sequence of numbers from 1 to 10
- Calculating the square of a number
- Calculating the square of each number in the sequence
- Calculating the sum of two numbers
- Calculating the sum of all squares

Separation of Concerns

- Creating the sequence of numbers from 1 to 10
- Calculating the square of a number
- Calculating the square of each number in the sequence
- Calculating the sum of two numbers
- Calculating the sum of all squares

Separation of Concerns

- Creating the sequence of numbers from 1 to 10

```
IntStream sequence = IntStream.rangeClosed(1, 10);
```

- Calculating the square of a number
- Calculating the square of each number in the sequence
- Calculating the sum of two numbers
- Calculating the sum of all squares

Separation of Concerns

- Creating the sequence of numbers from 1 to 10

```
IntStream sequence = IntStream.rangeClosed(1, 10);
```

- Calculating the square of a number

```
IntUnaryOperator square = x -> x*x;
```

- Calculating the square of each number in the sequence
- Calculating the sum of two numbers
- Calculating the sum of all squares

Separation of Concerns

- Creating the sequence of numbers from 1 to 10

```
IntStream sequence = IntStream.rangeClosed(1, 10);
```

- Calculating the square of a number

```
IntUnaryOperator square = x -> x*x;
```

- Calculating the square of each number in the sequence

```
IntStream squaredSequence = sequence.map(square);
```

- Calculating the sum of two numbers

- Calculating the sum of all squares

Separation of Concerns

- Creating the sequence of numbers from 1 to 10

```
IntStream sequence = IntStream.rangeClosed(1, 10);
```

- Calculating the square of a number

```
IntUnaryOperator square = x -> x*x;
```

- Calculating the square of each number in the sequence

```
IntStream squaredSequence = sequence.map(square);
```

- Calculating the sum of two numbers

```
IntBinaryOperator add = (x,y) -> x+y;
```

- Calculating the sum of all squares

Separation of Concerns

- Creating the sequence of numbers from 1 to 10

```
IntStream sequence = IntStream.rangeClosed(1, 10);
```

- Calculating the square of a number

```
IntUnaryOperator square = x -> x*x;
```

- Calculating the square of each number in the sequence

```
IntStream squaredSequence = sequence.map(square);
```

- Calculating the sum of two numbers

```
IntBinaryOperator add = (x,y) -> x+y;
```

- Calculating the sum of all squares

```
Integer sum = squaredSequence.reduce(0, add);
```

Combining the components

Java 8:

```
IntStream.rangeClosed(1, 10).map(x -> x*x).reduce(0, (x,y) -> x+y);  // 385
```

Combining the components

Java 8:

```
IntStream.rangeClosed(1, 10).map(x -> x*x).reduce(0, (x,y) -> x+y);  // 385
```

Haskell:

```
foldl (+) 0 (map (\x -> x*x) [1..10])                                 -- 385
```

Combining the components

Java 8:

```
IntStream.rangeClosed(1, 10).map(x -> x*x).reduce(0, (x,y) -> x+y);  // 385
```

Haskell:

```
foldl (+) 0 (map (\x -> x*x) [1..10])                                -- 385
```

or

```
(>.>) x f = f x
[1..10] >.> map (\x -> x*x) >.> foldl (+) 0                          -- 385
```

Phew!

OK, everybody take a deep breath :-)

Pattern Matching

Fibonacci-Function „naïve“:

```
fib x = if x < 2 then x else fib (x-1) + fib (x-2)
```

Pattern Matching

Fibonacci-Function „naïve“:

```
fib x = if x < 2 then x else fib (x-1) + fib (x-2)
```

Fibonacci-Function with Pattern Matching:

```
fib 0 = 0
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

Algebraic Datatypes

Binary tree:

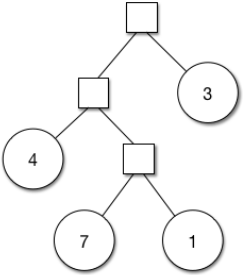
```
data Tree =
       Node Tree Tree
      | Leaf Int
```

Algebraic Datatypes

Binary tree:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```



Algebraic Datatypes

Binary tree:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summary function:

Algebraic Datatypes

Binary tree:

```
data Tree =
       Node Tree Tree
      | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summary function:

```
treeSum (Leaf x) = x
```

Algebraic Datatypes

Binary tree:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summary function:

```
treeSum (Leaf x) = x

treeSum (Node m n) = treeSum m + treeSum n
```

Algebraic Datatypes

Binary tree:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summary function:

```
treeSum (Leaf x) = x

treeSum (Node m n) = treeSum m + treeSum n
```

```
treeSum myTree                                   -- 15
```

Bottom line

- Functional programming is more common than you may have expected
- Some of it can be integrated into non-functional coding
- Many languages have functional aspects or additional modules

Bottom line

- Functional programming is more common than you may have expected
- Some of it can be integrated into non-functional coding
- Many languages have functional aspects or additional modules

Reference:

- Haskell: `http://www.haskell.org`

Thank you very much!

Code & slides on GitHub:

`https://github.com/NicoleRauch/FunctionalProgrammingForBeginners`

Nicole Rauch

E-Mail	`info@nicole-rauch.de`
Twitter	`@NicoleRauch`
Web	`http://www.nicole-rauch.de`