# Lecture 22

## Conclusions

SOEN 6441, Summer 2018



René Witte
Department of Computer Science
and Software Engineering
Concordia University

# Outline

1. Review and Outlook
2. Functional Programming
3. Summary
4. Notes and Further Reading

René Witte

Concordia University

# Motivation

René Witte

Concordia University



## Multi-core Systems

- Increasing data sizes to process
- Code must run faster on multi-core systems
- Difficult and error-prone with traditional object-oriented techniques (manipulating fields, external iteration, synchronizing threads)

## Solution: Functional Programming Techniques

- Functions without side-effects
- Immutable data structures
- Code as objects

# Review: Java 8





**Goal: reuse code, like `filter`**

Very verbose prior to Java 8 (anonymous classes)

## Behavior parameterization

- Passing a lambda

```
apple -> apple.getWeight() > 150
```

- Passing a method reference

```
Apple::isHeavy
```

# Review: Java 8 (II)





**Goal: Parallel processing of large data sets**

External iteration in old `Collections`

- Complex operations require multiple traversal of the same data set
- Difficult to parallelize

**Streams API**

Parallel, functional-style declarative processing of large collections.

- Internal iteration
- Passing behavior through lambdas

# Review: Java 8 (III)

René Witte



**Goal: Distribute processing on multiple cores**

- Java 5 `Future` could spawn a method call onto a new thread
- Not possible to join multiple futures together without blocking

**CompletableFuture**

Functional-style asynchronous computing using composable `Future`s

- non-blocking composition of futures, using lambdas
- using `thenCompose`, `thenCombine`, `allOf`, etc.

# Review: Java 8 (IV)

René Witte



## Optional

Functional-style modeling of missing values

- explicit modeling of missing values
- internal testing instead of external `null` checks
- functional style processing through `map`, `flatMap`, `filter`, etc.

## Default Methods

Implementations in interfaces; multiple inheritance of behavior.

# Review: Reactive Programming



## Reactive Manifesto

Responsive, Resilient, Elastic and Message Driven

## Programming Concepts

- Actor-based Programming, using Akka
- Asynchronous Programming (CompletableFuture)
- Functional Programming (lambdas)
- Reactive Stream Processing (Java 9)

# Java 9

## JDK 9

Released 2017-09-21 (Java SE 9), 2018-01-16 (Java SE 9.0.4)

### New Features

- Modularization of the JDK
- Java shell jshell (REPL)
- Ahead-of-Time Compilation (Graal compiler)
- XML Catalogs
- Java Linker jlink
- Immutable Collections
- Reactive Streams

### Java 9 Reactive Streams

Standard for asynchronous stream processing with non-blocking back pressure

- New `Flow` class in Java 9
- Designed by Netflix, Oracle, Typesafe, Twitter, Red Hat, and others
- Implementations: Akka Streams, Spring/Pivotal Reactor, Netflix RxJava, Slick

René Witte
Concordia University



## JDK 10

Released 2018-03-20 (Java SE 10), 2018-07-17 (Java SE 10.0.2)

### New Features

- Local-Variable Type Inference:

```
var list = new ArrayList<String>();   // infers ArrayList<String>
var stream = list.stream();           // infers Stream<String>
```

- New APIs for Creating Unmodifiable Collections

```
Stream.of("foo", "bar").collect(toUnmodifiableList());
```

- Some default Root Certificates in the JDK
- various other enhancements

### Short-term releases

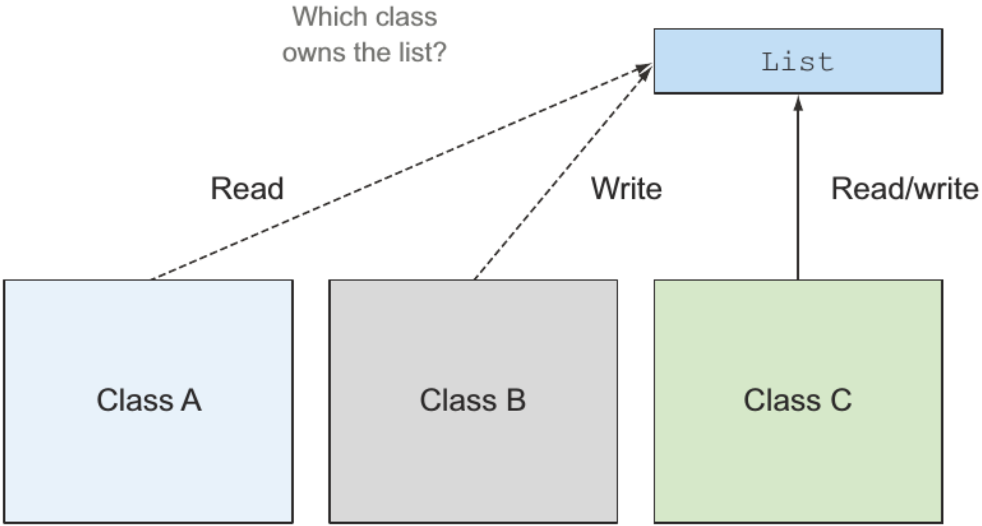
New JDK release cycles: Java 9 and 10 are short-term releases

- Support for 6 months only
- JDK 9 reached end-of-life in March 2018!
- Next long-term release (LTS) will be JDK 11

# Outline

# Shared Mutable Data

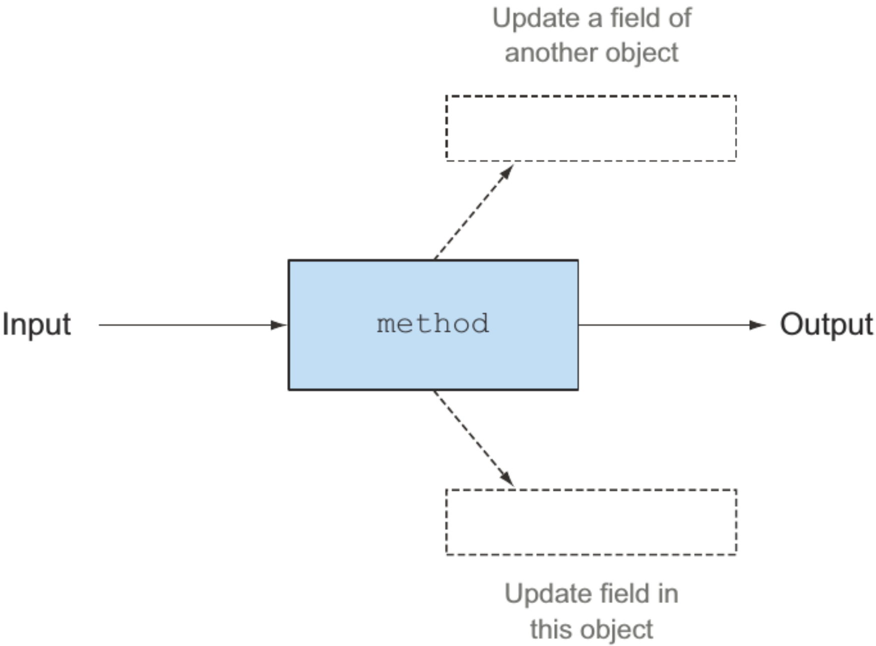
## Side effects vs. side-effect free (pure) programming



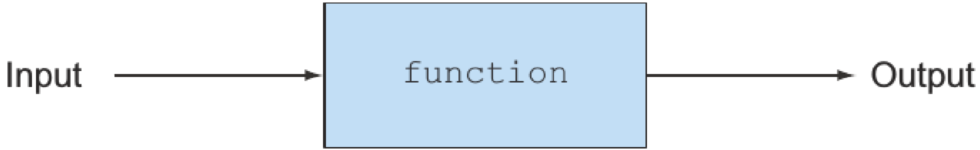



René Witte
Concordia University

# Side effects



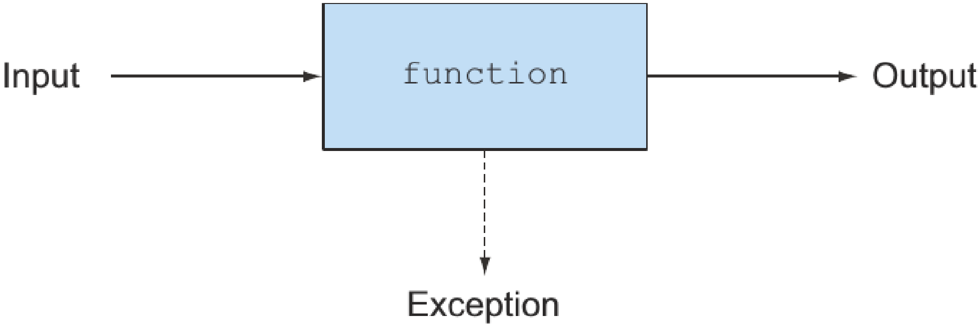
# Functional style: no side-effects!






## Rules

- Method can only mutate local variables
- Method cannot throw exceptions






René Witte
Concordia University

# Functions everywhere

René Witte



## Functions as data (“first-class functions”)

```
Function<String, Integer> strToInt = Integer::parseInt;
```

# Object-oriented vs. declarative programming



## Object-oriented style

```
Transaction mostExpensive = transactions.get(0);
if(mostExpensive == null)
    throw new IllegalArgumentException("Empty list of transactions")

for(Transaction t: transactions.subList(1, transactions.size())){
  if(t.getValue() > mostExpensive.getValue()){
    mostExpensive = t;
  }
}
```

## Declarative style

```
Optional<Transaction> mostExpensive =
  transactions.stream().max(comparing(Transaction::getValue));
```

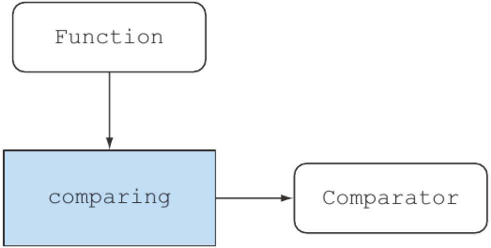
# Higher-order Functions





## Example

```
Comparator<Apple> c =
  comparing(Apple::getWeight);
```






### Higher-order functions in programming

Functions that can do at least one of the following:

- Take one or more functions as parameter
- Return a function as result

# Currying





### Named after Haskell Curry (1900–1982)

Currying: Translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

### Example

Method to convert units in programs (e.g., ℉ → ℃, € → $):

```
static double converter(double x, double f, double b) {
  return x * f + b;
}
```

(multiply by conversion factor, adjust baseline if relevant)

### Issue

Need to provide all three arguments for every conversion

- error-prone
- makes code bloated and harder to read

# Applying currying

René Witte



## Provide a Factory for one-argument conversion functions

```java
static DoubleUnaryOperator curriedConverter(double f, double b){
  return (double x) -> x * f + b;
}
```

## Using

Defining converter functions:

```java
DoubleUnaryOperator convertCtoF = curriedConverter(9.0/5, 32);
DoubleUnaryOperator convertUSDtoGBP = curriedConverter(0.6, 0);
DoubleUnaryOperator convertKmtoMi = curriedConverter(0.6214, 0);
```

Applying a converter function:

```java
double gbp = convertUSDtoGBP.applyAsDouble(1000);
```

## Theoretical View

$f(x, y) = (g(x))(y)$

# Persistent Data Structures

René Witte



## Functional Data Structures

E.g., `String.replace`:

```
"Doncordia".replace('D', 'C')
```

## Destructive Updates

E.g., `List.add()`

# Example: TrainJourney





## Mutable `TrainJourney` class

```
class TrainJourney {
  public int price;
  public TrainJourney onward;
  public TrainJourney(int p, TrainJourney t) {
    price = p;
    onward = t;
  }
}
```

## Linking two journeys

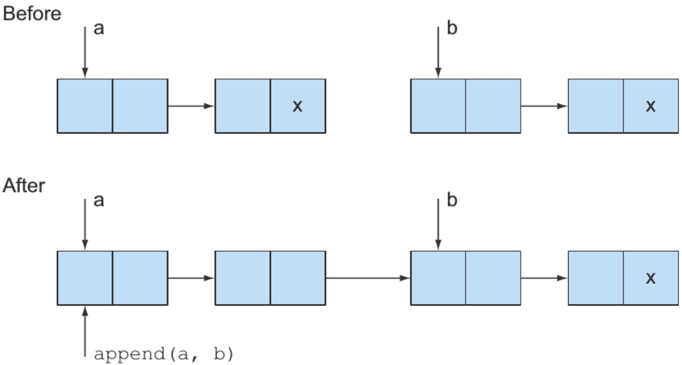
```
static TrainJourney link(TrainJourney a, TrainJourney b){
  if (a==null) return b;
  TrainJourney t = a;
  while(t.onward != null){
    t = t.onward;
  }
  t.onward = b;
  return a;
}
```

# The issue with destructive updates



## Example

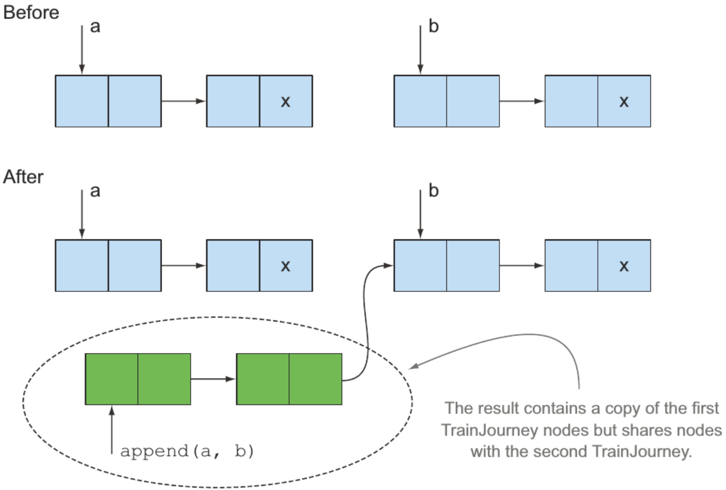
```
TrainJourney montrealToOttawa = ...;
TrainJourney ottawaToToronto = ...;

john.setJourney(mtlToOttawa);
jane.setJourney(mtlToOttawa);

jane.getJourney().link(ottawaToToronto);
```



# Solution

## Functional-style append

```
static TrainJourney append(TrainJourney a, TrainJourney b){
  return a==null ? b : new TrainJourney(a.price, append(a.onward, b));
}
```






René Witte
Université Concordia University

# Combinators





**Combinator**
Higher-order function that:

- accepts two (or more) functions as input
- produces another function combining these functions.

**Example: `CompletableFuture.thenCombine`**

```
thenCombine( CompletionStage<? extends U> other,
             BiFunction<? super T,? super U,? extends V> fn )
```

# Programming Combinators

## Repeat

Write a function `repeat` that applies a function repeatedly, e.g.,

```
repeat(3, (Integer x) -> 2*x);
```

which results in the function `x ->(2*(2*(2*x)))`.

## Solution

```
static <A> Function<A,A> repeat(int n, Function<A,A> f) {
  return n==0 ? x -> x : compose(f, repeat(n-1, f));
}
```

## Testing

```
System.out.println(repeat(3, (Integer x) -> 2*x).apply(10));
```

# Y Combinator

René Witte



### Discovered by Haskell Curry

In lambda calculus:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

See https://en.wikipedia.org/wiki/Fixed-point_combinator

### Example

Implement a recursive factorial ($n$!):

```
public static long factorial(long n) {
  return n == 1 ? 1 : n * factorial(n-1); }
```

*but without using the function name in the body!*

See http://rosettacode.org/wiki/Y_combinator#Java

### Importance

- Can be used to create recursion in non-recursive languages
- Important for proving that $\lambda$-calculus is Turing complete

# Summary



## Functional Programming

- Functional-style programming promotes side-effect-free methods and declarative programming.
- First-class functions are functions that can be passed as arguments, returned as results, and stored in data structures.
- A higher-order function is a function that takes at least one or more functions as input or returns another function. Typical higher-order functions in Java include `comparing`, `andThen`, and `compose`.
- Currying is a technique that lets you modularize functions and reuse code.
- A persistent data structure preserves the previous version of itself when it's modified. As a result, it can prevent unnecessary defensive copying.
- Combinators are a functional idea that combines two or more functions or other data structures.

# Outline

# Reading Material





## Required

- [UFM14, Chapter 13] (Thinking functionally)
- [UFM14, Chapter 14] (Functional programming techniques)
- [UFM14, Chapter 16] (Conclusions)

# References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
*Java 8 in Action: Lambdas, streams, and functional-style programming*.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.