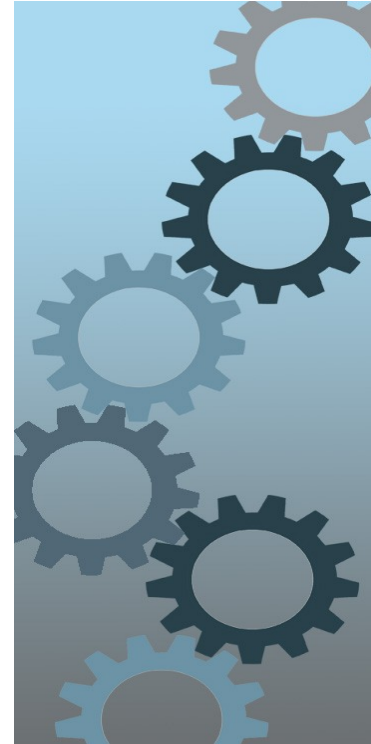


# *Programming Concepts And Paradigms*

Functional Programming  
Lambdas and Streams

Marcel Baumann



# Content

- Functional Approaches
  - Conditional expressions, switch, lambda
- Functional Programming
  - Composition
  - Streams Operations
  - Currying
- Standard Library

# Functional vs OOP

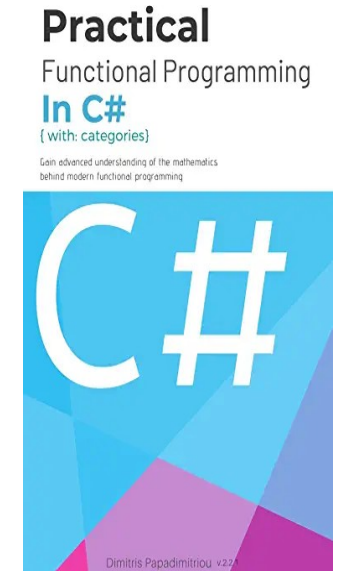
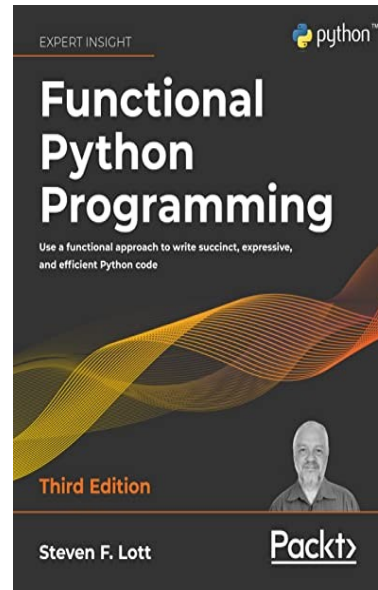
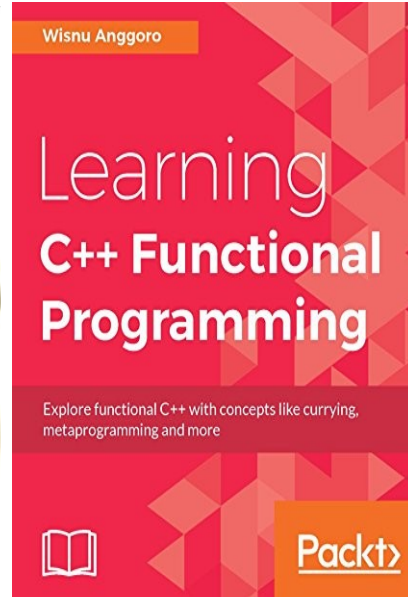
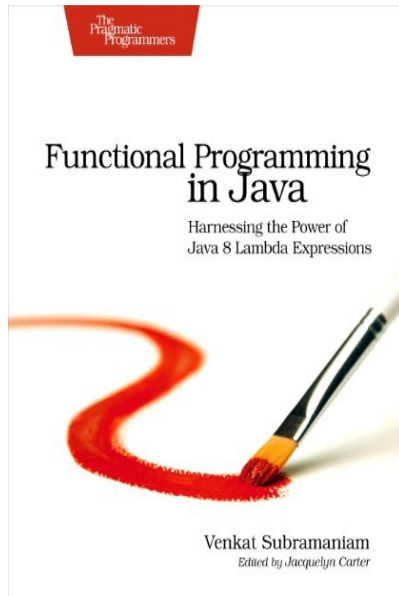
## Functional

- Order of execution isn't always important, it can be asynchronous
- **Functions** are basic building elements to work with data structures
- Follows the **declarative** paradigm
- Focuses on the **result** desired and its final conditions
- Ensures immutability; programs should be stateless
- Primary activity is writing **new functions** and **composing existing ones**

## Object-Oriented

- A specific execution sequence of statements is crucial
- **Objects** are the main abstractions for data
- Follows the **imperative** paradigm
- Focuses on **how** the desired result can be achieved
- **State changes** are an important part of the execution
- Primary activity is **building, extending** and **composing objects**

# Functional is Hype



# Build-in Functional Concepts in Java

- Ternary operator instead of *if () then {} else {}*
  - Part of the initial Java language definition 25 years ago
- Switch expression instead of switch statement
  - Quite new construct delivered with Java 12
- Streams instead of loop statements

# Conditional Expressions

```
return object != null ? object.getValue() : null;
```

Ternary operator is the expression construct for the if statement

# Functional Idioms

@Override

```
public boolean equals(Object obj) {  
    return (obj instanceof EntityImp o) &&  
        Objects.equals(oid(), o.oid());  
}
```

Match operator to access fields

# Switch Expressions

```
int numLetters = switch (day) {  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    case THURSDAY, SATURDAY -> 8;  
    case WEDNESDAY -> 9;  
};
```



# Lambdas

- Conciseness for anonymous classes
- Similar to function pointers in C or C++
- Similar to lambdas in Scheme

# Functional Programming

```
static final IntToDoubleFunction squares =  
    x -> (x == 0) ? 0 : Math.pow(x, 2) + Lecture5.squares.applyAsDouble(x - 1);
```

```
public static double meanOfSquaresRecursive(int n) {  
    return squares.applyAsDouble(n-1) / n;  
}
```

```
static final BiFunction<Integer, Double, Double> squaresTail =  
    (x, r) -> (x == 0) ? r : Lecture5.squaresTail.apply(x - 1, r + Math.pow(x, 2));
```

```
public static double meanOfSquaresTailRecursive(int n) {  
    return squaresTail.apply(n-1, 0D) / n;  
}
```

# Control Questions

1. What does  $(\textit{Integer } x, \textit{final Integer } y) \rightarrow x * y$  mean?
2. What does  $(\textit{var } x, \textit{final var } y) \rightarrow x * y$  mean?
3. What does  $(x, y) \rightarrow x * y$  mean?
4. What is the type of the lambda  $(\textit{Integer } x, \textit{final Integer } y) \rightarrow x * y$ ?
5. What does  $\langle ? \textit{super } T \rangle$  mean?
6. What does  $\langle ? \textit{extends } T \rangle$  mean?

# Functional Concepts (1/5)

- **Referential Transparency** (Clojure lectures)
  - A function, or more generally an expression, is called referentially transparent if a call can be replaced by its value without affecting the behavior of the program. Simply spoken, given the same input the output is always the same
- **Higher Order Functions** (Clojure lectures)
  - In computer science, a higher-order function is a function that does at least one of the following:
    - takes one or more functions as arguments
    - returns a function as its result

# Functional Concepts (2/5)

- Pure Functions (Clojure Lectures)
  - The function return values are identical for identical arguments (no variation with local static variables, non-local variables, mutable reference arguments or input streams)
  - The function application has no side effects (no mutation of local static variables, non-local variables, mutable reference arguments or input/output streams)

# Functional Concepts (3/5)

- Functional Composition (Clojure lectures)
  - In mathematics, function composition is an operation that takes two functions  $f$  and  $g$  and produces a function  $h$  such that  $h(x) = g(f(x))$
  - Composition requires higher order functions

# Functional Concepts (4/5)

- Map (Java Lectures)

- Stream map returns a stream consisting of the results of applying the given function to the elements of this stream. Stream map(Function mapper) is an intermediate operation

- Filter (Java Lectures)

- Stream filter returns a stream consisting of the elements of the given stream that match the given predicate. The filter() argument should be stateless predicate which is applied to each element in the stream to determine if it should be included or not

- Reduce (Java Lectures)

- Reducing is the repeated process of combining all elements. reduce operation applies a binary operator to each element in the stream where the first argument to the operator is the return value of the previous application and second argument is the current stream element

# Functional Concepts (5/5)

- **Lambda Lifting**

- Lambda lifting is a meta-process that restructures a computer program so that functions are defined independently of each other in a global scope. An individual "lift" transforms a local function into a global function. It is a two step process:
  - Eliminating free variables in the function by adding parameters
  - Moving functions from a restricted scope to broader or global scope

- **Currying**

- In computer science, currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each take a single argument



# Composition

You can compose functions. In mathematics, function composition is the application of one function to the result of another to produce a third function. For instance, the functions  $f: X \rightarrow Y$  and  $g: Y \rightarrow Z$  can be composed to yield a function  $h: g(f(x))$  which maps  $X \rightarrow Z$

# Function Composition

- **default** <V> Function<T,V> andThen  
(Function<? super R,? extends V> after)
- **default** <V> Function<V,R> compose  
(Function<? super V,? extends T> before)

# Predicate Composition

- **default** Predicate<T> and(Predicate<? super T> other)
- **default** Predicate<T> or(Predicate<? super T> other)
- **default** Predicate<T> negate()

# Comparator Composition

- **default** Comparator<T> thenComparing  
(Comparator<? super T> other)
- **default** <U> Comparator<T> thenComparing(  
Function<? super T,? extends U> keyExtractor,  
Comparator<? super U> keyComparator)
- **default** <U extends Comparable<? super U>>  
Comparator<T> thenComparing(  
Function<? super T,? extends U> keyExtractor)

# Map Operations on streams

- `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
- `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`
- `default <R> Stream<R> mapMulti(BiConsumer<? super T, ? super Consumer<R>> mapper)`

# Map Operations Examples (1/3)

```
void testMap() {  
    var list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);  
  
    System.out.println(list.stream().map(o -> o * o).map(String::valueOf)  
        .collect(Collectors.joining(", ", "(", ")")));  
}
```

```
void testMapToString() {  
    var list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);  
  
    System.out.println(list.stream().map(o -> String.valueOf(o * o))  
        .collect(Collectors.joining(", ", "(", ")")));  
}
```

# Map Operations Examples (2/3)

```
void testFlatMap() {  
    var list = List.of(List.of(1, 2, 3, 4, 5), List.of(6, 7, 8, 9, 10),  
                        List.of(11, 12, 13, 14, 15));  
  
    System.out.println(list.stream().flatMap(o -> o.stream())  
                        .map(o -> o * o).map(String::valueOf)  
                        .collect(Collectors.joining(", ", "(" , ")")));  
}
```

```
void testFlatMap() {  
    var list = List.of(List.of(1, 2, 3, 4, 5), List.of(6, 7, 8, 9, 10),  
                        List.of(11, 12, 13, 14, 15));  
  
    System.out.println(list.stream().flatMap(Collection::stream)  
                        .map(o -> o * o).map(String::valueOf)  
                        .collect(Collectors.joining(", ", "(" , ")")));  
}
```

# Map Operations Examples (3/3)

```
void testMapMulti() {  
    var list = List.of(List.of(1, 2, 3, 4, 5), List.of(6, 7, 8, 9, 10),  
                        List.of(11, 12, 13, 14, 15));  
  
    System.out.println(list.stream()  
        .mapMulti((o, c) -> o.forEach(o1 -> c.accept(o1 * o1)))  
        .map(String::valueOf).collect(Collectors.joining(", ", "(", ")")));  
}
```

```
void testMapMultiWithOptional() {  
    var list = List.of(Optional.of(1), Optional.of(2), Optional.of(3),  
                        Optional.empty(), Optional.of(4), Optional.of(5),  
                        Optional.empty());  
  
    System.out.println(list.stream().mapMulti(Optional::ifPresent)  
        .map(String::valueOf)  
        .collect(Collectors.joining(", ", "(", ")")));  
}
```



# Java Exception Pains (1/2)

```
@FunctionalInterface
public interface ThrowingFunction<T, R, E extends Exception> {
    R apply(T t) throws E;

    static <T, R, E extends Exception> Function<T, R>
        of(ThrowingFunction<T, R, E> f) {
        return t -> {
            try {
                return f.apply(t);
            } catch (Exception e) {
                throw new CompletionException(e);
            }
        };
    }
}
```

# Java Exception Pains (2/2)

```
Double function(Integer value) throws TimeoutException {
    if (value != 50) {
        return Double.valueOf(value);
    } else {
        throw new TimeoutException();
    }
}

@Test
void testWrappers() {
    assertThatExceptionOfType(RuntimeException.class)
        .isThrownBy(() -> IntStream.range(1, 101)
            .boxed().map(ThrowingFunction.of(this::function)).forEach(o -> {}));
}
```

# Partial Application and Currying

- Arity is a measure of the number of parameters a function takes.
  - Concept was introduced in Prolog lectures e.g. `is_bigger/2`
  - Java provides existing functional interfaces for nullary (Supplier), unary (Function), and binary (BiFunction), but that's it.
- Partial application allows you to derive a new function from an existing one by fixing some values
- Fixing one parameter is called currying

# Currying Simple Example

```
// factory creation pattern
static Letter of(String salutation, String body){
    return new Letter(salutation, body);
}

// functional building with a bi-function
BiFunction<String, String, Letter> letterCreator
    = (salutation, body) -> new Letter(salutation, body);

// functional currying building with a sequence of functions
Function<String, Function<String, Letter>> letterCreatorCurried
    = salutation -> body -> new Letter(salutation, body);

// Calling the functions
var letter1 = Letter.of(salutation, body);

var letter2 = letterCreator.apply(salutation, body);

var letter3 = letterCreatorCurried.apply(salutation).apply(body)
```

# Currying: More Complex Example

```
// factory creation pattern
```

```
static Letter of(String returnAddress, String insideAddress,  
    LocalDate date, String salutation, String body, String closing) {  
    return new Letter(returnAddress, insideAddress, date, salutation,  
        body, closing);  
}
```

```
// functional currying building with a sequence of functions
```

```
Function<String, Function<String, Function<LocalDate, Function<String,  
    Function<String, Function<String, Letter>>>>> letterCreator =  
returnAddress -> returnAddress -> insideAddress -> date -> salutation -> body  
-> closing  
-> new Letter(returnAddress, insideAddress, date, salutation, body, closing);
```

```
// Calling the functions
```

```
var letter1 = Letter.of(returnAddress, insideAddress, date, salutation, body, closing);  
var letter2 = letterCreator.apply(returnAddress).apply(insideAddress).apply(date)  
    .apply(salutation).apply(body).apply(closing);
```

# Currying Builder Example (1/2)

// All interfaces are SAM - Single Abstract Method

```
public static class Builder {
    public static ReturnAddress builder() {
        return returnAddress -> insideAddress -> dateOfLetter -> salutation -> body -> closing ->
            new Letter(returnAddress, insideAddress, dateOfLetter, salutation, body, closing);
    }

    public interface ReturnAddress {
        InsideAddress withReturnAddress(String returnAddress);
    }

    public interface InsideAddress {
        DateOfLetter withInsideAddress(String insideAddress);
    }

    public interface DateOfLetter {
        Salutation withDateOfLetter(LocalDate dateOfLetter);
    }

    public interface Salutation {
        Body withSalutation(String salutation);
    }

    public interface Body {
        Closing withBody(String body);
    }

    public interface Closing {
        Letter withClosing(String closing);
    }
}
```

# Currying Builder Example (2/2)

```
// create a letter with the functional builder  
  
var letter = Letter.Builder.builder().  
    .withReturnAddress(returnAddress)  
    .withInsideAddress(insideAddress)  
    .withDateOfLetter(dateOfLetter)  
    .withSalutation(salutation)  
    .withBody(body)  
    .withClosing(closing);
```

# Limits of Current Java

- Advanced functional programming concepts are a pain to write in Java
- Ruedi will show you more elegant constructs to implement the same concepts in other programming languages such as Kotlin



# Control Questions

1. What does functional composition mean?
2. What are examples of composition defined in the standard API?
3. What does currying mean?
4. How does streams handle checked exceptions?

# Solutions and Compromises

- Java is an object-oriented language adding functional features
- Java tries very hard to be backward compatible
- Often the solutions are compromises

# Optional<T>

- Optional<T> is a stream
  - Optional is a stream with zero or one item. It provides filter(), map(), flatmap()
  - It also provides stream() (*because it does inherit from Stream interface*)
- Modern Java programming idioms
  - Never return null from a function
  - Never pass an optional as a parameter – *except with streams or optionals (is also a stream) to stay functional* -
  - Never use an optional as instance variable
  - Try to use a functional style
- *Optional<T> is a monad*

# Future<T>

- Future class represents a future result of an asynchronous computation – a result that will eventually appear in the Future after the processing is complete
  - ExecutorService returns a future when starting an asynchronous job with Callable<T> or Runnable by
  - Returning Future<T> or Future<?>
- CompletableFuture<T>, ForkJoinTask, etc.

# Streams

- Functional programming
- Declarative iterators
- SQL like concepts → set theory
- Goodies JDK 16 → `toList()` on Stream

# Stream Example

```
class Scratch {  
    enum Tags {TAG_1, TAG_2, TAG_3, TAG_4,  
                TAG_5, TAG_6};  
    public static void main(String[] args) {  
        var data = tagsData().stream()  
            .flatMap(o -> o.stream())  
            .collect(Collectors.groupingBy(o -> o,  
                                           Collectors.counting()));  
        data.entrySet().stream()  
            .sorted((e1, e2) -> Long.compare(e2.getValue(),  
                                              e1.getValue()))  
            .limit(3).toList().forEach(System.out::println);  
    }  
}  
  
private static List<EnumSet<Tags>> tagsData() {  
    List<EnumSet<Tags>> tags = new ArrayList<>();  
    tags.add(EnumSet.of(Tags.TAG_1, Tags.TAG_2,  
                        Tags.TAG_3));  
    tags.add(EnumSet.of(Tags.TAG_1, Tags.TAG_2,  
                        Tags.TAG_3));  
    tags.add(EnumSet.of(Tags.TAG_4, Tags.TAG_5,  
                        Tags.TAG_6));  
    tags.add(EnumSet.of(Tags.TAG_4, Tags.TAG_5,  
                        Tags.TAG_6));  
    tags.add(EnumSet.of(Tags.TAG_1, Tags.TAG_6));  
    tags.add(EnumSet.of(Tags.TAG_6));  
    return tags;  
}
```

# Stream Programming

```
public static double meanOfSquaresModern(int n) {  
    return IntStream.range(0, n).asDoubleStream().map(i -> i * i).sum() / n;  
}
```

# Outlook Project Amber

- Concise Method Declaration  
(Draft JEP, open for years)

```
int length(String s) {  
    return s.length();  
}
```

```
int length(String s) -> s.length();
```

```
int length(String s) = String::length;
```



# Outlook Project Loom

- Tail-call elimination for recursion
  - Finally we could use recursion instead of loop statements
- Delimited continuation
- *Functional programming is fun ;-)*

# Links

- [Open JDK](#)
- [JEP List](#)
- [Project Amber](#)
- [Project Loom](#)

# Exercises

- Lambdas
- Currying and functional builder
- If you are interested in functional programming with Java look at [Vavr](#) or [FunctionalJava](#)