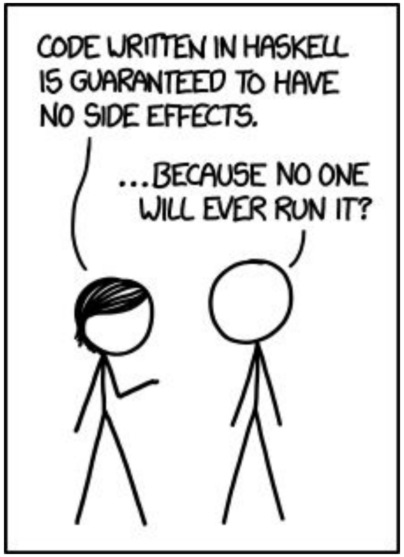
Funktionale Programmierung
in Java

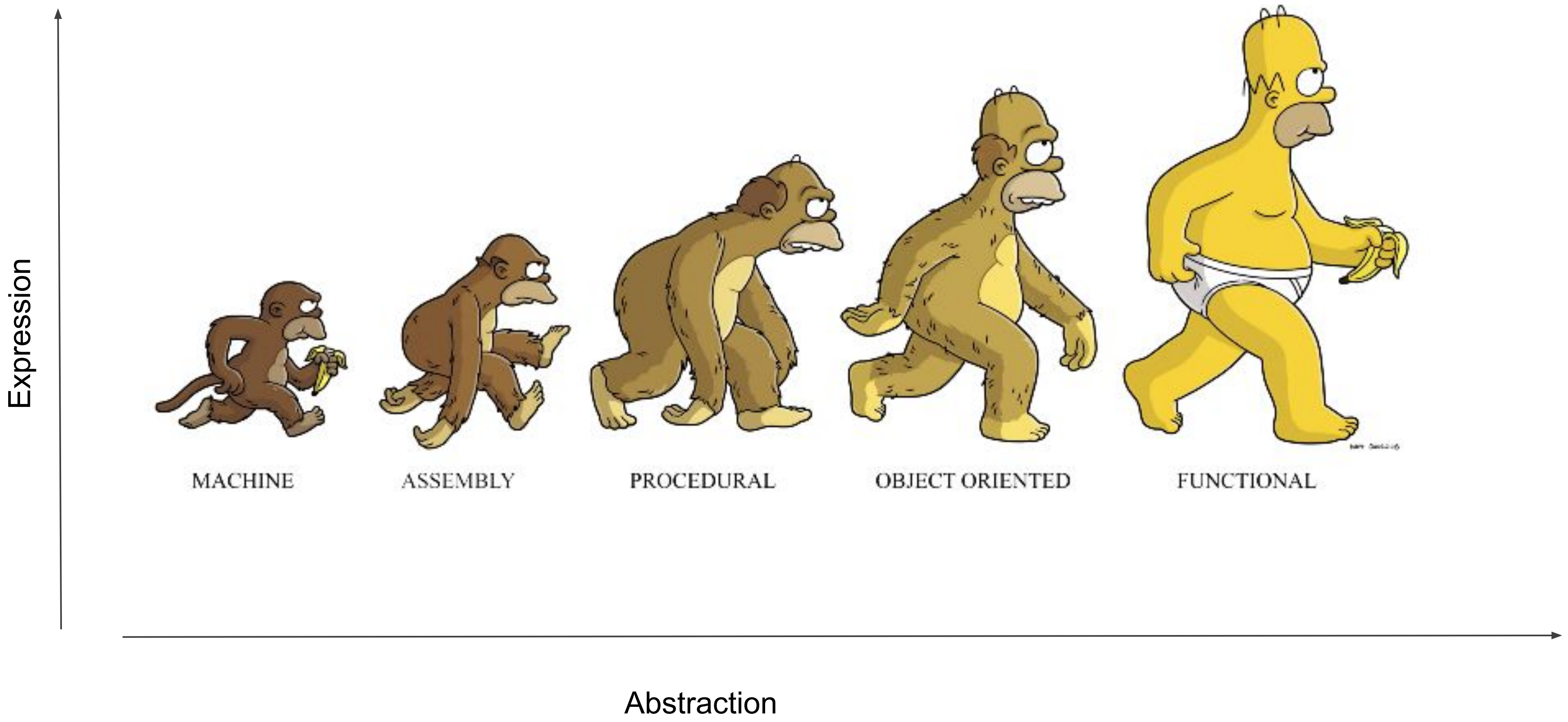


Lambda Calculus

$$
\begin{aligned}
0 &:= \lambda f.\lambda x.x \\
1 &:= \lambda f.\lambda x.f\ x \\
2 &:= \lambda f.\lambda x.f\ (f\ x) \\
3 &:= \lambda f.\lambda x.f\ (f\ (f\ x))
\end{aligned}
$$

```
Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)

.distinct()

.sorted((i,j) -> i-j )

.limit(3)

.forEach(System.out::println );
```

Übersicht


Expression
MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL
Abstraction

Übersicht

- Konzepte
- Lambda-Ausdrücke
- funktionale Interfaces
- Streams + Operations

Geschichte der funktionalen Programmierung

- 30s => Lambda-Kalkül

$0 := \lambda f.\lambda x.x$

$1 := \lambda f.\lambda x.f\ x$

$2 := \lambda f.\lambda x.f\ (f\ x)$

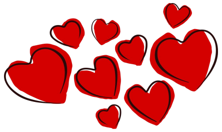
$3 := \lambda f.\lambda x.f\ (f\ (f\ x))$

(λx. +((λy. ((λx. * x y) 2)) x) y)

(λx. + ((λy. (* 2 y)) x) y)

Geschichte der funktionalen Programmierung

- 30s => Lambda-Kalkül
- 60s => die Programmiersprache LISP (inspiriert vom Lambda-Kalkül)
- Heute =>
  - Erlang
  - Haskell
  - Scala
  - ...
  - + (JS, Java,...??)



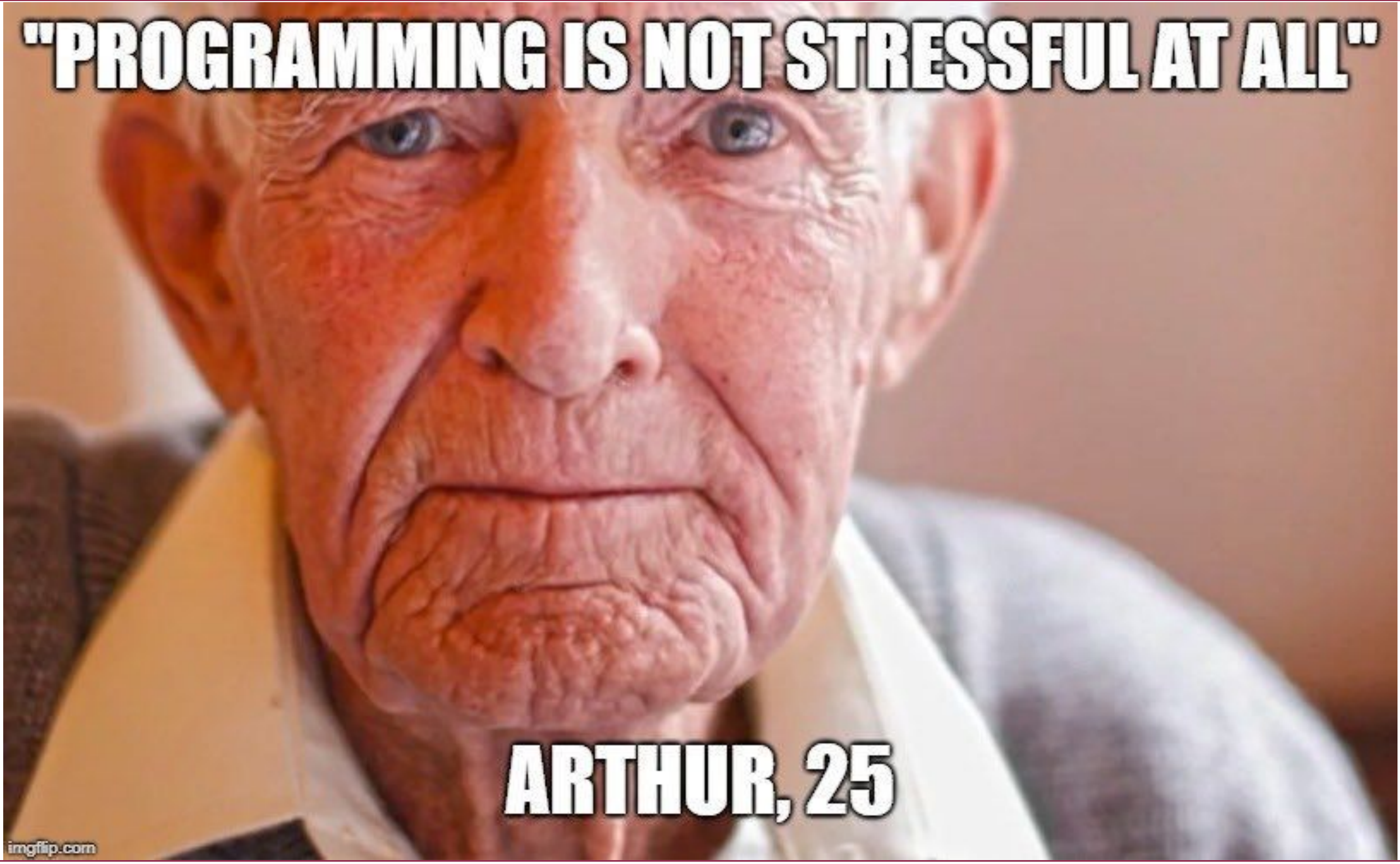
Funktionales Programmieren

- basiert auf Mathematik
- ist deklarativ (nicht imperativ)
- Innerhalb des Paradigmas ist der Ablauf eines Programms nicht definiert
- In eingeschränkter Form ist funktionale Programmierung die Programmierung ohne veränderliche Variablen, ohne Zuweisung und ohne Kontrollstrukturen.

Eigenschaften funktionaler Sprachen

- Immutability/Keine Variablen
- keine Seiteneffekte (pure functions)
- keine Zuweisung
- keine imperativen Kontrollstrukturen
- Funktionen => First Class Citizen

...ist einfach



Forget everything you know, you must. Yeessssss.

Pure Functions

- das Ergebnis hängt nur von den Argumenten ab
- die Funktion ändert den Umfang (des Aufrufers) nicht

```
public class SideEffectClass{

    private int state = 0;

    public int doSomething(int arg0){
        return arg0 + state;
    }
}
```

Funktionen => First Class Citizen

- Wie alle Daten können Funktionen innerhalb von Funktionen definiert werden
- Wie alle Daten können Funktionen an Funktionen übergeben und zurückgegeben werden
- Funktionen lassen sich mit anderen Funktionen zu neuen Funktionen verknüpfen
- Insbesondere gibt es Funktionen, die andere Funktionen auf Datenstrukturen anwenden

Funktionales Programmieren

- Programmieren = Definition von Funktionen
- Ausführung = Auswerten von Ausdrücken
- Einziges Resultat ist der Rückgabewert
- `Resultat = Funktion(Argument`$_1$`, . . . , Argument`$_n$`)`

Lambdas

ist ein Ausdruck, dessen Wert eine Funktion ist

```
btn.setOnAction(new EventHandler<ActionEvent>() {
     @Override
     public void handle(ActionEvent event) {
         System.out.println("Hello World!");
    }
});
```

```
btn.setOnAction(event -> System.out.println("Hello World!“));
```

Lambda Ausdrucks

- Anonyme Methode
- Prägnante Syntax, weniger Code, lesbarer
- ad-hoc Implementierung von Funktionalität

lambda = **ArgList** "->" **Body**

ArgList = Identifier

| "(" Identifier ["," Identifier]* ")"

| "(" Type Identifier ["," Type Identifier]* ")“

Body = Expression

| "{" [Statement ";"]+ "}"

Beispiele

- `(int x, int y) -> { return x+y; }`

- `// Argument type is inferred:`

  `(x, y) -> { return x+y; }`

- `// No brackets needed if only one argument`

  `x -> { return x+1; }`

- `// No arguments needed`

  `() -> { System.out.println(“I am a Runnable”); }`

Beispiele

- `// Lambda using a statement block`

  ```
  a -> {
   if (a.balance() < limit) a.alert();
  else a.okay();
  }
  ```

- `// Single expression`

  ```
  a -> (a.balance() < limit) ? a.alert() : a.okay()
  ```

- `// returns Account`

  ```
  (Account a) -> { return a; }
  ```

- `// returns int`

  ```
  () -> 5;
  ```

Beispiele

```java
class Person {
     private String name;
     private int age;

     public Person(String name, int age) {
        this.name = name;
        this.age = age;
     }

     public String getName() { return name; }
     public int getAge() { return age; }
     public String toString() {
        return "Person[" + name + ", " + age + "]";
     }
}

List<Person> persons = Arrays.asList(
    new Person("Hugo", 55),
    new Person("Amalie", 15),
    new Person("Anelise", 32)
);
```

Beispiele

```java
Collections.sort(
persons,
new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
    return o1.getAge() - o2.getAge();
    }
 });

 Collections.sort( persons,
    (Person o1, Person o2)
    -> { return o1.getAge() - o2.getAge(); }
 );

Collections.sort( persons,
    (o1, o2) -> o1.getAge() - o2.getAge()
);
```

Typen von Lambdas

- Functional Interface
- ≈ Interface mit einer abstrakten Methode

```
Consumer<Account> myLambda =
 (Account a) -> {if (a.balance() < limit) a.alert();};
```

Typen von Lambdas

```
interface op {
    int do (int a, int b);
}

interface message {
    void say (String message);
}
```

Typen von Lambdas

```
public static void main(...) {
   op add = (int a, int b) -> a + b;
   op mul = (int a, int b) -> { return a * b;};
   op div = (int a, int b) -> a / b;
   System.out.println(add.do(2,3));

   message msg =
      m -> System.out.println("Hello " + m);
   msg.say("World");
}
```

Typen von Lambdas

```
class T {
   private int exec(int a, int b, op oper) {
         return oper.do(a, b);
   }

   public static void main(...) {
      op add = (int a, int b) -> a + b;
      System.out.println(new T().exec(2,3, add));
   }
}
```

Functional Interface

- Vordefinierte Functional Interfaces

- Consumer: Kein Resultat
- Function: Produziert Resultat
- Operator: Produziert Resultat vom Argument-Typ
- Supplier: Produziert Resultat ohne Argument
- Predicate: Produziert boolean-Resultat

Predicate

Interface Predicate<T>

Type Parameters:

`T` - the type of the input to the predicate

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface Predicate<T>
```

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:

1.8

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`default Predicate<T>`	`and(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
`static <T> Predicate<T>`	`isEqual(Object targetRef)` Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`.
`default Predicate<T>`	`negate()` Returns a predicate that represents the logical negation of this predicate.
`default Predicate<T>`	`or(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
`boolean`	`test(T t)` Evaluates this predicate on the given argument.

Predicate

Interface Predicate<T>

Type Parameters:
`T` - the type of the input to the predicate

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression o

---

```
@FunctionalInterface
public interface Predicate<T>
```

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:
1.8

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`default Predicate<T>`	`and(Predicate<? super T> other)` Returns a composed predicate that r
`static <T> Predicate<T>`	`isEqual(Object targetRef)` Returns a predicate that tests if two
`default Predicate<T>`	`negate()` Returns a predicate that represents t
`default Predicate<T>`	`or(Predicate<? super T> other)` Returns a composed predicate that r
`boolean`	`test(T t)` Evaluates this predicate on the giver

```java
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;

public class Lambda_Ex {
  static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
    List<T> res = new LinkedList<>();
    for (T x: l) {
      if (pred.test(x)) {
        res.add(x);
      }
    }
    return res;
 }

public static void main(String[] args) {
    List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
    System.out.println(
    filterList(
    l,
    (x) -> x%2 == 0
    ));
 }

}
```

Consumer

Interface Consumer<T>

Type Parameters:
T - the type of the input to the operation

All Known Subinterfaces:
Stream.Builder<T>

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface Consumer<T>
```

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.

This is a functional interface whose functional method is `accept(Object)`.

Since:
1.8

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
void	`accept(T t)` Performs this operation on the given argument.
default Consumer<T>	`andThen(Consumer<? super T> after)` Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

Consumer

Interface Consumer<T>

Type Parameters:

T - the type of the input to the operation

All Known Subinterfaces:

Stream.Builder<T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression o

---

@FunctionalInterface
public interface **Consumer<T>**

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, C

This is a functional interface whose functional method is accept(Object).

Since:

1.8

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
void	accept(T t) Performs this operation on the given
default Consumer<T>	andThen(Consumer<? super T> aft Returns a composed Consumer that p

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

class WorkerOnList<T> implements Consumer<List<T>> {
   private Consumer<T> action;

   public WorkerOnList(Consumer<T> action) {
     this.action = action;
   }

  @Override
  public void accept(List<T> l) {
    for(T x: l) {
      action.accept(x);
    }
   }
}

public class Ex {
   public static void main(String[] args) {
      List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
      WorkerOnList<Integer> worker =
        new WorkerOnList<>( (i) -> System.out.println(i*10) );
      worker.accept(l);
   }
}
```

Function

Interface Function<T,R>

Type Parameters:

`T` - the type of the input to the function

`R` - the type of the result of the function

All Known Subinterfaces:

`UnaryOperator<T>`

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

Since:

1.8

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`default <V> Function<T,V>`	`andThen(Function<? super R,? extends V> after)` Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
`R`	`apply(T t)` Applies this function to the given argument.
`default <V> Function<V,R>`	`compose(Function<? super V,? extends T> before)` Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.
`static <T> Function<T,T>`	`identity()` Returns a function that always returns its input argument.

Function

Interface Function<T,R>

Type Parameters:
T - the type of the input to the function
R - the type of the result of the function

All Known Subinterfaces:
UnaryOperator<T>

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface
public interface **Function<T,R>**

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is apply(Object).

Since:
1.8

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
default <V> Function<T,V>	andThen(Function<? super R,? extends V> after) Returns a composed function that first applies this function to
R	apply(T t) Applies this function to the given argument.
default <V> Function<V,R>	compose(Function<? super V,? extends T> before) Returns a composed function that first applies the before fun
static <T> Function<T,T>	identity() Returns a function that always returns its input argument.

```java
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Function;

class ListTransformer<T,R> implements Function<List<T>, List<R>> {
  private Function<T, R> fun;

  public ListTransformer(Function<T, R> fun) {
    this.fun = fun;
   }

  @Override
  public List<R> apply(List<T> l) {
    List<R> res = new LinkedList<>();
    for(T x: l) {
      res.add(fun.apply(x));
    }
    return res;
   }

}

public class Lambda_Ex {
  public static void main(String[] args) {
  List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
  ListTransformer<Integer, Integer> worker = new
    ListTransformer<>( (i) -> i*10 );
  System.out.println(worker.apply(l));
 }
}
```

Methoden-Referenzen

- existierende Methoden einer Klasse als Lambda Ausdruck verwenden können
- Brauchen Kontext, damit korrekter Ziel-Typ abgeleitet werden kann
- Weniger Code, verständlicher

Methode als Predicate

Interface Predicate<T>

Type Parameters:

`T - the type of the input to the predicate`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression o`

```
@FunctionalInterface
public interface Predicate<T>
```

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:

1.8

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`default Predicate<T>`	`and(Predicate<? super T> other)` Returns a composed predicate that r
`static <T> Predicate<T>`	`isEqual(Object targetRef)` Returns a predicate that tests if two
`default Predicate<T>`	`negate()` Returns a predicate that represents t
`default Predicate<T>`	`or(Predicate<? super T> other)` Returns a composed predicate that r
`boolean`	`test(T t)` Evaluates this predicate on the giver

```java
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;

public class Lambda_Ex {
 static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
    List<T> res = new LinkedList<>();
    for (T x: l) {
      if (pred.test(x)) {
        res.add(x);
      }
    }
    return res;
  }
 static boolean even(int x) {
    return x % 2 == 0;
 }

public static void main(String[] args) {
  List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
  System.out.println(
  filterList(
  l,
  Lambda_Ex::even
  ));
 }
}
```

Erweiterungen bei Collections

`forEach(Consumer<T> c)`

```java
import java.util.Arrays;
import java.util.List;

public class Lambda_Ex {
   public static void main(String[] args) {
     List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
     l.forEach( x -> System.out.println(x) );
   }
}
```

Erweiterungen bei Collections

`removeIf(Predicate<T> p)`

```java
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class Lambda_Ex {
   public static void main(String[] args) {
      List<Integer> l = new LinkedList<>(Arrays.asList(1,2,3,4,5,6,7,8,
      l.removeIf( x -> x % 2 != 0 );
      l.forEach( x -> System.out.println(x) );
   }
}
```

Erweiterungen bei Collections

`replaceAll(UnaryOperator<T> operator)`

```java
import java.util.Arrays;
import java.util.List;

public class Lambda_Ex {
   public static void main(String[] args) {
      List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
      l.replaceAll( x -> 2 * x );
      l.forEach( x -> System.out.println(x) );
   }
}
```

Scoping

- Lambdas haben Zugriff auf lokale Variablen vom umschließenden Scope
- Lambdas führen keinen neuen Scope ein
- Variablen müssen final bzw. effective-final sein

```
int x = 5;
return y -> return x + y; // nicht final aber funktioniert
```

Scoping

- Variablen müssen final bzw. effective-final sein

```
int x = 5;
return y -> return x + y; // nicht final aber funktioniert

public Supplier<Interger> gen() {
    int x = 5;
    x++;
    return () -> return x + 1; // error
}
```

Streams

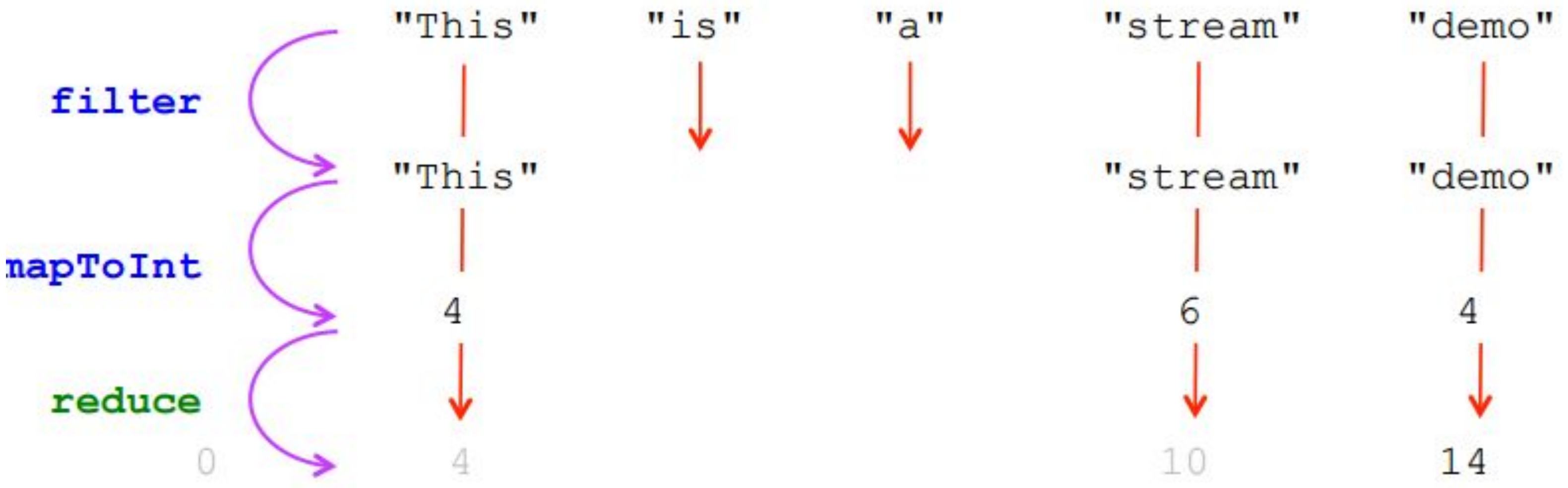
- `interface java.util.stream.Stream<T>`
- Eine Reihe von Elementen, die eine parallele Verarbeitung von Daten erlauben
- Funktionales Bearbeiten und Behandeln von Sequenzen
- Streams unterstützen `filter`, `map`, `reduce`, `findAny`, `skip`, `peek`
- Haben nichts mit java.io.InputStream, resp. java.io.OutputStream zu tun! -> als ob! ^^

Streams

- `Stream<T>` ist der Typ der Streams mit Objekten vom Typ T
- Streams für Elemente von diesen drei primitiven Datentypen
  - `IntStream`
  - `DoubleStream`
  - `LongStream`
- diese Versionen erben nicht von Stream
- Streams mit primitiven Daten arbeiten in vielen Fällen effizienter

Beispiel

```
String[] txt = { "This", "is", "a", "stream", "demo"};
Arrays.stream(txt)
      .filter(s -> s.length() > 3)
      .mapToInt(s -> s.length())
      .reduce(0, (l1, l2) -> l1 + l2);
```



Beispiel

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Ex {
   public static void main(String[] args) {
    List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
    List<Integer> ll = l.stream() // list -> stream
    .filter( (x) -> x%2 == 0 )
    .map( (x) -> 10*x )
    .collect( Collectors.toList() ); // back to a list

    ll.forEach( x -> System.out.println(x) );
   }
}
```

Erzeugung von Streams

- statische Methoden in `Arrays`

- ```
  IntStream isP =
      Arrays.stream(new int[]{1,2,3,4,5,6,7,8,9,0});
      // Stream of primitive data
  ```

- ```
  Stream<Integer> isO =
      Arrays.stream(new Integer[]{1,2,3,4,5,6,7,8,9,0});
      // Stream of objects
  ```

Erzeugung von Streams

- statische Methoden in `java.util.stream.Stream`
- mit `iterate` und `generate` hat man eine einfache Möglichkeit unendliche Ströme zu erzeugen

- Stream<Integer> is1a = Stream.of(1,2,3,4,5,6,7,8,9,0);

  // Object-Stream 1, 2, ... 9, 0
- IntStream is1b = IntStream.of(1,2,3,4,5,6,7,8,9,0);

  // int-Stream 1, 2, ... 9, 0
- Stream<Integer> is2 = Stream.iterate(1, ((x) -> x+1));

  // (infinite) Stream 1, 2, ...

Erzeugung von Streams

- statische range-Methoden in `IntStream` und `LongStream`

- Die Interfaces `IntStream` und `LongStream` enthalten jeweils zwei statische range-Methoden mit denen Streams erzeugt werden können

```
IntStream isPrimA = IntStream.range(1, 10);
// 1,2, .. 9
IntStream isPrimA = IntStream.rangeClosed(1, 10);
// 1,2, .. 9, 10
```

Pipeline-Operationen

- Streams werden typischerweise in einer Pipeline Struktur genutzt
- Verarbeitungs-Operationen transformieren die Elemente eines Streams
- Folge von Verarbeitungs-/Tranformationsschritten
- Abschluss mit einer terminalen Operation
- Intermediate und Terminal Operationen

Verarbeitungs Operationen

`filter(Predicate<T> pred)`

- entfernt alle Elemente für die das übergebene Prädikat false liefert

`map(Function<? super T,? extends R> mapper)`

- wendet auf jedes Element die übergebene Funktion an

Verarbeitungs Operationen

```
flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

- wendet die übergebene Funktion auf alle Elemente an
- die entstehenden Stream ist flach

```
peek(Consumer<? super T> action)
```

- wendet die übergebene ergebnislose Funktion auf alle Elemente an, ohne dabei den Stream selbst zu verändern

Beispiel

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class Ex {
   public static void main(String[] args) {
      List<Integer> is = IntStream.range(1,10)
      .filter( (i) -> i%2 != 0)
      .peek( (i) -> System.out.println(i) )
      .map( (i) -> 10*i )
      .boxed()
      .collect( Collectors.toList() );

      System.out.println(is);
   }
}
```

Beispiel

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;


public class Ex{
 static Stream<Integer> range(int from, int to) {
    return IntStream.range(from, to).boxed();
 }

 public static void main(String[] args) {
    List<Integer> is =
    Stream.of(0, 1, 2)
    .flatMap((i) -> range(10*i, 10*i+10))
    .collect(Collectors.toList());

    System.out.println(is);
 }

}
```

Beispiel

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Ex {
   public static void main(String[] args) {
      List<Integer> lst =
      Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)
      .distinct()
      .sorted( (i,j) -> i-j )
      .skip(1)
      .limit(3)
      .collect( Collectors.toList() );

      System.out.println(lst);
    }
}
```

Streams

- Es gibt Stream-Operationen, welche

- wieder einen Stream produzieren: `filter(), map()`
  - intermediate

- etwas anderes tun: `forEach(), reduce(), collect()`
  - terminal

- intermediate-Streams werden nicht direkt ausgewertet

Terminale Operationen

- ohne Ergebnis
- `forEach`

```
Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)
      .distinct()
      .sorted( (i,j) -> i-j )
      .limit(3)
      .forEach( System.out::println );
```

Terminale Operationen

- mit Array als Ergebnis
- `toArray`
- Die Methode `toArray` erzeugt einen Array aus den Elementen des Streams

```
Object[] a = Stream.of("1", "2", "3")
                   .map( Integer::parseInt )
                   .toArray();
```

Terminale Operationen

- mit Collections als Ergebnis
- `collect`
- erzeugt eine Collection aus den Elementen des Streams
- Für die Erzeugung einer Collection verwendet man typischerweise einen vordefinierten `Collector` aus `java.util.stream.Collectors`

```
List<Integer> l1 = Stream.of(1, 2, 3)
                                .collect(Collectors.toList());
```

Terminale Operationen

- mit Collections als Ergebnis
- `collect`
- erzeugt eine Collection aus den Elementen des Streams
- In Collectors findet sich auch die mit denen Maps erzeugt werden können

```
Map<String, Integer> m = Stream.of("1", "2", "3")
   .collect(Collectors.toMap((s) -> s, Integer::parseInt));
```

Terminale Operationen

Gruppieren und partitionieren

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Stream;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.partitioningBy;
import static java.util.stream.Collectors.counting;

public class Ex {
   public static void main(String[] args) {
      Map<Boolean, List<Integer>> oddAndEven =
        Stream.of(1, 2, 3, 4, 5, 6 ,7 ,8, 9, 0)
      .collect( partitioningBy( (x) -> x%2 == 0 ) );
      Map<Integer, List<Integer>> groupedMod3 =
        Stream.of(1, 2, 3, 4, 5, 6 ,7 ,8, 9, 0)
      .collect( groupingBy( (x) -> x%3 ) );
      Map<Integer, List<String>> groupedByLength =
        Stream.of("one", "two", "three", "four",
       "five", "six", "seven", "eight", "nine")
      .collect( groupingBy( (s) -> s.length() ) );
      Map<Integer, Long> countGroupsByLength =
        Stream.of("one", "two", "three", "four",
       "five", "six", "seven", "eight", "nine")
      .collect( groupingBy( String::length, counting() ) );
   }
 }
```

```
{false=[1, 3, 5, 7, 9], true=[2, 4, 6, 8, 0]}
{0=[3, 6, 9, 0], 1=[1, 4, 7], 2=[2, 5, 8]}
{3=[one, two, six], 4=[four, five, nine], 5=[three, seven, eight]}
{3=3, 4=3, 5=3}

Process finished with exit code 0
```

Terminale Operationen

Summe, Minimum, Maximum, Durchschnitt

```java
import java.util.OptionalDouble;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Ex {
    public static void main(String[] args) {
        long count = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .count();

        System.out.println("count = " + count);
        long sum = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .sum();

        System.out.println("sum = " + sum);
        OptionalDouble av = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .average();

        System.out.println("average = " + av);
    }
}
```

Terminale Operationen

```java
boolean allEven = Stream.of(2, 4, 6)
    .allMatch( (x) -> x%2 == 0 );
System.out.println(allEven); // => true

boolean anyEven = Stream.of(1, 2, 3, 4)
    .anyMatch( (x) -> x%2 == 0 );
System.out.println(anyEven); // => true

boolean noneEven = Stream.of(1, 2, 3, 4, 5)
    .noneMatch( (x) -> x%2 == 0 );
System.out.println(noneEven); // => false
```

```java
String concat = Stream.of
  ("one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine")
 .collect( joining("+") );

System.out.println(concat);
// => one+two+three+four+five+six+seven+eight+nine
```

Terminale Operationen

- reduzierende Operationen
- Produziert einzelnes Resultat aus allen Elementen

```
Optional<Integer> sumOfAll =
Stream.of(1, 2, 3, 4, 5) .reduce( (a, x) -> a+x );

Optional<Integer> subOfAll =
Stream.of(1, 2, 3, 4, 5) .reduce( (a, x) -> a-x );
```

Terminale Operationen

```java
import java.util.Iterator;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Fibonacci {
   static IntStream fibs() {
    int[] start = {1, 1};
    Stream<int[]> pairStream = Stream.iterate(start,
      (int[] p) -> new int[]{p[1], p[0]+p[1]});
    return pairStream.mapToInt((p) -> p[1]);
   }

  static int getNthFib(int n) {
     return fibs().skip(n-1).findFirst().getAsInt();
   }

  public static void main(String[] args) {
    Iterator<Integer> iter = fibs().iterator();
    int i = 1;
    while (iter.hasNext()) {
      int f = iter.next();
      System.out.println(i++ + " : " + f);
      if (i >= 10) break;
    }
    for (int n=1; n<10; n++) {
      System.out.println(n + " : " + getNthFib(n));
    }
  }
}
```