FUNKTIONALE PROGRAMMIERUNG IN JAVA

Ausgewählte Kapitel der Softwareentwicklung

Table of Contents

Einstieg Funktionale Programmierung

Programmieren mit Funktionen im mathematischen Sinne

- eindeutige Abbildung von Argumentwerten auf Ergebnis
- kein speicherndes Verhalten (keinen Zustand)

`x = y ➔ f(x) = f(y)` Ergebnis ist nur von Werten der Argumente abhängig ➔ Kein Seiteneffekt!!

Merkmale (rein) funktionaler Programmierung

- Programmieren ohne Seiteneffekte
  - kein veränderlicher Speicher
  - keine Wertzuweisung
  - nur unveränderliche Daten
- Referentielle Transparenz
  - Funktionen liefern bei gleichen Argumenten immer gleiches Ergebnis
- Substitutionsprinzip (als Folge der referentiellen Transparenz)
  - ein Symbol kann stets durch seine Definition ersetzt werden
  - die Reihenfolge der Ersetzung spielt keine Rolle für das Endergebnis

Funktionale Programmierung ist **deklarativ**
➔ definiert das **WAS** und nicht das **WIE**

Funktionale Programmierung gilt als **robuster** und **weniger fehleranfällig**
➔ z.B. weil **keine Seiteneffekte**

Funktionale Programme können einfach **parallelisiert** werden
➔ **Ausführungsreihenfolge** von Funktionen **beliebig**

Funktionale Programmierung günstig bei **nebenläufigen** und **verteilten** Systemen
➔ keine Seiteneffekte
➔ Prominente Beispiele: **Twitter**, **WhatsApp, ÖBB Scotty**

Imperative mit Schleife:

```
List<Integer> result = new ArrayList<Integer>();
for (int x : list) {
   result.add(square(x));
}
```

Funktional mit Higher-Order-Function:

```
List<Integer> result = list.stream().map(x -> square(x)).collect(Collectors.toList());
```

Lambda Expressions

Motivation

Gesucht: Variable, in der man Methoden speichern kann

```
Function f;   // f can contain a method
```

Lösungsmöglichkeit

```
interface Function {
    int exec (int x);
}
```

"***Funktionsinterface***"
(Interface mit nur 1 Methode)

Zuweisung eines Objekts einer implementierenden Klasse

```
Function f = new Square();
```

f.exec(10) ⇒ 100

```
class Square implements Function {
    int exec (int x) { return x * x; }
}
```

Zuweisung eines Objekts einer anonymen Unterklasse

```
Function f = new Function() {
    int exec (int x) { return x * x; }
}
```

f.exec(10) ⇒ 100

```
f = new Function() {
    int exec (int x) { return 2 * x + 1; }
}
```

f.exec(10) ⇒ 21

Lambda-Ausdruck

Kurzform für eine namenlose Methode

benannte Methode `int exec (int x) { return x * x; }`

namenlose Methode `x -> x * x` ← "*Lambda-Ausdruck*"

Begriff aus dem *Lambda-Kalkül*
z.B. λx . square x

Beispiel:

```
interface Function {
    int exec (int x);
}
```

```
Function f;
f = x -> x * x;
f = a -> a + a;
```

f.exec(10) ⇒ liefert 100
f.exec(10) ⇒ liefert 20

Compiler behandelt das wie folgt ⇓

```
f = new Function() {
        int exec (int a) { return a + a; }
    }
```

automatische *Typinferenz* (aus dem Kontext)

```
f = a -> a + a
    int  int
```

Lambda ⇒ Referenz auf Funktion

Lambdas zur Parametrisierung von Methoden

Zum Beispiel: Methode, die eine Funktion auf alle Elemente eines Arrays anwendet

```
int[] map (Function f, int[] data) {
   int[] result = new int[data.length];
   for (int i = 0; i < data.length; i++) result[i] = f.exec(data[i]);
   return result;
}
```

Kann mit unterschiedlichen Funktionen parametrisiert werden

```
int[] data = { 1, 2, 3, 4 };

int[] result = map(x -> x + 1, data);      ⇒  2, 3, 4, 5
int[] result = map(x -> x * x, data);      ⇒  1, 4, 9, 16
```

Weiteres Beispiel: Methode, die aus einem Array bestimmte Elemente herausfiltert

```
int[] filter (Predicate pred, int[] data) {
   int[] a = new int[data.length];
   int len = 0;
   for (int i = 0; i < data.length; i++) {
      if (pred.test(data[i])) a[len++] = data[i];
   }
   int[] result = new int[len];
   arraycopy(a, 0, result, 0, len);
   return result;
}
```

```
interface Predicate {
   boolean test (int x);
}
```

} gibt true | false zurück

Kann mit unterschiedlichen Funktionen parametrisiert werden

```
int[] data = { 1, 2, 3, 4, 5, 6 };

int[] result = filter(x -> x % 2 == 0, data);      ⇒  2, 4, 6
int[] result = filter(x -> x > 3, data);           ⇒  4, 5, 6
```

Lambdas mit mehreren Parametern

```
interface BinaryOp {
   int exec (int x, int y);
}
```

```
BinaryOp op;
op = (a, b) -> a * a + b * b;
op = (x, y) -> 2 * x + 2 * y;
```

```
op.exec(3, 4)   => 25
op.exec(3, 4)   => 14
```

falls mehrere Parameter => Klammern nötig

Man kann auch Parametertypen angeben
   op = (int x, int y) -> 2 * x + 2 * y;
obwohl meist nicht nötig (Typinferenz)

```
interface StringOp {
   String exec (String s, int a, int b);
}
```

```
StringOp op;
op = (s, a, b) -> s.substring(a, b);
op = (s, a, b) -> s.substring(0, a) +
                  s.substring(b);
```

```
op.exec("Hello", 2, 4)   => "ll"
op.exec("Hello", 2, 4)   => "Heo"
```

```
interface Random {
   int get ();
}
```

```
Random op;
op = () -> rand.nextInt();
```

```
op.get()   => ... Zufallszahl ...
```

Eigentlich gibt es 2 Arten von Lambda-Ausdrücken

- Parameter -> Ausdruck
- Parameter -> Anweisung

```
interface Action {
    void exec();
}
```

```
Action  hello = () -> System.out.println("Hello");
```

```
hello.exec();
```

```
interface Action2 {
    void exec(char ch, int n);
}
```

```
Action2  printLine = (ch, n) -> {
    for (int i = 0; i < n; i++) System.out.print(ch);
    System.out.println();
};
```

```
printLine.exec('*', 10)
```

```
interface Function2 {
    int exec(int a, int b);
}
```

```
Function2  ggt = (x, y) -> {
    int rest = x % y;
    while (rest != 0) {
        x = y; y = rest; rest = x % y;
    }
    return y;
};
```

```
ggt.exec(18, 12)
```

Beispiel: Threaderzeugung mit Lambdas. Java-Bibliothek enthält das Funktionsinterface Runnable

```
interface Runnable {
    void run ();
}
```

```
public class Thread {
    public Thread (Runnable target) { ... }
    ...
}
```

```
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 20; i++) {
        System.out.print(".");
        try { Thread.sleep(500); }
        catch (InterruptedException e) { break; }
    }
});
Thread t2 = new Thread(() -> {
    for (int i = 0; i < 20; i++) {
        System.out.print("*");
        try { Thread.sleep(1000); }
        catch (InterruptedException e) { break; }
    }
});
t1.start();
t2.start();
```

Ausgabe

```
.*..*..*..*..*..*..*..*..*..*..**********
```

Beispiel: Vergleiche mit Lambdas. Java-Bibliothek enthält das Funktionsinterface Comparator<T>

```
interface Comparator<T> {
    int compare (T a, T b);
}
```

```
public class Arrays {
    static <T> void sort (T[] a, Comparator<T> c) { ... }
    ...
}
```

```
String[] stringArray = ...;

Arrays.sort(stringArray, (s1, s2) -> s1.length() - s2.length());
```

sortiert *stringArray* nach Stringlänge

⇩

Typinferenz

- *stringArray* ist vom Typ *String[]* => *T* ist *String*
- Lambda-Ausdruck ist vom Typ *Comparator<String>*
- Parameter *s1* und *s2* sind vom Typ *String*
- Rückgabewert ist vom Typ *int*

```
Arrays.sort(stringArray, (s1, s2) -> s1.compareTo(s2));
```

sortiert *stringArray* alphabetisch

Geschachtelte Lambdas

Lambda-Ausdruck, der einen anderen Lambda-Ausdruck zurückgibt

```
interface Function {
  int exec(int x);
}
```

```
interface NestedFunction {
  Function exec(int x);
}
```

Deklaration

$$\texttt{NestedFunction f = x -> }\underbrace{\texttt{(y -> x / y)}}_{\textit{Function}}\texttt{;}$$

x: fix
y: variabel (Parameter)

```
NestedFunction f = x -> y -> x / y;
```

Verwendung

f $\xrightarrow{x = 10}$ x -> (y -> x / y) $\Rightarrow$ $\xrightarrow{y = 2}$ y -> 10 / y $\Rightarrow$ 5

Zugriff auf äußere lokale Variablen

Beispiel: liefere Position des ersten Arrayelements, das größer als limit ist

wichtig!
1ser Frage

```
void foo(int[] a) {
  int limit = readInt();
  int pos = position(a, x -> x > limit);
  ...
}
```

a | 3 | 4 | 6 | 9 | 11 | 12 | 15 | 19 | limit = 10

pos

Lambda-Ausdruck greift auf äußere Variable zu.

<u>Problem</u>: wenn Lambda ausgeführt wird, lebt *limit* vielleicht nicht mehr

=> <u>Capturing</u>: *limit* wird mit Lambda mitgespeichert

Voraussetzung: Variablen, die in Lambdas angesprochen werden, müssen "effectively final" sein

- final int limit = ...; oder
- int limit = ...;
  - nach der Initialisierung nicht mehr verändert → sonst Fehlermeldung

Kopie der Variable wird in Lambda gespeichert. Lambdas mit Zugriff auf äußere Variablen nennt man auch Closures.

Zugriff auf Felder der umgebenden Klasse

```
class C {
  int limit = ...;
  void foo (int[] a) {
    int pos = position(a, x -> x > limit);
    ...
  }
}
```

muss nicht als *final* deklariert werden

Zugriff auf Feld der Klasse *C*

Zugriff auf limit bedeutet eigentlich this.limit
=> Garbage Collector sammelt this nicht ein, solange es vom Lambda referenziert wird
=> this bezieht sich auf Objekt der umgebenden Klasse (nicht auf Objekt der anonymen Klasse)

Streams in Java

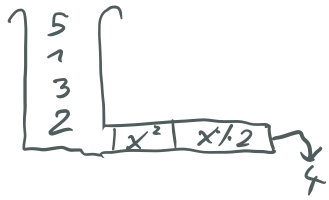
What is a Stream?

A stream is a sequence of data elements made available over time. The data originates from a source and can be processed.

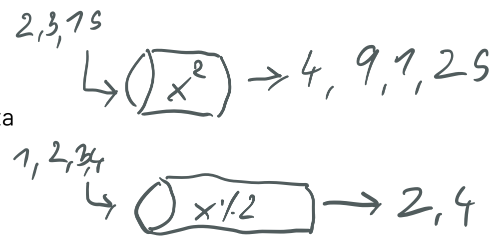
- Programming style is declarative and programmers use "building blocks"
- Pattern: map – filter – reduce
- Stream processing is lazy

Collections

- Store data
- does not offer operations on data
- External Iteration



Streams



- Does not store data
- Collection of operations to process data
- Internal Iteration

First Example

Find the name of all dishes with less than 400cal

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public enum Type {MEAT, FISH, OTHER }

    public Dish(String name, boolean vegetarian, int calories, Type type) { … }

    public String getName() { return name; }
    …
}

List<Dish> menu = Arrays.asList(
        new Dish("pork", false, 800, Dish.Type.MEAT), new Dish("beef", false, 700, Dish.Type.MEAT),
        new Dish("chicken", false, 400, Dish.Type.MEAT), new Dish("french fries", true, 530,
Dish.Type.OTHER),
        new Dish("rice", true, 350, Dish.Type.OTHER), new Dish("season fruit", true, 120,
Dish.Type.OTHER),
        new Dish("pizza", true, 550, Dish.Type.OTHER), new Dish("prawns", false, 400,
Dish.Type.FISH),
        new Dish("salmon", false, 450, Dish.Type.FISH)
);
```

A possible imperative solution using loops and temporary collections.

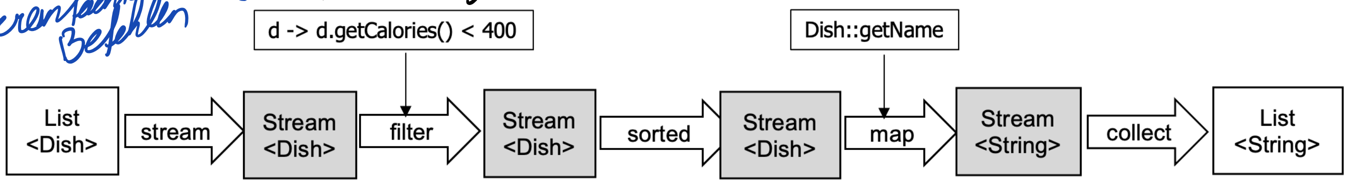
```java
public static List<String> getLowCaloricDishesNames (List<Dish> dishes){
        List<Dish> lowCaloricDishes = new ArrayList<>();
        for (Dish d: dishes){
            if (d.getCalories() < 400){
                lowCaloricDishes.add(d);
            }
        }
        List<String> lowCaloricDishesName = new ArrayList<>();
        Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
            public int compare(Dish d1, Dish d2){
                return Integer.compare(d1.getCalories(), d2.getCalories());
            }
        });
        for (Dish d: lowCaloricDishes){
            lowCaloricDishesName.add(d.getName());
        }
        return lowCaloricDishesName;
}
```

Functional programming using map – filter –reduce pattern:

```java
public static List<String> getLowCaloricDishesNames(List<Dish> dishes) {
        return dishes.stream()
                .filter(d -> d.getCalories() < 400)
                .sorted(comparing(Dish::getCalories))
                .map(Dish::getName)
                .collect(toList());
}
```

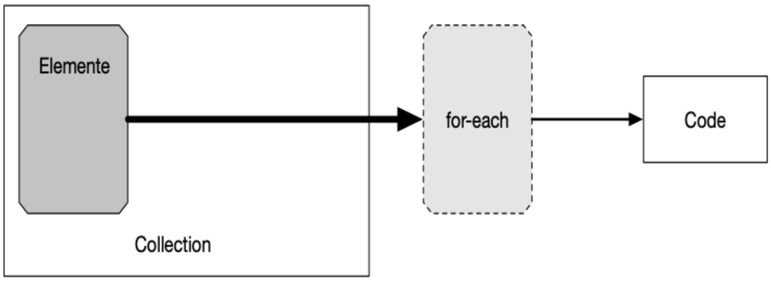
statt den vereinfachten Befehlen

```
{ .sorted(a,b) -> Integer.compare(a.getCalories(), b.getCalories())
  .map(a -> a.getName
```



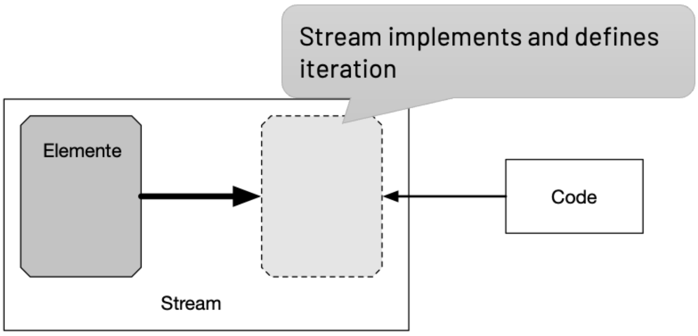
Imperative style ➔ External iteration

```
public static List<String> externalIteration(List<Dish> dishes) {
    List<String> names = new ArrayList<>();
    for (Dish d : dishes) {
        names.add(d.getName());
    }
    return names;
}
```



Functional style ➔ Internal iteration

```
public static List<String> internalIteration(List<Dish> dishes) {
    return dishes.stream().map(Dish::getName).collect(Collectors.toList());
}
```



External iteration ensures that elements are processed sequentially. The programmer defines **HOW** the elements are processed and **WHICH** operation is applied.

➔ Over-specified: in many cases the **WHAT** would be sufficient
➔ Performance disadvantage

```
List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});
for(String letter: alphabets) {
    letter.toUpperCase();
}
```

Internal iteration allows to only specify the operation (i.e., WHICH operation should be applied to the elements).

➔ How the elements are processed is not specified but can be defined by the stream
➔ Better for optimization. E.g. parallization.

```
List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});
alphabets.forEach(l -> l.toUpperCase());
```

Lazy Evaluation Maturathema

Streams process elements **lazily**, i.e., operations are applied to the elements on demand.
On demand: If an element is needed for materialization (terminal operation).

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares = numbers.stream().filter(n -> {
  System.out.println("filtering " + n);
  return n % 2 == 0;
}).map(n -> {
  System.out.println("mapping " + n);
  return n * n;
}).limit(2)
.collect(toList());
```

Output:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Streams for Primitives

Generic stream

```
public interface Stream<T>
```

Streams for primitives **int**, **long**, and **double**

```
public interface IntStream
public interface LongStream
public interface DoubleStream
```

Generische Typen viel langsamer => deshalb primitive Streams

Stream Operations

Create streams (Source → Stream)

from collections	`coll.stream(), Stream.of(T... values)`
Generator methods	`generate(Supplier<T> s), iterate(T seed, UnaryOperator<T> f)`
Library methods	`Files.lines(), …`

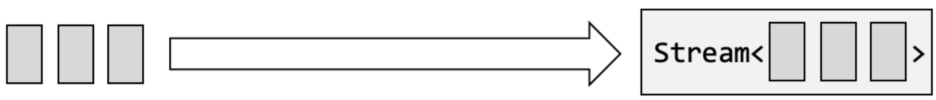
Intermediate operations (Stream → Stream)

Mapping	`map(Function<? super T, ? extends R> mapper),` `flatMap, mapToInt, mapToDouble, … analog`
Filtering	`filter(Predicate<? super T> predicate)`
Sorting	`sorted(), sorted(Comparator<? super T> comparator)`
Subsets	`limit(long maxSize), skip(long n), distinct()`

Terminal operations (Stream → Result)

Iteration	`forEach(Consumer<? super T> action)`
Reduce	`reduce(T identity, BinaryOperator<T> accumulator)`
Collection	`collect`
Grouping	`collect + Collector`
Finding elements	`min(), max(), findFirst(), findAny()`
Other	`count(), sum(),  allMatch(), anyMatch(), noneMatch()`

Building Streams



Streams from Collections

Stream from collections

```
List<String> words = new ArrayList<String>();
words.add("A"); ...

Stream<String> wordStream = words.stream();
```

Stream from arrays

```
int[] numbers = new int[] { 1, 2, 3 };
IntStream numberStream = Arrays.stream(numbers);
```

Stream from list of values

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Stream<String> single = Stream.of("Me");
Stream<String> all = Stream.concat(single, names);
Stream<String> none = Stream.empty();
```

Generators

Stream.generate: Uses a Supplier<T> to create a stream

```
final Random r = new Random();
IntStream randStream = IntStream.generate(() -> r.nextInt(100));
```

Stream.iterate: Starting at a **seed** value; Computes the next value based on the previous value

```
Stream<Point> randomWalk = Stream.iterate(new Point(0, 0),
  p -> new Point(p.x + r.nextInt(DIST), p.y + r.nextInt(DIST)));
```

Others

Streams are very useful to traverse folders and files!

Get lines of a file:

```
static Stream<String> lines(Path path)
static Stream<String> lines(Path path, Charset cs)
```

Get files in a folder:

```
static Stream<Path> list(Path dir)
```

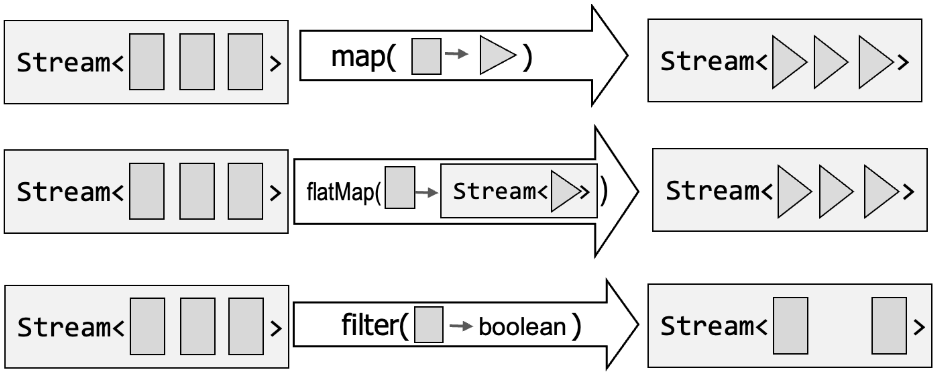
Search for a file:

```
static Stream<Path> find(Path start, int maxDepth,
                       BiPredicate<Path, BasicFileAttributes> matcher, FileVisitOption... options)
```

Walk an entire folder:

```
static Stream<Path> walk(Path start, FileVisitOption... options)
static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options)
```

Intermediate Operations



Mapping

Map:

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

Example:

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Stream<String> initials = names.map(a -> a.substring(0, 1));
```

Result:

```
A,P,M,J
```

mapToInt, mapToLong, mapToDouble

```
IntStream       mapToInt(ToIntFunction<? super T> mapper)
LongStream      mapToLong(ToLongFunction<? super T> mapper)
DoubleStream    mapToDouble(ToDoubleFunction<? super T> mapper)
```

Example:

```
IntString length = names.mapToInt(a -> a.length());
```

Result:

```
3,3,4,3
```

FlatMap:

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

Example:

```
Stream<Integer> posInts = Stream.iterate(0, x -> x + 1);
posInts.limit(5)
   .flatMapToInt(i -> IntStream.iterate(1, j -> j + 1).limit(i));
```

Result:

```
1,   1,2,    1,2,3,     1,2,3,4
```

Example: All characters of all words

```
IntString allChars = names.flatMapToInt(w -> w.chars());
```

Result:

```
65,110,110,80,97 …
```

Filtering

```
Stream<String> shortNames = names.filter(name -> name.length <= 3);
```

Result:

```
"Ann", "Pat", "Joe"
```

Others

limit: cut of stream after maxSize elements

Example:

```
IntStream numbers =  IntStream.of(10,9,8,7,6,5,4,3,2,1);
numbers.limit(5);
```

Result:

```
10,9,8,7,6
```

skip: skip first n elements

```
numbers.skip(5);
```

Result:

```
5,4,3,2,1
```

distinct: remove duplicates (using equals)

Example:

```
IntStream numbers =  IntStream.of(10,9,8,7,6,6,7,8,9,10);
stream.distinct();
```

Result:

```
10,9,8,7,6
```

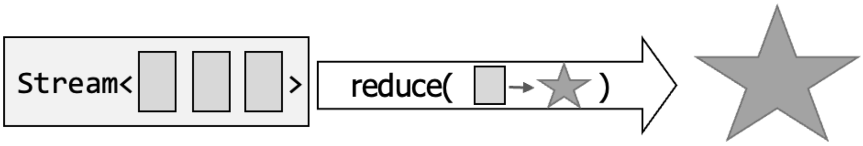
sorted: sort elements

```
Stream<String> namesSorted = names.sorted();
```

Result:

```
"Ann", "Joe", "Mary", "Pat"
```

Terminal Operations



Iteration

forEach: iterate over all elements and apply a Consumer<T>

Example:

forEach

```
sortedNames.filter(name -> {
  System.println(name);
});
```

Result:

```
Ann
Joe
Mary
Pat
```

Reduce

Combine elements of type T to a result of Type U

- provide an initial value
- use a function U x T → U to add elements to a partial result
- use a binary operator U x U → U to combine partial results

Maturafrage

ATTENTION: reduce relies on *immutable reduction*, i.e., combining two values must result a NEW value

Example: Sum of all elements in a Stream<Integer>

```
Stream<Integer> numbers =  Stream.of(10, 9, 8, 7, 6, 6, 7, 8, 9, 10);
int sum = numbers.reduce(0,  (partialSum, x) -> partialSum + x,  (partial1, partial2) -> partial1 +
partial2 );
```

Another example: Combine elements of type T using a binary operator

- use **identity** of type T
- binary operator **T x T → T**
- result of type **T**

```
int count = numbers.reduce(0,  (n, s) -> n + 1);
```

One can alos only provide an accumulator: Combine elements of type T using a binary operator

- binary operator **T x T → T**
- result of type **Optional<T>**

```
Optional<Integer> maxOpt = numbers.reduce((currentMax, x) -> (x > currentMax) ? x : currentMax);
```

Collect

Accumulate elements of type **T** to result of type **U**

- collect changes a container and accumulates the result
- can be used with very powerful **Collectors** class

Grouping data iteratively:

```
Map<Currency, List<Transaction>> transactionsByCurrencies = new HashMap<>();
for (Transaction transaction : transactions) {
  Currency currency = transaction.getCurrency();
  List<Transaction> transactionsForCurrency = transactionsByCurrencies.get(currency);
  if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies.put(currency, transactionsForCurrency);
  }
  transactionsForCurrency.add(transaction);
}
```

Grouping using a Collector:

```
Map<Currency, List<Transaction>> transactionsByCurrencies = transactions.stream()
        .collect(Collectors.groupingBy(Transaction::getCurrency));
```

Collect vs. Reduce

Reduce: combines two values and produces a new one ➔ immutable
Collect: modifies a container to accumulate result

WRONG: Using reduce to collect elements to an ArrayList

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
ArrayList<String> nameList = names.reduce(
        new ArrayList<String>(),
        (l, x) -> {
                        l.add(x);
                        return l;
                },
        (l1, l2) -> {
                        l1.addAll(l2);
                        return l2;
                });
```

Collectors

Collectors allow combining all elements into a Collection

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
List<String> list = names.map(n -> n.substring(0, 1))
                 .collect(Collectors.toList());
```

Result

```
[A,P,M,J]
```

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Map<Integer, List<String>> namesByLength =
      names.collect(Collectors.groupingBy(n -> n.length()));
```

Result

```
[3 -> [Ann, Pat, Joe],
 4 -> [Mary]
]
```

Collectors provide many functions to combine a stream to a single value

Average:

```
menu.stream().collect(Collectors.averagingInt(Dish::getCalories));
```

```
477.77777777777777
```

Statistics:

```
menu.stream().collect(Collectors.summarizingInt(Dish::getCalories));
```

```
IntSummaryStatistics{count=9, sum=4300, min=120, average=477.777778, max=800}
```

Others

Count

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
long count = names.count();
```

Min

```
IntStream stream =  IntStream.of(9,1,7,3,8,2,4,6,5);
OptionalInt min = stream.min();
```

Max

```
IntStream stream =  IntStream.of(9,1,7,3,8,2,4,6,5);
OptionalInt max = stream.max();
```

Sum

```
IntStream stream =  IntStream.of(9,1,7,3,8,2,4,6,5);
OptionalInt sum = stream.sum();
```

findFirst

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Optional<String> firstWord = names.findFirst();
```

findAny

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Optional<String> any = names.findAny();
```

toArray

```
int arr[] = IntStream.range(0, 100).toArray();
```

Hints

Order of Execution

```
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {
  System.out.println("filter: " + s);
  return true;
}).forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
filter: d2
forEach: d2
filter: a2
forEach: a2
filter: b1
forEach: b1
filter: b3
forEach: b3
filter: c
forEach: c
```

```
Stream.of("d2", "a2", "b1", "b3", "c").map(s -> {
  System.out.println("map: " + s);
  return s.toUpperCase();
}).anyMatch(s -> {
  System.out.println("anyMatch: " + s);
  return s.startsWith("A");
});
```

Output

```
map: d2
anyMatch: D2
map: a2
anyMatch: A2
```

Swap map and filter

```
Stream.of("d2", "a2", "b1", "b3", "c").map(s -> {
  System.out.println("map: " + s);
  return s.toUpperCase();
}).filter(s -> {
  System.out.println("filter: " + s);
  return s.startsWith("A");
}).forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
map: d2
filter: D2
map: a2
filter: A2
forEach: A2
map: b1
filter: B1
map: b3
filter: B3
map: c
filter: C
```

```
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {
  System.out.println("filter: " + s);
  return s.startsWith("a");
}).map(s -> {
  System.out.println("map: " + s);
  return s.toUpperCase();
}).forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
filter: d2
filter: a2
map: a2
forEach: A2
filter: b1
filter: b3
filter: c
```

```
Stream.of("d2", "a2", "b1", "b3", "c").sorted((s1, s2) -> {
  System.out.printf("sort: %s; %s\n", s1, s2);
  return s1.compareTo(s2);
}).filter(s -> {
  System.out.println("filter: " + s);
  return s.startsWith("a");
}).map(s -> {
  System.out.println("map: " + s);
  return s.toUpperCase();
}).forEach(s -> System.out.println("forEach: " + s));
```

Result:

```
sort: a2; d2
sort: b1; a2
sort: b1; d2
sort: b1; a2
sort: b3; b1
sort: b3; d2
sort: c; b3
sort: c; d2

filter: a2
map: a2
forEach: A2
filter: b1
filter: b3
filter: c
filter: d2
```

```
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {
  System.out.println("filter: " + s);
  return s.startsWith("a");
}).sorted((s1, s2) -> {
  System.out.printf("sort: %s; %s\n", s1, s2);
  return s1.compareTo(s2);
}).map(s -> {
  System.out.println("map: " + s);
  return s.toUpperCase();
}).forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
filter: d2
filter: a2
filter: b1
filter: b3
filter: c
map: a2
forEach: A2
```

Never reuse a stream

```
IntStream stream = IntStream.of(1, 2);
stream.forEach(System.out::println);
// That was fun! Let's do it again!
stream.forEach(System.out::println);
```

Endless streams

```
// Grab a coffee!
int sum = IntStream.iterate(0, i -> i + 1).sum();
```

```
// Grab another coffee!
IntStream.iterate(0, i -> (i + 1) % 2)
  .distinct()
  .limit(10)
  .forEach(System.out::println);
```

The order matters!

```
IntStream.iterate(0, i -> i + 1)
  .limit(10)
  .skip(5)
  .forEach(System.out::println);
```

Output:

```
5,6,7,8,9
```

```
IntStream.iterate(0, i -> i + 1)
  .skip(5)
  .limit(10)
  .forEach(System.out::println);
```

Output:

```
5,6,7,8,9,10,11,12,13,14
```