TECHNISCHE HOCHSCHULE LÜBECK

Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 04

Einführung in die funktionale Programmierung

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter - Map - Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

Unit 04

Einführung in die Funktionale Programmierung

Code as Parameter

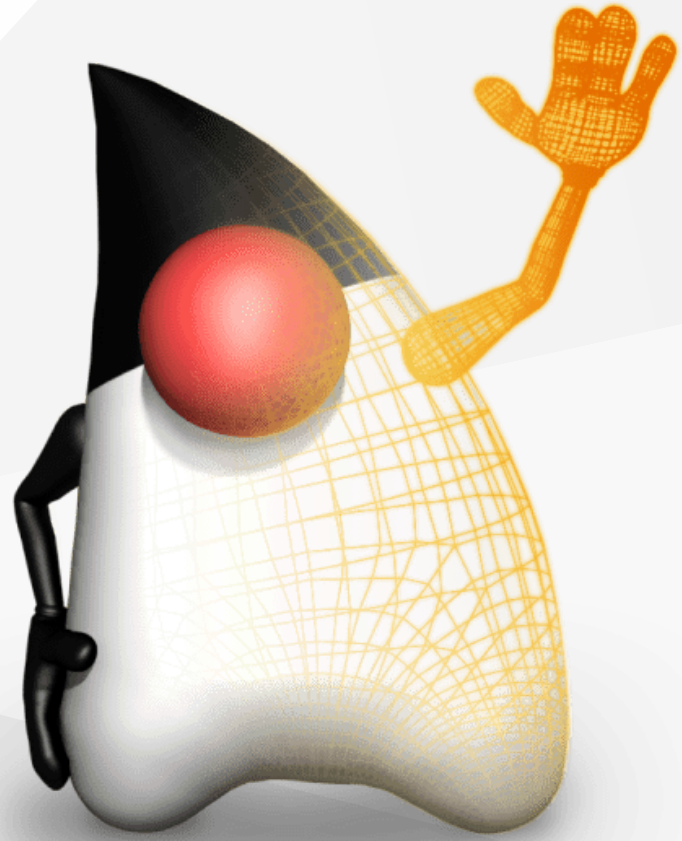
- Wiederverwendbarkeit durch Parameterisierung von Logik

Lambda Funktionen

- Wie ruft man Lambda Funktionen auf?
- Syntax von Lambda Funktionen
- Funktionsinterface bestimmt den Typ einer Lambda-Funktion (Function, Operator, Predicate, Supplier, Consumer, etc.)

Übungsaufgaben

- Definition von Operatoren, Funktionen, Prädikaten
- Methoden die Lambda-Funktionen als Parameter nutzen



Code as Parameter

Häh?

Einführung in funktionales Denken

Angenommen Sie sollen in einer Liste alle geraden (oder auch ungeraden) Werte selektieren.

```
public List<Integer> filter(List<Integer> vs) {
  List<Integer> filtered = new ArrayList<>();
  for (int v : values) {
    if (v % 2 == 0) { // <= einziger Unterschied
      filltered.add(v);
    }
  }
  return filtered;
}
```

Angenommen Sie sollen in einer Liste alle Werte unterhalb des Wertes 42 selektieren.

```
public List<Integer> filter(List<Integer> vs) {
  List<Integer> filtered = new ArrayList<>();
  for (int v : values) {
    if (v < 42) { // <= einziger Unterschied
      filltered.add(v);
    }
  }
  return filtered;
}
```

Die Logik ist in beiden Fällen gleich, einzig und allein die logische Bedingung zur Selektion von Werten unterscheiden sich.

- Im Falle von Werten würde man einen Parameter einführen.
- Aber wie parametersiert man Bedingungen wie `v % 2 == 0` oder `v < 42`?

Beispiel:
Parametrisierung beliebiger Bedingungen mittels `Predicate`

Um die Bedingungen wie `v % 2 == 0` oder `v < 42` dynamisch in einer Methode zu handhaben, können wir diese als Platzhalter mittels eines Datentyps `Predicate` im Code vorsehen. Dieser repräsentiert keinen Wert, sondern **ausführbaren Code**.

Zum Zeitpunkt der Programmierung wissen wir noch nicht, welche Bedingungen genau geprüft werden (nur das Bedingungen geprüft werden).

```
import java.util.function.Predicate; // <= Wir brauchen in dieser Unit das function Package

public static List<Integer> filter(List<Integer> vs, Predicate<Integer> predicate) {
  List<Integer> result = new ArrayList<>();
  for (int v : vs) {
    if (predicate.test(v)) { // <= Dies parametrisiert die Bedingung
      result.add(v);
    }
  }
  return result;
}
```

Beispiel:
Parametrisierung beliebiger Bedingungen mittels `Predicate`

Wir führen also einen Parameter namens `predicate` ein, der eine beliebige Bedingung ausdrücken kann, die **außerhalb** der Methode definierbar ist.

```
public static List<Integer> filter(List<Integer> vs, Predicate<Integer> predicate) {
  List<Integer> result = new ArrayList<>();
  for (int v : vs) {
    if (predicate.test(v)) { // <= Dies parametrisiert die Bedingung
      result.add(v);
    }
  }
  return result;
}
```

Dann können wir unterschiedlichste Bedingungen mittels sogenannter **Lambda-Funktionen** definieren.

```
List<Integer> sample = List.of(11, 12, 13, 14, 15, 16, 17, 18, 19);
List<Integer> evens = filter(sample, v -> v % 2 == 0); // => [12, 14, 16, 18] (nur gerade Werte)
List<Integer> lucky = filter(sample, v -> v != 13); // => [11, 12, 14, 15, 16, 17, 18, 19] (ungleich 13)
List<Integer> endsOn7 = filter(sample, v -> v % 10 == 7); // => [17] (alles was auf 7 endet)
```

Was sind Lambda Funktionen?

Der Begriff "Lambda" stammt aus dem Lambda-Kalkül, einer formalen Sprache, die in den 1930er Jahren von Alonzo Church entwickelt wurde, um Funktionen zu definieren und anzuwenden.

1. **Anonyme Funktionen**: Lambda-Ausdrücke haben **keinen** spezifischen **Namen**. Sie können als Objekte behandelt werden, die wie jede andere Variable gespeichert und übergeben werden können.
2. **Syntax**: Ein Lambda-Ausdruck besteht aus einer Parameterliste, der Zeichenfolge `->` (Pfeil) und einem Ausdruck, der den Rückgabewert der Funktion definiert. Zum Beispiel: `x -> 2 * x`
3. **Funktionale Schnittstellen**: In Java sind Lambda-Ausdrücke typischerweise an **funktionale Schnittstellen** gebunden, die genau eine abstrakte Methode definieren. Dies ermöglicht es, Lambda-Ausdrücke als Implementierungen dieser Schnittstellen zu verwenden.

Naming Things ist eines der "schwierigen" Probleme der Informatik.

Kompakte Syntax-Regeln für Lambdas

Ausgehend von einer "klassischen" Methode werden wir diese nun "schrittweise" zu einer einfach Lambda-Funktion vereinfachen:

```
public int add(int x, int y) { return x + y; }
```

1. Man kann diese anonym (d.h. ohne Namen) wie folgt als Lambda Ausdruck definieren:

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

2. Auf die Angaben von Typen kann man bei den Parametern verzichten (Typinferenz):

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

3. Besteht der Rumpf nur aus einer `return` Anweisung kann diese weiter vereinfacht werden

   ```
   (x, y) -> x + y
   ```

4. Hat die Funktion nur einen Parameter, kann auch auf die Klammern um die Parameter verzichtet werden:

   ```
   x -> x + x
   ```

Weitere Syntax-Regeln für Lambdas

Lambda-Rümpfe können mehr als eine Anweisung umfassen (auch wenn das kein klassischer Einsatzzweck von Lambdas ist und vermieden werden sollte).

Beispiel: Summe einer Liste von Werten

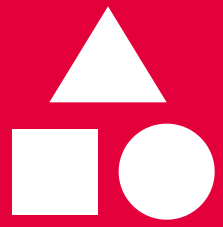
```
(int[] vs) -> {
  int sum = 0;
  for (v : vs) sum += v;
  return sum;
}
```

Lambda-Funktionen können auch keinen Parameter haben (wir nennen diese dann `Supplier`):

Beispiel: Erzeugung einer Zufallszahl bzw. Konsolenausgabe

```
() -> Math.random()
```

```
() -> System.out.println("Hello World")
```



Arten von Lambda Funktionen

Aber vorab:
Wie ruft man eine Lambda Funktion überhaupt auf?

Aufruf von Lambda Funktionen

Um eine Lambda-Funktion programmatisch in Java aufzurufen, wird eine Funktionsvariable benötigt, die einen Lambda-Ausdruck repräsentiert. Diese Funktionsvariable ist in der Regel einem **Funktionsinterface** zugeordnet, das eine Methode definiert, die die Lambda-Funktion implementiert.

Beispiel (ein zweistelliger Operator):

```java
import java.util.function.BinaryOperator; // BiOperator = Funktionsinterface

BinaryOperator<Integer> adder = (x, y) -> x + y; // adder = Funktionsvariable
System.out.println(adder.apply(46, -4)); // => 42
```

- Die Lambdas `Function` und `Operator` werden mit der `apply()`-Methode aufgerufen.
- `Predicate` (log. Bedingungen) allerdings nicht. Diese wird mit `test()` aufgerufen (da etwas geprüft wird).

Beispiel (ein einstelliges Prädikat):

```java
import java.util.function.Predicate; // Predicate = Funktionsinterface

Predicate<Integer> isEven = x -> x % 2 == 0; // isEven = Funktionsvariable
System.out.println(isEven.test(42)); // => true
```

Welche Funktionstypen von Lambdas gibt es?

Übersicht der (wichtigsten) definierten Funktionsinterfaces im `java.util.function`-Paket:

Funktions Interface	Funktionale Methode	Beschreibung
`Function<T,R>`	`R apply(T)`	Wandelt das Argument `t` in einen anderen Wert vom Typ `R` um. Nicht Typerhaltend.
`BiFunction<T,U,R>`	`R apply(T, U)`	Erzeugt aus den Argumenten `t` und `u` ein Ergebnis vom Typ `R`. Nicht Typerhaltend.
`UnaryOperator<T>`	`T apply(T)`	Wendet eine Operation auf `t` an und liefert ein `T`-Objekt. Typerhaltend.
`BinaryOperator<T>`	`T apply(T,T)`	Verknüpft zwei Argumente `p` und `q` mit einem Operator zu einem neuen Wert `T`. Typerhaltend.
`Predicate<T>`	`boolean test(T)`	Prüft, ob eine Bedingung für das Argument `t` erfüllt ist. Liefert immer `boolean`.
`BiPredicate<T, R>`	`boolean test(T,R)`	Prüft, ob eine Bedingung für die Argumente `t` und `r` erfüllt sind. Liefert immer `boolean`.
`Supplier<T>`	`T get()`	Liefert ein Ergebnis.
`Consumer<T>`	`void accept(T)`	Verarbeitet das Objekt `t`.
`Comparator<T>`	`int compare(T,T)`	Vergleicht zwei Objekte `p` und `q` und gibt einen ganzzahligen Wert zurück.

Funktionen: $f : X \to Y$ bzw. $f : X \times Y \to Z$

In der Mathematik wird eine Funktion als eine Zuordnung oder Abbildung beschrieben, die jedem Element einer Menge (X, Definitionsmenge) genau ein Element einer anderen Menge (Y, Wertebereich oder Bildmenge) zuordnet. X und Y können, müssen aber nicht identisch sein.

Beispiel einer einstelligen Funktion:

```
Function<String, Integer> length = s -> s.length();
length.apply("Hallo"); // => 5
```

Funktionen können auch mehrstellig sein.

Beispiel einer zweistelligen Funktion, die Zeichenketten n-mal hintereinander hängt:

```
BiFunction<String, Integer, String> repeat = (s, n) -> {
  String ret = "";
  for (int i = 0; i < n; i++) ret += s;
  return ret;
}
repeat.apply("Hallo", 3); // => HalloHalloHallo
```

Drei- und mehrstellige Funktionen

Im `java.util.function` Package gibt es aber keine drei-, vier- oder gar mehrstelligen Funktionsinterfaces. Eine dreistellige Funktion kann man aber wie folgt definieren (vier- und mehrstellig analog):

```
interface TriFunction<A, B, C, R> {
  R apply(A a, B b, C c);
}
```

Beispiel einer dreistelligen Funktion:
Eine Zeichenkette wird n-mal hintereinander hängt und durch Trennzeichen getrennt.

```
TriFunction<String, Integer, Character, String> repeatWithSeparator = (text, n, sep) -> {
  String result = "";
  for (int i = 0; i < n; i++) {
    if (i != 0) result += sep;
    result += text;
  }
  return result;
};
```

```
repeatWithSeparator.apply("Hallo", 3, '-'); // => "Hallo-Hallo-Hallo"
```

Operatoren: $op : X \to X$ bzw. $op : X \times X \to X$

Mathematisch betrachtet sind Operatoren eine spezielle Art von Funktionen, da sie typerhaltend sind. D.h. alle Operanden und das Result einer Verknüpfung auf diesen Operanden haben denselben Datentyp.

`UnaryOperator<T>` für einstellige Operatoren
Beispiel: Ziehen von Zufallszahlen zwischen $[0, n[$

```
import java.util.Random;
import java.util.function.UnaryOperator;

UnaryOperator<Integer> randomizer = n -> {
  Random random = new Random();
  return random.nextInt(n);
};

randomizer.apply(5); // => z.B. 3
```

Das generelle Function Interface wäre auch gegangen.
`Function<Integer, Integer> rand = ...`

`BinaryOperator<T>` für zweistellige Operatoren
Beispiel: Ziehen von Zufallszahlen zwischen $[n, m[$

```
import java.util.Random;
import java.util.function.BinaryOperator;

BinaryOperator<Integer> rand = (n, m) -> {
  Random random = new Random();
  return random.nextInt(m - n + 1) + n;
};

rand.apply(7, 13); // => z.B. 9
```

Das generelle BiFunction Interface ginge auch.
`BiFunction<Integer, Integer, Integer> rand = ...`

Anmerkung • Drei- und mehrstellige Operatoren analog zu Funktionen, also bspw.:
`interface TriOperator<T> { T apply(T a, T b, T c); }`

Prädikate

$pred : X \to \mathbb{B}$ bzw. $pred : X \times X \to \mathbb{B}$

Mathematisch betrachtet sind auch Prädikate eine spezielle Art von Funktionen, die alle Eingabewerte auf boolesche Werte abbilden.

Beispiel:

```
BiPredicate<String> equallyGolden = (t1, t2) -> t1.contains("gold") == t2.contains("gold");
equallyGolden.test("Zum goldenen Hirsch", "Zum goldenen Anker"); // => true
equallyGolden.test("Zum braunen Hirsch", "Zum goldenen Anker"); // => false
equallyGolden.test("Zum braunen Hirsch", "Zum schwarzen Abt"); // => true
```

Auch hier hätte man das Prädikat wieder durch eine zweistellige Funktion ausdrücken können.

```
BiFunction<String, String, Boolean> equallyGolden = (t1, t2) -> t1.contains("gold") == t2.contains("gold");
```

Merke: Alle Funktionen, die auf `Boolean` mappen, sind Prädikate.

Drei- und mehrstellige Prädikate analog `Operator` und `Function`, d.h.

```
interface TriPredicate<T> { boolean test(T a, T b, T c); }
```

Supplier und Consumer

Supplier: Funktionale Schnittstelle, die eine Methode `get()`, definiert ohne dafür Eingaben zu benötigen. Wird normalerweise genutzt, um Dinge zu erzeugen (oft zufällig oder aus externen Quellen), z.B. um Zufallszahlen zwischen $[0, 100[$ zu generieren.

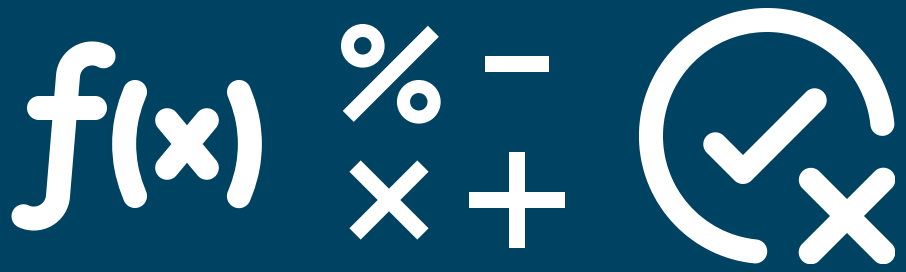
```
import java.util.Random;
import java.util.function.Supplier;

Supplier<Integer> zufallsZahlSupplier = () -> new Random().nextInt(100);
System.out.println("Zufallszahl: " + zufallsZahlSupplier.get());
```

Consumer: Eine funktionale Schnittstelle, die eine Methode `accept(T)` definiert. Consumer werden oft verwendet, um Eingabedaten zu verarbeiten oder zu modifizieren. Zum Beispiel kann ein Consumer dazu verwendet werden, Werte zu drucken, in einer Datenbank zu speichern oder in eine Datei zu schreiben.

```
List<String> cities = List.of("Hamburg", "München", "Lübeck");
Consumer<String> hanseaticPrint = city ->
  System.out.print(city.equals("Lübeck") ? city + " (Königin der Hanse)" : city + ", ");
cities.stream().forEach(hanseaticPrint);
```

`Hamburg, München, Lübeck (Königin der Hanse)` *(zur Bedeutung von* `stream()` *kommen wir nächste Woche)*



Übungsaufgaben

Lambdas: Funktion • Operator • Prädikat

"Ob Du wirklich richtig stehst, siehst Du wenn das Licht an geht."

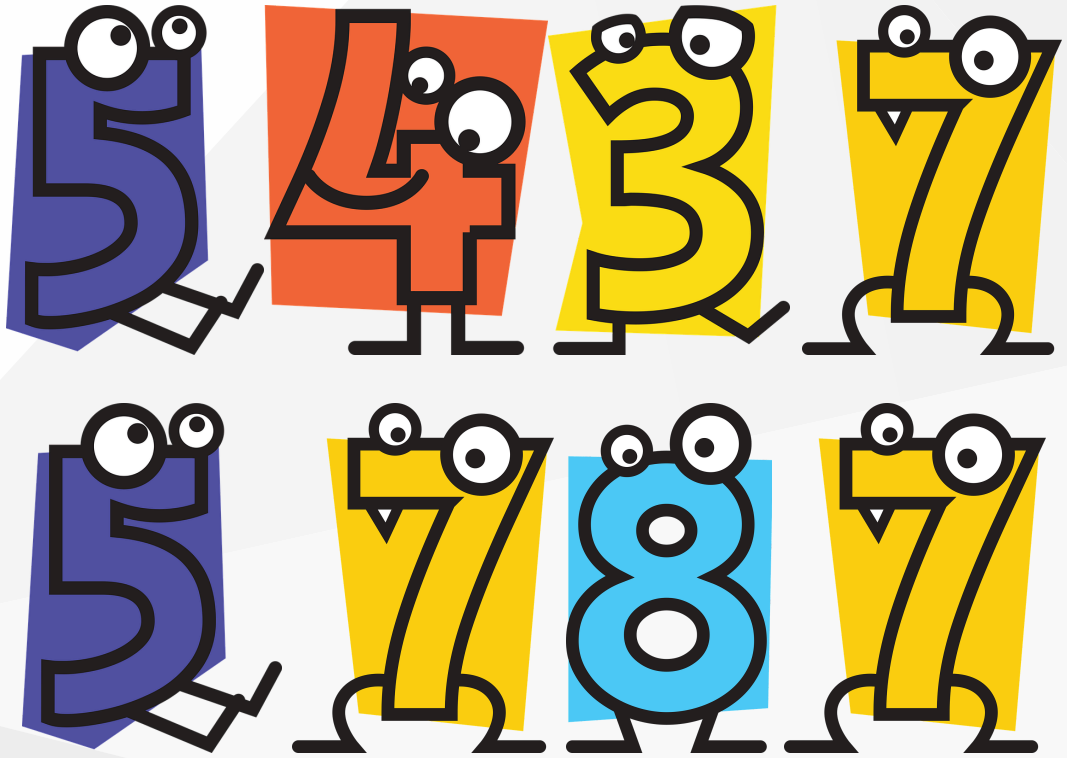
1. Aufgabe:

Erstellen Sie nun einen Lambda-Ausdruck, der bestimmt, welche Ziffern in einer positiven Zahl wie häufig vorkommen.

```
int value = 54375787;
Function|Operator|Predicate count = (...) -> { ... };
Map<Integer, Long> r = count.apply(value);
System.out.println(r);
// => { 3=1, 4=1, 5=2, 7=3, 8=1 }
```

Lösung:

>>> YOUR-TURN <<<



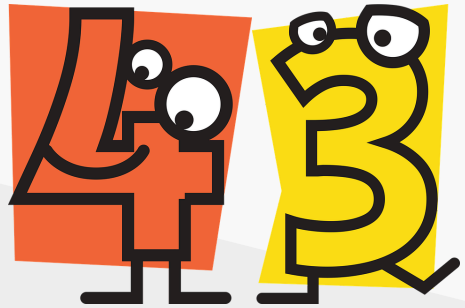
Bildquelle: BenKirb (pixabay.com)

2. Aufgabe:

Schreiben Sie nun eine Lambda-Funktion, die bestimmt ob zwei Zahlen "verliebt" sind. Zwei Zahlen seien verliebt, wenn ihre Quersummen gleich sind.

Beispiel:

```
int a = 81;
int b = 72;
int c = 34;
int d = 70;
Function|Operator|Predicate inLove = (...) -> { ... };
System.out.println(inLove.test(a, b)); // => true
System.out.println(inLove.test(a, c)); // => false
System.out.println(inLove.test(c, d)); // => true
```



Tipp:

Lösen Sie das Problem der Quersummen-Bildung und des Vergleichs in zwei separaten Lambda-Funktionen.

3. Aufgabe:

Schreiben Sie nun eine Methode namens `reduce()`, die auf einer Liste von Integer-Werten, einen zweistelligen Operator anwendet, der von links nach rechts immer das vorderste Element mit dem Resultat der letzten Operation zusammenfasst, beginnend ab einem Startwert.

Beispiele:

```
List<Integer> values = List.of(1, 2, 3, 4, 5);
int result = reduce(0, values, (a, b) -> a + b);
System.out.println(result); // => 15

result = reduce(1, values, (a, b) -> a * b);
System.out.println(result); // => 120

result = reduce(0, values, (a, b) -> b - a);
System.out.println(result); // => 3
```

Veranschaulichung:

```
0  [1, 2, 3, 4, 5] mit (a, b) -> a + b
1  [2, 3, 4, 5]
3  [3, 4, 5]
6  [4, 5]
10 [5]
15 []
```

```
0  [1, 2, 3, 4, 5] mit (a, b) -> b - a
1  [2, 3, 4, 5]
1  [3, 4, 5]
2  [4, 5]
2  [5]
3  []
```

4. Aufgabe:

Schreiben Sie nun eine Methode namens `generate()`, die eine Liste von n Werten durch einen Supplier erzeugt.

Beispiele:

```java
import java.util.Random;

List<Integer> randoms = generate(5, () -> new Random().nextInt(100));
System.out.println(randoms); // => z.B. [34, 17, 22, 0, 99]

List<Integer> notSoRandom = generate(3, () -> 42);
System.out.println(notSoRandom); // => [42, 42, 42]

notSoRandom = generate(0, () -> Integer.MAX_VALUE);
System.out.println(notSoRandom); // => []
```

Bildquelle: BenKirb (pixabay.com)

Zusammenfassung

- Code as Parameter
  Oft braucht man immer dieselben Code Muster, um Probleme zu lösen, die sich nur an wenigen Stellen Code unterscheiden (z.B. Filtern). Diese wenigen verbleibenden problemspezifischen Code-Stellen kann man mit sogenannten Lambda-Funktionen parametrisieren.
- Lambda Funktionen
  Lambda-Ausdrücke sind anonyme Funktionen in Java, die zur Implementierung von funktionalen Interfaces verwendet werden und es ermöglichen, Code in einer kompakten Notation zu schreiben, und anderen Methoden als Parameter zur Ausführung übergeben zu können.
- Funktionale Interfaces
  Ein funktionales Interface ist eine Schnittstelle in Java, die genau eine abstrakte Methode enthält und die Nutzung funktionaler Programmierkonzepte (Lambdas) ermöglicht. Die wichtigsten funktionalen Interfaces in Java sind `Function`, `Operator`, `Predicate`, `Consumer` *und* `Supplier` *mit denen man die gebräuchlisten funktionalen Methoden definieren kann.*

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe

```java
import java.util.*; import java.util.function.*;

public class Main {
  public static void main(String[] args) {
    int value = 54375787;

    Function<Integer, Map<Integer, Long>> count = x -> {
      Map<Integer, Long> counted = new TreeMap<>();
      while (x > 0) {
        int digit = x % 10;
        counted.put(digit, counted.getOrDefault(digit, 0L) + 1);
        x /= 10;
      }
      return counted;
    };

    Map<Integer, Long> r = count.apply(value);
    System.out.println(r);
  }
}
```

Lösungsvorschlag zur 2. Aufgabe

```java
import java.util.*; import java.util.function.*;

public class Main {

  public static void main(String[] args) {
    int a = 81; int b = 72; int c = 34; int d = 70;

    UnaryOperator<Integer> querSumme = x -> {
      int sum = 0;
      while (x > 0) { sum += x % 10; x /= 10; }
      return sum;
    };

    BiPredicate<Integer, Integer> inLove = (x, y) -> querSumme.apply(x) == querSumme.apply(y);
    System.out.println(inLove.test(a, b)); // => true
    System.out.println(inLove.test(a, c)); // => false
    System.out.println(inLove.test(c, d)); // => true
  }
}
```

Lösungsvorschlag zur 3. Aufgabe

```java
import java.util.*; import java.util.function.*;

public class Main {

  public static int reduce(int start, List<Integer> data, BinaryOperator<Integer> accumulator) {
    int aggregat = start;
    for (int x : data) aggregat = accumulator.apply(aggregat, x);
    return aggregat;
  }

  public static void main(String[] args) {
    List<Integer> values = List.of(1, 2, 3, 4, 5);
    int result = reduce(0, values, (a, b) -> a + b);
    System.out.println(result); // => 15

    result = reduce(1, values, (a, b) -> a * b);
    System.out.println(result); // => 120

    result = reduce(0, values, (a, b) -> b - a);
    System.out.println(result); // => 3
  }
}
```

Lösungsvorschlag zur 4. Aufgabe

```java
import java.util.*; import java.util.function.*; import java.util.Random;

public class Main {
  public static List<Integer> generate(int n, Supplier<Integer> newValue) {
    List<Integer> values = new ArrayList<>();
    while (n-- > 0) values.add(newValue.get());
    return values;
  }

  public static void main(String[] args) {
    List<Integer> randoms = generate(5, () -> new Random().nextInt(100));
    System.out.println(randoms); // => z.B. [34, 17, 22, 0, 99]

    List<Integer> notSoRandom = generate(3, () -> 42);
    System.out.println(notSoRandom); // => [42, 42, 42]

    notSoRandom = generate(0, () -> Integer.MAX_VALUE);
    System.out.println(notSoRandom); // => []
  }
}
```