Java: Lambda-Ausdrücke und Streams

- Den Zusammenhang zw. Lambda-Ausdrücken und Funktionalen Interfaces verstehen
- Den Begriff Funktionales Interface verstehen
- Syntaktische Vereinfachungen bei Lambda-Ausdrücken anwenden können
- Wichtige Funktionale Interfaces des Pakets `java.util.function` kennen
- Erkennen wie im Lambda-Ausdruck auf umschließende Objekte und lokale Variablen zugegriffen werden kann
- Mit Methoden- und Konstruktor-Referenzen umgehen können
- Mit Streams arbeiten können
- Arrays und Listen in Streams umwandeln und von Streams zurückwandeln können
- Zw. Intermediären und Terminalen Operationen bei Streams unterscheiden können
- Die Verarbeitungsreihenfolge der Operationen begreifen
- Streams erstellen können, die wiederverwendet werden können

Einführendes Beispiel

Ein String-Array soll sortiert werden, dabei sollen führende und abschließende Leerzeichen ignoriert werden

Unter Zuhilfenahme von `Arrays.sort()` und des `Comparators`

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

interface Comparator<T>
{
  int compare(T o1, T o2);
}
```

Im Package `java.util` vorhanden

Konkret:

```
String[] words = { "M", "\nSonne", " Q", "\t\tAm Abend" };
Arrays.sort(words, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.trim().compareTo(s2.trim());
  }
});
System.out.println(Arrays.toString(words));
```

Kürzer mit einem *Lambda-Ausdruck* (ab Java 8):

```
Arrays.sort(words, (String s1, String s2) -> {
  return s1.trim().compareTo(s2.trim());
});
```

Oder noch kürzer:

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

Auch die Zuweisung an eine lokale Variable ist möglich:

```
Comparator<String> c = (s1, s2) -> s1.trim().compareTo(s2.trim());
Arrays.sort(words, c);
```

Ein Lambda-Ausdruck implementiert kompakt ein Interface mit genau einer „**abstrakten**“ Methode = *Funktionales Interface*

Funktionale Interfaces sind im Paket `java.util.function` vorhanden

Ein Lambda-Ausdruck präsentiert nur den Java-Code und verbirgt das, was Compiler aus dem Kontext herleiten kann. In Wirklichkeit wird ein Objekt, welches das *Funktionale Interface* implementiert, erstellt

Syntaktische Vereinfachungen

- *Datentypen* in der Parameterliste können weggelassen werden, da diese aus der Bindung zum Interface ermittelt werden können
- Besteht Parameterliste nur aus *einem Parameter* können die runden Klammern weggelassen werden
- Besteht Methode aus *einer Anweisung* können geschweifte Klammern und das Schlüsselwort `return` entfallen

Beispiel

```
public class ComputeSum
{
  public static double sum(DoubleFunction<Double> f,int s,int e) {
    double ret = 0.0;
    for (int i = s; i <= e; i++)
      ret += f.apply(i);
    return ret;
  }

  public static void main(String[] args) {
    // 1 + 2 + 3 + ... + n ergibt n(n+1)/2
    System.out.println(sum(x -> x, 1, 100));
    // 1 + 1/2 + 1/4 + 1/8 + ... strebt gegen 2
    System.out.println(sum(x -> 1 / Math.pow(2, x), 0, 40));
    // 1 - 1/3 + 1/5 - 1/7 + ... strebt gegen pi/4
    DoubleFunction<Double> f = x -> Math.pow(-1, x) / (2 * x + 1);
    System.out.println(sum(f, 0, 10000000));
  }
}
```

Start

Definition

```
5050.0
1.9999999999990905
0.7853981883974454
```

Funktionen werden definiert und an anderer Stelle erst gestartet

FRAGE: Das *Funktionale Interface* `DoubleFunction` ist im Package `java.util.function` definiert. Wie lautet seine genaue Definition?

Wichtige Funktionale Interfaces des Packages `java.util.function`

Funktionales Interface	Abbildung
`Consumer<T>`	`(T) -> void`
`BiConsumer<T, U>`	`(T, U) -> void`
`DoubleConsumer, IntConsumer, LongConsumer`	
`Supplier<T>`	`() -> T`
`DoubleSupplier, IntSupplier, LongSupplier, BooleanSupplier`	
`Predicate<T>`	`(T) -> boolean`
`BiPredicate<T, U>`	`(T, U) -> boolean`
`DoublePredicate (double) -> boolean, IntPredicate, LongPredicate`	
`Function<T, R>`	`(T) -> R`
`BiFunction<T, U, R>`	`(T, U) -> R`
`DoubleFunction (double) -> R, IntFunction, LongFunction`	
`LongToDoubleFunction`	`(long) -> double`
`LongToIntFunction, IntToLongFunction, IntToDoubleFunction, …`	
`UnaryOperator<T>`	`(T) -> T`
`DoubleUnaryOperator, IntUnaryOperator, LongUnaryOperator`	
`BinaryOperator<T>`	`(T, T) -> T`
`DoubleBinaryOperator, IntBinaryOperator, LongBinaryOperator`	

Lambda-Ausdrücke und ihre Umgebung

- Lambda-Ausdrücke können *lokale Variablen* enthalten, wenn Compiler diese als „effektiv“ unveränderlich erkennt oder diese als `final` deklariert wurden[1)]
- Lambda-Ausdrücke können auf *Membervariablen* des umschließenden Objekts zugreifen und diese auch verändern[2)]
- Lambda-Ausdrücke können den `this`-Zeiger enthalten, wobei dabei wiederum auf das umschließende Objekt zugegriffen wird

```
public class PrintSupplier
{
  public static void printSupplier(Supplier<Integer> s) {
    System.out.println(s.get());
  }
}
```

```
public class PrintSupplierTest
{
  private int x = 1;
  private static int y = 2;

  public static void main(String[] args) {
    PrintSupplierTest pt = new PrintSupplierTest();
  }

  public PrintSupplierTest() {
    int z = 0;
    PrintSupplier.printSupplier(() -> {
      System.out.println(this.getClass().getName());
      this.x++[2)];
      y++;
      return x + y + z[1)];
    });
    System.out.println(this.x);
    System.out.println(PrintSupplierTest.y);
  }
}
```

```
PrintSupplierTest
5
2
3
```

AUFGABE: Schreiben Sie den Lambda-Ausdruck um, indem Sie an dessen Stelle eine *anonyme Klasse* verwenden. Funktioniert der Aufruf der Methode `get()` wenn sie die identischen vier Programmzeilen enthält?

Methoden- und Konstruktor-Referenzen

Besteht Lambda-Ausdruck lediglich aus einem Methodenaufruf, kann Methode auch durch *Klassenname::Methodenname* aufgerufen werden

```
public static double sum(double x, double y) {
  return x + y;
}

public static void printTable(DoubleBinaryOperator f,
  int cols, int rows) {
  for (int i = 1; i <= rows; i++) {
    for (int j = 1; j <= cols; j++)
      System.out.print(f.applyAsDouble(i, j) + " ");
    System.out.println();
  }
}

printTable((x, y) -> Math.pow(x, y), 3, 5);
printTable(Math::pow, 3, 5);
printTable(MethodReferenceTest::sum, 3, 5);
```

```
1.0 1.0 1.0
2.0 4.0 8.0
3.0 9.0 27.0
4.0 16.0 64.0
5.0 25.0 125.0

1.0 1.0 1.0
2.0 4.0 8.0
3.0 9.0 27.0
4.0 16.0 64.0
5.0 25.0 125.0

2.0 3.0 4.0
3.0 4.0 5.0
4.0 5.0 6.0
5.0 6.0 7.0
6.0 7.0 8.0
```

Mit *Klassenname::new* kann der passende Konstruktor aufgerufen werden der zum Funktionalen Interface passt

```
public static ArrayList<Integer> makeList(
  IntFunction<Integer> f, int start, int end) {
  ArrayList<Integer> ret = new ArrayList<Integer>();
  for (int i = start; i <= end; i++)
    ret.add(f.apply(i));
  return ret;
}

IntFunction<Integer> f1 = x -> x + 1;
IntFunction<Integer> f2 = Integer::valueOf;
IntFunction<Integer> f3 = Integer::new;
System.out.println(makeList(f1, 1, 3));
System.out.println(makeList(f2, 1, 3));
System.out.println(makeList(f3, 1, 3));
```

Integer valueOf(int i)

Integer(int i)
Konstruktor

```
[2, 3, 4]
[1, 2, 3]
[1, 2, 3]
```

Beispiel Objektfabrik

```
public class User
{
  private String name = null;
  private Integer age = null;

  public User(String name, Integer age) {
    this.name = name;
    this.age = age;
  }
  …
}
```

```
public interface UserFactory
{
  User getUser(String name, Integer age);
}
```

```
UserFactory userFactory = User::new;
User user = userFactory.getUser("Clark Kent", 29);
System.out.println(user);
```

Der Konstruktor ist gleich dem Funktionalen Interface.

`getUser()` ruft intern den Konstruktor auf

Lambda-Ausdrücke und Streams

Ein Stream repräsentiert eine Reihe von Elementen und erlaubt anhand unterschiedlicher Operationen diese zu bearbeiten.

Array in Stream

```
String[] myArray = {"a1", "a2", "b1", "c2", "c1"};
Arrays.stream(myArray)
  .filter(s -> s.startsWith("c"))
  .map(String::toUpperCase)
  .sorted()
  .forEach(System.out::println);
```

List in Stream

```
List<String> myList = Arrays.asList("a1", "b1", "a1", "c1", "d1");
myList.stream()
  .distinct()
  .reduce((x, y) -> x + "," + y)
  .ifPresent(System.out::println);[1]
```

Stream in Array

```
Object[] newArray= myList.stream()
  .limit(2)
  .toArray();
System.out.println(Arrays.toString(newArray));
```

Stream in List

```
List<String> newList = myList.stream()
  .skip(2)
  .collect(Collectors.toList());
newList.forEach(System.out::println);
```

```
C1
C2
a1,b1,c1,d1
[a1, b1]
a1
c1
d1
```

- Aus *Arrays*, *Listen*, *Collections*, *Objekten* oder unter Zuhilfenahme eines `Stream.Builder` kann Stream erzeugt werden
- Stream ändert ursprüngliche Datenstruktur nicht
- Operationen können nacheinander oder parallel ausgeführt werden
- Neben `Stream` gibt es noch `IntStream`, `LongStream` und `DoubleStream` um primitive Datentypen zu speichern
- Bereitgestellte Methoden haben als Argument meist Lambda-Ausdrücke und werden eingeteilt in:

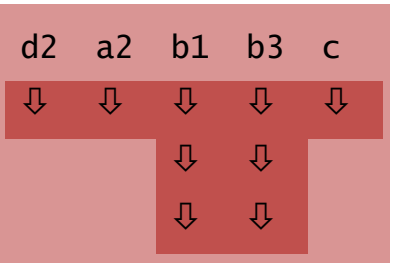
[1] Mit `get()` kann String-Ergebnis erfragt werden

Streamoperationen

Intermediäre Operationen (engl. Intermediate Operations)	Terminale Operationen (engl. Terminal Operations)
liefern wiederum einen Stream der weiterverarbeitet werden kann	führen Operationen auf Elemente aus, können einen Wert liefern und beenden den Stream
`filter()`, `map()`, `distinct()`, `sorted()`, `limit()`, `skip()`, …	`forEach()`, `reduce()`, `toArray()`, `count()`, `anyMatch()`, `collect()`, …

Verarbeitungsreihenfolge

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




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

- Reihenfolge der Operationen bestimmt maßgeblich die Verarbeitungsgeschwindigkeit
- Stream-Operationen werden nur dann ausgeführt, wenn s einer terminaler Operation abgeschlossen werden

Streams wiederverwenden

PROBLEM: Streams können – nachdem sie einmal geschlossen wurden – nicht mehr verwendet werden

```
Stream<String> stream =
  Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> s.startsWith("a"));          // ergibt true
stream.noneMatch(s -> s.startsWith("a"));         // wirft Exception
```

LÖSUNG:

```
Supplier<Stream<String>> sSupplier =
  () -> Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> s.startsWith("a"));

sSupplier.get().anyMatch(s -> s.startsWith("a"));  // ergibt true
sSupplier.get().noneMatch(s -> s.startsWith("a")); // ergibt false
```