Inhalt der Vorlesung

1. Klassen und Objekte
2. Vererbung
3. Enums, Wrapper und Autoboxing
4. Interfaces
5. Generics
6. Exceptions
7. Polymorphismus
8. Grafische Benutzeroberflächen mit JavaFX
9. Streams und Lambda Expressions
10. Leichtgewichtige Prozesse – Threads

Kapitel 9: Streams und Lambda Expressions

Inhalt

Kapitel 9: Streams und Lambda Expressions

Lernziele

[LZ 9.1] Erklären können, was ein Lambda Ausdruck ist
[LZ 9.2] Funktionale Interfaces definieren und anwenden können
[LZ 9.3] Wichtige Funktionale Interfaces kennen
[LZ 9.4] Lambdas zur Filterung und Verarbeitung von Daten einsetzen können
[LZ 9.5] Erklären können, was ein Stream ist und wie er arbeitet
[LZ 9.6] Wichtige create-, intermediate- und terminal-Operationen kennen und diese anwenden können

In vielen Situationen ist es erforderlich, einer Methode ein „Stück Funktionalität“ als Parameter zu übergeben, beispielsweise als Reaktion auf einen Button-Klick, oder wenn eine Liste nach bestimmten Kriterien sortiert werden soll. Eine in Java häufig genutzte Lösung sind anonyme Klassen, wie im nachfolgenden Beispiel dargestellt:

```
button.setOnAction(new EventHandler() {
    public void handle(Event event) {
        System.out.println("Button-Klick!");//Reaktion auf Button-Klick
    }
});
```

Obiges Code-Fragment definiert eine anonyme (=unbenannte) Klasse, welche das `EventHandler`-Interface mit genau einer Methode „`handle`“ implementiert. Die Klasse wird hierbei gleichzeitig definiert und instanziiert. Im Ergebnis wird der `onAction`-Methode ein Objekt übergeben, auf welchem dann die `handle`-Methode aufgerufen wird, wenn der Button „`button`“ angeklickt wird.
Anonyme Klassen sind für die Übergabe eines relativ kleinen Anweisungsblocks (s. oben) umständlich und schlecht lesbar. Lambda Expressions (kurz Lambdas) bieten für diese und ähnliche Situationen eine Alternative. Mit einer Lambda lässt sich obiger Code wesentlich kompakter so schreiben:

```
button.setOnAction(event -> System.out.println("Button-Klick!"));
```

9. Streams und Lambda Expressions

9.1 Lambda Expressions

Ein Lambda ist vergleichbar mit einer namenlosen Methode ohne Rückgabetyp und Angabe ausgelöster Exceptions. Ein Lambda besteht aus einer Parameterliste gefolgt von „->“ und einem Ausdruck (Expression) bzw. einem Anweisungsblock:

```
<Parameterliste> -> <Ausdruck> | <Block>
```

Parameterliste

Die Parameterliste benötigt Klammern (falls mehr als ein Parameter vorhanden ist oder Datentypen angegeben werden), die Parameter sind durch Kommas zu trennen. Die Datentypen sind optional.

```
(int a, int b)
(a, b)
```

Ausdruck

Wird ein Ausdruck verwendet, so ergänzt der Compiler implizit ein „return“, so dass zur Laufzeit der Ausdruck ausgewertet und als Ergebnis zurückgegeben werden kann.

Beispiel: p sei ein Parameter vom Typ Person

```
p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25
```

Der Ausdruck liefert true für Männer im Alter von 18 bis einschl. 25 Jahren

Block

Anstelle eines Ausdrucks kann ein Block eingesetzt werden, dann aber mit einem „return“:

```
p -> {
      return p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25;
}
```

Lambdas besitzen viele Anwendungsgebiete, von denen einige vorgestellt werden.

Lambdas in GUI-Anwendungen

s. einführendes Beispiel (Button-Klick)

Lambdas zum Sortieren von Listen (Comparator-Interface!)

```
Comparator<Integer> intComparator = (int1, int2) -> int1-int2;
List<Integer> integers = Arrays.asList(1, 2, 13, 4, 15, 6, 17, 8, 19);
Collections.sort(integers, intComparator);
System.out.println(integers);
```

Anstelle eines Comparator-Objekts zum Vergleich von Integer-Objekten wird das Lambda „intComparator“ übergeben, das bewirkt, dass die int-Liste in absteigender Reihenfolge ausgegeben wird.
Alternativ könnte man das Lambda auch schreiben als

```
Comparator<Integer> intComparator2 = (int1, int2) -> {
    if (int1 < int2)
        return 1;
    else if (int1 > int2)
        return -1;
    return 0;
};
```

Funktionale Interfaces (functional interfaces)
Bevor weitere Anwendungsmöglichkeiten vorgestellt werden können, ist der Begriff des *functional interface* zu klären. Ein *functional interface* ist ein Interface mit genau einer Methode, etwa das Interface Comparator<T>, welches die Methode compare fordert.
Mit Java 8 werden Functional Interfaces als eigener Datentyp eingeführt, um Lambda Ausdrücke als Instanzen von Functional Interfaces (=Objekte, die ein Functional Interface implementieren) definieren zu können.
Beispiel für ein Functional Interface:

```
@FunctionalInterface
interface MathOperation { int operation(int a, int b); }
```

Das Interface ist ein Datentyp, dem jeder Lambda-Ausdruck entspricht, der zwei int-Parameter entgegennimmt und ein int-Ergebnis liefert.
Beispiel:

```
MathOperation addition = (int a, int b) -> a + b;
int operate(int a, int b, MathOperation mathOperation) {
     return mathOperation.operation(a, b);
}
```

Das Lambda „passt“ zu `MathOperation`, entsprechend ist folgender Aufruf korrekt:

```
int ergebnis = operate(3, 5, addition);
```

Über den “.”-Operator wird die auszuführende Methode (hier: operation) bezeichnet. “operate“ führt beliebige Operationen auf den Argumenten aus, woraus sich eine große Flexibilität ergibt!

Das Interface Predicate<T>
Ein vordefiniertes Functional Interface ist Predicate<T> für die Prüfung von Bedingungen (=Prädikat) zu einem Objekt vom Typ T:

```
@FunctionalInterface
interface Predicate<T> { boolean test(T objectToBeTested); }
```

Jedes Objekt, welches Predicate<T> implementiert, bietet also eine Methode test, der ein Objekt vom Typ T übergeben wird und die true oder false zurückgibt.
Beispiel:

```
Predicate<Person> checkAlter = p -> p.getAlter() >= 18;
```

`checkAlter` prüft, ob eine gegebene Person volljährig ist.

Das Interface Consumer<T>
Ein Consumer-Objekt bietet eine Methode accept(T objectToBeConsumed), die das übergebene Objekt vom Typ T verarbeitet (z.B. auf die Console ausgibt):

```
@FunctionalInterface
interface Consumer<T> { void accept(T objectToBeConsumed); }
```

Beispiel:

```
Consumer<Person> printPerson = p -> System.out.println(p);
```

`printPerson` gibt das übergebene Person-Objekt auf die Console aus.

Lambdas zum Filtern und Bearbeiten von Listen von Elementen

Gegeben Sie eine Klasse Person, siehe Klassendefinition unten. Eine Personenliste soll nach flexibel festzulegenden Kriterien gefiltert werden, anschließend möchte man eine beliebige Aktion auf jedem Element der Liste ausführen. Im Beispiel wird die Liste nach volljährigen Personen gefiltert und dann ausgegeben.

```
class Person {
    private String name; private int alter;
    public Person(String name, int alter) {
        this.name = name; this.alter = alter; }
    public String toString() {
        return "Person: name = " + name + ", alter = " + alter; }
    // getter aus Platzgründen weggelassen }
```

Die Personenliste sei folgendermaßen definiert:

```
List<Person> personen = Arrays.asList(
    new Person("Anna", 19), new Person("Hugo", 24),
    new Person("Codie", 18), new Person("Susi", 14));
```

Nun kann mit Hilfe zweier Lambdas in der Methode `processPersons` gefiltert und anschließend ausgegeben werden:

```
public void processPersons(List<Person> liste,
        Predicate<Person> tester, Consumer<Person> action) {
    for (Person p : liste)
        if (tester.test(p))
            action.accept(p);
}
```

Die Methode `processPersons` iteriert lediglich über die Personenliste, prüft jede Person mit den Prädikat und leitet diese – falls das Prädikat true liefert – an den Consumer weiter.
Aufruf:

```
Predicate<Person> checkAlter = p -> p.getAlter() >= 18;
Consumer<Person> printPerson = p -> System.out.println(p);
processPersons(personen, checkAlter, printPerson);
```

Der große Vorteil dieser Vorgehensweise liegt in der Flexibilität von processPersons. Über die beiden Parameter kann die Filterung und die Verarbeitung beliebig verändert werden!

Eine weitere Einsatzmöglichkeit für Lambdas ist der Einsatz von Lambdas in Aggregatfunktionen zur Verarbeitung von Streams. Hierfür sei auf die Beispiele unter Abschnitt 9.2 verweisen.

Collections (z.B. eine Personenliste) werden häufig dazu verwendet, Teile der Collection mit bestimmten Eigenschaften herauszufiltern (z.B. alle weiblichen Personen), diese zu verändern bzw. diese als Eingabe weiterverarbeitende Methoden zu verwenden (z.B. schicke eine E-Mail an alle Frauen der Liste).
Zur Optimierung solcher Vorgänge sowie zur Vermeidung der hierfür typischen Iterationen wurde mit Java 8 das Stream-Konzept eingeführt. Ein Stream ist eine Art Fließband, auf dem Operationen auf alle Elemente einer Collection angewendet werden. Es werden drei Typen von Operationen auf Streams unterschieden:

Daten (Array oder Collection) => Stream => $Operation_1$ => … => $Operation_n$ => Ergebnis

Create-Operation | Intermediate-Operation | Terminal-Op.

Beispiel:

```
List<Person> personen = ...;
List<Person> erwachsene = personen.stream()                  // create
                          .filter(p -> p.getAlter() >= 18) // intermediate
                          .collect(Collectors.toList());   // terminal
```

Mit der stream-Operation wird ein Stream-Objekt zu `personen` erzeugt. Die filter-Methode reduziert die Menge der Personen-Objekte auf die Erwachsenen (Alter >= 18!) während die vorgegebene toList-Methode den resultierenden gefilterten Stream wieder in eine Liste umwandelt.

Einige create-Operationen

Umwandlung Array → Stream: Person[] personen = …;
final Stream<Person> Arrays.stream(personen);

Umwandlung Liste → Stream: final Stream<Person> s = personen.stream();

Einige intermediate-Operationen

Stream<T> filter(Predicate<T> p):	liefert einen Stream von T-Objekten, die p erfüllen
Stream<R> map(Function<T, R> f):	liefert einen Stream von R-Objekten, die durch f aus dem Ausgangs-Stream gebildet werden
Stream<T> sorted(Comparator<T> c):	liefert Stream mit Sortierung gemäß c

Einige Terminal-Operationen

void forEach(Consumer<T> action):	Führt action auf jedem Element des Streams aus
Object[] toArray():	Liefert ein Array mit den Objekten des Streams
T reduce(T identity, BinaryOperator<T> op):	Reduktionsoperation, siehe Beispiel Folgeseite
T min(Comparator<T> c):	Liefert das Minimum des Streams gemäß c
T max(Comparator<T> c):	Liefert das Maximum des Streams gemäß c
long count():	liefert die Anzahl Elemente des Streams
T findFirst():	liefert das erste Element des Streams

Beispiel

```
public static void main(String[] args) {
    List<Person> personen = Arrays.asList(
        new Person("Anna", 19), new Person("Hugo", 24),
        new Person("Codie", 18), new Person("Susi", 14));
    int summeAlter = personen.stream()
                        .mapToInt(p -> p.getAlter())
                        .reduce(0, (x, y) -> x + y);
    double durchschnittsAlter = (double) summeAlter / personen.size();
    System.out.printf("Anzahl Personen: %d, Durchschnittsalter: %f\n",
        personen.size(), durchschnittsAlter);
}
```

Die Personenliste wird zunächst in einen *Stream* umgewandelt, die Methode `mapToInt` (s. Java API) wandelt den `Person-Stream` in einen `int`-Stream um, indem von jeder Person das Alter verwendet wird. Dann führt `reduce` eine Summation der Alterswerte durch, beginnend bei 0. Abschließend wird das Durchschnittsalter berechnet und ausgegeben.

Weitere Informationen:

- Stream-API: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- Collector-API: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

online: Oracle Java Tutorial

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://docs.oracle.com/javase/tutorial/collections/streams/

Buch

- M. Inden: Java 8 – Die Neuerungen; dpunkt.verlag; 1. Auflage; 2014