universität innsbruck

Funktionale Programmierung in Java

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Funktionale Programmierung

- Allgemein:
  - Programm wird als Aneinanderreihung von mathematischen Funktionen repräsentiert.
  - Daten sind stets unveränderbar (immutable).
  - Frei von Seiteneffekten.
  - Funktionen können an weitere Funktionen übergeben werden (Code-as-Data).
  - Funktionen können ohne die Erzeugung eines Objekts (Binden) an andere Objekte/Methoden übergeben werden.
- In Java:
  - Funktionen werden durch Funktionsobjekte dargestellt.
  - Wie jedes andere Objekt können Funktionsobjekte:
    - Variablen zugewiesen werden.
    - Als Parameter übergeben werden.
    - Von Methoden zurückgegeben werden.
  - Nicht alle Daten sind unveränderbar.
  - Nicht zwingend frei von Seiteneffekten.

Lambdas - Einführung

- Lambda-Ausdrücke sind das Hauptfeature von Java 8
- Lambda-Ausdrücke (kurz Lambdas) sind ein wesentlicher Teil der funktionalen Programmierung in Java.
- Lambda-Ausdrücke können als Alternative zum Erstellen von Funktionsobjekten durch anonyme Klassen verwendet werden.
- Lambda-Ausdrücke können als anonyme Methoden gesehen werden.

Lambdas – erstes Beispiel

Comparator mit anonymer innerer Klasse:

```
Comparator<Rectangle> areaComparator = new Comparator<Rectangle>() {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return Integer.compare(r1.getArea(), r2.getArea());
    }
};
```

Comparator mit Lambda:

```
Comparator<Rectangle> areaComparator =
     (r1, r2) -> Integer.compare(r1.getArea(), r2.getArea());
```

Lambdas (1)

- Ein Lambda-Ausdruck ist eine unbenannte Methode mit formalen Parametern, dem Operator `->` und einem Methodenrumpf.
- Bei den formalen Parametern wird zwischen explizit und implizit typisierten Parametern unterschieden.
- Beim Methodenrumpf wird zwischen Anweisungsrumpf und Ausdrucksrumpf unterschieden.
- Beispiele:
  - Explizit typisiert & Anweisungsrumpf: `(int x) -> { return x * x; }`
  - Implizit typisiert & Anweisungsrumpf: `(x, y) -> { return x + y; }`
  - Explizit typisiert & Ausdrucksrumpf: `(String s) -> s.toLowerCase()`
  - Implizit typisiert & Ausdrucksrumpf: `x -> x * x`

Lambdas (2)

- Variablen, welche im Lambda-Ausdruck verwendet werden, aber nicht als Parameter spezifiziert sind, werden als freie Variablen bezeichnet.
- Im Methodenrumpf des Lambdas dürfen lokale Variablen nur als freie Variablen verwendet werden, wenn Sie final bzw. effektiv final sind.
  - Eine Variable ist effectively final, wenn der Wert der Variable nach der Initialisierung nicht mehr verändert wird.

Favor Lambdas over Anonymous Classes

```
Comparator<Rectangle> areaComparator = new Comparator<Rectangle>() {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return Integer.compare(r1.getArea(), r2.getArea());
    }
};
```

```
Comparator<Rectangle> areaComparator =
     (r1, r2) -> Integer.compare(r1.getArea(), r2.getArea());
```

- Vorher:
  - Anonyme innere Klasse bläst Code auf
  - Unübersichtlicher Code
- Nachher:
  - Konziser und lesbar Code

Methoden-Referenzen

- Für Lambda-Ausdrücke, welche im Methodenrumpf nur eine Methode aufrufen, wurde in Java 8 eine verkürzte Schreibweise eingeführt.
- Es gibt vier verschiedene Anwendungsmöglichkeiten:
  - Statische Methoden
  - Objektmethoden
  - Objektmethoden eines bestimmten Exemplars
  - Konstruktoren
- Methoden-Referenzen werden mit Hilfe von „`::`“ angegeben

Anwendungsmöglichkeit	**Als Methoden-Referenz**	**Als Lambda**
Statische Methode	`Integer::signum`	`i -> Integer.signum(i)`
Objektmethode	`Object::toString`	`o -> o.toString()`
Objektmethode eines Objekts str	`str::length`	`() -> str.length()`
Konstruktor	`ArrayList::new`	`() -> new ArrayList<>()`

Vertiefung: Interfaces

- Es existiert kein „Wurzelinterface“ von dem alle anderen Interfaces erben.
- Elemente eines Interface-Typs sind:
  - Alle Elemente, welche im Interface selbst definiert wurden.
  - Alle Elemente, welche von direkten Superinterfaces geerbt wurden.
    - Statische innere Klassen
    - Objektmethoden
    - Konstanten
  - Hat ein Interface keine direkten Superinterfaces, deklariert es implizit eine `public abstract` Methode für jede `public` Methode der Klasse `Object`, welche nicht bereits, mit kompatibler `throws` Klausel, explizit im Interface deklariert wurde. Die so deklarierten Methoden haben:
    - Die selbe Signatur der Methode aus `Object`.
    - Den selben Rückgabewert der Methode aus `Object`.
    - Die selbe `throws`-Klausel der Methode aus `Object`.

Functional Interfaces (1)

- Um Lambdas zu ermöglichen, wurden sogenannte „Functional Interfaces“ eingeführt.
- Das sind Interfaces, welche genau eine abstrakte Methode beinhalten (abgesehen von den Methoden, welche auch in `Object` vorkommen).
- Sie werden auch Single Abstract Method Interfaces (SAM) genannt.
- Beispiele: `Callable`, `Comparator`, `Runnable`

→ Lambdas und Functional Interfaces stehen eng in Beziehung zueinander, denn Lambdas können überall dort eingesetzt werden, wo zuvor ein SAM-Interface durch eine anonyme Klasse implementiert wurde.

Functional Interfaces (2)

- Lambdas können stets nur eine Methode definieren.
- Die Annotation `@FunctionalInterface` kann auf Functional Interfaces angewendet werden, um Compiler-Fehlermeldungen zu erhalten, falls mehr als eine abstrakte Methode deklariert wird.
  - Diese Annotation ist optional. Allerdings wird empfohlen diese Annotation zu verwenden.
  - SAM-Typen können auch ohne die Annotation verwendet werden.
- Java bietet in `java.util.function` weit verbreitete und oft verwendete Functional Interfaces an.
- Standard Functional Interfaces verwenden!
- Eigene Functional Interfaces nur dann deklarieren, wenn kein Standard Functional Interface passt!

Standard Functional Interfaces (1)

- Auszug von wichtigen Standard Functional Interfaces:

`BiFunction<T, U, R>`

- Repräsentiert eine Funktion, die zwei Parameter erwartet und ein Ergebnis erzeugt.

`BinaryOperator<T>`

- Repräsentiert eine Operation auf zwei Operanden.
- Spezialisierung von `BiFunction<T, U, R>`.

`Consumer<T>`

- Verarbeitet ein Element.
- Erwartet einen Seiteneffekt.

`Function<T, R>`

- Repräsentiert eine Funktion, die einen Parameter erwartet und ein Ergebnis erzeugt.

`Predicate<T>`

- Repräsentiert ein Prädikat mit einem Parameter.

`Supplier<T>`

- Erzeugt ein Objekt.

`UnaryOperator<T>`

- Repräsentiert eine Operation auf einem Operanden.
- Spezialisierung von `Function<T, R>`.

Standard Functional Interfaces (2)

- Beispiele

Interface	Abstrakte Methode	Beispiel
`Consumer<T>`	`void accept(T t)`	`System.out::print`
`Function<T, R>`	`R apply(T t)`	`Arrays::asList`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Supplier<T>`	`T get()`	`Math::random`
`UnaryOperator<T>`	`T apply(T t)`	`Character::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`Double::sum`

Standard Functional Interfaces (3)

- Im Paket `java.util.function` gibt es spezielle Interfaces für die primitiven Datentypen `int`, `long` und `double`.
  - Für eine bessere Performance (um Boxing und Unboxing zu vermeiden) sind diese Interfaces interessant.
- Der Bezeichner der Functional Interfaces für die primitiven Datentypen verwendet jeweils den Typ als Präfix.
- Beispiel Interfaces für `int`:

Interface	Abstrakte Methode	Beispiel
`IntBinaryOperator`	`int applyAsInt(int l, int r)`	`Integer::sum`
`IntConsumer`	`void accept(int value)`	`System.out::println`
`IntFunction<R>`	`R apply(int value)`	`Integer::toString`
`IntPredicate`	`boolean test(int value)`	`v -> v > 10`
`IntSupplier`	`int getAsInt()`	`() -> Integer.MIN_VALUE`
`IntUnaryOperator`	`int applyAsInt(int operand)`	`Integer::signum`

Streams

- Streams ermöglichen Folgen von Verarbeitungsschritten auf Daten.
- Es besteht eine Ähnlichkeit zu Collections.
  - Streams können allerdings nur einmal traversiert werden.
  - Streams arbeiten mit interner Iteration.
  - Streams speichern keine Daten, sondern stellen nur Zugriff auf Daten bereit.
  - Streams werden nur dann Ausgewertet, wenn das Ergebnis benötigt wird.
- Die Interfaces `Stream`, `IntStream`, `LongStream` und `DoubleStream` sind Streams von Objekten bzw. den primitiven Datentypen `int`, `long` und `double`.
- Bei der Verarbeitung wird zwischen folgenden Typen von Operationen unterschieden:
  - Stream-Erzeugung: Operation, um einen Stream zu erzeugen.
  - Zwischenberechnung (intermediate operations): Operationen, um einen Stream in einen anderen Stream zu überführen.
  - Ergebniserzeugung (terminal operation): Operation, um aus einem Stream ein Endergebnis zu erzeugen.

Stream-Erzeugung

- Für die Erstellung von Streams gibt es eine Vielzahl von Operationen
- Beispiele:
  - Stream aus Array
    - `Arrays.stream(T[] array)`
  - Stream aus dem Collection-Exemplar `myCollection`
    - `myCollection.stream()`
  - Stream für vordefinierte Wertebereiche
    - `Stream.of (T... values)`
    - `IntStream.range(int startInclusive, int endExclusive)`
    - `"Stream for a string".chars()`
  - Generieren von Werten
    - `Stream.generate(Supplier<T> s)`
    - `Stream.iterate(T seed, UnaryOperator<T> f)`

```
Stream<Character> stream1 = Stream.of('a', 'b', 'y', 'z');
IntStream stream2 = IntStream.range(0, 100);
Stream<Integer> stream3 = Stream.iterate(1, x -> x + 2);
```

Zwischenberechnungen (1)

- Diese Verarbeitungsschritte können hintereinander ausgeführt werden.
  - Der Rückgabetyp der Zwischenberechnungen sind neue Streams.
- Überblick über Zwischenoperationen bei Streams:

Kategorie	Methoden
Abbilden	`flatMap, map, mapMulti, mapToInt`
Filtern	`filter, distinct`
Intervall	`dropWhile, limit, skip, takeWhile`
Sortieren	`sorted`

Zwischenberechnungen (2)

- Es wird zwischen zustandslosen und zustandsbehafteten Zwischenberechnungen unterschieden.
  - Bei zustandslosen Schritten ist die Aktion unabhängig von allen anderen Elementen des Streams anwendbar (z.B. filtern).
  - Bei zustandsbehafteten Schritten ist die Aktion abhängig von anderen Elementen des Streams (z.B. sortieren).
- Zwischenberechnungen, welche auch für unendliche Input Streams möglicherweise endliche Output Streams produzieren, werden als short-circuiting intermediate operations bezeichnet.
  - Beispiele: `limit`, `takeWhile`

Zwischenberechnungen - Stream

- Beispiele für zustandslose Zwischenberechnungen von `Stream`

  `filter(Predicate<? super T> predicate)`
  - Entfernt alle Elemente, die `predicate` nicht erfüllen.

  `map(Function<? super T, ? extends R> mapper)`
  - Wendet die Funktion auf jedem Element des Streams an.

  `flatMap(Function<? super T, ? extends Stream<? extends R>> m)`
  - Wendet die Funktion auf jedem Element des Streams an und bildet verschachtelte Streams auf einen flachen Stream ab.

- Beispiele für zustandsbehaftete Zwischenberechnungen von `Stream`

  `sorted()`
  - Sortierung der Elemente basierend auf der natürlichen Ordnung.

  `distinct()`
  - Entfernt alle Duplikate gemäß der `equals`-Methode.

  `limit(long maxSize)`
  - Begrenzt die maximale Anzahl der Elemente im Stream.

  `skip(long n)`
  - Überspringt die ersten `n`-Elemente des Streams.

Ergebniserzeugung

- Erst durch diese Operation werden die spezifizierten Operationen auch wirklich ausgeführt.
- Überblick über Terminal-Operationen bei Streams:

Kategorie	Methoden
Boolsche Quantoren	anyMatch, allMatch, noneMatch
Iterieren	forEach, forEachOrdered
Reduzieren	count, findAny, findFirst, max, min, reduce
Sammeln	collect, toArray

- Bei Short-Circuiting Terminal-Operationen müssen für die Berechnung des Ergebnisses nicht immer alle Elemente betrachtet werden.

Ergebniserzeugung - Stream (1)

- Beispiele für Ergebniserzeugung von `Stream` mit Short-Circuit-Evaluierung:

  `findAny()`
  - Liefert ein Element des Streams, sofern der Stream nicht leer ist.

  `findFirst()`
  - Liefert das erste Element des Streams, sofern der Stream nicht leer ist.

  `anyMatch(Predicate<? super T> predicate)`
  - Prüft, ob mindestens ein Element das Prädikat erfüllt.

  `allMatch(Predicate<? super T> predicate)`
  - Prüft, ob alle Elemente das Prädikat erfüllen.

Ergebniserzeugung - Stream (2)

- Beispiele für Ergebniserzeugung von `Stream`:

  `forEach(Consumer<? super T> action)`
  - Führt action für jedes Element des Streams aus.

  `collect(Collector<? super T, A, R> collector)`
  - Sammeln von Elementen (beispielsweise übertragen in eine Collection).

  `reduce(BinaryOperator<T> accumulator)`
  - Akkumulierung aller Elemente zu einem Ergebniswert.

  `count()`
  - Ermittelt die Anzahl der Elemente im Stream.

  `toList()`
  - Sammeln der Elemente in einer nicht modifizierbaren Liste.

  `min(Comparator<? super T> comparator)`
  - Liefert das kleinste Element des Streams basierend auf dem Komparator.

  `max(Comparator<? super T> comparator)`
  - Liefert das größte Element des Streams basierend auf dem Komparator.

Collectors-Klasse

- Bietet viele statische Methoden zum Erzeugen von Objekten, die das Interface `Collector` implementieren, an.
- Diese Objekte können mit der Methode `collect` eingesetzt werden.
- Beispiele einiger statischer Methoden:

  `groupingBy(Function<? super T,? extends K> classifier)`
  - Gruppieren der Elemente anhand von `classifier`.

  `joining(CharSequence delimiter)`
  - Verknüpfung der Elemente zu einem String, wobei `delimiter` zwischen den einzelnen Elementen eingesetzt wird.

  `partitioningBy(Predicate<? super T> predicate)`
  - Partitionieren der Elemente anhand von `predicate`.

  `toList()`, `toSet()`, …
  - Sammeln der Elemente in einer Collection.

Stream - Beispiel

```
List<Integer> values = Arrays.asList(5, 1, 4, 5, 2, 1);
long count1 =
        values.stream()
                .filter(v -> v == 1)
                .count();

List<Integer> larger3 =
        values.stream()
                .filter(v -> v > 3)
                .collect(Collectors.toList());

List<Integer> distinctSorted =
        values.stream()
                .distinct()
                .sorted()
                .collect(Collectors.toList());

System.out.println(count1);
System.out.println(larger3);
System.out.println(distinctSorted);
```

Ausgabe:

```
2
[5, 4, 5]
[1, 2, 4, 5]
```

Stream – Beispiel Produkt (1)

```
public class Product {
    private final int productID;
    private final BigDecimal price;
    private final String name;

    public Product(int productID, BigDecimal price, String name) {
        this.productID = productID;
        this.price = price;
        this.name = name;
    }

    public int getProductID() {
        return productID;
    }

    public BigDecimal getPrice() {
        return price;
    }

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

src/at/ac/uibk/pm/lambdas/streams/Product.java

Stream – Beispiel Produkt (2)

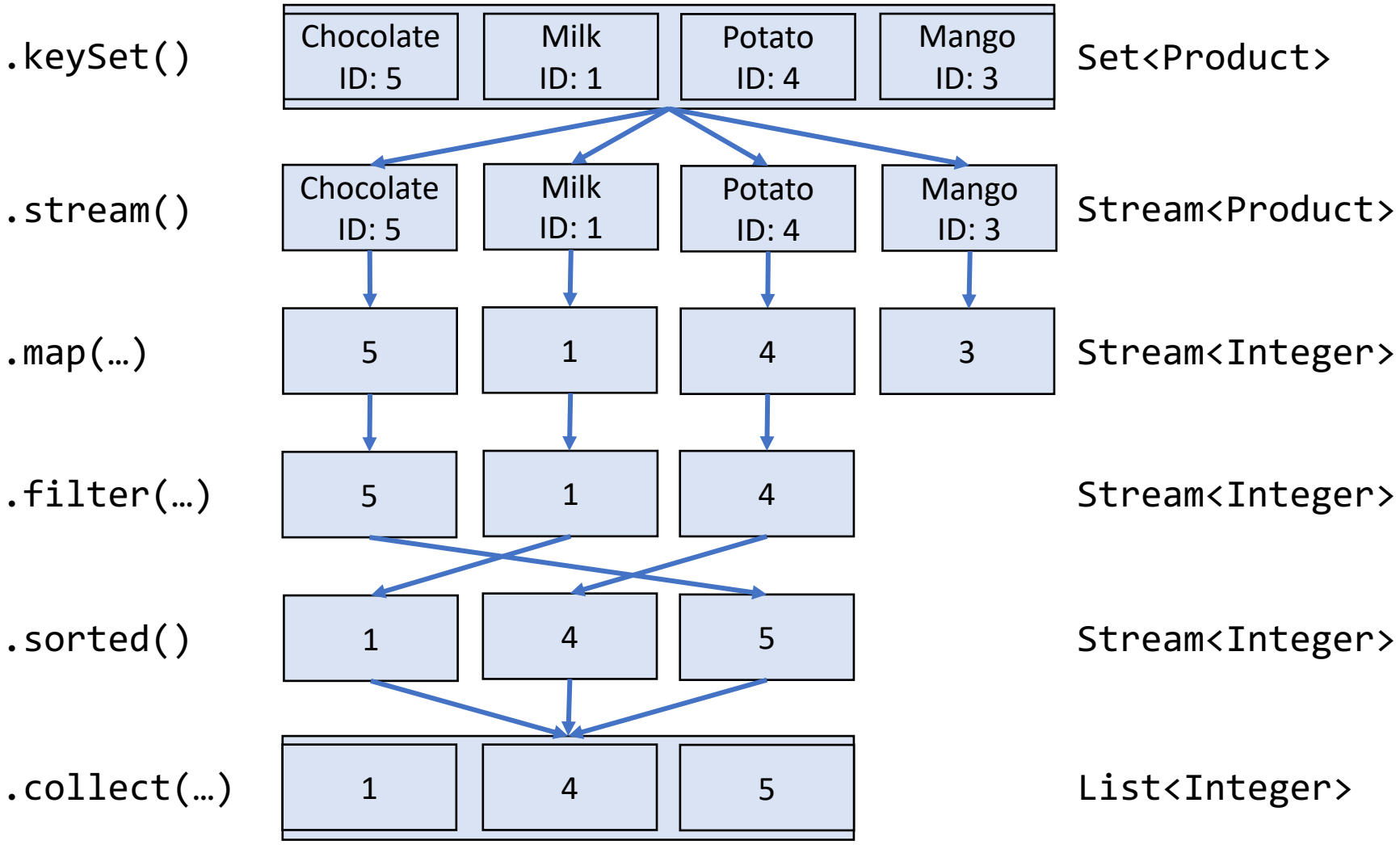
```
Map<Product, Integer> shoppingList = new LinkedHashMap<>();
shoppingList.put(new Product(5, new BigDecimal("1"), "Chocolate"), 2);
shoppingList.put(new Product(1, new BigDecimal("1.2"), "Milk"), 2);
shoppingList.put(new Product(4, new BigDecimal("2.5"), "Potato"), 1);
shoppingList.put(new Product(3, new BigDecimal("1.1"), "Mango"), 3);

Set<Integer> unavailableProducts = Set.of(3, 8, 9);

List<Integer> shoppingBasket =
        shoppingList
                .keySet()
                .stream()
                .map(Product::getProductID)
                .filter(productID -> !unavailableProducts.contains(productID))
                .sorted()
                .collect(Collectors.toList());
System.out.printf("Product ids in shopping basket:%n%s%n", shoppingBasket);
```

Ausgabe:

```
Product ids in shopping basket:
[1, 4, 5]
```

src/at/ac/uibk/pm/lambdas/streams/ProductApplication.java

Stream – Beispiel Produkt (3)



Exkurs: Lokale Type-Inference (1)

- Typen für lokale Deklarationen können über Type Inference hergeleitet werden.

```
var numbers = List.of(1, 2, 3, 4, 5);

for (var number : numbers) {
    System.out.println(number);
}
```

- Vorteil:
  - Kürzerer, kompakterer Code.
  - Typsicherheit bleibt durch Inference erhalten.
- Nachteil:
  - Explizite Typinformation geht verloren (Lesbarkeit).
  - Es wird immer der exakte Typ verwendet und nicht ein Basistyp.

Exkurs: Lokale Type-Inference (2)

- Einschränkungen:
  - Kann nicht bei einer Deklaration ohne Initialisierung verwendet werden.
  - Kann nicht bei Arrays verwendet werden, wenn eine Initialisierungliste verwendet wird.

```
var numbers; // Compile-Error!
var array = {1, 2, 3, 4, 5}; // Compile-Error!
```

- Bei der Verwendung von `var` und dem Diamant-Operator wird für den Typparameter `Object` inferiert!

```
List<Integer> numbers1 = new ArrayList<>(); // Type = List<Integer>
var numbers2 = new ArrayList<>(); // Type = ArrayList<Object>
```

Interface Comparator (1)

- Mit Java 8 wurde das Interface `Comparator` mit hilfreichen default und statischen Methoden erweitert, wodurch ein Komparator kompakt implementiert werden kann.
- Auszug einiger statischer Methoden:

  `naturalOrder()`
  - Gibt einen Komparator basierend auf der natürlichen Ordnung zurück.

  `reverseOrder()`
  - Gibt einen Komparator basierend auf der umgekehrten natürlichen Ordnung zurück.

  `nullsFirst()`
  - Gibt einen Komparator zurück, der `null` als das kleinste Element betrachtet.

  `comparing(Function<? super T, ? extends U> keyExtractor)`
  - Definiert einen Komparator basierend auf Elementen vom Typ `U`, welche von `T` durch `keyExtractor` extrahiert wurden.
  - Falls Elemente vom Typ `U` nicht `Comparable` implementieren, muss zusätzlich ein Komparator mitgegeben werden.

Interface Comparator (2)

- Auszug einiger default Methoden:

  `reversed()`

  - Gibt einen Komparator zurück, der Elemente in umgekehrter Reihenfolge sortiert.

  `thenComparing(Function<? super T, ? extends U> keyExtractor)`

  - Lexikografische Verkettung von einem zusätzlichen Vergleichskriterium.

- Um Autoboxing zu umgehen gibt es zusätzliche Methoden für die primitiven Datentypen `int`, `long` und `double`.
  - `comparingInt`
  - `thenComparingInt`
  - …

Beispiel Comparator

```
public class Rectangle {
    …
    @Override
    public int compareTo(Rectangle o) {
        int result = Integer.compare(width, o.width);
        if (result != 0) {
            return result;
        }
        return Integer.compare(length, o.length);
    }
    …
}
```

```
public class Rectangle {
    …
    private static final Comparator<Rectangle> COMPARATOR =
            Comparator.<Rectangle>comparingInt(r -> r.width)
                    .thenComparingInt(r -> r.length);
    …
    @Override
    public int compareTo(Rectangle o) {
        return COMPARATOR.compare(this, o);
    }
    …
}
```

src/at/ac/uibk/pm/lambdas/Rectangle.java

Optional-Klasse (1)

- Bei Berechnungen kann mitunter auch in einem Normalfall kein Ergebnis ermittelt werden.
  - Beispielsweise wenn ein Element in einer bestimmten Menge nicht gefunden werden kann.
- Für die Modellierung von optionalen oder nicht vorhanden Werten kann `null` verwendet werden.
  - Wird dieser Rückgabewert nicht entsprechend behandelt, kann es zu einer `NullPointerException` kommen.
- Die Klasse `Optional` wurde in Java 8 als neue Klasse eingeführt, um NullPointerExceptions vorzubeugen und optionale Werte ausdrücklich kommunizieren zu können.
- Entweder ist ein Wert, welcher nicht `null` ist, im Container oder der Container ist leer.
- Angelehnt an funktionale Programmiersprachen.

Optional-Klasse (2)

- Statische Methoden:

  `Optional.of(T value)`
  - Erzeugt ein `Optional`-Exemplar mit einem gegeben Wert, welcher nicht `null` ist.

  `Optional.ofNullable(T value)`
  - Erzeugt ein `Optional`-Exemplar mit einem gegeben Wert.
  - Falls `value` gleich `null` ist, wird ein leeres `Optional`-Exemplar zurückgegeben.

  `Optional.empty()`
  - Erzeugt ein leeres `Optional`-Exemplar.

Optional-Klasse (3)

- Auszug Objektmethoden:

`isPresent()`

- Überprüft ob ein Wert vorhanden ist.

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

- Führt die eine Aktion aus, wenn ein Wert vorhanden ist.

```
ifPresentOrElse(Consumer<? super T> action,
                Runnable emptyAction)
```

- Führt die eine Aktion aus, wenn ein Wert vorhanden ist.
- Anderenfalls wird `emptyAction` ausgeführt.

`get()`

- Gibt das Element des Containers zurück.
- Wirft eine `NoSuchElementException`, wenn der Container leer ist.

`orElse(T other)`

- Gibt das Element des Containers zurück, wenn er nicht leer ist.
- Anderenfalls wird `other` zurückgegeben.

`orElseGet(Supplier<? extends T> supplier)`

- Gibt das Element des Containers zurück, wenn er nicht leer ist.
- Anderenfalls wird das Auswertungsergebnis von `supplier` zurückgegeben.

Beispiel ohne Optional

```
public static Map.Entry<Product, Integer> findBestseller(
        Map<Product, Integer> shoppingList) {
    Map.Entry<Product, Integer> max = null;
    for (var entry : shoppingList.entrySet()) {
        if (max == null || entry.getValue() > max.getValue()) {
            max = entry;
        }
    }
    return max;
}
```

```
Map<Product, Integer> emptyShoppingList = Map.of();
var bestseller = findBestseller(emptyShoppingList);
System.out.println(bestseller.getKey()); // Runtime-Error!
```

```
Map<Product, Integer> emptyShoppingList = Map.of();
var bestseller = findBestseller(emptyShoppingList);
if (bestseller != null) {
    System.out.printf("The bestseller is: %s!%n", bestseller.getKey());
} else {
    System.out.println("No bestseller found!");
}
```

Beispiel mit Optional

```
public static Optional<Map.Entry<Product, Integer>> findBestseller(
        Map<Product, Integer> shoppingList) {
    Map.Entry<Product, Integer> max = null;
    for (var entry : shoppingList.entrySet()) {
        if (max == null || entry.getValue() > max.getValue()) {
            max = entry;
        }
    }
    return Optional.ofNullable(max);
}
```

```
Map<Product, Integer> emptyShoppingList = Map.of();
var bestseller = findBestseller(emptyShoppingList);
bestseller.ifPresentOrElse(
        e -> System.out.printf("The bestseller is: %s!%n", e.getKey()),
        () -> System.out.println("No bestseller found!"));
```

Beispiel mit Optional und Stream

```
public static Optional<Map.Entry<Product, Integer>> findBestseller(
        Map<Product, Integer> shoppingList) {
    return shoppingList.entrySet()
            .stream()
            .max(Map.Entry.comparingByValue());
}
```

```
Map<Product, Integer> emptyShoppingList = Map.of();
var bestseller = findBestseller(emptyShoppingList);
bestseller.ifPresentOrElse(
        e -> System.out.printf("The bestseller is: %s!%n", e.getKey()),
        () -> System.out.println("No bestseller found!"));
```

Quellen

- Herbert Prähofer: **Funktionale Programmierung in Java: Eine umfassende Einführung**, dpunkt.verlag, 2020
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (***Java SE 17 Edition***)**, Oracle, 2021