Tutorium zur Vorlesung Programmieren

Sondertutorium

Joshua Gleitze

9. Januar 2017

IPD Koziolek

Generische Typen

Generische Typen

Klassen können Typvariablen deklarieren.

```
1  public Interface List<E> {
2      boolean add(E e);
3      E remove (int index);
4      ...
5  }
```

```
List<String> names = new ArrayList<String>();
```

Generische Typen

Methoden können Typvariablen deklarieren.

java.util.Collections (Auszug)

```
1  static <T> void fill(List<? super T> list, T obj)
```

Diamant-Operator

Kurzzschreibweise, falls die Typparameter aus dem Kontext klar sind.

```
Map<Person, Set<Person>> siblings = new HashMap<Person, Set<Person>>();
```

```
Map<Person, Set<Person>> siblings = new HashMap<>();
```

Typeinschränkungen

Bei Deklaration einer Typvariablen gibt

`extends X`
an, dass die Typvariable nur mit Typen, die `X` sind oder von `X` erben, belegt werden kann.

`super X`
an, dass die Typvariable nur mit Typen, die `X` sind oder von denen `X` erbt, belegt werden kann.

Wildcard

Bei der Parameterisierung kann statt eines Typs oder einer Typvariablen auch der Wildcard-Operator `?` verwendet werden.

Bedeutung: „Ein beliebiger, aber nicht bekannter Typ“.

java.util.Collections (Auszug)

```
static void reverse(List<?> list);
```

Oft wird ein Wildcard weiter eingeschränkt.

Herleitung: Typ von Arrays#parallelSort

1. Versuch:

java.util.Arrays

```
1  public static void sort(Object[] arr) {
2      ...
3  }
```

Herleitung: Typ von Arrays#parallelSort

1. Versuch:

java.util.Arrays

```
1  public static void sort(Object[] arr) {
2      ...
3  }
```

Die Elemente können nicht verglichen werden!

Herleitung: Typ von Arrays#parallelSort

2. Versuch:

```
1  public static void sort(Comparable[] arr) {
2      ...
3  }
```

Herleitung: Typ von `Arrays#parallelSort`

2. Versuch:

```
1  public static void sort(Comparable[] arr) {
2      ...
3  }
```

Wie sieht `Comparable` aus?

Herleitung: Typ von `Arrays#parallelSort`

2. Versuch:

```
public static void sort(Comparable[] arr) {
    ...
}
```

Wie sieht `Comparable` aus?

```
public interface Comparable {
    int compareTo(Object other);
}
```

Herleitung: Typ von Arrays#parallelSort

2. Versuch:

```
public static void sort(Comparable[] arr) {
    ...
}
```

Wie sieht `Comparable` aus?

```
public interface Comparable {
    int compareTo(Object other);
}
```

```
public class Weapon implements Comparable {
    public int compareTo(Object other) {
        return this.getPrice() - ???;
    }
}
```

Herleitung: Typ von `Arrays#parallelSort`

2. Versuch:

```
public static void sort(Comparable[] arr) {
    ...
}
```

So sieht `Comparable` aus:

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

Herleitung: Typ von Arrays#parallelSort

2. Versuch:

```
public static void sort(Comparable[] arr) {
    ...
}
```

So sieht `Comparable` aus:

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

```
public class Weapon implements Comparable<Weapon> {
    public int compareTo(Weapon other) {
        return this.getPrice() - other.getPrice();
    }
}
```

Herleitung: Typ von Arrays#parallelSort

3. Versuch:

```
1  public static void sort(Comparable<?>[] arr) {
2      ...
3  }
```

Herleitung: Typ von Arrays#parallelSort

3. Versuch:

```
1  public static void sort(Comparable<?>[] arr) {
2      ...
3      if (arr[i].compareTo(arr[i + 1] < 0) {
4      ...
5  }
```

Herleitung: Typ von Arrays#parallelSort

3. Versuch:

```
1  public static void sort(Comparable<?>[] arr) {
2      ...
3      if (arr[i].compareTo(arr[i + 1] < 0) {
4      ...
5  }
```

```
error: incompatible types: Comparable<CAP#1> cannot be converted to CAP#2
        arr[i].compareTo(arr[i + 1]);
                         ^
  where CAP#1,CAP#2 are fresh type-variables:
    CAP#1 extends Object from capture of ?
    CAP#2 extends Object from capture of ?
```

Herleitung: Typ von Arrays#parallelSort

4. Versuch:

```
1  public static <T extends Comparable<T>> void sort(T[] arr) {
2      ...
3  }
```

gut. besser:

```
1  public static <T extends Comparable<? super T>> void sort(T[] arr) {
2      ...
3  }
```

Herleitung: Typ von `Arrays#parallelSort`

Aus Gründen der Abwärtskompatibilität hat nur `Arrays#parallelSort` diese Signatur:

java.util.Arrays (Auszug)

```
static <T extends Comparable<? super T>> void parallelSort(T[] a);

static void sort(Object[] a);
```

Typelöschung

- Java-Laufzeitumgebung hat keine generischen Typen im Typsystem
- Java-Compiler löscht alle generisch Typinformationen bei der Übersetzung
- Jede Typvariable wird zu ihrem allgemeinsten Typen *(Erasure Type)*
- Wenn nötig werden Casts Zwischenmethoden *(bridge methods)* eingefügt

Beispiel: Typlöschung

Mit generischen Typen

java.util.List (OpenJDK 8)

```
111 public interface List<E> extends Collection<E> {

152     Iterator<E> iterator();

238     boolean add(E e);

261     E remove(int index);
```

Beispiel: Typlöschung

Nach Typlöschung

java.util.List (OpenJDK 8)

```
111 public interface List extends Collection {

152     Iterator<Object> iterator();

238     boolean add(Object e);

261     Object remove(int index);
```

Typlöschung: Konsequenzen

- Kein Zugriff auf Typvariablen möglich
- Es können keine Felder einer Typvariablen erstellt werden
- Casts zu Typvariablen sind unsicher

Beispiel: Kein Zugriff auf Typvariablen

```
1  public class Test<T> {
2      public void foo() {
3          final Class<?> tclass = T.class;
4      }
5  }
```

```
error: cannot select from a type variable
        final Class<?> tclass = T.class;
                                ^
```

Beispiel: Keine Felder generischer Typen

```java
public class Test<T> {
    public void foo() {
        final T[] tArray = new T[5];
    }
}
```

```
error: generic array creation
        final T[] tArray = new T[5];
                           ^
```

Beispiel: Unsichere Casts

```
1  public class MyArrayList<T> {
2      private Object[] array;
3      private int firstFree;
4
5      public void add(T element) {
6          this.array[firstFree] = element;
7      }
8
9      public T getLast() {
10         return (T) this.array[firstFree - 1];
11     }
12 }
```

```
Note: /pfad/MyArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Beispiel: Unsichere Casts

```
1  public class MyArrayList<T> {
2      private Object[] array = new Object[10];
3      private int firstFree;
4
5      public void add(T element) {
6          this.array[firstFree++] = element;
7      }
8
9      @SuppressWarnings("unchecked")
10     public T getLast() {
11         return (T) this.array[firstFree - 1];
12     }
13 }
```

Beispiel: Unsichere Casts

```
1  public class Main {
2      public static void main(String... args) {
3          MyArrayList<Integer> list = new MyArrayList<>();
4          list.add(4);
5          @SuppressWarnings("unchecked")
6          MyArrayList<Double> doubleList = (MyArrayList) list;
7          Double four = doubleList.getLast();
8      }
9  }
```

kompiliert, wirft aber `ClassCastException` in Zeile 7.

Vergleichen

Comparator vs Comparable

`java.lang.Comparable<T>`	`java.util.Comparator<T>`
Eine Klasse, deren Instanzen eine natürliche Reihenfolge haben	Definiert, wie zwei Elemente zu vergleichen sind

Comparable und Comparator: Funktion

`a.compareTo(b) < 0`	„a < b“
`comparator.compare(a, b) < 0`	
`a.compareTo(b) == 0`	„a = b“
`comparator.compare(a, b) == 0`	
`a.compareTo(b) > 0`	„a > b“
`comparator.compare(a, b) > 0`	

Comparable und Comparator: Einsatzzwecke

- `java.util.Arrays#binarySearch`
- `java.util.Arrays#sort`, `java.util.Arrays#parallelSort`
- `java.util.Collections#sort`
- `java.util.PriorityQueue`, `java.util.SortedSet`,
- `java.util.TreeMap`

Anonyme Klassen, Lambdas

Beispiel: Comparator

Inventory.java

```java
public class Inventory {
    private Weapon[] weapons;

    public Weapon[] getWeaponsSortedByDamage() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, new WeaponByDamageComparator());
        return sortedWeapons;
    }

    public Weapon[] getWeaponsSortedByPrice() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, new WeaponByPriceComparator());
        return sortedWeapons;
    }
}
```

Beispiel: Comparator

WeaponByDamageComparator.java

```java
public class WeaponByDamageComparator implements Comparator<Weapon> {
    @Override
    public int compare(Weapon left, Weapon right) {
        return left.getDamage() - right.getDamage();
    }
}
```

Beispiel: Comparator

WeaponByPriceComparator.java

```java
public class WeaponByPriceComparator implements Comparator<Weapon> {
    @Override
    public int compare(Weapon left, Weapon right) {
        return left.getPrice() - right.getPrice();
    }
}
```

Beispiel: Comparator mit inneren Klassen

Inventory.java

```java
public class Inventory {
    private Weapon[] weapons;

    public Weapon[] getWeaponsSortedByDamage() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, new WeaponByDamageComparator());
        return sortedWeapons;
    }

    public Weapon[] getWeaponsSortedByPrice() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, new WeaponByPriceComparator());
        return sortedWeapons;
    }

    private static class WeaponByPriceComparator implements Comparator<Weapon> {
        @Override
        public int compare(Weapon left, Weapon right) {
            return left.getPrice() - right.getPrice();
        }
    }

    private static class WeaponByDamageComparator implements Comparator<Weapon> {
        @Override
        public int compare(Weapon left, Weapon right) {
            return left.getDamage() - right.getDamage();
        }
    }
}
```

Anonyme Klassen

Kurzzschreibweise, falls man eine Instanz einer Klasse nur an einer Stelle braucht.

Syntax:

```
new Typ(Konstruktorargumente) {
    // Klassendefinition
}
```

`Typ` kann auch in Interface sein!

Beispiel: Comparator mit anonymen Klassen

Inventory.java

```java
public class Inventory {
    private Weapon[] weapons;

    public Weapon[] getWeaponsSortedByDamage() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, new Comparator<Weapon>() {
                @Override
                public int compare(Weapon left, Weapon right) {
                    return left.getDamage() - right.getDamage();
                }
            });
        return sortedWeapons;
    }

    public Weapon[] getWeaponsSortedByPrice() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, new Comparator<Weapon>() {
                @Override
                public int compare(Weapon left, Weapon right) {
                    return left.getPrice() - right.getPrice();
                }
            });
        return sortedWeapons;
    }
}
```

Funktionales Interface

Interface, das nur eine abstrakte Methode definiert

Zum Beispiel `Comparator`, `Runnable`, …

Für die gängisten Fälle stehen generische funktionale Interfaces in `java.util.function` bereit.

Lambdas

Kurzzschreibweise für anonyme Klassen, die ein funktionales Interface implementieren.

Syntax:

```
(Typ1 Parameter1, Typ2 Parameter2) -> Ausdruck
```

oder

```
(Typ1 Parameter1, Typ2 Parameter2) -> Anweisung
```

oder

```
(Typ1 Parameter1, Typ2 Parameter2) -> {
    return Ausdruck;
}
```

Typinferenz

Lambdas sind immer Objekte mit einem Typ!

Meistens kann das Typsystem aber den Typ des Lambdas und damit den Typ der Parameter selbst ableiten.

Die Typen der Paramter können dann weggelassen werden:

```
(parameter1, Parameter2) -> Ausdruck
```

oder

```
parameter -> Ausdruck
```

Beispiel: Comparator mit Lambdas

Inventory.java

```java
public class Inventory {
    private Weapon[] weapons;

    public Weapon[] getWeaponsSortedByDamage() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons,
            (left, right) -> left.getDamage() - right.getDamage());
        return sortedWeapons;
    }

    public Weapon[] getWeaponsSortedByPrice() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons,
            (left, right) -> left.getPrice() - right.getPrice());
        return sortedWeapons;
    }
}
```

Methodenreferenzen

Gibt es eine Methode, die ein funktionales Interface erfüllt, kann diese mit `::` referenziert werden.

Für Methoden von Instanzen:

```
public Interface Supplier<T> {
    T get();
}
```

```
Sword mySword = new Claymore();
Supplier<Integer> sup = mySword::getPrice;
```

$\hat{=}$

```
Sword mySword = new Claymore();
Supplier<Integer> sup = () -> mySword.getPrice();
```

Methodenreferenzen

Gibt es eine Methode, die ein funktionales Interface erfüllt, kann diese mit `::` referenziert werden.

Für Konstruktoren:

```
public Interface Supplier<T> {
    T get();
}
```

```
Supplier<Claymore> sup = Claymore::new;
```

$\hat{=}$

```
Supplier<Claymore> sup = () -> new Claymore();
```

Methodenreferenzen

Gibt es eine Methode, die ein funktionales Interface erfüllt, kann diese mit `::` referenziert werden.

Für Intanzmethoden mit Instanz als Argument:

java.util.function.Function (Auszug)

```
public interface Function<T, R> {
    R apply(T);
}
```

```
Function<Weapon, Integer> func = Weapon::getPrice;
```

≙

```
Function<Weapon, Integer> func =  weapon -> weapon.getPrice();
```

Beispiel: Comparator mit Methodenreferenzen

Inventory.java

```java
public class Inventory {
    private Weapon[] weapons;

    public Weapon[] getWeaponsSortedByDamage() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, Comparator.comparingInt(Weapon::getDamage));
        return sortedWeapons;
    }

    public Weapon[] getWeaponsSortedByPrice() {
        Weapon[] sortedWeapons = weapons.clone();
        Arrays.sort(sortedWeapons, Comparator.comparingInt(Weapon::getPrice));
        return sortedWeapons;
    }
}
```

Java-API

Optional

Dass Referenzen `null` sein können ist manchmal ärgerlich und führt zu Programmierfehlern.

```
1 String version = computer.getSoundcard().getUSB().getVersion();
2 version = version == null ? "" : version;
```

Optional

Dass Referenzen `null` sein können ist manchmal ärgerlich und führt zu Programmierfehlern.

```
String version = computer.getSoundcard().getUSB().getVersion();
version = version == null ? "" : version;
```

```
String version = "";
if(computer != null) {
    Soundcard soundcard = computer.getSoundcard();
    if(soundcard != null) {
        USB usb = soundcard.getUSB();
        if(usb != null) {
            version = usb.getVersion();
        }
    }
}
```

Optional

Dass Referenzen `null` sein können ist manchmal ärgerlich und führt zu Programmierfehlern.

```
1 String version = computer.getSoundcard().getUSB().getVersion();
2 version = version == null ? "" : version;
```

Ab jetzt alles auf `null` prüfen?

Optional

Dass Referenzen `null` sein können ist manchmal ärgerlich und führt zu Programmierfehlern.

```
1 String version = computer.getSoundcard().getUSB().getVersion();
2 version = version == null ? "" : version;
```

> *„I call it my billion-dollar mistake. It was the invention of the null reference in 1965. I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.“*
>
> – Tony Hoare

Optional

Dass Referenzen `null` sein können ist manchmal ärgerlich und führt zu Programmierfehlern.

```
1 String version = computer.getSoundcard().getUSB().getVersion();
2 version = version == null ? "" : version;
```

`Optional` to the rescue!

Optional

Wenn eine Referenz vom Typ `X` nicht vorhanden sein kann, bekommt sie den Typ `Optional<X>`.

```
1  public class Computer {
2      private Optional<Soundcard> soundcard;
3      public Optional<Soundcard> getSoundcard() { ... }
4      ...
5  }
6  public class Soundcard {
7      private Optional<USB> usb;
8      public Optional<USB> getUSB() { ... }
9  }
10 public class USB {
11     public String getVersion(){ ... }
12 }
```

Optional

Wenn eine Referenz vom Typ `X` nicht vorhanden sein kann, bekommt sie den Typ `Optional<X>`.

```
1  public class Computer {
2      private Optional<Soundcard> soundcard;
3      public Optional<Soundcard> getSoundcard() { ... }
4      ...
5  }
6  public class Soundcard {
7      private Optional<USB> usb;
8      public Optional<USB> getUSB() { ... }
9  }
10 public class USB {
11     public String getVersion(){ ... }
12 }
```

Der Programmierer wird vom Typsystem gezwungen, den Fall, dass kein Wert vorhanden ist, zu prüfen.

Optional erstellen

- `Optional.empty()` erstellt ein leeres Optional
- `Optional.of(x)` erstellt ein vorhandenes Optional. `x` darf nicht `null` sein.
- `Optional.ofNullable(x)` erstellt ein Optional, das vorhanden ist, falls `x != null`

Optional verwenden

- `Optional#get`: Wert oder `NoSuchElementException` bekommen
- `Optional#isPresent`: `true`, falls ein Wert vorhanden ist
- `Optional<T>#orElse(T other)`: bekomme Wert, oder `other` falls nicht vorhanden
- `Optional<T>#orElseGet(Supplier<? extends T> other)`: bekomme Wert, oder berechne Wert aus `other` falls nicht vorhanden

Optional: jetzt wird's praktisch

- `Optional<T>#map(Function<? super T,? extends U> mapper)` Bilde das Optional auf ein anderes Optional ab. Das andere Optional ist leer, falls dieses Optional leer ist, ansonsten enthält es `mapper` angewendet auf den Wert dieses Optionals.
- `Optional<T>#flatMap(Function<? super T,Optional<U>> mapper)` Wie oben. Allerdings gibt `mapper` direkt das neue Optional zurück.

Schöner mit Optional

```
String version = computer.flatMap(Computer::getSoundcard)
    .flatMap(Soundcard::getUSB)
    .map(USB::getVersion)
    .orElse("");
```

java.nio.file

`java.io.File` ist tot, lang lebe `java.nio.file.Path`!

java.nio.file

Verwende grundsätzlich `Path` statt `File`. Konvertiere wenn nötig mit `File#toPath` und `Path#toFile`.

Nützliche Klassen in `java.nio.file`

- `java.nio.file.Files`: statische Methode für alle gängigen Dateisystem-Operationen
- `java.nio.file.Paths#get`: Erstellung von Pfaden, z.B.: `Paths.get("path", "to", "my", "file")`

`java.util.stream.Stream`

Datenströme

Collections zu Stream: `Collection#stream`

Streams: Zwischenoperationen

- `#map`: Vorbeifließendes Element auf ein anderes abbilden
- `#flatMap`: Vorbeifließendes Element auf mehrere Elemente abbilden (die alle in den Hauptstrom kommen)
- `#filter`: Filter
- `#peek`: `Consumer` für vorbeifließende Elemente

Streams: Lazy Evaluation

Es werden nur so lange Elemente durch ein `Stream` geschleust, wie nötig.

Zwischenoperationen werden erst angewandt, wenn eine terminale Operation ausgeführt wird.

Beispiel: Lazy Evaluation bei Streams

```
1  Stream.of("a3", "a6", "a2", "a1")
2      .map(s -> s.substring(1))
3      .mapToInt(Integer::parseInt)
4      .peek(System.out::println)
5      .filter(number -> number > 3)
6      .findAny()
7      .ifPresent(number -> System.out.println("found: " + number));
```

Beispiel: Lazy Evaluation bei Streams

```
1  Stream.of("a3", "a6", "a2", "a1")
2      .map(s -> s.substring(1))
3      .mapToInt(Integer::parseInt)
4      .peek(System.out::println)
5      .filter(number -> number > 3)
6      .findAny()
7      .ifPresent(number -> System.out.println("found: " + number));
```

```
3
6
found: 6
```

Streams: terminale Operationen

- `#count`
- `#allMatch`
- `#anyMatch`
- `#findAny`
- `#reduce`
- `#collect` (siehe `java.util.stream.Collectors`, um Streams in gängigen Datenstrukturen zu sammeln)

Streams: Sonderklassen für primitive Typen

Primitive Typen können nicht als Instanzen von Typvariablen verwendet werden. Aber: primitive Typen sind schneller.

„Lösung“:

- `java.util.stream.DoubleStream`
- `java.util.stream.IntStream`
- `java.util.stream.LongStream`

Streams: Sonderklassen für primitive Typen

Primitive Typen können nicht als Instanzen von Typvariablen verwendet werden. Aber: primitive Typen sind schneller.

„Lösung“:

- `Stream#mapToInt`, `Stream#flatMapToInt`
- `Stream#mapToDouble`, `Stream#flatMapToDouble`
- `Stream#mapToLong`, `Stream#flatMapToLong`
- analog für `DoubleStream`, `IntStream`, `LongStream`
- `DoubleStream#mapToObj`, `IntStream#mapToObj`, `LongStream#mapToObj`