Java OOP

Konrad Raue, Oliver Scholz

26. November 2019

Gliederung

1. Nachtrag
2. Debuggen in IntelliJ
3. Comparator
4. Anonyme Klassen
5. Lambda-Ausdruck
6. Nächste Woche

Nachtrag

continue, break

- continue: um diesen einen Schleifendurchlauf zu beenden
- break: für switch und um Schleifen (vorzeitig) zu beenden
- class Klasse extends Vererbung implements Interface1, Interface2, ...
- Interface klasse = new Klasse();
- AbstrakteKlasse klasse = new Klasse();

Debuggen in IntelliJ

Comparator

Comparable

```java
public class Vergleichsobjekt implements Comparable{
    private int id;
    private String content;

    public Vergleichsobjekt(int id, String content){
        this.id = id;
        this.content = content;
    }

    public int getId(){
        return id;
    }

    public String getContent(){
        return content;
    }

    @Override
    public int compareTo(Object o){
        Vergleichsobbjekt other = (Vergleichsobjekt) o;
        if(id < other.getId()){
            return -1;
        }else if(id > other.getId()){
            return 1;
        }else{
            return content.compareTo(other.getContent());
        }
    }
}
```

Jetzt kann Collections.sort(List<Vergleichsobjekt>); verwendet werden, um eine Liste zu sortieren.

Comparator

```
1  public class Vergleicher implements Comparator<Vergleichsobjekt>{
2      @Override
3      public int compare(Vergleichsobjekt first, Vergleichsobjekt
       second){
4          if(first.getId() < second.getId()){
5              return -1;
6          }else if(first.getId() > second.getId()){
7              return 1;
8          }else{
9              return first.getContent().compareTo(
10             second.getContent());
11         }
12     }
13 }
```

Jetzt kann Collections.sort(List<Vergleichsobjekt>, new Vergleicher()); verwendet werden, um eine Liste zu nach dem im Vergleicher festgelegten Prinzip zu sortieren.

Anonyme Klassen

anonyme Klasse

```
1  Collections.sort(List<Vergleichsobjekt>,
2  new Comparator<Vergleichsobjekt>(){
3      @Override
4      public int compare(Vergleichsobjekt first, Vergleichsobjekt
       second){
5          if(first.getId() < second.getId()){
6              return -1;
7          }else if(first.getId() > second.getId()){
8              return 1;
9          }else{
10             return first.getContent().compareTo(
11             second.getContent());
12         }
13     }
14 });
```

Lambda-Ausdruck

lambda expression

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

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

lambda expression

```
myList.forEach(new Consumer<String>() {
    public void accept(String element) {
        System.out.println(element);
    }
});
```

Dies ist ein FunctionalInterface, d.h. es gibt nur eine Methode von der Klasse Object in diesem Interface.

```
myList.forEach((String element) -> System.out.println(element));
```

element ist der einzige Parameter, d.h. sein Typ wird vom Kontext selbstständig abgeleitet.

```
myList.forEach(element -> System.out.println(element));
```

Nächste Woche

Nächste Woche

- Übungen
- Übungen
- Übungen