Programmierung 2

DHBW Mannheim - WWI17SCB - May 2018

Slides

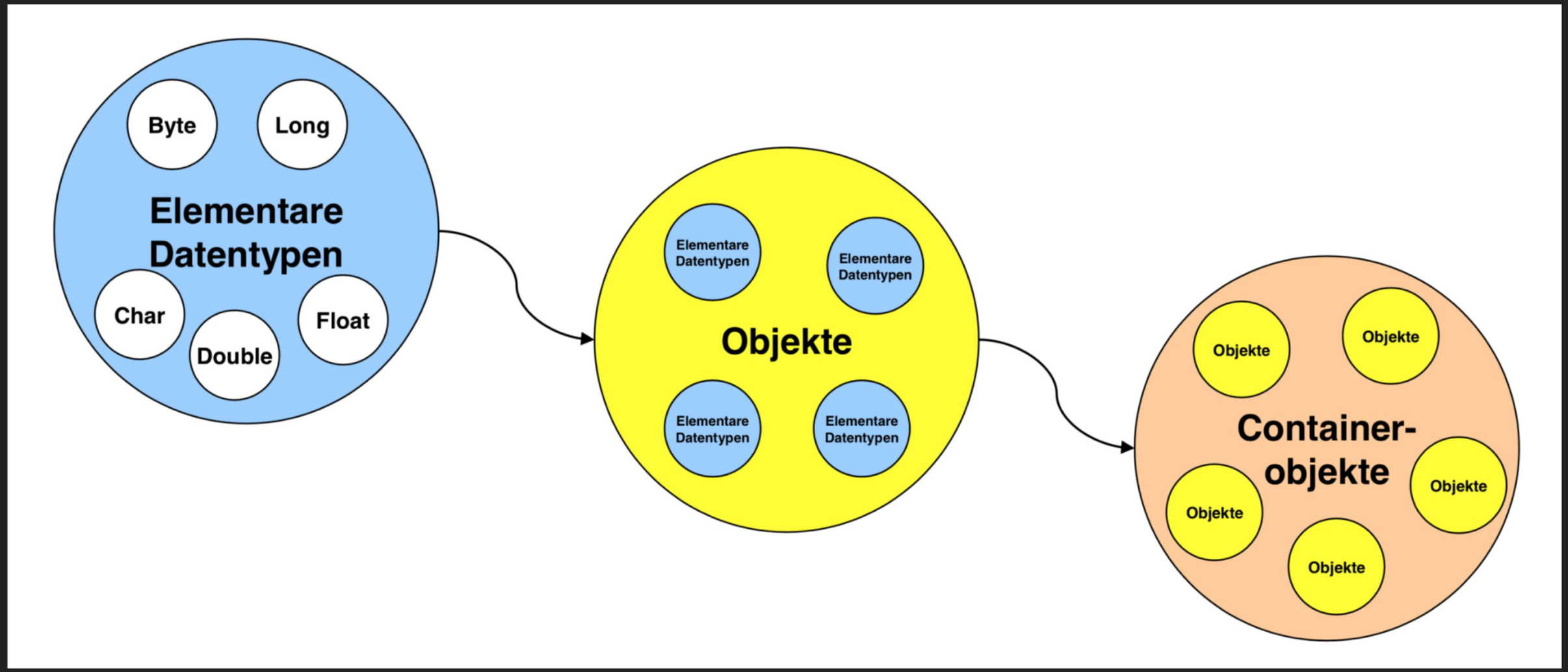
https://goo.gl/U45wY3

3. Collection Framework

Collections - Lernziele

* Wissen, wozu das Collection Framework da ist
* Unterschiede der 3 Objekt-Containerarten erklären
* Objekte in den Containern einfügen, löschen und finden
* Mit Iteratoren die Container durchlaufen
* Sortierbare Container mit Comparable und Comparator sortieren
* Die equals()- und die hashCode()-Methode in eigenen Klassen überschreiben

Datenstrukturen und Container



Generische Container

* Können verschiedenste Objekte enthalten
* Können beliebig viele Objekte aufnehmen
* Können auf bestimmte Objekte typisiert werden

* Offizielle Dokumentation:

  https://docs.oracle.com/javase/tutorial/collections/

Das Java Collection Framework

A collections framework is a unified architecture for representing and manipulating collections.

Hä?

Das Java Collection Framework bietet...

* **Interfaces**

Abstrakte Datentypen, die Collections repräsentieren. Interfaces erlauben es, Collections unabhängig von deren Implementierungsdetails zu verwenden.

Das Java Collection Framework bietet...

* **Implementierungen**

Wiederverwendbare Standard-Implementierungen von Collection-Objekten.

Das Java Collection Framework bietet...

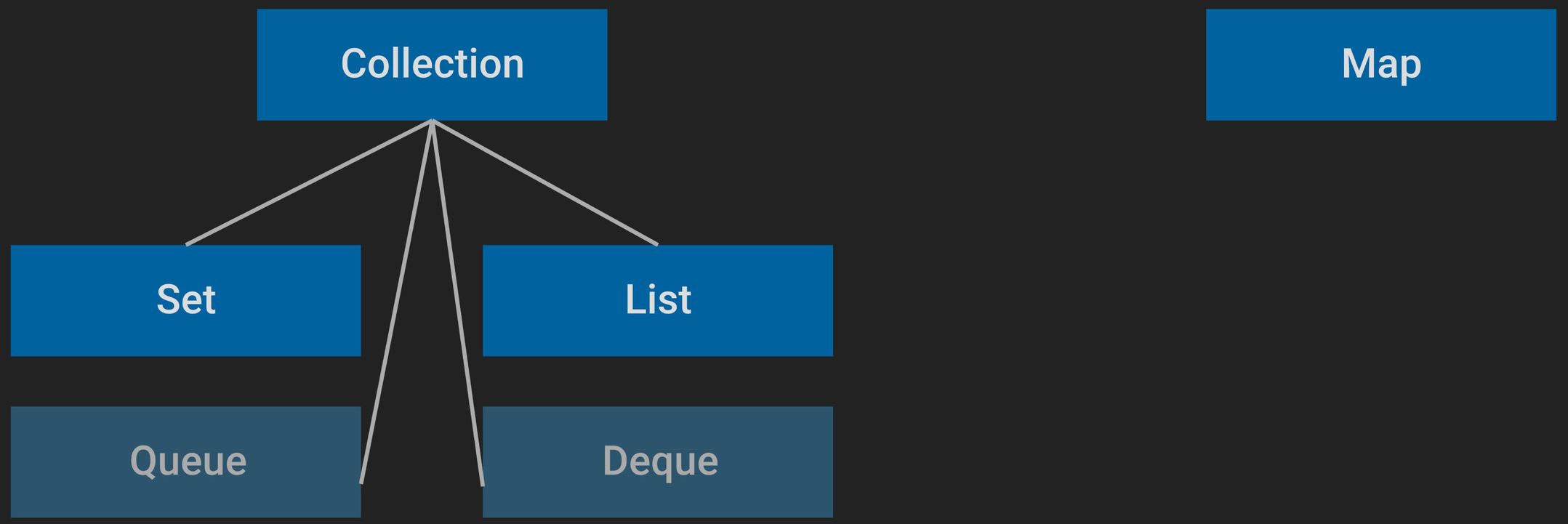
* **Algorithmen**

Methoden, die häufig benötigte Funktionen zur Verfügung stellen, wie z.B. Durchsuchen oder Sortieren einer Collection.

Das Java Collection Framework - Vorteile

* Weniger Programmieraufwand
* Wiederverwendung von Softwarekomponenten
* Bessere Performance und Softwarequalität
* Erlaubt Interoperabilität zwischen verschiedenen Softwarekomponenten
* Reduzierter Lernaufwand für neue APIs

Interfaces im Collection Framework



interface java.util.Collection

* Eine Gruppe von Objekten ("Elemente" der Collection)
* ***Collection*** ist der kleinste gemeinsame Nenner aller Collection-Interfaces
* Wird an Stellen verwendet, wo Collections sehr generisch behandelt werden
* Manche Arten von Collections erlauben Duplikate, andere nicht
* Manche Arten von Collections sind geordnet, andere nicht
* Es gibt keine direkte Implementierung von ***Collection***

java.util.Collection - Wichtige Methoden

* `int size()`
* `boolean isEmpty()`
* `boolean contains(Object o)`
* `Object[] toArray()`
* `boolean add(Object o)`
* `boolean remove(Object o)`
* `Iterator iterator()`
* `void clear()`

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

List, Set und Map

Listen (List)

* Zugriff sequentiell oder wahlfrei, Duplikate erlaubt

* Reihenfolge des Einfügens bleibt erhalten

Mengen (Set)

* Zugriff erfolgt über Iteratoren, keine Duplikate

* Reihenfolge des Einfügens bleibt nicht erhalten

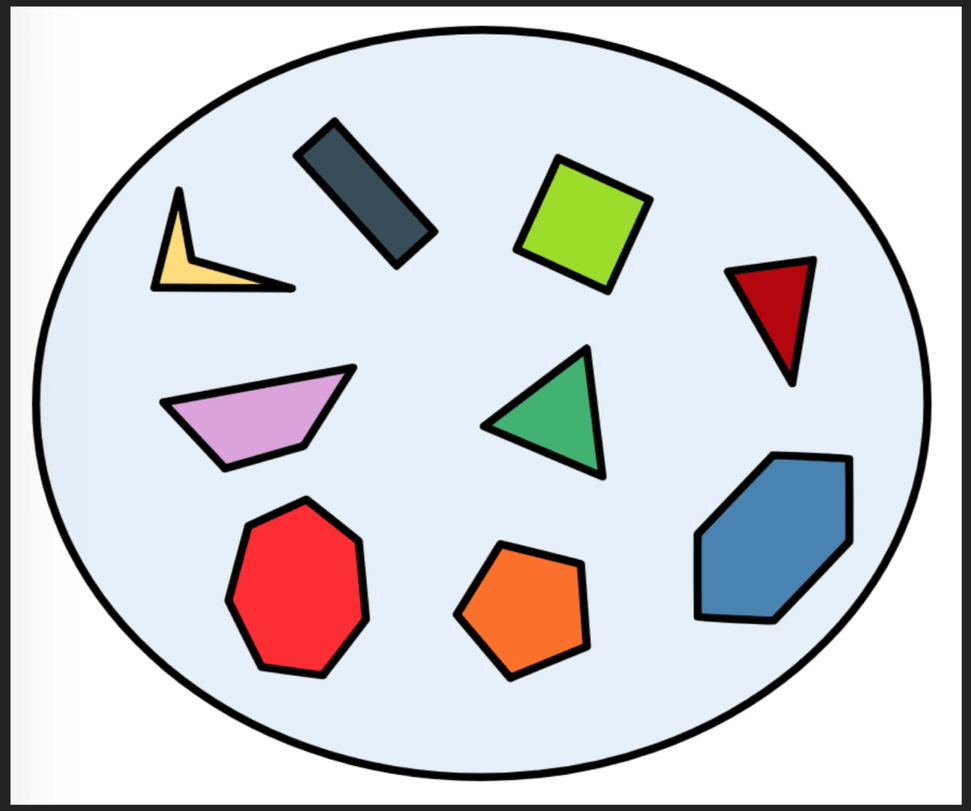
Schlüssel-Werte-Paare (Map)

* Zusammengehörige Objektpaare

* Schlüssel sind immer eindeutig, Zugriff über Schlüssel

interface java.util.Set

* Collection **ohne Duplikate**
* Keine eigenen Methoden
* Zugriff über Iterator
* Reihenfolge **nicht definiert**
* Entspricht einer math. Menge
* Kann z.B. verwendet werden für:
  - Karten einer Pokerhand
  - Vorlesungen eines Studenten



interface java.util.List

* **Geordnete** Collection - Zuerst Reihenfolge d. Einfügens
* Zugriff sequenziell, indexbasiert oder per Iterator
* Index beginnt mit 0 und endet bei n Elementen bei n-1
* Größe der Liste wird dynamisch beim Einfügen oder Löschen von Elementen angepasst
* Sortier- und durchsuchbar
* Kann z.B. verwendet werden für:
  - Auftragspositionen

java.util.List - Wichtige Methoden

* `boolean add(int index, Object o)`
* `Object get(int index)`
* `Object set(int index, Object o)`
* `int indexOf(Object o) / int lastIndexOf(Object o)`
* `Object remove(int index)`
* `void sort(Comparator c)`

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

interface java.util.Queue

* Warteschlange
* Reihenfolge d. Einfügens bleibt erhalten
* Üblicherweise FIFO-Queue
* Übliche Nutzung: Puffer für Elemente, die verarbeitet werden sollen (z.B. über eine Netzwerkverbindung verschickt)

interface java.util.Deque

* Deque = "Double-Ended Queue"
* FIFO- oder LIFO-Queue

interface java.util.Map

* Es werden immer Schlüssel-Werte-Paare eingefügt
* Jeder Schlüssel ist eindeutig
* Zugriff auf die Werte-Objekte erfolgt über die Schlüssel
* Zwei wesentliche Vertreter:
  * TreeMap: Einträge werden nach Schlüsseln sortiert -> Schlüssel-Klasse muss das Interface Comparable implementieren
  * HashMap: Es wird eine interne Position (Bucket) berechnet, an der das Schlüssel- Werte-Paar in die Map aufgenommen wird

java.util.Map - Wichtige Methoden

* boolean containsKey(Object k)
* boolean containsValue(Object v)
* Object get(Object key)
* Object put(Object key, Object value)
* Object remove(Object key)
* boolean replace(Object key, Object value)
* Set keySet()
* Collection values()

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Wichtige Implementierungen

HashSet - *Set*

TreeSet - *Set, SortedSet*

ArrayList - *List*

LinkedList - *List, Queue, Deque*

HashMap - *Map*

TreeMap - *Map, SortedMap*

Iterieren über Collections

class java.util.Iterator

* Einheitlicher Standard zum Traversieren von Collections
* Collection wird sequentiell durchlaufen
* Es können keine Elemente übersprungen werden
* Die Collection kann nur „vorwärts“ durchlaufen werden
* Bei Änderung des Inhalts der Collection muss der Iterator neu erzeugt werden

class java.util.Iterator - Wichtige Methoden

* `boolean hasNext()`

  überprüft, ob das aktuelle Element im Container noch einen Nachfolger hat

* `Object next()`

  greift auf das nächste Element des Containers zu

* `void remove()`

  löscht das Element aus dem Container, welches zuletzt vom Iterator gelesen wurde

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Listen: class java.util.ArrayList

* Einfachste Liste
* Für die meisten Anwendungsfälle geeignet
* "Das bessere Array"

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Collections - Codebeispiel

```
ArrayList liste = new ArrayList();
liste.add("Otto");
liste.add("Ludwig");
liste.add("Helmut");
liste.add(2, "Paul");
liste.set(1, "Überschreibt den Ludwig");
```

```
0 -> "Otto"
1 -> "Überschreibt den Ludwig"
2 -> "Paul"
3 -> "Helmut"
```

Collections - Codebeispiel

```java
System.out.println(liste.contains("Otto"));
System.out.println(liste.indexOf("Ludwig"));
System.out.println(liste.get(3));
System.out.println(liste.size());
```

```
true
-1
Helmut
4
```

Collections - Codebeispiel

```
Iterator i = liste.iterator();
while (i.hasNext()) {
  System.out.println(i.next());
}
```

```
Otto
Überschreibt den Ludwig
Paul
Helmut
```

Collections - Codebeispiel 1

```
liste.clear();
System.out.println(liste.size());
```

```
0
```

Coding Time!

Collections - Codingaufgabe 1

30m

* Implementiert das Uni-Beispiel von letzter Vorlesung mit java.util.ArrayList anstelle von Arrays
* Welche Vorteile hat das?

Collections - Codingaufgabe 1

30m

Modelliert eine Uni mit folgenden Objekten:

* Universität (Name, n Vorlesungen, m Studenten)

* Vorlesung (Titel, Studenten)

* Student (Name, hört 1-n Vorlesungen)

Legt einige Objekte an und weist Studenten an Vorlesungen zu. Am Ende sollte die Uni eine print-Methode haben, die ca. das tut:

```
Uni 'DHBW Mannheim' hat insgesamt 3 Studenten und 4 Vorlesungen.

Paul Müller hört 'Programmierung 2', 'Rhetorik'
Lisa Engel hört 'Rhetorik', 'Bilanzierung'
Franz Faul hört 'Einführung in die Wirtschaftsinformatik'
```

Generics

Generics

Ein paar Problemchen...

```
30
31      public boolean addStudent(Student s) {
32          return students.add(s)
33      }
35      |
```

Unchecked call to 'add(E)' as a member of raw type 'java.util.ArrayList' more... (⌘F1)

Wieso muss ich hier explizit casten?

```
while (i.hasNext()) {
  Student s = (Student) i.next();
  s.print();
}
```

Generics

* Wäre doch praktisch, wenn die Collection wüsste, welche Objekte in ihr drin stecken.
* Vorteile:
  * Typechecks zur Compilierzeit.
  * Keine Casts mehr.

https://docs.oracle.com/javase/tutorial/java/generics/why.html

Generics

* Dazu gibt es in Java "Generics". Aus:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

wird:

```
List list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

Generics

* Ein "generischer Typ" ist ein Datentyp, der mit einem zweiten Datentyp parametrisierbar ist.
* Beispiel für einen nicht-generischen Typen:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Generics

* Derselbe Typ als generischer Typ.
* Die *Typparameter* schreibt man in spitze Klammern. Sie gehören zur Klassendefinition dazu:

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Generics - Mehrere Typparameter

* Mehrere Typparameter schreibt man hintereinander
* Auch Interfaces können generische Typen sein.

```
public interface Map<K,V> {
  ...
}
```

Generics - Namenskonventionen

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value
* S,U,V etc. - 2nd, 3rd, 4th types
  * Alternativ: T1, T2, T3...

Generics - Verschachtelt

* Natürlich können generische Typen wiederum generische Typen enthalten

```
Map<Integer, Box<String>> mapOfBoxes = new HashMap<Integer, Box<String>>();
```

Wird dann beim Lesen etwas sperrig...

Generics - Der "Diamond"

* Erlaubt es, beim Konstruktoraufruf die Typparameter wegzulassen solange der Compiler sie automatisch aus dem Typ der Variable ableiten kann

```
TreeSet<Student> set = new TreeSet<>();

// Kein doppelter Diamond nötig:
Map<Integer, ArrayList<String>> mapOfBoxes = new HashMap<>();
```

Generics - Fazit

* Insbesondere mit Bezug auf Vererbung und Polymorphie gibt es hier noch einiges mehr zu lernen - das machen wir in der VL nicht
* Es macht Sinn, eure Collections zu parametrisieren, also:

```
public class Universitaet {
  private ArrayList<Student> studenten;
  private ArrayList<Vorlesung> vorlesungen;

  ...
}
```

Generics - Fazit

* Generische Parametrisierung von *Iterator*:

```
Iterator<Student> i = studenten.iterator();
while (i.hasNext()) {
  Student s = i.next();
  s.print();
}
```

Coding Time!

Collections - Codingaufgabe 2

15m

* Baut eure Aufgabe von eben so um, dass sie Generics verwendet.
* Entfernt die Casts aus den Iteratoren.

Collections mit definierter Reihenfolge

interface java.util.SortedSet<E>

* Collection, die immer eine interne Ordnung ("total ordering") hat
* Entsprechend der "natural order" der Elemente (z.B. aufsteigend für int)
* Bei selbst definierten Objekten als Elemente müssen diese
  das Interface ***Comparable*** implementieren

https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html

Sets: class java.util.TreeSet<E>

* Zugriff auf den Container erfolgt sequentiell über Iteratoren
* Größe der Liste wird dynamisch beim Einfügen/Löschen angepasst
* Duplikate sind nicht erlaubt (Vergleich über die equals-Methode)
* Reihenfolge, in der Elemente eingefügt werden, bleibt nicht erhalten
* Sortieren nach durch Implementierung des Comparable-Interface
* Das ***Comparable***-Interface muss auf jeden Fall implementiert werden, wenn Objekte in ein TreeSet eingefügt werden

https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Codebeispiel

```java
Student helga = new Student("Helga", "Hansen");
TreeSet<Student> set = new TreeSet<>();
set.add(helga);
```

```
Exception in thread "main" java.lang.ClassCastException:
collections.sortierung.Student cannot be cast to java.base/java.lang.Comparable
    at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
    at java.base/java.util.TreeMap.put(TreeMap.java:536)
    at java.base/java.util.TreeSet.add(TreeSet.java:255)
    at collections.sortierung.CollectionsSortDemo.main(CollectionsSortDemo.java:15)

Process finished with exit code 1
```

interface java.lang.Comparable<T>

* Elemente sortieren sich selbst beim Einfügen in Sets oder Maps
* Definiert die natürliche Ordnung von Objekten einer Klasse
* Beinhaltet genau eine Methode: `public int compareTo(T o)`
* Bedeutung der Rückgabewerte

  Wert < 0: Einzufügendes Element liegt vor dem Vergleichsobjekt

  Wert = 0: Einzufügendes Element und das Vergleichsobjekt sind gleich

  Wert > 0: EinzufügendeS Element liegt hinter dem Vergleichsobjekt

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Codebeispiel

```
public class Student implements Comparable<Student> {
    private String vorname;
    private String nachname;
    private String matrikelNr;

    public int compareTo(Student s) {
        return Integer.parseInt(matrikelNr) - Integer.parseInt(s.getMatrikelNr());
    }

    ...
}
```

Coding Time!

Collections - Codingaufgabe 3

20m

* Implementiert eine (neue) Klasse "Student" mit den Attributen Vorname, Nachname, Matrikelnummer und einer "print" Methode, die alle drei Attribute ausgibt
* Implementiert für diese Klasse das interface ***Comparable***
* Initialisiert ein neues TreeSet und fügt 5 Studenten hinzu
* Iteriert über das TreeSet und beachtet die Reihenfolge der Ausgabe

interface java.util.Comparator<T>

* Sortiert Elemente beim Einfügen in Sets oder Maps
* Sortierung erfolgt nach einer beliebigen Sortierreihenolge und übersteuert die natürliche Ordnung
* Comparator sollte in eigener Klasse implementiert werden
* Zur Verwendung des Comparators wird die implementierende Klasse dem Konstruktor des Sets oder der Map übergeben

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

interface java.util.Comparator<T>

* Wichtigste Methode: `public int compare(T o1, T o2)`
* Bedeutung der Rückgabewerte:
  * Wert < 0: o1 liegt vor o2
  * Wert = 0: o1 und o2 sind gleich
  * Wert > 0: o1 liegt hinter o2

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Codebeispiel

```
public class AutoComparator implements Comparator<Auto> {
    public int compare(Auto a, Auto b) {
        return b.leistung - a.leistung;
    }
}
```

Comparable vs Comparator

* Comparable wird in Klassen selbst implementiert und definiert deren natürliche Ordnung
* Comparator ist eine externe, selbst definierte Klasse, die die natürliche Ordnung übersteuern kann (z.B. "AutoComparator")
* Comparatoren können Collections bei deren Initialisierung übergeben werden

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Coding Time!

Collections - Codingaufgabe 4

15m

* Kopiert euer Beispiel von eben und baut einen "StudentComparator", der nach Matrikelnummern in absteigender Reihenfolge sortiert (=> s2 - s1 statt s1 - s2)
* Übergebt ein Objekt dieser Klasse dem TreeSet bei Initialisierung
* Beachtet die neue Sortierreihenfolge bei der Ausgabe.

Sortieren von Collections

Sortieren von Listen

* Listen (Vector, ArrayList, ...) sind normalerweise unsortiert
* Die Klasse java.util.Collections bietet eine überladene Sortiermethode zum Sortieren von List-Objekten an
* `static void sort(List<T> list)`
  * Sortiert die Liste nach der natürlichen Ordnung
  * Elemente müssen dazu das Interface Comparable implementieren
* `static void sort(List<T> list, Comparator<? super T> c)`
  * Sortiert die Objekte der Liste über den entsprechenden Comparator c

https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Codebeispiel

```
Student klaus = new Student("Klaus", "Klausen", "123456");
Student peter = new Student("Peter", "Petersen", "456789");
Student helga = new Student("Helga", "Hansen", "234567");

ArrayList<Student> list = new ArrayList<>();

list.add(klaus);
list.add(peter);
list.add(helga);
```

Codebeispiel

```
Iterator<Student> it = list.iterator();

while(it.hasNext()) {
  Student s = it.next();
  s.print();
}
```

```
Klaus Klausen 123456
Peter Petersen 456789
Helga Hansen 234567
```

Codebeispiel

```
list.sort(new StudentComparator());
Iterator<Student> it = list.iterator();

while(it.hasNext()) {
  Student s = it.next();
  s.print();
}
```

```
Peter Petersen 456789
Helga Hansen 234567
Klaus Klausen 123456
```

Vergleich von Objekten

Vergleich von Objekten

* Collections müssen wissen, ob ein Objekt einem anderen gleicht, um z.B. "contains()" korrekt implementieren zu können
* Hierzu wird auf `Object.equals()` zurückgegriffen
* "equals" macht einen inhaltlichen Vergleich - es geht nicht um Referenzgleichheit (entsprechend dem "==" Operator)

```
List list = new ArrayList();
list.add("123");

boolean contains123 = list.contains("123");
```

Vergleich von Objekten

* "equals" wird auch verwendet um Objekte aus Collections zu entfernen

```
List list = new ArrayList();
list.add("123");

boolean removed = list.remove("123");
```

* Für eigene Klassen muss daher "equals" richtig implementiert werden, wenn sie in Collections korrekt funktionieren sollen.

Codebeispiel - "contains" ohne "equals"

```
ArrayList<Student> list = new ArrayList<>();

Student klaus = new Student("Klaus", "Klausen", "123456");
list.add(klaus);

System.out.println(list.contains(klaus));
System.out.println(list.contains(new Student("Klaus", "Klausen", "123456")));
```

```
true
false
```

Was muss "equals" prüfen?

Für direkte Sub-Klassen von ***Object***

1. Alias-Check mit dem ==-Operator
2. Test auf null
3. Typverträglichkeit überprüft, ob es sich um Instanzen der gleichen Klasse handelt
4. Feld-Vergleich überprüft die inhaltliche Gleichheit der Attribute

Was muss "equals" prüfen?

Für indirekte Sub-Klassen von ***Object***

1. Alias-Check mit dem ==-Operator
2. Delegation an die Oberklasse ermöglicht die Prüfung der Gleichheit der von der Oberklasse geerbten Anteile
3. Feld-Vergleich überprüft die inhaltliche Gleichheit der Attribute der Sub-Klasse

Der Kontrakt von "equals"

* Reflexiv: jedes Objekt liefert beim Vergleich mit sich selbst true
* Symmetrisch: x verglichen mit y liefert das gleiche Ergebnis, wie der Vergleich von y mit x
* Transitiv: wenn x gleich y und y gleich z ist, dann ist auch x gleich z
* Konsistent: solange sich zwei Objekte nicht verändern, liefert der Vergleich der beiden Objekte immer das gleiche Ergebnis
* Objekte müssen von null verschieden sein

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

Codebeispiel - direkte Subklasse von Object

```
public class Haustier {
  private String art;
  private int gewicht;

  public boolean equals(Object objekt) {
    // Alias-Check
    if (this == objekt) return true;
    // Test auf null
    if (objekt == null) return false;
    // Typverträglichkeit
    if (objekt.getClass() != this.getClass()) return false;
    // Feldvergleich
    if(!this.art.equals(((Haustier) objekt).getArt())) return false;
    if(!(this.gewicht == ((Haustier) objekt).getGewicht())) return false;
    // Alles gleich!
    return true;
  }
}
```

Codebeispiel - indirekte Subklasse von Object

```
public class Hund extends Haustier {
  private String rasse;

  public boolean equals(Object objekt) {
    // Alias-Check
    if (this == objekt) return true;
    // Delegation an super
    if (!super.equals(objekt)) return false;
    // Feldvergleich
    if (!this.rasse.equals(((Hund) objekt).getRasse())) return false;
    // Alles gleich!
    return true;
  }
}
```

Wann sind Objekte gleich?

* Natürlich müssen nicht zwingend alle Felder verglichen werden
* Gleichheit ergibt sich aus dem Domänenmodell
* Für Studenten reicht z.B. vielleicht die Matrikelnummer?

Hashing von Objekten - hashCode()

* Hashes von Objekten werden benötigt, um sie in Hash-basierte Collection-Objekte (z.B. HashSet, HashMap) zu stecken
* Warum und wie das intern funktioniert, lernt ihr in "Algorithmen und Datenstrukturen"
* Wir merken uns: bei Nutzung von HashMaps brauchen wir "hashCode"

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--

Zusammenhang hashCode() und equals()

* Korrekte Verwaltung der Einträge basiert auf folgender Bedingung:

```
if o1.equals(o2) == true
then o1.hashCode() == o2.hashCode()
```

Es folgt: sobald die equals()-Methode überschrieben wird, muss auch die hashCode()-Methode überschrieben werden

!!! Es folgt NICHT: !!!

```
if o1.hashCode() == o2.hashCode()
then o1.equals(o2) == true
```

Zusammenhang hashCode() und equals()

Vorschlag zur Implementierung

* Verwendung der Attribute, die bei der Implementierung der equals()-Methode verwendet werden
* Ermittlung der Hash-Codes der ausgewählten Attribute einer Klasse
* Addition oder bitweise Verknüpfung mit exklusivem Oder der einzelnen Hash-Codes

hashCode() - Codebeispiel

^ ist XOR

```
public class Haustier {
  private String art;
  private int gewicht;

  public int hashCode() {
    return this.getArt().hashCode() ^ this.getGewicht();
  }
}
```

hashCode() - Codebeispiel

Geht aber auch einfacher, z.B.:

```
public class Student {
  private int matrikelNr;

  public int hashCode() {
    return this.matrikelNr;
  }
}
```

Maps

Maps: class java.util.HashMap<K, V>

* Hashtable-basierte Implementierung des Map-Interfaces
* Auf Basis der hashCode()-Methode der Schlüsselklasse wird eine interne Position (Bucket) berechnet, an der das Schlüssel- Werte-Paar in die Map aufgenommen wird
* Sortierung/Reihenfolge nicht definiert

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

java.util.Map - Wichtige Methoden

* boolean containsKey(Object k)
* boolean containsValue(Object v)
* Object get(Object key)
* Object put(Object key, Object value)
* Object remove(Object key)
* boolean replace(Object key, Object value)
* Set keySet()
* Collection values()

Wrapper-Klassen

* Statt elementarer Datentypen werden Objekte erwartet
* Um elementare Datentypen in Objekten zu kapseln, gibt es die Wrapper-Klassen
* Methoden zur Ein- und Ausgabe sowie zur Manipulation
* Methoden zur Umwandlung von Datentypen
* Wrapper-Klassen existieren für folgende Datentypen:

  boolean, byte, char, double, float, int, long, short

Wrapper-Klassen - Autoboxing

* Seit Java 5 gibt es "Autoboxing", d.h. Maps wrappen elementare Datentypen selbst in die Wrapper-Klassen
* Ich schreibe also statt:

```
map.put(new Integer(1), paul);
```

einfach:

```
map.put(1, paul);
```

HashMap - Codebeispiel

```
HashMap pairs = new HashMap<Integer, Hund>();
pairs.put(10,new Hund("Collie"));
pairs.put(20,new Hund("Bernhardiner"));
// Schlüssel holen
Set<Integer> keys = pairs.keySet();
// Über Schlüssel iterieren
Iterator<Integer> it = keys.iterator();
while (it.hasNext()) {
  int key = it.next();
  // Mithilfe des Schlüssels aus der HashMap lesen
  Hund value = pair.get(key);
  System.out.println("Schlüssel: " + key + " Wert: " + value.getRasse());
}
```

Die Qual der Wahl...

Wann nehme ich welche Collection?

* Schlüssel-Wert-Paare: Maps
* Mengen ohne Duplikate: Sets
* Listen, ggf. mit Duplikaten: List

* Oft hat die Wahl der Implementierung große Auswirkungen auf das Laufzeit- und Performanceverhalten (z.B. HashMap vs TreeMap)
* Mehr dazu lernt ihr in "Algorithmen und Datenstrukturen"
* Hier brauchen wir nur HashMap, TreeSet und ArrayList

Coding Time!

Collections - Codingaufgabe 5

20m

* Modelliert eine Klasse "Student" mit allen nötigen Methoden für Vergleich und Speicherung (compareTo, equals, hashCode)
* Speichert Studenten in einer HashMap
* Probiert aus, was passiert, wenn ihr containsValue() verwendet und einen neuen Student mit gleichen Daten übergebt
* Iteriert über die Hashmap (holt euch vorher die Schlüssel mit `keySet()`) und gebt alle Studenten aus