Inhalt der Vorlesung

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

Kapitel 5: Generics

Inhalt

Kapitel 5: Generics

Lernziele

[LZ 5.1] Die Vorteile von typsicheren Datenstrukturen kennen
[LZ 5.2] Parametrisierbare Datentypen instanziieren und nutzen können
[LZ 5.3] Collection-Klassen und dazugehörige Interfaces kennen und anwenden können
[LZ 5.4] Verstehen, warum auch die Collection-Interfaces generisch sein müssen
[LZ 5.5] Generische Interfaces nutzen und implementieren können
[LZ 5.6] Das Iterator-Konzept verstehen und anwenden können
[LZ 5.7] Eigene Klassen und Interfaces parametrisierbar definieren und einsetzen können

Die Java-Laufzeitbibliothek bietet eine Klasse `ArrayList` als sogenannte `Collection` an (mehr dazu unter Abschnitt 5.2), die eine Liste von Objekten verwalten kann und dabei einen schnellen Index-Zugriff erlaubt (daher der Name). Eine Liste von Quadraten lässt sich folgendermaßen erzeugen:

```
ArrayList liste = new ArrayList();
liste.add(new Quadrat(1.0));
liste.add(new Quadrat(2.0));
```

Die Collections genannten Standard-Listen sind sehr praktisch und ersparen das mühevolle Kodieren eigener Klassen für Listen. Jedoch ist es bei obiger Anwendung von `ArrayList` möglich, beliebige Objekte in einer `ArrayList` zu speichern:

```
liste.add(new Student(4711, "Codie Coder")); // Student hinzufügen
```

Das bedeutet, dass die Liste nicht typsicher ist, d.h. es lassen sich Objekte unterschiedlichen Typs darin speichern.

Bei der Ausgabe der Quadratflächen mit

```
for (int i = 0; i < liste.size(); ++i)
    System.out.println(((Quadrat) liste.get(i)).flaeche());
```

würde eine `ClassCastException` beim Versuch, das Student-Objekt in ein Quadrat-Objekt umzuwandeln („zu cast'en") geworfen werden und das Programm würde terminieren.

Abhilfe verschafft hier eine Typ-Abfrage mit `instanceof` in der Schleife:

```
if (liste.get(i) instanceof Quadrat)
    System.out.println(((Quadrat) liste.get(i)).flaeche());
```

Die `instanceof`-Abfrage löst das eigentliche Problem nicht, welches in der fehlenden Typsicherheit besteht. Die Liste müsste „wissen", dass sie nur Quadrat-Referenzen aufnehmen darf!
Als Lösung für dieses Problem wurden Klassen und Interfaces parametrisierbar gemacht. Auf das Beispiel bezogen bedeutet das, dass das ArrayList-Objekt den Typ Quadrat als Elementtyp bei der Erzeugung als Parameter erhält.

Damit kann der Compiler zur Übersetzungszeit (nicht erst zur Laufzeit!) prüfen, ob eine add-Methode korrekt mit einer Quadrat-Referenz parametrisiert wird:

```
ArrayList<Quadrat> liste = new ArrayList()<Quadrat>;
liste.add(new Quadrat(1.0));     // ok
liste.add(new Quadrat(2.0)); // ok
liste.add(new Student(4711, "Codie Coder")); // Compilerfehler!
```

Auch die Ausgabeschleife vereinfacht sich, da die get-Methode so definiert ist:

```
E get(int index); // E ist der Typ-Parameter, hier Quadrat
```

Somit liefert `get` bereits den richten Typ, ein Class-Cast ist nicht notwendig:

```
for (int i = 0; i < liste.size(); ++i) {
     System.out.println(liste.get(i).flaeche());
```

Obiger Code funktioniert, weil ArrayList parametrisierbar definiert ist (s. Java-API):

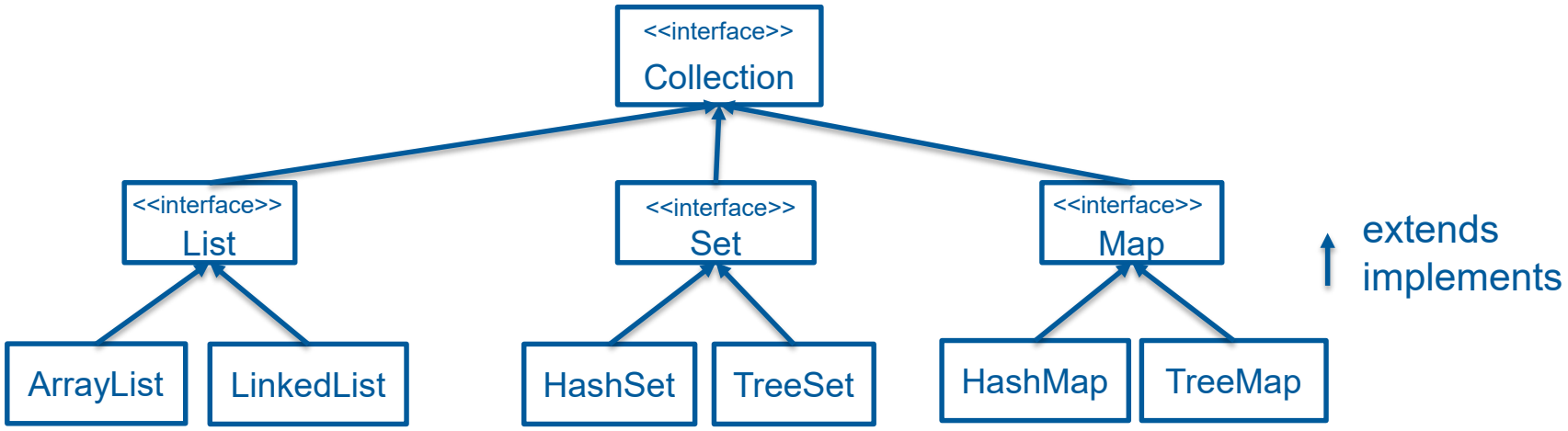
```
class ArrayList<E> {
   public boolean add(E param) {
       …
   }
   …
}
```

Anmerkungen

- wird der bei einer parametrisierbaren Klasse wie `ArrayList` der Typparameter weggelassen, so nutzt man den „raw type“ und der Compiler zeigt eine Warnung an. Die Warnung kann mit Hilfe der Annotation @SuppressWarnings("rawtypes") unterdrückt werden.
- Bei Deklaration mit gleichzeitiger Initialisierung kann der Typparameter entfallen:
  `ArrayList<Quadrat> liste = new ArrayList()<>;`
- Klassen und Interfaces können mehrere Typparameter besitzen. Beispielsweise hat die Klasse HashMap (s. Abschnitt 5.2) zwei Typparameter: `class HashMap<K, V>`
- Generics können nur mit Objekten genutzt werden, entsprechend wird Autoboxing und Autounboxing für primitive Typen in Listen benötigt:

```
ArrayList<Integer> liste = new ArrayList<>();
liste.add(2);                    // Autoboxing int → Integer
int erstesInt = liste.get(0);    // Autounboxing Integer → int
```

Die Java-Laufzeitbibliothek bietet im Paket „java.util“ parametrisierbare Klassen zur effizienten Speicherung von Daten in häufig verwendeten Datenstrukturen wie Liste, Hashtabelle bzw. Baum an. Neben den Klassen gibt es zudem eine Menge von parametrisierbaren Interfaces, die gemeinsame Methoden der Klassen festlegen und Java-Programme damit flexibler machen. Das folgende UML-Modell zeigt die Vererbungs- und Implementierungsbeziehungen (Ausschnitt):



List: listenartige Datenstrukturen

Set: Menge von Elementen, keine mehrfach vorkommenden Elemente möglich

Map: speichert/liest einen Wert zu einem Schlüssel

Erläuterung der hier behandelten Klassen:

- Eine Collection<E> ist eine ungeordnete Gruppe von Elementen der Klasse E (E ist der Parameter). Bei List kann ein Element mehrfach vorkommen, bei Set und Map nicht.

- Operationen auf Collections:

boolean add(E e)	Fügt e hinzu
void clear()	Löscht alle Elemente
boolean contains(Object o)	Liefert true, falls Referenz o enthalten ist
Iterator<E> iterator()	Liefert iterator-Objekt zur Iteration, s. Abschnitt 5.3
boolean remove(Object o)	Löscht o falls vorhanden und gibt true zurück, false sonst
int size()	Liefert Anzahl Elemente

- Implementierungen zu List<E> verwalten Objekte der Klasse E in Form linearer Listen. Sie bieten die Collection-Methoden, darüber hinaus können Elemente an beliebiger Stelle eingefügt bzw. gelöscht werden. ArrayList<E> bietet einen performanten lesenden Zugriff, LinkedList<E> dagegen performantes Einfügen und Löschen von Elementen.

- Operationen auf Listen zusätzlich zu den von Collection geerbten:

boolean add(E e)	hängt e an Listenende, liefert true falls erfolgreich
void add(int index, E e)	Fügt e an index-te Stelle ein
E get(int index)	Liefert das Element an der Stelle index
E remove(int index)	Löscht das index-te Element und gibt es zurück

Sets speichern Mengen von Objektreferenzen vom Typ E (siehe Parametriesierung), wobei jede Referenz nur einmal vorkommen darf.
Implementierung des Set-Interfaces sind

- **HashSet<E>**: speichert Elemente vom Typ E in zufälliger Reihenfolge
- **TreeSet<E>**: speichert Elemente vom Typ E in aufsteigender Reihenfolge

Ein Set<E> bietet die **Operationen**

boolean add(E e)	Fügt e hinzu, falls es nicht bereits enthalten ist
void clear()	Löscht alle Elemente
boolean contains(Object o)	Liefert true, falls Referenz o enthalten ist
Iterator<E> iterator()	Liefert iterator-Objekt zur Iteration, s. Abschnitt 5.3
boolean remove(Object o)	Löscht o falls vorhanden und gibt true zurück, false sonst
int size()	Liefert Anzahl Elemente

Beispiel: Erzeugung und Ausgabe der Lottozahlen in aufsteigender Reihenfolge

```
class Lotto {
    public static void main(String[] args) {
        TreeSet<Integer> ziehung = new TreeSet<Integer>();
        while (ziehung.size() < 6) // 6 (aus 49) Zahlen werden benötigt
            ziehung.add((int) (Math.random() * 49) + 1); // nur noch nicht Vorhandene werden hinzugefügt!
        System.out.println("Lottozahlen: " + ziehung); // Ausgabe aller 6 Zahlen
    }
}
```

Ein **Map<K, V>** ist eine Datenstruktur, die beliebig viele Paare aus Schlüsseln vom Typ K (*Key*) und Werten vom Typ V (*Value*) speichert. K und V können beliebige Klassen sein. Wesentlich ist, dass die vewendeten Schlüssel eindeutig sind, z.B. Matrikelnummer bei Studierenden (bezogen auf die jeweilige Hochschule!).

Gegeben sei beispielsweise ein Objekt, welches das Map-Interface implementiert:

```
Map<Integer, Student> studenten = ...; // Integer sind die Schlüssel (keys)
```

Dann werden Studenten mit deren Matrikelnummern als Schlüssel folgendermaßen eingefügt:

```
studenten.put(4711, new Student(4711, "Codie Coder");
studenten.put(4712, new Student(4712, "Bart Simpson");
```

Das Lesen eines Elements zu einer gegebenen Matrikelnummer 4711 gelingt mit:

```
Student codie = studenten.get(4711);
```

Die hier betrachteten Map-Implementierungen sind HashMap und TreeMap. HashMap speichert Elemente in ungeordneter, TreeMap in aufsteigender Reihenfolge.

Beispiele für das Erzeugen von Map-Objekten:

```
Map<Integer, Student> studenten = new HashMap<>();
Map<Integer, Student> studenten = new TreeMap<>();
```

Welche Implementierung gewählt wird, hängt davon ab, welche Anforderungen in Bezug auf Element-Ordnung bzw. Performance (HashMap ist schneller als TreeMap) gestellt werden.

Wichtige Operationen eines Map<K, V> sind:

void clear()	Löscht alle key-value-Paare
V get(K key)	Liefert Wert zu key oder null, falls key nicht existiert
V put(K key, V value)	Fügt value zu Schlüssel key ein
int size()	Liefert die Anzahl key-value-Paare
V remove(Object key)	Löscht das key-value-Paar zu key
Set<K> keySet()	Liefert die Schlüssel als Set, z.B. für Iteration
Collection<V> values()	Liefert alle Werte als Collection, z.B. für Iteration

Beispiel zu Maps

```
class Adresse {
    private String name, ort, strasse; private int plz, nr;
    public Adresse(String name, ...) { ... }
    public String getName() { return name; }
    public String toString() { return "Adresse [" + name + ... }
}
// Nutzung:
Map<String, Adresse> kontakte = new TreeMap<>();
Adresse peter = new Adresse("Peter", 12345, "Spielstadt", "Beispielweg", 3);
Adresse sarah = new Adresse("Sarah", 54321, "Javatown", "Glücksgasse", 1);
// Kontakte eintragen
kontakte.put(sarah.getName(), sarah);kontakte.put(peter.getName(), peter);
// alle Kontakte ausgeben
for (Adresse adr : kontakte.values())
    System.out.println(adr);
// Kontakt Sarah suchen
Adresse sarahAdresse = kontakte.get("Sarah");
if (sarahAdresse != null)
    System.out.println("Gefunden: " + sarahAdresse);
```

Die Ausgabe erfolgt durch die TreeMap-Verwendung aufsteigend nach Namen sortiert. Verwendet man ein HashMap, so ist die Reihenfolge unbestimmt. Alternativ kann auch über die Schlüssel ausgegeben werden:

```
for (String name : kontakte.keySet())
    System.out.println(kontakte.get(name));
```

In den vorangegangenen Abschnitten wurde erklärt, dass auch Interfaces parametrisierbar sind. Diese Aussage soll anhand der Interfaces Comparable<T> und Iterable<T> veranschaulicht werden.

Vergleich zweier Objekte vom gleichen Typ T mit Comparable<T>

```
class Student implements Comparable<Student> {
    private int matrNr;
    private String name;
    ... // CTOR etc.

    public int compareTo(Student s) { // Vergleich nach matrNr
        return matrNr – s.matrNr;
    }
}
```

nur Student-Objekte sind für compareTo zulässig

Auswirkung der Parametrisierung: Student statt Object

Im Unterschied zur nicht-parametrisierten Version ist der Parameter vom Typ Student, d.h. man benötigt weder Typüberprüfung noch Typecast. Der Compiler kann bereits zur Übersetzungszeit prüfen, ob eine passende Referenz übergeben wird (Student oder Subklassen davon).

Iteration über eine Menge von Elementen vom Typ T mit Iterable<T>

Zur Iteration über Collections oder Arrays mit Elementen vom Typ T hat Java ein Standard-Interface: das Interface Iterable<T> sowie das Interface Iterator<T>:

```
interface Iterable<T> {
    Iterator<T> iterator();
}
interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

ein iterierbares Objekt bietet eine `iterator`-Methode, die ein Iterator-Objekt erzeugt

ein Iterator-Objekt ermöglicht die Iteration über alle Elemente der Collection mithilfe der Methoden `hasNext` und `next` (siehe unten)

Eine Iteration wird damit folgendermaßen durchgeführt:

```
List<String> stringListe = new ArrayList<>(
    Arrays.asList(new String[] { "Java", "ist", "super" }));
Iterator<String> iterator = stringListe.iterator();
while (iterator.hasNext())
    System.out.println(iterator.next());
```

Durch die Parametrisierung des Interfaces liefert `next` bereits den richtigen Typ (`String`), somit entfallen Typecasts und der Compiler kann die richtige Verwendung zur Übersetzungszeit prüfen.

Wie können eigene Klassen iterierbar gemacht werden? Das folgende Beispiel zeigt eine `Iterable<T>`-Implementierung am Beispiel einer einfachen Integer-Liste:

```
class IntListeIterator implements Iterator<Integer> {
    private int index = 0;         // für Iteration, Start bei 0
    private List<Integer> liste;   // Liste, über die iteriert wird
    public IntListeIterator(List<Integer> liste) {
        this.liste = liste;
    }
    public boolean hasNext() {
        return index < liste.size();
    }
    public Integer next() {
        return (hasNext() ? liste.get(index++) : null);
    }
}
class IntListe implements Iterable<Integer> {
    private List<Integer> liste = new ArrayList<>();
    public void hinzufuegen(int wert) {
        liste.add(wert); }
    public Iterator<Integer> iterator() {
        return new IntListeIterator(liste);   // Iterator erzeugen!
    }
}
/* Anwendung: */
IntListe l = new IntListe();
for (int i : l)
    System.out.println(i);
```

Umsetzung durch Compiler

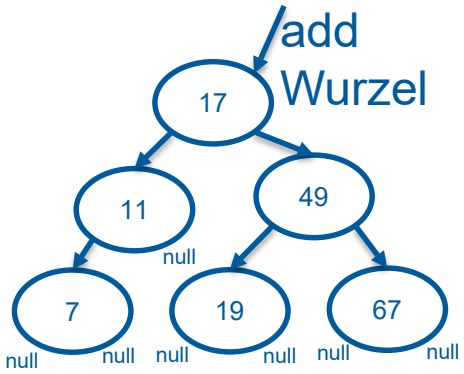
```
Iterator<Integer> iterator = l.iterator();

while (iterator.hasNext())
    System.out.println(iterator.next());
```

Erläuterung des letzten Beispiels:
- Für die Iteration erzeugt IntListe einen neuen Iterator.
- Dieser gibt, beginnend beim Index 0, alle Elemente der Liste zurück.
- Die For-Each-Schleife wird vom Compiler in das bekannte Konstrukt übersetzt:

```
while (iterator.hasNext())
    iterator.next();
```

Gegeben sei die nachfolgende Implementierung eines binären Suchbaums. Jedes Baum-Element besteht aus einem Datenanteil vom Typ int sowie einem linken und rechten Unterbaum jeweils vom Typ BinTree (rekursive Datenstruktur). Die add-Methode sortiert „value“ entweder im rechten oder im linken Unterbaum ein, abhängig davon, ob „value“ kleiner oder größer als das Wurzelelement „data“ des aktuell aufgerufenen Baums ist. Der rechts dargestellte Binärbaum verdeutlicht dieses Prinzip, wobei add stets an der Wurzel des Baumes aufgerufen wird.



5.4 Eigene Klassen parametrisieren

```
public class BinTree {
    private int data;     // Nutzdaten dieses Knotens
    private BinTree left; // linker Teilbaum oder null
    private BinTree right;// rechter Teilbaum oder null

    public BinTree(int data) {
        this.data = data;  this.left = left; this.right = right;
    }

    public void add(int value) {
        if (value < data) {                 // in linken Teilbaum einfügen
            if (left == null)
                 left = new BinTree(value); // neuer Knoten im linken Teilbaum
            else
                 left.add(value);           // rekursiv in linken Teilbaum einfügen
        } else if (value > data) {          // in rechten Teilbaum eifügen
            if (right == null)
                 right = new BinTree(value);// neuer Knoten im rechten Teilbaum
            else
                 right.add(value);          // rekursiv in rechten Teilbaum einfügen
}}}
// Aufruf:
BinTree tree = new BinTree(17);
tree.add(49);     // einhängen in rechtem Teilbaum, da 49 > 17
```

5. Generics

5.4 Eigene Klassen parametrisieren

Die Klasse BinTree kann beliebig viele int-Werte speichern. Sollen jedoch anstelle von „int“-Werten Strings gespeichert werden, so muss die Implementierung im „copy&paste“-Stil angepasst werden. Diese Vorgehensweise ist fehleranfällig und nicht gut wartbar. Die Lösung des Problems besteht in der Parametrisierung der Klasse.

```
public class BinTree<T extends Comparable<T>> {
    private T data;                 // Nutzdaten des Baumknotens
    private BinTree<T> left;        // linker Teilbaum oder null (kein Teilbaum vorh.)
    private BinTree<T> right;       // rechter Teilbaum oder null (kein Teilbaum vorh.)

    public BinTree(T d) { // erzeugt einen Baumknoten ohne Teilbäume
        data = d; left = null; right = null;
    }
    // fügt value dem Baum hinzu, d.h. value wird entsprechend seines Wertes einsortiert
    public void add(T value) {
        if (value.compareTo(data) < 0) {        // value < data => in linken Teilbaum
            if (left == null)
                left = new BinTree<T>(value);   // neuen Teilbaum erzeugen, Teilbaum in Wurzel einhängen
            else
                left.add(value);                // rekursiv in linken Teilbaum einfügen
        } else if (value.compareTo(data) > 0) { // value > data => in rechten Teilbaum
            if (right == null)
                right = new BinTree<T>(value);  // neuen Teilbaum erzeugen, Teilbaum in Wurzel einhängen
            else
                right.add(value);               // rekursiv in rechten Teilbaum einfügen
            }
            // hier wird nichts gemacht: der Wert existiert bereits!
    }
}
// Test: Baum für Integer erzeugen und füllen
BinTree<Integer> intTree = new BinTree<>(17);
intTree.add(49); // 49 wird im rechten Teilbaum eingefügt, da 49 > 17
```

Erläuterungen:

- Die Klasse `BinTree` wird mit dem Typ `T` parametrisiert. Bzgl. `T` wird zusätzlich noch eine Bedingung aufgestellt: `T extends Comparable<T>`. Das bedeutet, dass nur Parameter-Klassen erlaubt sind, die das `Comparable-Interface` implementieren. Die Bedingung ist erforderlich, da in der `add-Methode` ein Vergleichsoperator bzgl. der Daten vom Typ `T` benötigt wird, um den neuen Wert `value` einzufügen.
- In vielen Fällen genügt die Parametrisierung ohne eine Bedingung, wie sie oben existiert. Dadurch wird die Parametrisierung einer gegebenen Klasse relativ einfach: man stellt den Parameter (ein Grossbuchstabe, z.B. T) hinter den Klassennamen und nutzt T in der Klassendefinition an allen erforderlichen Stellen. Dies gilt für Interfaces analog.