

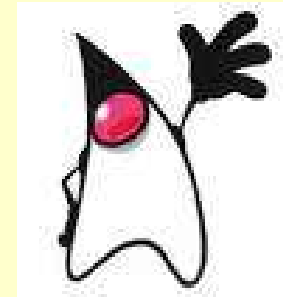
JAVA 5 – Generics

Inhalt

1. Die Programmiersprache Java
2. Simple Generics
3. Das Beispielprogramm
4. Tieferer Einstieg in Generics
5. Arrays
6. Kritische Betrachtung von Generics

1.1. Die Geschichte von Java

- ◆ 1991 - The Green Projekt
- ◆ James Gosling und 12 weitere Entwickler
- ◆ 18 Monate
- ◆ Ziele:
 - ◆ Mit den Nachteilen von C++ aufräumen
 - ◆ Netzwerkorientiert
 - ◆ Klein, sicher und portabel
- ◆ Kein Happyend für das Projekt !?



1.1. Die Geschichte von Java

- ◆ Doch ein Happeyend durch das Internet – HotJava
- ◆ Was hat das ganz eigentlich mit Kaffee zu tun ?



1.2. Grundprinzipien: Objektorientierung

- ◆ Die Struktur von Objekten steht im Mittelpunkt
- ◆ Ein Programm besteht aus Objekten
- ◆ Vorteile der Kapselung:
 - ◆ übersichtlicher
 - ◆ wieder verwendbarer
 - ◆ sicherer
- ◆ Ausnahme in Java: primitive Datentypen

1.2. Grundprinzipien: Vererbung

- ◆ Konstruktoren, Attribute und Methoden werden an die abgeleitete Klasse weitergegeben
- ◆ Die Unterklasse kann erweitern oder überschreiben
- ◆ Keine Mehrfachvererbung in Java
- ◆ Stattdessen Implementierung von mehreren Interfaces
- ◆ Vererbungshierarchie:
 - `java.lang.Object`
 - `java.lang.Number`
 - `java.lang.Integer`

1.3. Polymorphie

- ◆ Enthaltende Polymorphie – Subtyping:
 - ◆ Typumwandlung – Casts
 - ◆ Überladen – von Methoden
- ◆ Parametrische Polymorphie – Generizität

2.1. Warum Generics?

- ◆ Typisches Einsatzgebiet Collection API
- ◆ Collections verwalten Daten des gleichen Typs – Object

```
ArrayList intList = new ArrayList();  
intList.add(new Integer(1));  
  
Integer num = (Integer) intList.get(0);
```

2.1. Warum Generics?

- ◆ unübersichtlicher Code durch Casts
- ◆ Casts können ClassCastExceptions verursachen
- ◆ Compiler kann Typsicherheit nicht sicherstellen

```
ArrayList intList = new ArrayList();  
intList.add(new Integer(1));  
  
String text = (String) intList.get(0);
```

2.2. Ein Beispiel

```
ArrayList<Integer> intList = new ArrayList<Integer>();  
intList.add(new Integer(1));  
  
Integer num = intList.get(0);
```

- ◆ Kein Cast – potenzielle Fehlerquelle umgangen
- ◆ Kann vom Compiler geprüft werden:
 - ◆ Compiler meldet Fehler beim Einstellen eines Elements von einem falschen Typ
 - ◆ Compiler kennt den Typ eines Listenelements beim Herausnehmen

2.3 Das Konzept von Generics

- ◆ Generizität bedeutet Verwendung von Typparametern

```
public interface List <E>{  
    void add (E x) ;  
    ...  
}
```

- ◆ Generische Klassen oder Methoden sind in Abhängigkeit eines Typparameters
- ◆ Begriffe
 - ◆ <E> formal type parameter
 - ◆ <Integer> actual type parameter
 - ◆ LinkedList<Integer> parameterized type

2.4 *Typeerasure*

- ◆ Möglichkeit der Umsetzung durch Kopie des Codes
- ◆ stattdessen Code Sharing
- ◆ JVM unterstützt nicht Generizität
- ◆ Dekompilierte Anweisung mit Casts
- ◆ Die Typinformation steht nur während der Kompilierung zur Verfügung

2.4 Typeerasure

- ◆ Compiler entfernt Typinformation = Typeerasure
- ◆ Bytecode mit Typ Object
- ◆ `ArrayList<Integer> == ArrayList<String>`
- ◆ gibt zur Laufzeit `true` zurück

2.4 Typeerasure

- ◆ Gründe für die Umsetzung mit Typeerasure:
 - ◆ Kompatibilität mit altem Code
 - ◆ Einfache Umsetzung von Generics
 - ◆ Keine Änderung der JVM nötig
 - ◆ Ein Nachteil entsteht nur bei der längeren Compilezeit, die meist irrelevant ist

3. *Das Beispielprogramm*

- ◆ Es soll ein Programm erstellt werden, das eine Liste von Zeichenelementen:
 - ◆ ausgibt mit `println`,
 - ◆ zeichnet,
 - ◆ um Elemente erweitert und
 - ◆ ein Zeichenelement aus der Liste löscht, wenn es einen bestimmten Punkt enthält.

3. Das Beispielprogramm

- ◆ In der Bibliothek gibt es bereits das Interface Shape, das verwendet werden soll, sowie diverse Implementierungen davon z.B. als Ellipse
- ◆ Die Methoden sollen polymorph sein
- ◆ Es soll möglich sein sicherzustellen, dass nur ein Typ von Zeichenelement in der Liste ist z.B. nur Rechtecke
- ◆ Als Listentyp soll ArrayList verwendet werden

4.1 Subtyping

```
ArrayList<Ellipse> ellipseList = new ArrayList<Ellipse>();  
ArrayList<Shape> shapeList = ellipseList;  
  
shapeList.add(new Rectangle());  
Ellipse ellipse = ellipseList.get(0);
```

- ◆ Zeile 2 ergibt einen Kompilierungsfehler !
- ◆ ArrayList<Ellipse> ist keine Subklasse von ArrayList<Shape>

4.2 Wildcards

```
public void printAll(AbstractList <Object> list) {  
    for (Object object : list)  
        System.out.println(object) ;  
}
```

- ◆ Nicht verwendbar für z.B. `ArrayList<Ellipse>`
- ◆ Wildcards lösen das Problem

```
public void printAll(AbstractList<?> list) {  
    for (Object object : list)  
        System.out.println(object) ;  
}
```

4.2 Wildcards

```
AbstractList<?> list = new ArrayList<Ellipse>();  
list.add (new Object());
```

- ◆ Nicht möglich
- ◆ Wildcards erlauben nur lesenden Zugriff
- ◆ Schreiben ist nicht möglich

4.3 Bounded Wildcards

- ◆ Rückgabebetyp Object
- ◆ Teilweise ist aber eine Einschränkung nötig um z.B. bestimmte Methoden anwenden zu können

```
public void drawAll(AbstractList<? extends Shape> list) {  
    for (Shape shape : list)  
        draw(shape);  
}
```

- ◆ Die Methode akzeptiert alle Arten von Shapelisten d.h. auch ein Liste mit einem Subtyp von Shape

4.3 Bounded Wildcards

- ◆ Upper bound durch Angabe von extends
- ◆ Mit upper bound aber nur lesender Zugriff
- ◆ Lower bound durch Angabe von super
- ◆ <? super shape> alle Supertypes werden akzeptiert
- ◆ Mit lower bound schreibender Zugriff

```
ArrayList<? super Ellipse> list;  
list.add(new Ellipse());
```

4.4 Generische Methoden

- ◆ Methoden können in Abhängigkeit von Typparametern sein

```
public <E extends Shape> E remove(ArrayList<E> list, Point point){
    for (E shape : list) {
        if (shape.contains(point)) {
            list.remove(shape);
            return shape;
        }
    }
    return null;
}
```

4.4 Generische Methoden

- ◆ Zur Laufzeit muss aber ein spezifischer Typ zugewiesen werden
- ◆ Der Compiler leitet aus den Parametern der Methode den spezifischen Typ ab: Typinferenz
- ◆ Elemente können der Collection auch hinzugefügt werden

```
public <E> void addElem(AbstractList<E> list, E elem) {  
    list.add(elem);  
}
```

4.5 Generische Methoden oder Wildcards ?

```
public void drawAll(AbstractList<? extends Shape> list){  
    for (Shape shape : list)  
        draw(shape);  
}
```

```
public <T> void drawAll(AbstractList<T extends Shape> list){  
    for(T shape: list)  
        draw(shape);  
}
```

- ◆ Grundsatz: Wenn ein Typparameter nur einmal verwendet wird, sollte man Wildcards benutzen.

4.6 Generische Klassen

- ◆ Collections sind Beispiele für generische Klassen

```
public interface List <E>{  
    void add (E x);  
    ...  
}
```

4.6 Generische Klassen

```
public class ShapeList<E extends Shape> extends ArrayList<E> {  
  
    public void drawAll( ) {  
  
        for (E shape : this)  
            draw(shape);  
    }  
  
    public E remove(Point point) {  
  
        for (E shape : this) {  
            if (shape.contains(point)) {  
                this.remove(shape);  
                return shape;  
            }  
        }  
        return null;  
    }  
  
}
```

4.6 Generische Klassen

- ◆ Es gilt das Gleiche wie für generische Methoden:
 - ◆ gebundene oder nicht gebundene Typparameter
 - ◆ mit einem oder mehreren Typparametern

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

4.6 Generische Klassen

```
public <E> void addNewElem(AbstractList<E> list) {  
    E elem = new E( );  
    list.add(elem);  
}
```

Funktioniert nicht

Neu Elemente des Parameters können nicht erzeugt werden

Zur Runtime fehlt die Typinformation durch Typeerasure

new E() wäre zur Runtime: new Object()

4.6 Generische Klassen

- ◆ Keine primitiven Datentypen
- ◆ Keine instanceof Prüfung möglich
- ◆ Generische Typen können nicht in statischen Elementen verwendet werden

```
public class myList<E> {  
    static E firstElem;  
    ...  
}
```

- ◆ extends, implements nicht mit typisierten Parametern

```
public class myList<E> extends E{...}  
public class myList<E> implements E{...}
```

5.1 Vergleich Arrays und generischen Collections

- ◆ Kovarianz: ein Typ kann spezialisiert werden
- ◆ Typparameter sind nicht kovariant
- ◆ Wir haben festgestellt:
 - ◆ `ArrayList<Object>` ist nicht Supertyp von `ArrayList<String>`
- ◆ Arrays sind kovariant:
 - ◆ `Object[]` ist Supertyp von `String[]`

5.1 Vergleich Arrays und generischen Collections

```
Object[ ] objArr;  
String[ ] strArr = new String[10];  
objArr = strArr;  
  
objArr[0] = new Integer(5);  
String string = strArr[0];
```

- ◆ Die letzte Zeile würde einem String einen Integer zuweisen
- ◆ Kommt aber nicht soweit:
 - ◆ vorletzte Zeile wirft `ArrayStoreException`
- ◆ Aber erst zur Runtime !

5.2 Arrays von parametrisierten Typen

```
ArrayList<Integer> [ ] intListArr = new ArrayList[10];  
Object [ ] objArr = intListArr;  
objArr[0] = new ArrayList<String>();
```

- ◆ Das Gleiche wie vorher ?
- ◆ Zur Runtime wird in der letzten Zeile keine Exception geworfen
- ◆ Es fehlt die Typ-Information wegen Typeerasure

5.2 Arrays von parametrisierten Typen

- ◆ Fehler taucht noch später auf
- ◆ Es kann z.B eine ClassCastException bei der Entnahme eines Elements geworfen werden
- ◆ Folge:
 - ◆ Der eigentliche Fehler ist schwer zu lokalisieren
 - ◆ Keine Typsicherheit trotz Generics

5.2 Arrays von parametrisierten Typen

```
ArrayList<Integer> [ ] intListArr = new ArrayList[10];
```

- ◆ Der Compiler gibt bei der Zeile schon eine unchecked Warnung

```
ArrayList<Integer>[ ] intListArr = new ArrayList<Integer>[10];
```

- ◆ Der Compiler wirft einen Fehler, dass er kein generisches Array erstellen kann.
- ◆ Das Gleiche gilt für Arrays von Typparametern.

```
public <T> T[ ] fromListToArray (AbstractList<T> list) {  
    T[ ] array = new T[10];  
    ...  
}
```

5.2 Arrays von parametrisierten Typen

```
List<?>[ ] array = new List<?>[10];  
List<Integer> intList = new ArrayList<Integer>( );  
intList.add(new Integer(2));  
  
array[1] = intList;  
Integer i = (Integer) array[1].get(0);
```

- ◆ Kann ClassCastException werfen
- ◆ Schlussfolgerung:
 - ◆ Man kann Arrays und generische Typen nicht typsicher kombinieren
 - ◆ Arrays passen so nicht in das Konzept von Generics und Typerasure

6.1 Sinnvoller Einsatz von Generics

- ◆ Grundsätzlich:
 - ◆ Generics sind eine Form von Polymorphie
 - ◆ Dieser ist immer dann sinnvoll, wenn der Code wieder verwendbar sein soll.
- ◆ Generics kontra Polymorphie durch Subtyping:
 - ◆ Generics für die Typsicherheit
 - ◆ Vorteil gegenüber Casts durch frühere Fehlererkennung

6.1 Sinnvoller Einsatz von Generics

- ◆ Der Entwickler wird nicht gezwungen Generics zu verwenden
- ◆ Eine unchecked Warnung weist darauf hin, dass die Typsicherheit nicht gewährleistet ist.

6.2 Fazit

- ◆ Generics in Java bringen praktisch keine Nachteile
- ◆ Das Wegfallen von Casts macht den Code übersichtlicher
- ◆ Bedingungen lassen sich besser und einfacher ausdrücken
- ◆ Fehler treten an den Stellen auf, wo sie verursacht werden
- ◆ In Bezug auf Arrays ist das Konzept noch nicht ausgereift