Weiterführende Konzepte des Programmierens mit Java
Generische Datentypen

HOCHSCHULE DER MEDIEN

Prof. Dr. Peter Thies
Prof. Dr. Christian Rathke
Hochschule der Medien (HdM)

{thies|rathke}@hdm-stuttgart.de
http://www.hdm-stuttgart.de/{thies|rathke}

Übersicht

- Generische Datentypen (*generics*) erleichtern das Aufspüren von Programmierfehlern zur Compilierzeit.
- Sie werden hauptsächlich in sog. *Collections* eingesetzt.

Generisch *(von lat. gigno 3. genui, genitus, ‚zeugen', ‚hervorbringen', ‚verursachen') ist die Eigenschaft eines materiellen oder abstrakten Objekts, insbesondere eines Begriffs, nicht auf Spezifisches, also auf unterscheidende Eigenheiten Bezug zu nehmen, sondern im Gegenteil sich auf eine ganze Klasse, Gattung oder Menge anwenden zu lassen bzw. eine solche gleichsam hervorzubringen oder stellvertretend dafür zu stehen.*
(Wikipedia)

Warum Generics?

- erlauben es Typen (Klassen und Interfaces) als Parameter bei der Definition von Klassen, Interfaces und Methoden zu verwenden.
- Stärkere Überprüfung von Typen zur Compile-Zeit.
- Vermeidung von Type-Casts:

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

kann mit Generics so formuliert werden:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```


Eine einfache Klasse für "Schachteln"

- Eine nicht(!) generische Implementierung der Klasse **Box** für Objekte beliebigen Typs mit Methoden zum Hinzufügen und Herausgeben eines Objekts:

```
public class Box {

    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }

}
```

- Wie kann man, je nach Bedarf, Instanzen von **Box** erzeugen, die nur Objekte eines bestimmten Typs aufnehmen dürfen?

Beispiel: Eine Box für Instanzen von Integer

```
public class BoxDemo1 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

- Die im Kommentar ausgedrückte Zusicherung ist für den Compiler nicht interpretierbar und führt im folgenden Beispiel erst zur Laufzeit (!) zu einem Fehler:

Fehler wird erst zur Laufzeit erkannt!

```
public class BoxDemo2 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        // Imagine this is one part of a large application
        // modified by one programmer.
        integerBox.add("10"); // note how the type is now String

        // ... and this is another, perhaps written
        // by a different programmer
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

```
Exception in thread "main"
        java.lang.ClassCastException:
           java.lang.String cannot be cast to java.lang.Integer
           at BoxDemo2.main(BoxDemo2.java:6)
```


Definition der Klasse Box mit Verwendung von Generics

```
public class Box<T> {

    private T something; // T stands for "Type"

    public void add(T thing) {
        something = thing;
    }

    public T get() {
        return something;
    }
}
```

- **`public class Box<T>`** erzeugt eine *generische* Typendeklaration (kann auch mit Interfaces verwendet werden)
- **`T`** ist eine sog. Typen-Variable (bzw. ein *formaler Typen-Parameter*); normalerweise ein Großbuchstabe
- Im Beispiel wurden alle Vorkommen von **`Object`** durch **`T`** ersetzt.

Generische Typen

- Der "Wert" von "T" kann ein Klassentyp, ein Interfacetyp oder selbst wieder eine Typenvariable sein (aber kein primitiver Typ)
- "Box<T>" ist ein *generischer* Typ, weil erst bei Verwendung T durch einen Wert ersetzt und damit der Typ festgelegt wird.
- Bei Verwendung erfolgt der *Aufruf des generischen Typs*, bei dem T durch einen konkreten Wert ersetzt wird, z.B.:

```
Box<Integer> integerBox;
```

- Der Aufruf eines generischen Typs wird auch *parametrisierter Typ* genannt.
- Bei Instanziierung wird das Typ-Argument angegeben, z.B.:

```
integerBox = new Box<Integer>();
```

Beispiel für die Verwendung

```
public class BoxDemo3 {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no cast!
        System.out.println(someInteger);
    }
}
```

- Ein Type Cast ist nicht (!) erforderlich.
- Bei fehlerhafter Verwendung wird ein Fehler zur Compilierzeit erzeugt, z,B.:

```
BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>
    cannot be applied to (java.lang.String)
        integerBox.add("10");
                   ^
    1 error
```


Regeln und Konventionen

- Typen-Variablen sind selbst keine Datentypen,
- generische Datentypen können mehr als einen Typen-Parameter haben (z.B.: `class name<T1, T2> { … }`),
- üblicherweise bestehen Typen-Parameter-Namen aus einem Großbuchstaben,
- die am meisten verwendeten Namen für Typen-Parameter:
  - E (Elementtyp, wird z.B. im Java Collections Framework verwendet)
  - K (Keytyp)
  - N (Numbertyp)
  - T (Typ)
  - V (Valuetyp)
  - S, U, V, etc. (zweite, dritte, vierte, usw. Typen)

Mehrere Typenparameter

- Beispiel: die generische „OrderedPair“-Klasse implementiert das generische „Pair“-Interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```


Mehrere Typenparameter (fortg.)

- Die folgenden Anweisungen erzeugen zwei Instantiierungen der Klasse „OrderedPair“:

```
Pair<String, Integer> p1 =
     new OrderedPair<String, Integer>("Even", new Integer(8));
Pair<String, String>  p2 =
     new OrderedPair<String, String>("hello", "world");
```

geht auch (wg. „Autoboxing“):

```
Pair<String, Integer> p1 =
     new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 =
     new OrderedPair<String, String>("hello", "world");
```

geht auch (wg. Typerschließung):

```
Pair<String, Integer> p1 = new OrderedPair<>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

Generische Methoden

- definieren ihre *eigenen* Typenparameter;
- der Gültigkeitsbereich ist auf die Methode begrenzt;
- die Typenparameter werden vor dem Rückgabetyp der Methode in spitzen Klammern angegeben:

```
public class Util {
   // Generic static method(2 type parameters)
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
   }
}
```


Generische Methoden mit generischer Klasse

```
public class Util {
   // Generic static method(2 type parameters)
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
   }}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

```
boolean same = Util.compare(p1, p2);
```


Generische Methoden und Konstruktoren

```
public class Box<T> {

    private T something;

    public void add(T thing) {
        someting = thing;
    }

    public T get() {
        return something;
    }

    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.<String>inspect("some text");
    }
}
```

Ausgabe des Programms:

```
T: java.lang.Integer
U: java.lang.String
```


Generische Methoden und Konstruktoren (fortg.)

- Weiteres Beispiel (auch für die Verwendung einer Typenvariable als Typenparameter):
  - Statische Methode, die ein Element in mehrere Schachteln steckt:

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
       for (Box<U> box : boxes) {
           box.add(u);
       }
   }
```

  - Nutzung:

```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;
Box.<Crayon>fillBoxes(red, crayonBoxes);
```

  - Wegen automatischer *Typenableitung* ebenso möglich:

```
Box.fillBoxes(red, crayonBoxes);
```

Begrenzte Typparameter

```
public class Box<T> {

    private T something;

    public void add(T thing) {
         something = thing;
    }

    public T get() {
        return something;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + something.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // error: this is a String!
    }
}
```

- Z.B. zur Einschränkung des Typparameters auf den Typ "Number" oder eines seiner Subtypen.
- Verwendung von *extends* nach dem Parameter

Beispielverwendung in der Main-Methode führt jetzt zum Compilerfehler

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

- *Mehrere einzuhaltende Typen* werden mit dem Zeichen & hinzugefügt:

```
<U extends Number & MyInterface>
```


Begrenzte Typparameter (fortg.)

- Begrenzte Typparameter erlauben den Aufruf von Methoden, die in der Typenbegrenzung definiert sind.

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```


Generische Methoden und begrenzte Typenparameter

- Mit begrenzten Typenparametern lassen sich vorteilhaft generische Algorithmen implementieren.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

- Problem: „>“ kann nicht auf Objekte angewendet werden

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

public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Vererbung und Subtypen

- Solange Datentypen kompatibel sind, können Zuweisungen von unterschiedlichen Objekttypen problemlos erfolgen, z.B.:

```
Object someObject = null;
Integer someInteger = new Integer(10);
someObject = someInteger; // OK
```

- Solche Beziehungen werden "ist ein" (is-a) Beziehungen genannt: Ein Integer *ist ein* Objekt bzw. "jeder Wert vom Typ **`Integer`** ist auch ein Wert vom Typ **`Object`**"
- Ebenso ist ein Integer oder ein Double auch eine Number:

```
public void someMethod(Number n){
    // method body omitted
}

someMethod(new Integer(10)); // OK
someMethod(new Double(10.1)); // OK
```


Generics, Vererbung und Subtypen (fortg.)

- Dasselbe gilt auch für diese Verwendung von Generics:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

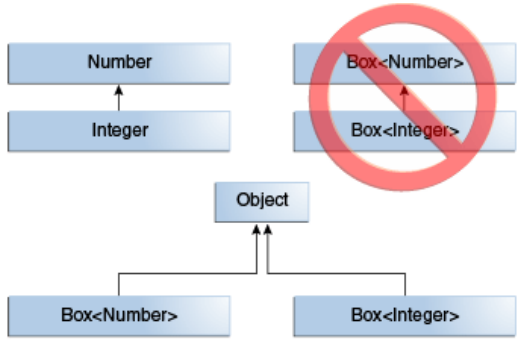
- Wie verhält es sich aber mit Subtypen der generischen Typen selbst? Welche Werte können z.B. als Argument für die folgende Methode verwendet werden?

```
public void boxTest(Box<Number> n){
        // method body omitted
    }
```

- Kann die Methode z.B. mit einer Instanz von **`Box<Integer>`** aufgerufen werden?
- Die Antwort ist "nein", denn **`Box<Integer>`** oder **`Box<Double>`** sind *keine* Subtypen von **`Box<Number>`**.

Generics, Vererbung und Subtypen (fortg.)

- Box<Integer> ist *kein* Subtyp von Box<Number> obwohl Integer ein Subtyp von Number ist.



Subtypen generischer Typen folgen nicht der Typenhierarchie ihrer Parameter!

```
// A cage is a collection of things, with bars to keep them in.
interface Cage<E> extends Collection<E> {}
```

```
interface Lion extends Animal {}
Lion king = ...;

Cage<Lion> lionCage = ...;
lionCage.add(king);
```

```
interface Butterfly extends Animal {}
Butterfly monarch = ...;

Cage<Butterfly> butterflyCage = ...;
butterflyCage.add(monarch);
```

```
Cage<Animal> animalCage = ...;
animalCage.add(king);
animalCage.add(monarch);
```

```
animalCage = lionCage;     // compile-time error
animalCage = butterflyCage; // compile-time error
```

Subtypen generischer Typen folgen nicht der Typenhierarchie ihrer Parameter! (fortg.)

- **`Cage<Animal>`** bedeutet "Alle-Tiere-Käfig"; daher kann **`Cage<Lion>`** ("Löwenkäfig") auch nicht Subtyp sein, denn folgende Bedingung wäre verletzt:
  - Wenn x (z.B. der Wilhelma-Löwenkäfig) ein Wert vom Typ A (z.B. Löwenkäfig) ist, dann ist x auch Wert vom Supertyp (z.B. "Alle-Tiere-Käfig") von A.

- Benötigt wird ein generischer Typ für "Irgendein-Tier-Käfig":

```
Cage<? extends Animal> someCage = ...;
```

- **`Cage<Lion>`** und **`Cage<Butterfly>`** sind Subtpyen von **`Cage<? extends Animal>`**

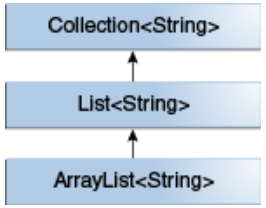
```
someCage = lionCage; // OK
someCage = butterflyCage; // OK
```


Generics und Untertypen

- Man kann den Untertyp einer generischen Klasse oder eines generischen Interfaces *explizit* durch Angabe von „extends“ oder „implements“ bilden.

```
List<E> extends Collection<E>
ArrayList<E> implements List<E>
```

- Solange der Parametertyp gleich bleibt, wird die Untertypenbeziehung automatisch hergestellt.
- Ein weiterer Untertyp kann hinzudefiniert werden, z.B.:




```
interface Payload<E,P> extends List<E> {
   void setPayload(int index, P val);
   …
}
```


Erweiterungen

- Ableiten von Datentypen
- Platzhalter mit Typeneinschränkung und Untertypen
- Auslöschen generischer Typen und Methoden
- Einschränkungen für generische Datentypen

Ableiten von Datentypen

- Der Java-Compiler kann oft Typargumente aus anderen beteiligten Datentypen ableiten, so dass man sie dann nicht angeben muss.
- Beispiel:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Ein Datentyp des zweiten Arguments ist „Serializable“

Ableiten von Datentypen in generischen Methoden

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
             boxContents.toString() + "]");
      counter++;
    }
  }

  public static void main(String[] args) {
    java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
      new java.util.ArrayList<>();
    BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
    BoxDemo.outputBoxes(listOfIntegerBoxes);
  }
}
```

Typenbestätigung

Typenableitung

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```


Ableitung von Datentypen und Instanziierung generischer Klassen

- Solange der Compiler die Datentypen von Konstruktorargumenten ableiten kann, darf der Konstruktor ohne Parametrisierungen, aber mit dem „Diamant“ verwendet werden, z.B. statt

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

kann

```
Map<String, List<String>> myMap = new HashMap<>();
```

Der „Diamant“

verwendet werden.

- Das Weglassen des Diamanten führt zu einer „unchecked conversion“ Warnung

Ableitung von Datentypen in generischen Konstruktoren von generischen und nicht-generischen Klassen

- Kontruktoren können Typparameter deklarieren, sowohl von generischen als auch nicht generischen Klassen.

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

```
new MyClass<Integer>("")
```

- Der Datentyp „String“ für den Argumentdatentyp des Konstruktors kann abgeleitet werden.
- In diesem Beispiel kann zusätzlich „Integer“ für den Typenparameter „X“ von „MyClass“ abgeleitet werden:

```
MyClass<Integer> myObject = new MyClass<>("");
```


Zieldatentypen

- Ein Zieldatentyp eines Ausdrucks ist der Datentyp, den der Java-Compiler in Abhängigkeit vom Verwendungsort erwartet.
- z.B. Beispiel ist die Klassemethode „Collections.emptyList“ wie folgt definiert:

```
static <T> List<T> emptyList();
```

- Der Zieldatentyp im folgenden Ausdruck ist „List<String>“:

```
List<String> listOne = Collections.emptyList();
```

Also kann der Datentyp „String“ für „T“ abgeleitet werden.

Wildcards (Platzhalter)

- Das als Platzhalter (*wildcard*) bezeichnete Fragezeichen (?) kennzeichnet einen unbekannten Datentyp.
- Es kann als Datentyp für einen Parameter, für ein Attribut oder für eine lokale Variable verwendet werden.
- Es kann nicht als Typargument eines generischen Methodenaufrufs, bei der Instanziierung einer generischen Klasse oder als Supertyp verwendet werden.

Durch Super-Datentypen eingeschränkte Platzhalter (Upper Bound Wildcards)

- Beispiel: Methodenparameter, der für `List<Integer>`, `List<Double>` und `List<Number>` zulässig ist:
  `List<? extends Number>`
- Beispiel:

weniger restriktiv als List<Number>

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```


Nicht eingeschränkte Platzhalter

- … werden durch das Fragezeichen gekennzeichnet, z.B. wie in `List<?>`
- Dies wird bezeichnet als „Liste eines unbekannten Datentyps“.
- Beispiel:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

- Dies ist weniger restriktiv als die Verwendung von `<List<Object>>`, weil jeder(!) Datentyp der Form `<List<A>>` Subtyp von `<List<?>>` ist.

Durch Sub-Datentypen eingeschränkte Platzhalter (Lower Bounded Wildcards)

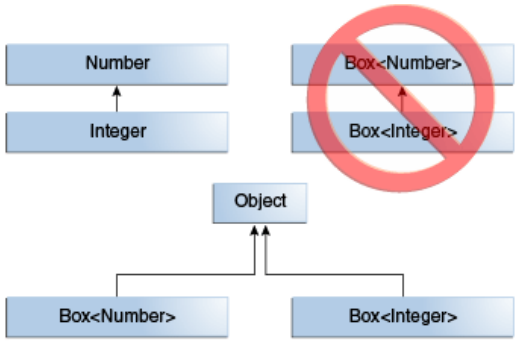
- … wird durch Verwendung von `super` nach dem Platzhaltersymbol gekennzeichnet.
- Beispiel:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

- Erlaubt Listen, deren Elemente vom Datentyp Integer und Super-Datentypen von Integer sind.

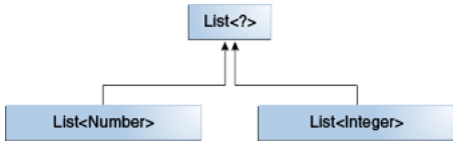
Platzhalter und Sub-Datentyen

- Box<Integer> ist *kein* Subtyp von Box<Number> obwohl Integer ein Subtyp von Number ist.



Platzhalter und Sub-Datentypen

- Durch die Verwendung von ? werden folgende Beziehungen ermöglicht:



- `List<Number>` und `List<Integer>` stehen in keiner direkten Beziehung.
- Die folgenden Beziehungen existieren aufgrund der Tatsache, dass `Number` Super-Datentyp von `Integer` ist: