

# Generische Klassen&Interfaces

Lukas Abelt

lukas.abelt@airbus.com

DHBW Ravensburg  
Wirtschaftsinformatik

Ravensburg  
25. März 2019

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Java-Klassenbibliothek

## Kurzüberblick

- Java bietet Vielzahl an „fertigen“ Klassen
- Zusammengefasst in Packages
- Diese implementieren Standardfunktionalitäten wie z.B.
  - Ein- und Ausgabefunktionalitäten
  - Grafische Oberflächen
  - Netzwerkkommunikation
  - Datum- und Zeit, Internationalisierung
  - Und viele mehr...

# Wichtige Packages I

... der Klassenbibliothek

Die wichtigsten Standardpackages im schnellen Überblick:

- ▣ *java.lang*: Integriert die fundamentalen Klassen, die in der Regel immer zur Java Entwicklung benötigt werden wie zum Beispiel String Object oder auch die Wrapper Klassen der primitiven Datentypen (Integer, Boolean, Double usw.). **Muss nicht explizit importiert werden!**
- ▣ *java.util*: Häufig benötigte Klassen, wie Listenstrukturen (List, Stack), Klassen zur Verarbeitung von Datum und Uhrzeit (Calendar) oder Zufallszahlengeneratoren (Random)
- ▣ *java.io*: Klassen zur Ein- und Ausgabe über Streams
- ▣ *java.net*: Klasse zur Implementierung von Netzwerkkommunikation

# Wichtige Packages II

## ... der Klassenbibliothek

- *java.rmi*: Klassen zur Entwicklung verteilter Programme unter Nutzung von Remote Method Invocation
- *java.awt*: Grundlegendes Package für die Entwicklung grafischer Oberflächen
- *java.swing*: Erweiterte Komponente zur Entwicklung von grafischen Oberflächen. Baut auf *java.awt* auf, bietet jedoch mehr Funktionalität
- *javax.crypto* und *java.security*: Klassen zur Umsetzung von sicherheitsrelevanten Aspekte (Zugriffsschutz, Rechteverwaltung etc.)
- *java.sql*: Package zur Interaktion mit SQL Datenbanken

# Arbeiten mit der Klassenbibliothek

- Packages über `import` in Klasse einbinden
- Oracle bietet umfangreiche Dokumentation zu allen Klassen der Standard-API
- Schnellster Weg zur Doku:
  - In den meisten IDEs sowieso integriert
  - Sonst Google: „Java 8/9/10 API *PackageName*“

# API-Dokumentation

## Links

### Java 8 API

<https://docs.oracle.com/javase/8/docs/api/index.html>

### Java 9 API

<https://docs.oracle.com/javase/9/docs/api/index.html>

### Java 10 API

<https://docs.oracle.com/javase/10/docs/api/index.html>



# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Generische Klassen

## Was ist das und Warum?

- Ist eine Methode Klassen deutlich versatiler zu machen
  - Und dadurch wiederverwenbar
  - Bei geringerem Implementierungsaufwand
- Bisher: Festlegen von Datentypen bei Design der Klasse
- Jetzt: Festlegen von Datentypen bei Verwendung der Klasse (Zumindest für einige)
- Funktioniert für:
  - Member Variablen
  - Funktionsparameter
  - Rückgabewerte

# Generische Klassen

## Beispiel: Benannte Werte

- Angenommen man erhält folgende Anforderungen für eine Klasse
  - Die Klasse soll einen Wert speichern
  - Dieser soll vom Typ `Integer`, `String` oder `Boolean` sein
  - Die Klasse soll einen Namen für diesen Wert als `String` speichern können
- Mögliche Ansätze ohne generische Klassen:
  - Implementierung einer Klasse `NamedValue`, die drei Member der entsprechenden Typen hat
  - Implementierung einzelner Klassen `NamedInteger`, `NamedString` und `NamedBoolean`

# Variante 1: One class for all!

## Implementierung

```
1  class NamedValue{
2      private Integer intValue;
3      private String stringValue;
4      private Boolean boolValue;
5      private String name;
6
7      void set(Integer newInt);
8      void set(String newString);
9      void set(Boolean newBool);
10     void setName(String newName);
11
12     Integer getIntegerValue();
13     String getStringValue();
14     Boolean getBooleanValue();
15     String getName();
16 }
```

# Variante 1: One class for all!

## Probleme

- Es sollte **ein** Wert gespeichert werden
  - Unsere Klasse speichert (theoretisch) bis zu drei verschiedene Werte
  - *Könnte* man abfangen
  - Erhöht jedoch weiter den Implementierungsaufwand
- Keine einheitliche Schnittstelle um Wert abzurufen
- Erhöhter Aufwand bei Erweiterung der Klasse
  - Wert soll jetzt auch von Typ `Color` sein
  - Hinzufügen neuer Member Variable `colorValue`
  - Hinzufügen neuer set-/get-Methoden

# Variante 2: Viel hilft viel!

## Implementierung NamedInteger

```
1  class NamedInteger{
2      private Integer value;
3      private String name;
4
5      void set(Integer newValue);
6      Integer get();
7
8      void setName(String newName);
9      String getName();
10 }
```

# Variante 2: Viel hilft viel!

## Implementierung NamedString

```
1  class NamedString{
2      private String value;
3      private String name;
4
5      void set(String newValue);
6      String get();
7
8      void setName(String newName);
9      String getName();
10 }
```



# Variante 2: Viel hilft viel!

## Implementierung NamedBoolean

```
1  class NamedString{
2      private Boolean value;
3      private String name;
4
5      void set(Boolean newValue);
6      Boolean get();
7
8      void setName(String newName);
9      String getName();
10 }
```

# Variante 2: Viel hilft viel!

## Probleme

- ❑ Löst einige Probleme der ersten Variante...
  - ❑ Tatsächlich nur ein Wert gespeichert
  - ❑ Einheitliche Schnittstelle
- ❑ ...Aber eben nicht alle
- ❑ Copy-Paste-Code → Nach Möglichkeit zu vermeiden
- ❑ Problem bei Erweiterung bleibt ähnlich
  - ❑ Würde hier neue Klasse erfordern

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- **Syntax&Eigenschaften**
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Generische Klassen

## Die Lösung des Problems!

- Angabe von „Platzhaltern“ bei Definition der Klasse
  - ▣ Namen sind theoretisch beliebig wählbar
  - ▣ ...Es gibt jedoch Naming conventions dazu
- Diese repräsentieren den Datentypen
- Die Spezifizierung des Typs erfolgt erst bei Deklaration einer Variable vom Typ der Klasse

# Generische Klassen

## Syntax

```
1  class NamedValue<T>{
2      private T value;
3      private String name;
4
5      void set(T newValue);
6      T get();
7
8      void setName(String newName);
9      String getName();
10 }
11 //Verwendung:
12 NamedValue<Integer> namedInteger;
13 NamedValue<String> namedString;
14 NamedValue<Boolean> namedBoolean;
```

# Eigenschaften von generischen Klassen I

- Schnittstellen bleiben einheitlich (Im Rahmen des spezifizierten Typs)
- Kein Problem mit Anpassungen an neue Typen → Keine Änderung notwendig
- Definieren mehrerer generischer Typen möglich:

```
1 class name<T1, T2, ..., Tn> {/*Klasseninhalt*/}
```

# Eigenschaften von generischen Klassen II

- Naming Conventions für Typen:
  - In der Regel ein Buchstabe
  - T - Type
  - E - Element
  - N - Number
  - K - Key
  - V - Value

# Generische Klassen

## Begrenzen von Typen

- Problem: Generics bieten nur begrenzte Funktionalität
  - Garantiert sind nur Methoden die in `Object` implementiert sind
  - `equals()`, `toString()`, `hashCode()`...
- In manchen Fällen werden aber mehr Funktionalitäten benötigt
- Hierfür lassen sich Generics einschränken
  - Nach benötigter Superklasse
  - Nach benötigten Interfaces



# Begrenzung von Typen

## Beispiel 1: Spezifizieren der Superklasse

```
1  class BoxedNumber<T extends Number>{
2      private T number;
3
4      void set(T newNumber);
5      T get();
6  }
7  //Verwendung:
8  BoxedNumber<Integer> boxInt = new BoxedNumber<>();           //
   ↳ OK, Integer erbt von Number
9  BoxedNumber<Double> boxDbl = new BoxedNumber<>();           //
   ↳ OK, Double erbt von Number
10 BoxedNumber<String> error = new BoxedNumber<>();             //
    ↳ Compilerfehler, String ist keine Unterklasse von
    ↳ Number
```

# Begrenzung von Typen

## Beispiel 2: Spezifizieren von Interfaces

```
1 class BoxedComparable<T extends Comparable<T>>{
2     private T value;
3     /* set()-/get()-Methoden */
4     boolean isSmaller(T other){
5         return value.compareTo(other)<0;
6     }
7 }
8 //Verwendung:
9 BoxedComparable<Integer> boxInt;           //OK, Integer
10    ↪ implementiert Comparable
11 BoxedComparable<String> boxString;         //OK
12 BoxedComparable<Color> error;              //Compilerfehler
```

**Achtung:** Auch für Interfaces wird in diesem Fall das keyword `extends` genutzt!

# Begrenzung von Typen

## Beispiel 3: Spezifizieren von Superklasse&Interfaces

```
1  class ComparableNumber<T extends Number&Comparable<T>>{
2      private T value;
3
4      void set(T newValue);
5      T get();
6      boolean isSmaller(T other){
7          return value.compareTo(other)<0;
8      }
9  }
10 //Verwendung:
11 ComparableNumber<Integer> compInt;           //OK
12 ComparableNumber<Double> compDbl;           //OK
13 ComparableNumber<AtomicInteger> error;       //Compilerfehler
```

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- **Generische Interfaces**
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Generische Interfaces

...Gibt es natürlich auch noch

- Das ganze funktioniert analog mit Interfaces
- Klassen die das Interface implementieren müssen nicht generisch sein
- Spezifizierung des Typen erfolgt bei Implementierung
  - Entweder direkte Angabe des Typen
  - ...oder „durchreichen“ von Typparametern der Klasse
- Bekanntester Vertreter aus der Standardbibliothek: `Comparable<T>`

# Implementierung generischer Interfaces

## Beispiel 1: Direkte Angabe des Typen

```
1  class BoxedInt implements Comparable<Integer>{
2      private Integer number;
3
4      void set(Integer newNumber);
5      Integer get();
6
7      int compareTo(Integer i){
8          return number.compareTo(i)
9      }
10 }
```

# Implementierung generischer Interfaces

## Beispiel 2: Durchreichen von Typparametern

```
1  class BoxedValue<T> implements Comparable<T>{
2      private T value;
3
4      void set(T newValue);
5      T get();
6
7      int compareTo(T o){
8          if(number.hashCode()<i.hashCode()){
9              return -1;
10         } else if(number.hashCode()>i.hashCode()){
11             return 1;
12         } else{
13             return 0;
14         }
15     }
16 }
```

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

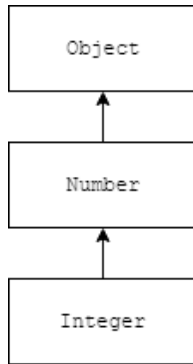


# Vererbung

## Kurze Wiederholung

- ▣ Objekte von Subklassen können einem Objekt der Superklasse zugewiesen werden
- ▣ So kann ein Integer beispielsweise einem Object zugewiesen werden

```
1 Integer someInt = new Integer(42);  
2 Object someObject = new Object();  
3 someObject = someInt;    //OK
```



# Vererbung

## In generischen Klassen

- Gleiches gilt natürlich auch für den generischen Typ selbst
- Zum Beispiel bei Funktionsaufrufen können auch Objekte der Subklasse übergeben werden:

```
1  class BoxValue<T>{
2      private T val;
3
4      void set(T t);
5      T get();
6  }
7  //Erlaubte Verwendung:
8  BoxValue<Number> boxNum = new BoxValue();
9  boxNum.set(new Double(2.34));    //OK
10 boxNum.set(new Integer(7));      //OK
```

# Vererbung

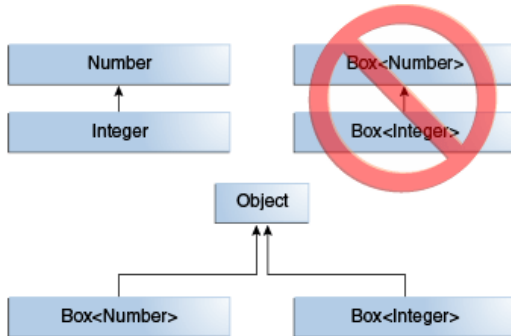
## Von generischen Klassen

```
1 void someMethod(BoxValue<Number> box){/* ... */}
```

- Welche Typen werden akzeptiert?
  - BoxValue<Number>
  - BoxValue<Integer>?
  - BoxValue<Double>?
- **NUR** BoxValue<Number> wird als Typ akzeptiert
- Integer erbt zwar von Number...
  - ...Aber BoxValue<Number> und BoxValue<Integer> stehen in keiner Beziehung zueinander!

# Vererbung

## Von generischen Klassen



Quelle: <https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html>

<https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html>

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- **Wildcards**
- Einschränkungen
- Aufgaben

# Wildcards

Was ist das jetzt schon wieder?

- ❑ Weiteres Werkzeug für generischen Code, wenn der Typ nicht explizit bekannt sein muss
- ❑ Dargestellt über `<?>` im Code
- ❑ Wildcards können nach „oben“ oder „unten“ (Im Sinne der Vererbungshierarchie) eingeschränkt werden
- ❑ Vereinfacht die Implementierung von Methoden, die Generische Klassen verwenden
- ❑ Wird **nicht** verwendet für:
  - ❑ Erzeugung von Objekten einer generischen Klasse
  - ❑ Als Typargument bei Aufruf generischer Methoden
  - ❑ Zur spezifizierung eines Supertyps

# Unbounded Wildcards

- Schränken den Typ nicht ein
- Nützlich, wenn:
  - Die Methode sich allein über die Funktionalitäten der `Object` Klasse realisieren lässt
  - Die verwendeten Methode der generischen Klasse nicht vom Typ abhängig sind
    - Zum Beispiel: `List.size()` oder `List.clear()`

# Unbounded Wildcards

## Codebeispiel

```
1 public static void printList(List<?> list){  
2     for (Object elem : list){  
3         System.out.println(elem.toString);  
4     }  
5 }
```



# Upper-Bounded Wildcards

Du und alle unter dir

- Schränkt die Wildcard auf eine Klasse sowie alle ihre Subklassen ein
- Definition erfolgt über die Nutzung des `extends` Keywords
  - Beispiel: `List<? extends Number>`
- Upper-Bounded Wildcards stellen sicher, dass Methoden der Superklasse verwendet werden können
- Vereinfachen somit Implementierung von Methoden, die für eine gesamte „Klassenfamilie“ funktionieren sollen (Beispielsweise alle `Number` Klassen)

# Upper-Bounded Wildcards

Codebeispiel: Summe einer Liste

```
1 public static double sumOfList(List<? extends Number> list)
    ↪ {
2     double s = 0.;
3     for (Number n : list){
4         s += n.doubleValue();
5     }
6     return s;
7 }
```

# Lower-Bounded Wildcards

Du und alle über dir!

- ▣ Schränkt die Wildcard auf eine Klasse und all ihre Superklassen ein
- ▣ Definition erfolgt über das Keyword `super`
  - ▣ Beispiel `List<? super Double>`

# Upper-Bounded Wildcards

Codebeispiel: Summe einer Liste

```
1 public static void addNumbers(List<? super Integer> list) {  
2     for (int i = 1; i <= 10; i++) {  
3         list.add(i);  
4     }  
5 }
```

# Wildcards

## Guidelines zur Benutzung

- ❑ Wildcards sollten nicht in Rückgabewerten verwendet werden
- ❑ Generell lassen sich die zu verwendenden Wildcards nach Art der Variable unterscheiden:
  - ❑ `in` Variable: Upper-Bounded
    - Falls ausschließlich Methoden aus `Object` benötigt werden: Unbounded
  - ❑ `out` Variable: Lower-Bounded
  - ❑ Wenn die Variable sowohl als `in` und `out` verwendet wird: Keine Wildcard nutzen

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- **Einschränkungen**
- Aufgaben

# Einschränkungen von Generics

## Keine Verwendung von primitiven Typen

- Typparameter müssen immer Klassen sein
- Dadurch keine Nutzung von primitiven Typen bei Instanziierung möglich
- Stattdessen müssen die entsprechenden Wrapper-Klassen (Integer, Boolean, Double etc.) verwendet werden
- Beispiel: Pair Klasse

# Einschränkungen von Generics

## Beispiel

```
1  class Pair<K, V>{
2      private K key;
3      private V value;
4
5      public Pair(K key, V value) {
6          this.key = key;
7          this.value = value;
8      }
9
10     // ...
11 }
12
13 Pair<int, char> p = new Pair<>(8, 'a'); //Compilerfehler
14 Pair<Integer, Character> p = new Pair<>(8, 'a'); //OK
```



# Einschränkung von Generics

## Instanziieren von Typparametern

- Instanziieren von Typparametern ist nicht erlaubt
- Kann umgangen werden über Reflection API

```
1 public <E> void append(List<E> list){
2     E elem = new E();    //Compilerfehler
3     list.add(elem);
4 }
5 {public <E> void append(List<E> list, Class<E> cls)
6     ↪ throws Exception{
7     E elem = cls.newInstance();    //OK
8     list.add(elem);
9 }
10 //Verwendung:
11 List<String> ls = new ArrayList<>();
12 append(ls, String.class);}
```

# Einschränkungen von Generics

## Statische Variablen

- Typparameter können nicht für statische Variablen verwendet werden
- Kommt dadurch, dass die Variable für alle Instanzen der Klasse gleich sein muss
- Unterschiedliche Instanzen der Klasse könnten sonst die Variable unterschiedlich typisieren

```
1  class StaticGeneric<T>{  
2      private static T var;  
3      //...  
4  }  
5  StaticGeneric<Integer> intS = new StaticGeneric();  
6  StaticGeneric<String> strS = new StaticGeneric();  
7  StaticGeneric<Boolean> boolS = new StaticGeneric();
```

# Einschränkungen generischer Klassen

## Weitere Einschränkungen

- ❑ Keine Arrays von parametrisierten Typen möglich
  - Runtime könnte bestimmte Fehlerfälle nicht mehr korrekt abfangen
- ❑ Generische Klassen können nicht direkt oder indirekt von `Throwable` ableiten
- ❑ `catch` und `throw` können nicht mit parametrisierten Typen verwendet werden
  - `throws` **jedoch schon!**
- ❑ Funktionssignaturen können nicht allein durch einen anderen Typargument überladen werden

# Inhalt

## 1 Java-Klassenbibliothek

## 2 Generische Klassen

- Motivation
- Syntax&Eigenschaften
- Generische Interfaces
- Vererbung in Generischen Klassen
- Wildcards
- Einschränkungen
- Aufgaben

# Kontakt

- E-Mail: [lukas.abelt@airbus.com](mailto:lukas.abelt@airbus.com)
- GitHub: <https://www.github.com/LuAbelt>
- GitLab: <https://www.gitlab.com/LuAbelt>
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt