Programmierkurs Java

UE 23 - Generics

Dr.-Ing. Dietrich Boles

Gliederung

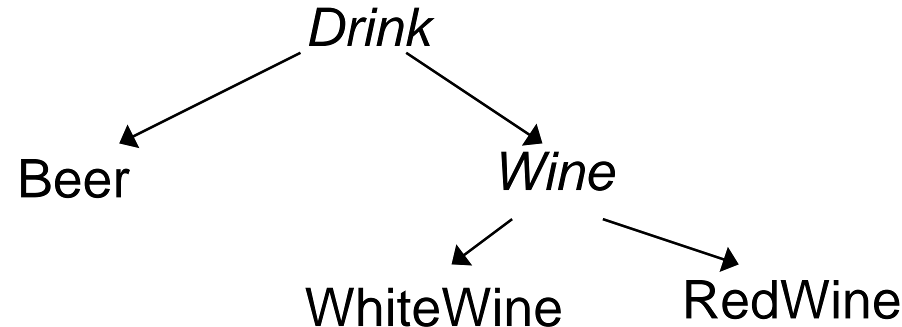
- Generics / Motivation

- Generische Klassen
  - Klassendefinition
  - Typ-Parameter mit Einschränkungen
  - Wildcards
  - Einschränkungen

- Generische Methoden

Motivation / Problem

- Simulation eines Getränkehandels
- Quelle: J. Nowak: Fortgeschr. Programmierung mit Java 5,dpunkt.
- Gegeben: vielerlei Getränke



- Gewünscht: typisierte Flaschen, die nur mit Bier, Rotwein oder Weißwein gefüllt werden können

Motivation / Ausgangslage

```
abstract class Drink { }

class Beer extends Drink {
  private String brewery;
  public Beer(String brewery) { this.brewery = brewery; }
  public String getBrewery()  { return this.brewery; }
  public String toString()    { return this.getClass().getName() +
                                   "[" + this.brewery + "]"; }
}

abstract class Wine extends Drink {
  private String region;
  public Wine(String region) { this. region = region; }
  public String getRegion()  { return this.region; }
  public String toString()   { return this.getClass().getName() +
                                   "[" + this.region + "]"; }
}

class WhiteWine extends Wine {
  public WhiteWine(String region)  { super(region); }
}

class RedWine extends Wine {
  public RedWine(String region)    { super(region); }
}
```

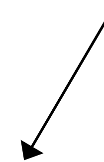
Motivation / erster Lösungsversuch

```
class Bottle {
  private Drink content;
  public boolean isEmpty() {
    return this.content == null;
  }
  public void fill(Drink content) {
    this.content = content;
  }
  public Drink empty() {
    Drink content = this.content;
    this.content = null;
    return content;
  }
}
...
Bottle beerBottle = new Bottle();
beerBottle.fill(new WhiteWine("Burgunder"));
...
Beer beer = (Beer)beerBottle.empty(); // ClassCastException!
```

Lösung: polymorphe Klasse

Probleme:

- keine korrekte Lösung
- Typ-Unsicherheit



Motivation / zweiter Lösungsversuch

```
abstract class DrinkBottle {}

class BeerBottle extends DrinkBottle {
  private Beer cont;
  public void fill(Beer content) { this.cont = content;  }
  public Beer empty() { Beer c=this.cont; this.cont=null; return c; }
}

abstract class WineBottle extends DrinkBottle {}

class WhiteWineBottle extends WineBottle {
  private WhiteWine cont;
  public void fill(WhiteWine content) { this.cont = content; }
  public WhiteWine empty() { WhiteWine c=this.cont; this.cont=null; return c; }
}

class RedWineBottle extends WineBottle {
  private RedWine cont;
  public void fill(RedWine content) { this.cont = content; }
  public RedWine empty() { RedWine c = this.cont; this.cont = null; return c; }
}
```

Lösung: Getränk-spezifische Flaschen
Problem: Keine Vererbung, keine wirkliche Polymorphie!

Motivation / Lösung (1)

```
class Bottle<T> {

  private T content;

  public boolean isEmpty() { return this.content == null; }
  public void fill (T content) { this.content = content; }
  public T empty() {
    T content = this.content;
    this.content = null;
    return content;
  }
}
```

Lösung:

- Generische (parametrisierte) Klasse
- **T** ist formaler Typ-Parameter (Typ-Variable) der Klasse **Bottle**

Motivation / Lösung (2)

```
Bottle<Beer> beerBottle = new Bottle<Beer>();
Bottle<WhiteWine> whiteWineBottle = new Bottle<WhiteWine>();
```

```
beerBottle.fill(new Beer("Veltins"));
beerBottle.fill(new WhiteWine("Burgunder"));
   // Fehlermeldung durch Compiler!
whiteWineBottle.fill(new WhiteWine("Burgunder"));

Beer beer = beerBottle.empty ();
System.out.println (beer);
WhiteWine whiteWine = whiteWineBottle.empty ();
System.out.println (whiteWine);
```

Generische Klassen / Klassendefinition

```
class C<T>    { ... }
```

T kann im weiteren Klassenkopf und im Klassenrumpf (fast) überall da verwendet werden, wo Klassennamen stehen können

```
class D<T> extends T { }
class E<T> { T obj; }

interface I<T1, T2> {
  public void setze(T1 obj1, T2 obj2);
}

class F<T4, T5, T6> implements I<T4, T6> {
  public T5 liefere() { ...}
  public void setze(T4 obj1, T6 obj2) { ... }
}
```

Generische Klassen / Objekte und Objektvariablen

```
class C<T> {
```

Formaler Typ-Parameter

```
  public T f(T t) {...}
}

class A { }
class B { }
```

Parametrisierte Klasse / Typ

```
C<A> obj = new C<A>();
```

Aktueller Typ-Parameter (Klasse, kein Standarddatentyp)

```
C<B> obj2 = new C<B>();
A a = obj.f(new A());
B b = obj2.f(new B());
A a2 = obj.f(new B()); // Compilierfehler
```

Generische Klassen / Übersetzung

```
class C<T> {
  T t;
  T f(T t) {
    T t1 = t; this.t = t1;
    String str = t.toString();
    return this.t;
  }
}
```

↓ Compiler

```
class C {
  Object t;
  Object f(Object t) {
    Object t1 = t; this.t = t1;
    String str = t.toString();
    return this.t;
  }
}
```

+ Meta-Informationen!

Raw-Type

Daher erlaubt:

```
C obj = new C();
```

Typ-Parameter mit Einschränkungen

- Motivationsproblem: **`Bottle<Object>`** ist keine Getränkeflasche!
- Lösung: Einschränkung des Typ-Parameters

```
class Bottle<T extends Drink> { /* wie auf Folie 7 */ }

class Petrol { }
class Stout extends Beer { ... }

Bottle<Drink> drinkBottle = new Bottle<Drink>();
Bottle<Beer> beerBottle = new Bottle<Beer>();
Bottle<Stout> stoutBottle = new Bottle<Stout>();

new Bottle<Object>();   // Compilierfehler
new Bottle<Petrol>();   // Compilierfehler
```

Typ-Parameter mit Einschränkungen / Übersetzung

```
class A {
  public void doIt() {...}
}

class C<T extends A> {
  T t;
  T f(T t) {
    T t1 = t;
    this.t = t1;
    this.t.doIt();
    return this.t;
  }
}
```

Compiler →

```
class C {
  A t;
  A f(A t) {
    A t1 = t;
    this.t = t1;
    this.t.doIt();
    return this.t;
  }
}
```

Arrays mit parametrisierten Klassen (1)

- Motivationsproblem: Klasse für Getränke-spezifische Getränkekästen
- Problem: Arrays mit parametrisierten Klassen sind nicht erlaubt

```
class BottleBox<T extends Drink> {

  private Object[] bottles; private int count = 0;

  public BottleBox(int number) {
    // this.bottles = new T[number]; nicht erlaubt
    this.bottles = new Object[number];
  }
  public void add(Bottle<T> bottle) {
    this.bottles[this.count] = bottle; this.count++;
  }
  public Bottle<T> getBottle(int index) {
    return (Bottle<T>)this.bottles[index];
  }
}
```

Internes sicheres (!) Cast

Arrays mit parametrisierten Klassen (2)

```
BottleBox<Beer> beerBottleBox = new BottleBox<Beer>(6);
// Bierkasten füllen
for (int i = 0; i < 6; i++) {
  Bottle<Beer> beerBottle = new Bottle<Beer>();
  beerBottle.fill(new Beer("Jever"));
  beerBottleBox.add(beerBottle);
}
Bottle<RedWine> wineBottle = new Bottle<RedWine>();
beerBottleBox.add(wineBottle); // Compilierfehler
...
// Bierkasten leeren
for (int i = 0; i < 6; i++) {
  Bottle<Beer> beerBottle = beerBottleBox.getBottle(i);
  Beer beer = beerBottle.empty ();
  System.out.println(beer);
}
```

Wildcards (1)

CARL VON OSSIETZKY universität OLDENBURG

- Motivationsproblem: Klasse für Getränkekästen mit beliebigen Flaschen
- Lösung: „Wildcards“

```
class BottleBox {
  private Object[] bottles;  private int count = 0;

  public BottleBox(int number) {
    this.bottles = new Object[number];
  }
  public void add(Bottle<? extends Drink> bottle) {
    this.bottles[this.count] = bottle; this.count++;
  }
  public Bottle<? extends Drink> getBottle(int index) {
    return (Bottle<? extends Drink>)this.bottles[index];
  }
}
```

Wildcards (2)

```
BottleBox box = new BottleBox(6);

// Füllen
Bottle<Beer> beerBottle = new Bottle<Beer>();
beerBottle.fill(new Beer("Veltins"));
box.add (beerBottle);

Bottle<WhiteWine> whiteWineBottle = new Bottle<WhiteWine>();
whiteWineBottle.fill (new WhiteWine("Burgunder"));
box.add (whiteWineBottle);

// Leeren
for (int i = 0; i < 6; i++) {
  Bottle<? extends Drink> bottle = box.getBottle (i);
  Drink drink = bottle.empty ();
  System.out.println(drink);
}
```

Wildcards (3)

- Motivationsproblem: Klasse für Getränke-spezifische Getränkekästen
- Problem der Lösung auf Folie 14: `BottleBox<Beer>`

  ↖ Semantische Ungereimtheit

```
class BottleBox<T extends Bottle<? extends Drink>> {

  private Object[] bottles; private int count = 0;

  public BottleBox(int number) {
    this.bottles = new Object[number];
  }
  public void add(T bottle) {
    this.bottles[this.count] = bottle; this.count++;
  }
  public T getBottle(int index) {
    return (T)this.bottles[index];
  }
}
```

Wildcards (4)

```
BottleBox<Bottle<Beer>> beerBottleBox =
  new BottleBox<Bottle<Beer>>(6);

// Bierkasten füllen
for (int i = 0; i < 6; i++) {
  Bottle<Beer> beerBottle = new Bottle<Beer>();
  beerBottle.fill(new Beer("Jever"));
  beerBottleBox.add(beerBottle);
}
...
// Bierkasten leeren
for (int i = 0; i < 6; i++) {
  Bottle<Beer> beerBottle = beerBottleBox.getBottle(i);
  Beer beer = beerBottle.empty ();
  System.out.println(beer);
}
```

Generics / Einschränkungen

- Keine Arrays vom formalen Typ-Parameter:

```
new T[number] // Compilierfehler
```

- Keine Objekte vom formalen Typ-Parameter:

```
new T() // Compilierfehler
```

- Keine statischen Attribute vom formalen Typ-Parameter

```
static T t; // Compilierfehler
```

Generische Methoden / Motivation (1)

- Motivationsproblem: Umfüllen von Flaschen
- Mögliche Lösung:

```
class BottleTransfuser<T extends Drink> {
  void transfuse(Bottle<T> fromBottle,
                 Bottle<T> toBottle) {
    T drink = fromBottle.empty();
    toBottle.fill(drink);
  }
}
BottleTransfuser<Beer> beerTransfuser = new
  BottleTransfuser<Beer>();
...
```

- **Problem: beerTransfuser kann nur Bierflaschen umfüllen!**

Genererische Methoden / Motivation (2)

- allgemeinere Lösung: Generische Methoden

```
class BottleTransfuser {
    <T extends Drink> void transfuse (Bottle<T> fromBottle,
                                      Bottle<T> toBottle) {
    T drink = fromBottle.empty();
    toBottle.fill(drink);
  }
}

BottleTransfuser transfuser = new BottleTransfuser ();

Bottle<Beer> b1 = new Bottle<Beer>(); b1.fill(new Beer("Jever"));
Bottle<Beer> b2 = new Bottle<Beer>();
transfuser.transfuse(b1, b2);

Bottle<WhiteWine> b3 = new Bottle<WhiteWine>();
b3.fill(new WhiteWine("Burgunder"));
Bottle<WhiteWine> b4 = new Bottle<WhiteWine>();
transfuser.transfuse(b3, b4);

transfuser.transfuse(b1, b4); // Fehlermeldung
```

Generische Methoden / Beispiele

```
class A { public void f() {} }
class B extends A { }

class C {
  public <T> void f1(T t) {
    String str = t.toString();
  }

  public static <T extends A> void f2(T t) {
    t.f();
  }
}

C obj = new C();      C.f2(new A());
obj.f1("Hallo");      C.f2(new B());
obj.f1(4711);         C.f2("Hallo"); // Compilierfehler
```

Zusammenfassung

- Generics: Parametrisierte Klassen
- Sinn und Zweck:
  - Semantische Korrektheit
  - Vermeidung von Type-Casts
  - Vermeidung von ClassCast-Exceptions