Java 5 Typsystem

Generische Klassen und ihre Auswirkungen auf das Typsystem

Sebastian Buchwald

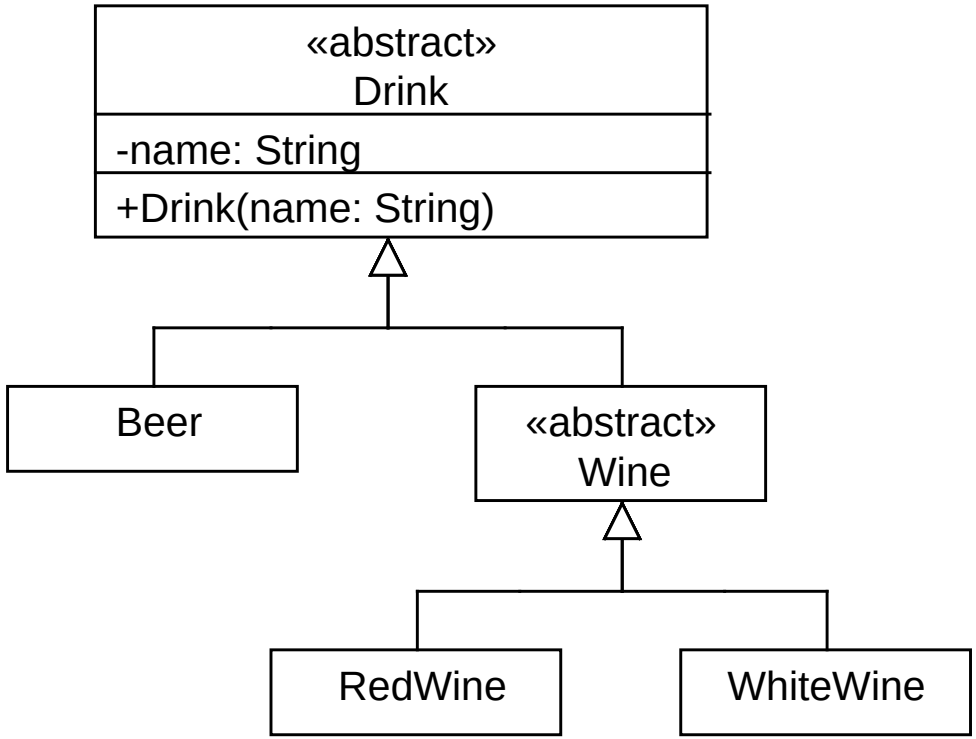
Universität Karlsruhe

12. Juli 2007

Gliederung

Gliederung

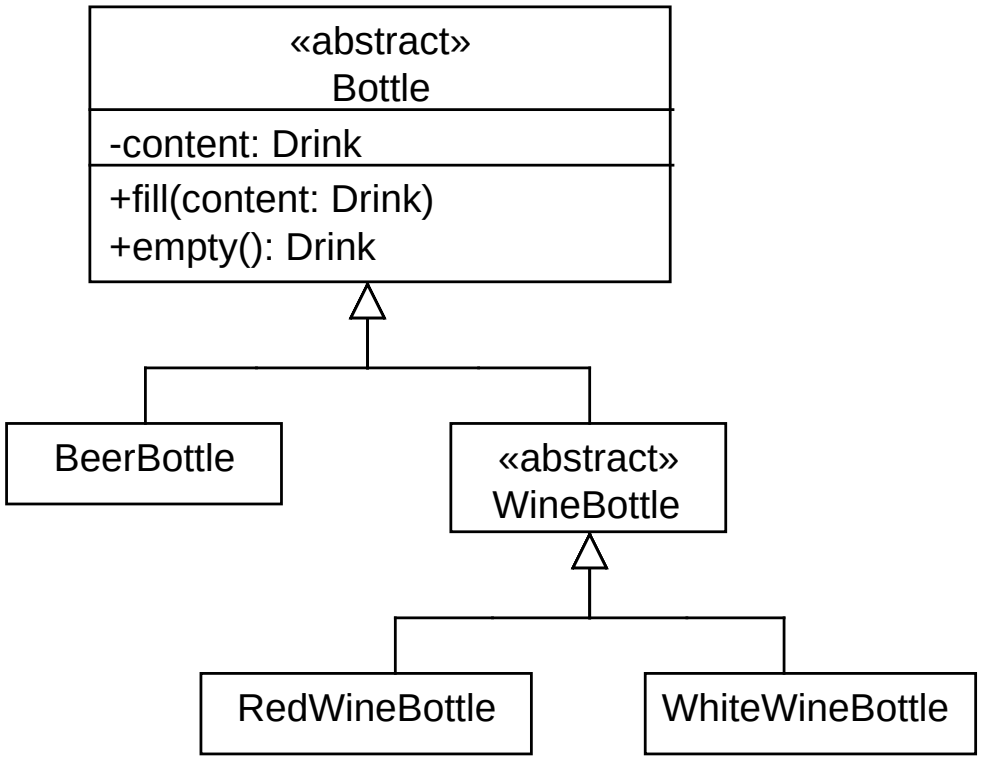
Betrachte folgende Klassenhierarchie



Wir wollen für jedes Getränk eine passende Flasche haben.

- in Bierflaschen darf nur Bier gefüllt werden
- in Rotweinflaschen nur Rotwein
- usw.

Wir müssen für jede Unterklasse von `Drink` eine eigene passende `Bottle`-Klasse anlegen.



Lösung mit Generics

- die Klassen unterscheiden sich nur durch die Art des Getränks
- generische Klassen lösen dieses Problem
- die Getränke-Klasse wird als Parameter übergeben

```
class Bottle<T> {
  private T content;

  public T empty(){...}
  public void fill(T drink){...}
}
```

Beispiel

```
Bottle<Beer> beerBottle = new Bottle<Beer>();
Bottle<Wine> wineBootle = new Bottle<Wine>();

Beer beer = new Beer("Hoepfner");
RedWine red = new RedWine("Rot");
WhiteWine white = new WhiteWine("Weiss");

beerBottle.fill(beer);
wineBootle.fill(red);
wineBootle.fill(white);
Wine wine = wineBootle.empty();
```

Vorteile von Generics

- weniger Klassen
- weniger Fehler
- Typprüfung vom Compiler, statt zur Laufzeit
- keine Casts mehr nötig

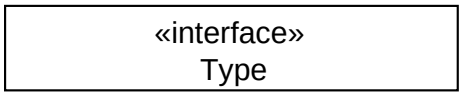
Gliederung

Konventionen

- Klassen: A, B, C, ...
- Interfaces: I, I1, I2, ...
- Typvariablen: T, T1, T2, ...

Was ist überhaupt ein Typ?

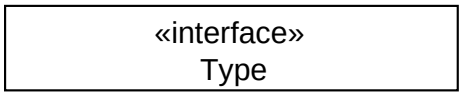
Man wird in java.lang.reflect fündig:



- Subinterfaces
  - GenericArrayType
  - ParameterizedType
  - TypeVariable<D>
  - WildcardType
- Implementing Classes
  - Class

Was ist überhaupt ein Typ?

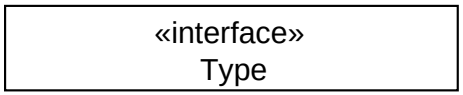
Man wird in java.lang.reflect fündig:



- Subinterfaces
  - GenericArrayType
  - ParameterizedType
  - TypeVariable<D>
  - WildcardType
- Implementing Classes
  - Class

Was ist überhaupt ein Typ?

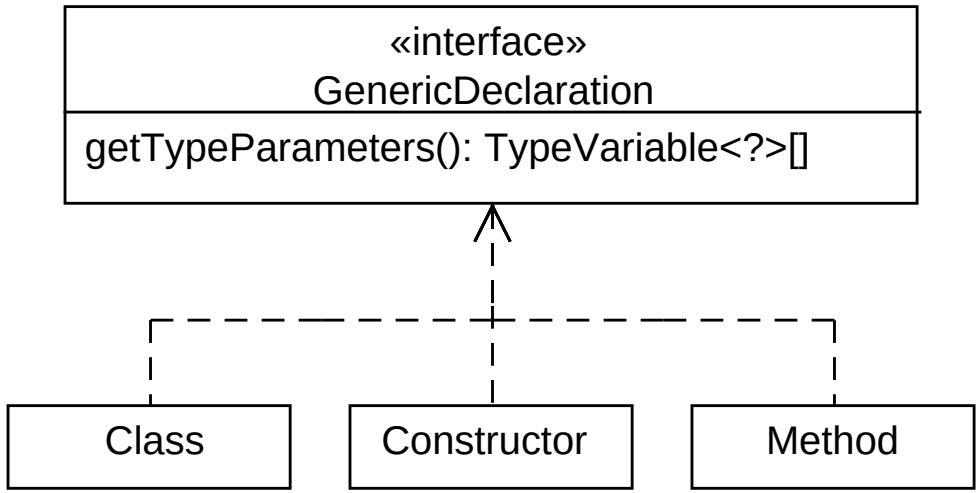
Man wird in java.lang.reflect fündig:



- Subinterfaces
  - GenericArrayType
  - ParameterizedType
  - TypeVariable<D>
  - WildcardType
- Implementing Classes
  - Class

GenericDeclaration

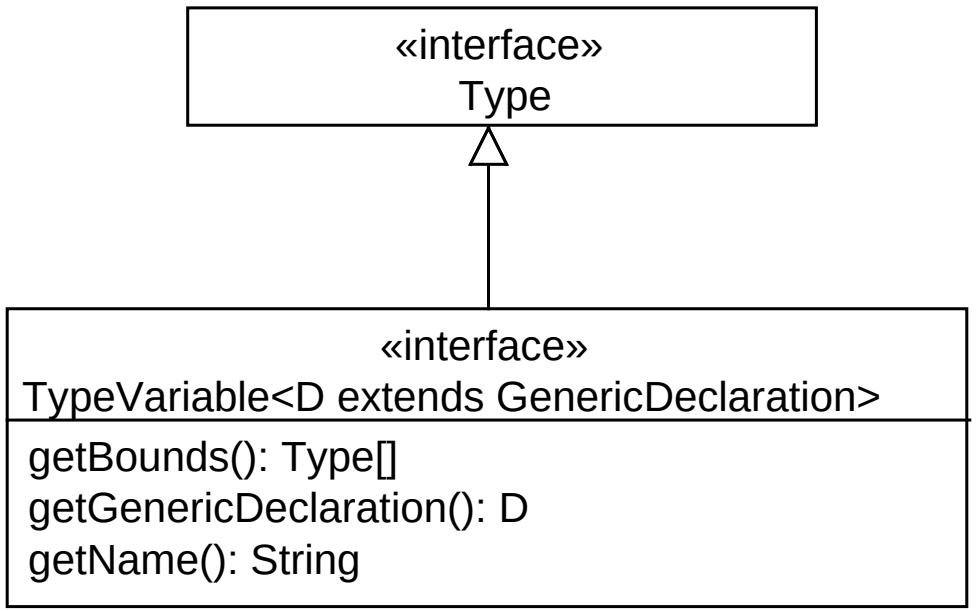
Interface für Entities die Typvariablen deklarieren können.



Beispiel

```
// Parametrisierte Klasse
public class C<T> {
    private T t;

    // Parametrisierte Methode
    public <T1> String foo(T1 t1) {
        return t.toString() + t1.toString();
    }
}
```

TypeVariable<D extends GenericDeclaration>



Einschränkungen einer Typvariable

```
public class D<T extends C & I1 & I2> {
    ...
}
```

Die Typvariable T wird durch die Klasse C und die beiden Interfaces I1 und I2 eingeschränkt.
`D<T>` entspricht `D<T extends Object>`.

Beispiel

Für

```
public class D<T extends C & I1 & I2> {
    ...
}
```

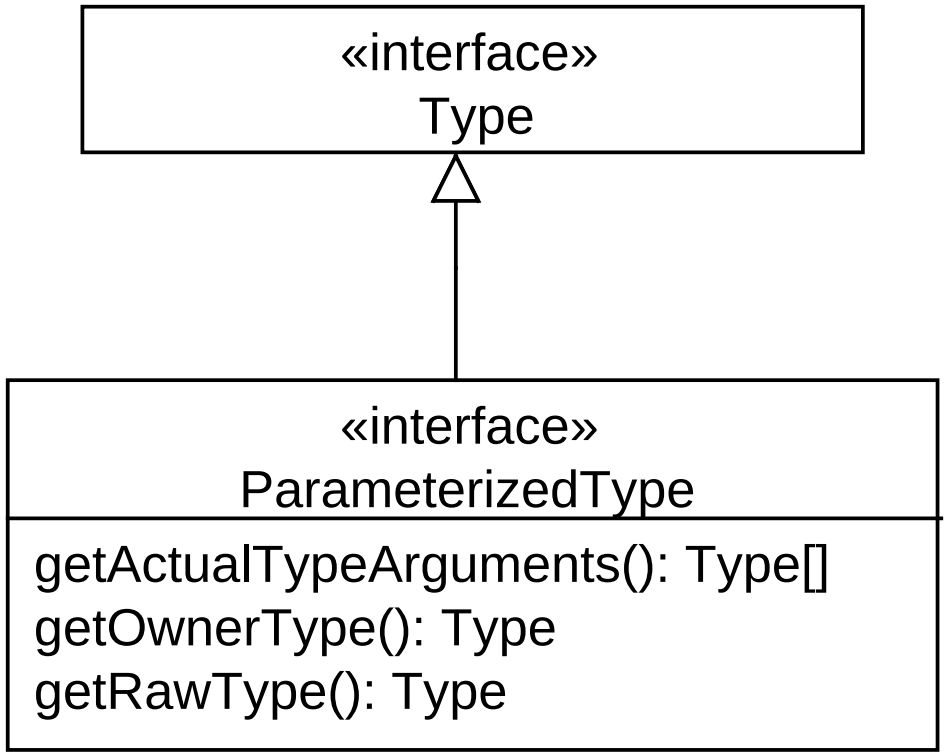
erhalten wir für das zu T gehörige TypeVariable-Objekt

```
getBounds() = {C.class, I1.class, I2.class}
```

```
getGenericDeclaration() = D.class
```

```
getName() = "T"
```

Interface ParameterizedType



Beispiel

Für

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

lässt sich schreiben

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

Dabei ist `C<A>` ein parametrisierter Typ.

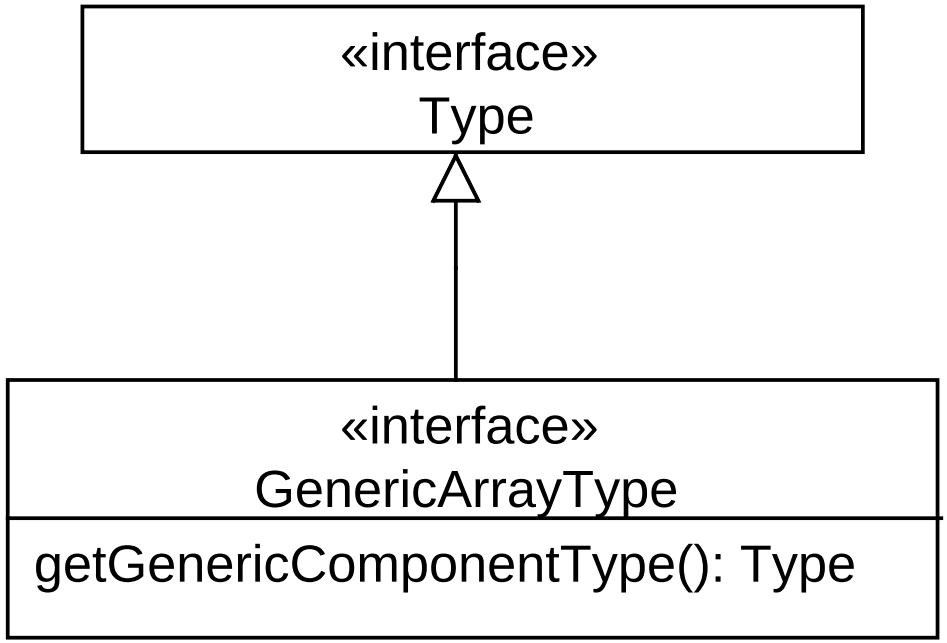
Das zu `C<A>` gehörige ParameterizedType-Objekt liefert

```
getActualTypeArguments() = {A.class}
```

```
getOwnerType() = null
```

```
getRawType() = C.class
```

Interface GenericArrayType



Für

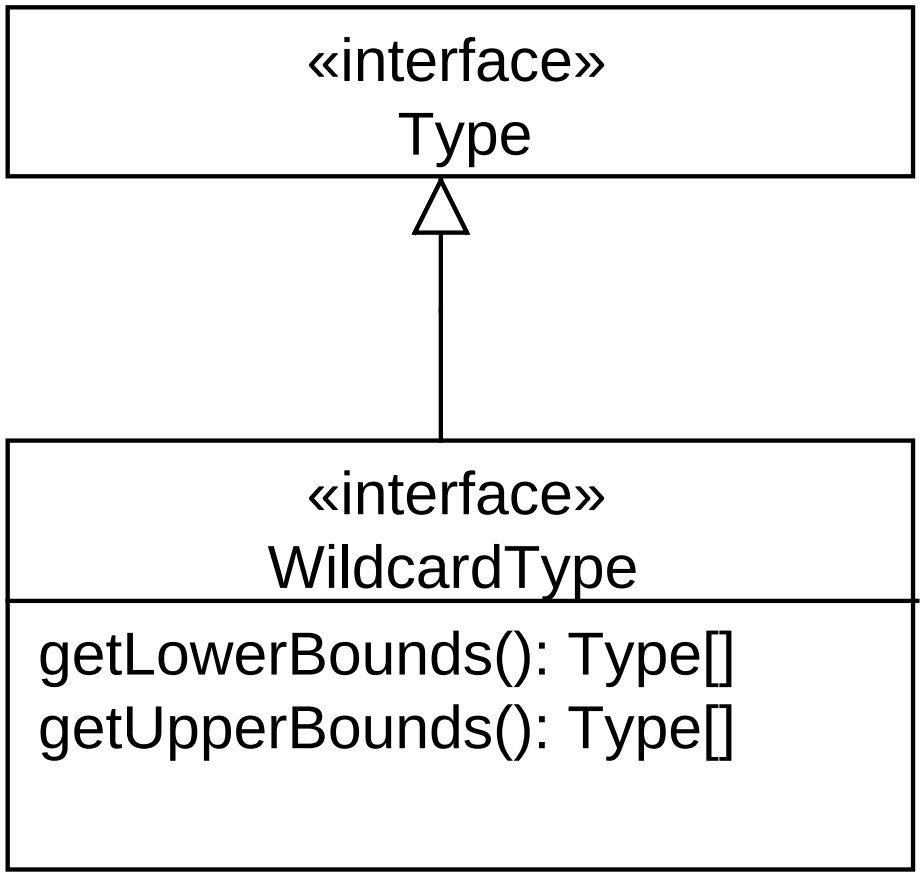
```
public class C<T> {
    ...
}
```

lässt sich schreiben

```
C<A>[] ca = null;
```

Dabei ist `C<A>[]` generischer Array-Typ.

Interface WildcardType



1. Beispiel

Wildcard-Typen ähneln den parametrisierten Typen, allerdings haben sie „irgendeinen“ Typ als aktuelles Argument. Für

```
public C<T> {}
```

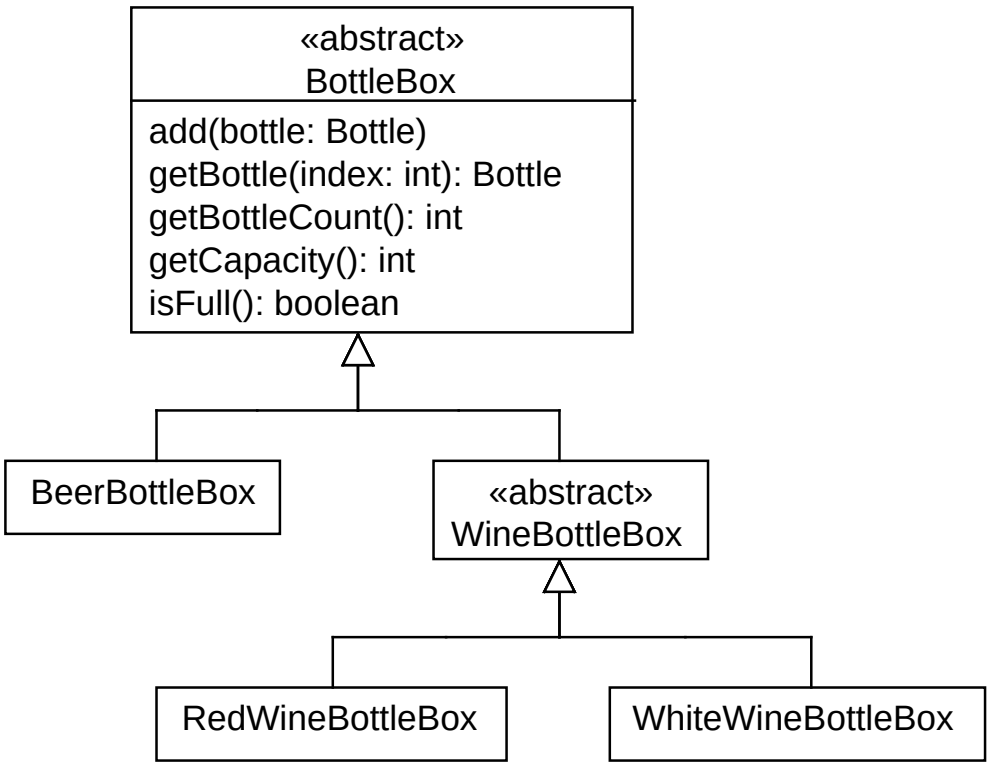
sind die zu

```
C<?> cWild;
C<? extends A> cWildExt;
C<? super B> cWildSup;
```

gehörigen Type-Objekte vom Wildcard-Typ.

2. Beispiel

Wir wollen nun Getränkekästen für unsere Flaschen modellieren.

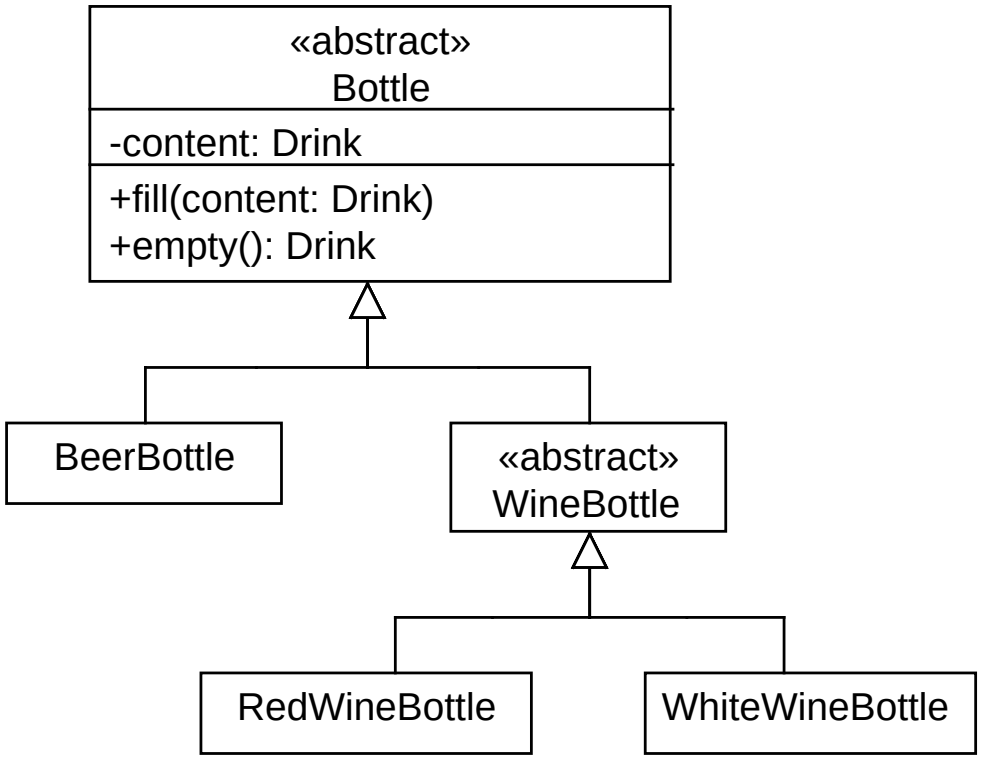


```
public class
BottleBox<T extends Bottle<? extends Drink>> {
    private Object[] bottles;
    private int count = 0;
    ...
    public void add(T bottle) {
        bottles[count] = bottle;
        count++;
    }

    public T getBottle(int index) {
        return (T) this.bottles[index];
    }
}
```

Gliederung

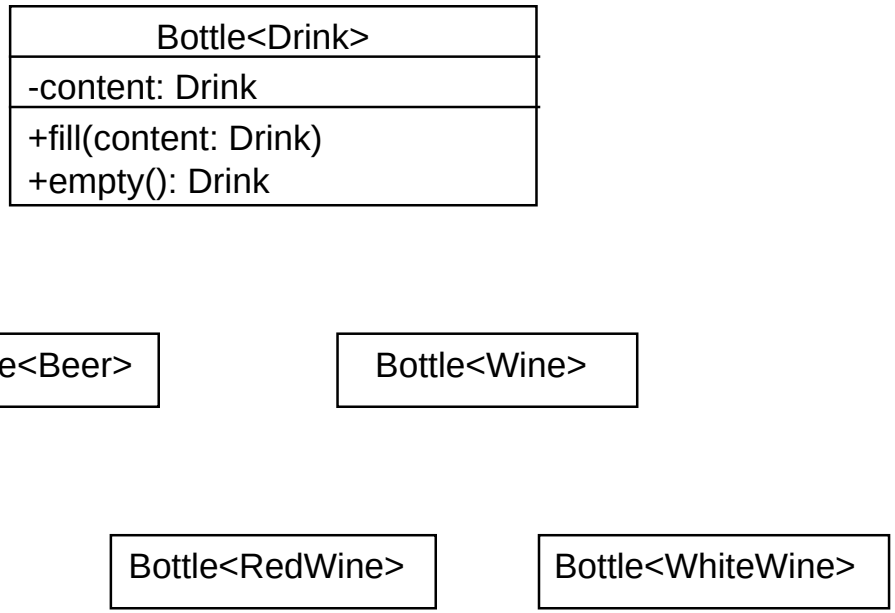
Vererbung bei parametrisierten Typen

Erinnern wir uns an unser Flaschen-Beispiel.



Vererbung bei parametrisierten Typen

Erinnern wir uns an unser Flaschen-Beispiel.



In der generische Lösung existiert keine Vererbungshierarchie.

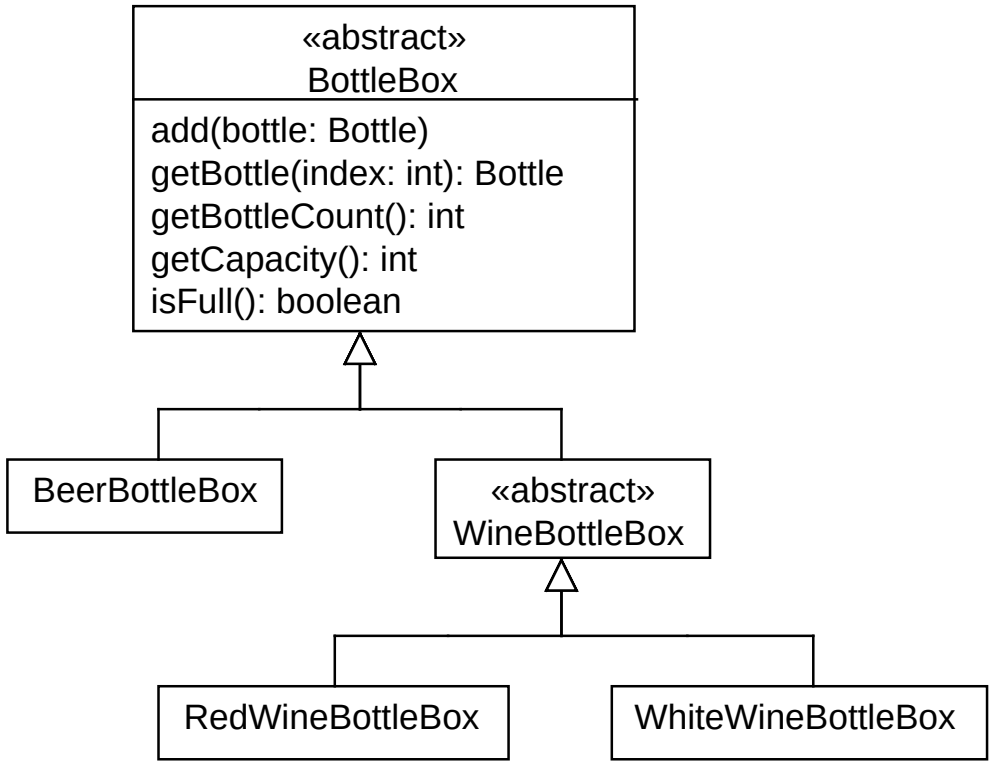
Was wäre wenn doch?

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

RedWine red = new RedWine("Rot");

drinkBottle = beerBottle;

drinkBottle.fill(red);
```

Wir können nun Wein in eine Bierflasche füllen und sind somit nicht mehr typsicher.

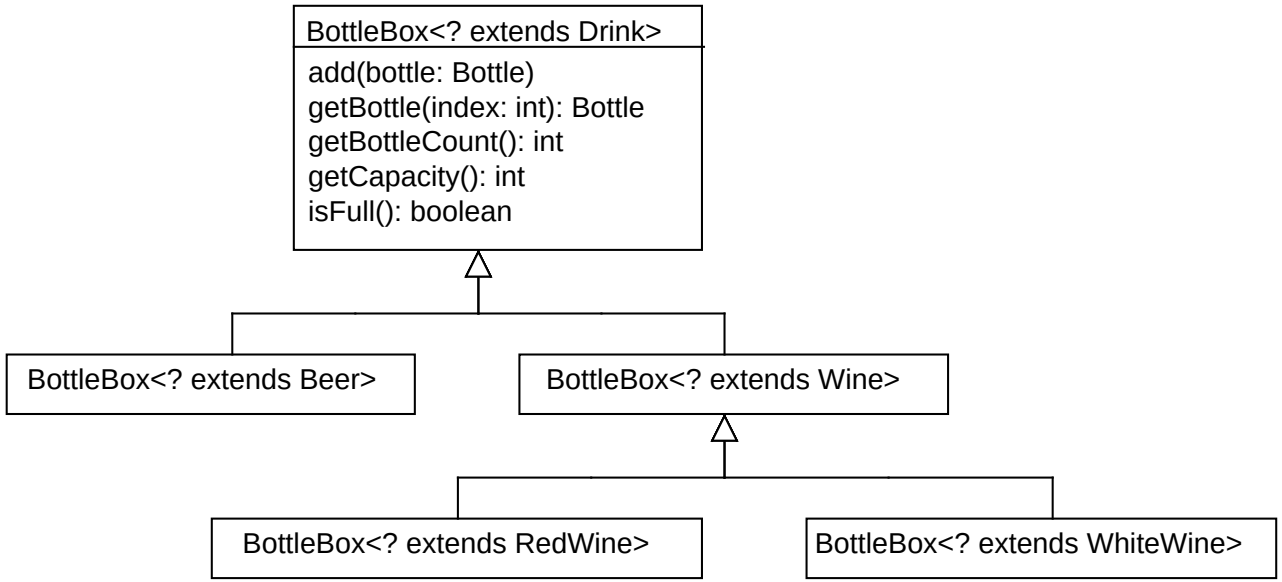
Vererbung bei Wildcard-Typen

Unsere generischen Getränkekästen basieren auf Wildcards.



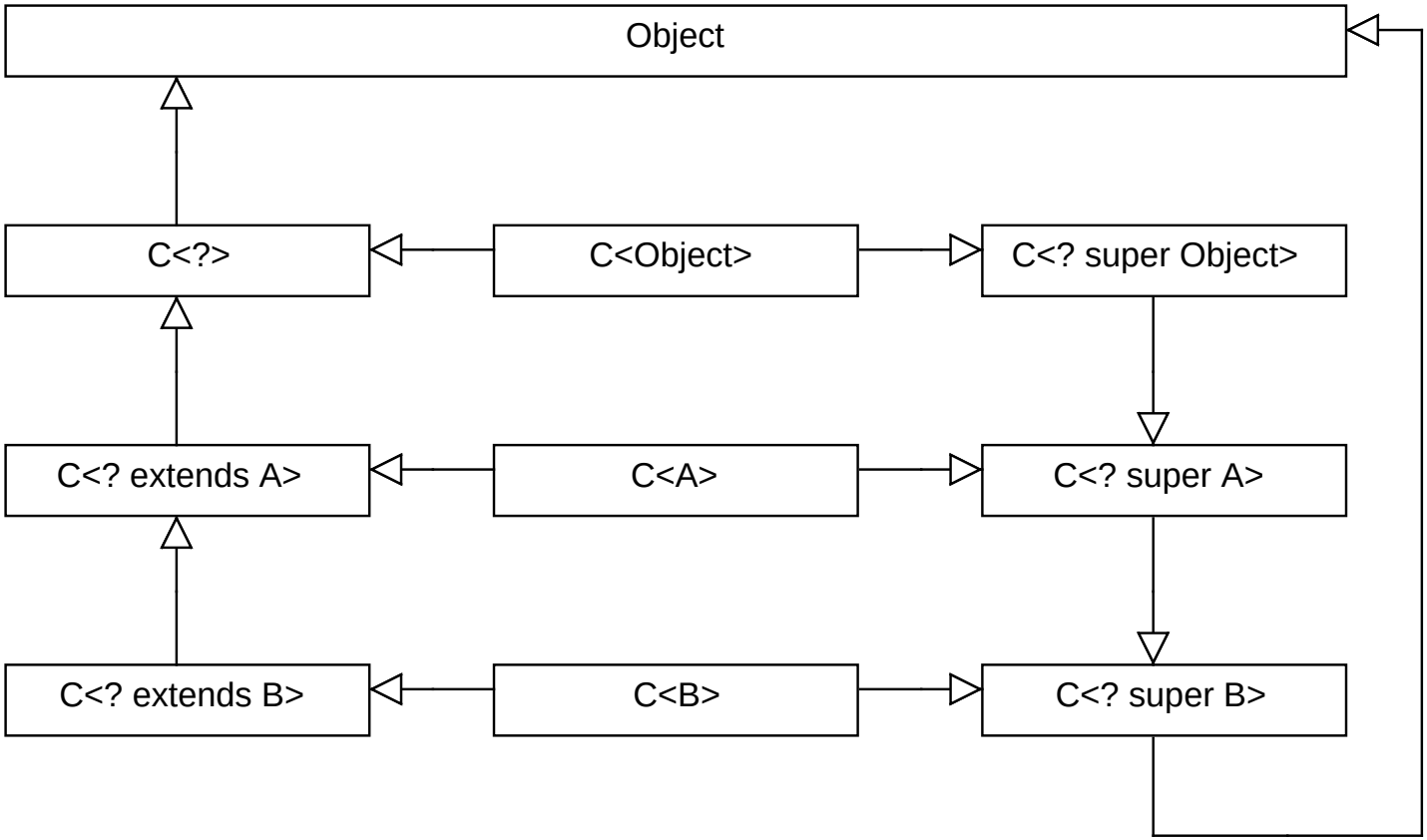
Vererbung bei Wildcard-Typen

Unsere generischen Getränkekästen basieren auf Wildcards.



Dort bleibt die Vererbung erhalten.

Übersicht


Object
C<?>
C<Object>
C<? super Object>
C<? extends A>
C<A>
C<? super A>
C<? extends B>
C<B>
C<? super B>

Gliederung

Eine parametrisierte Methode muss (und darf) nicht mit einem expliziten Typargument aufgerufen werden.

```
public <T> T choose(T a, T b) {
  // gibt zufällig a oder b zurück
}
```

Ein passendes Typargument wird automatisch ermittelt.

Beispiel für Typinferenz

```
Set<Integer> intSet = new TreeSet<Integer>();
List<String> stringList = new ArrayList<String>();

Object result = choose(intSet, stringList);

Collection<? extends Comparable<?>>
resultCollection = choose(intSet, stringList);
```

Gliederung

Man will für eine beliebige Mengen eine ReadOnly-Sicht der Menge haben.

```
<T> Set<T> unmodifiableSet(Set<T> set){...}
```

Die Methode lässt sich auch für einen Wildcard-Typ verwenden:

```
Set<?> set= new TreeSet<String>();
set = c.unmodifiableSet(set);
```

Der aktuelle Typ hinter `?` wird in `T` festgehalten („Capture“).

Zusammenfassung Generics

Generics

- bieten eine sinnvolle Erweiterung des Java-Typsystems.
- ermöglichen elegante Lösungen von existierenden Problemen.
- verbessern Lesbarkeit und Wartbarkeit von Programmen.
- sind typsicher.

Diskussion

1. Typsystem
   - Motivation generischer Klassen
   - Einführung
2. Vererbung
3. Sonstiges
   - Typinferenz
   - Wildcard Capture

Verbesserungsvorschläge?

```
class Bottle<T> {
  private T content;

  public T empty(){...}
  public void fill(T drink){...}
}
```

Literatur

Johannes Nowak.
Fortgeschrittene Programmierung mit Java 5.
dpunkt, 1. aufl. edition, 2005.

Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter.
Adding wildcards to the java programming language.
Journal of Object Technology, 3(11):97–116, December 2004.