Programmieren 3

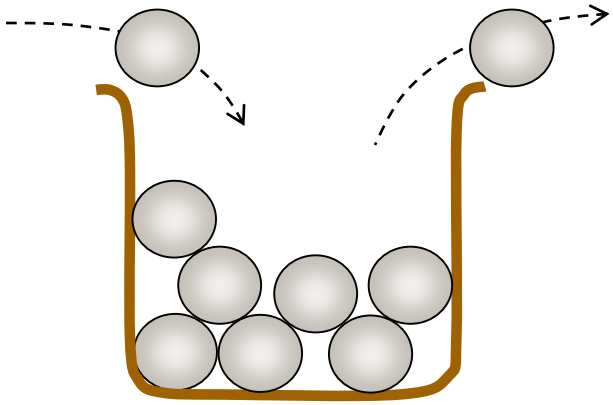
Kapitel 1: Generics

- Beispiel
- Type-Erasure
- Typsystem und Polymorphie
- Generische Methoden
- Bounds und Wildcards
- Einschränkungen

Martin Binder, FH Rosenheim, © 2015

Einfaches Beispiel – „Urne“ für ganze Zahlen

- Mindestanforderungen
  - Möglichkeit, Elemente in Urne abzulegen
  - Möglichkeit, ein Element aus Urne zu ziehen
- Implementierungsidee
  - Klasse mit privatem Feld
  - 1 öffentliche **Methode zum Einbringen** eines neuen Elements; Element wird in privates Feld eingetragen
  - 1 öffentliche **Methode zum Ziehen** eines Elements; Element wird per Zufall ermittelt und aus privatem Feld entfernt



Einfaches Beispiel – „Urne“ für ganze Zahlen

- Mindestanforderungen
  - Möglichkeit, Elemente in Urne abzulegen
  - Möglichkeit, ein Element aus Urne zu ziehen
- Implementierungsidee
  - Klasse mit privatem Feld
  - 1 öffentliche **Methode zum Einbringen** eines neuen Elements; Element wird in privates Feld eingetragen
  - 1 öffentliche **Methode zum Ziehen** eines Elements; Element wird per Zufall ermittelt und aus privatem Feld entfernt

```
public class UrneZahl {

  private Integer[] urne;
  private int anzahl;

  public UrneZahl() {
    urne = new Integer[100];
    anzahl = 0;
  }

  public void reinlegen(Integer neu) {
    urne[anzahl++] = neu;
  }

  public Integer rausnehmen() {
    anzahl--;

    int ziehung =
      (int) Math.floor(anzahl * Math.random());
    Integer wert = urne[ziehung];

    for(int i = ziehung; i < anzahl-1; i++)
      urne[i] = urne[i+1];

    return wert;
  }
}
```

Weiteres Beispiel – „Urne“ für Zeichenketten

```
public class UrneZahl {

  private Integer[] urne;
  private int anzahl;

  public UrneZahl() {
    urne = new Integer[100];
    anzahl = 0;
  }

  public void reinlegen(Integer neu) {
    urne[anzahl++] = neu;
  }

  public Integer rausnehmen() {
    anzahl--;

    int ziehung =
      (int) Math.floor(anzahl * Math.random());
    Integer wert = urne[ziehung];

    for(int i = ziehung; i < anzahl-1; i++)
      urne[i] = urne[i+1];

    return wert;
  }
}
```

```
public class UrneZeichenkette {

  private String[] urne;
  private int anzahl;

  public UrneZeichenkette() {
    urne = new String[100];
    anzahl = 0;
  }

  public void reinlegen(String neu) {
    urne[anzahl++] = neu;
  }

  public String rausnehmen() {
    anzahl--;

    int ziehung =
      (int) Math.floor(anzahl *
Math.random());
    String wert = urne[ziehung];

    for(int i = ziehung; i < anzahl-1; i++)
      urne[i] = urne[i+1];

    return wert;
  }
}
```

Problem: Eine Klasse für unterschiedliche Datentypen…

- … aber dieselbe Logik der Operationen?
  (`reinlegen`, `rausnehmen`)
- Eine Möglichkeit: Urne für **`Object`** Elemente

```
public class UrneObject {

  private Object[] urne;
  private int anzahl;

  public UrneZahl() {
    urne = new Object[100];
    anzahl = 0;
  }

  public void reinlegen(Object neu) {
    urne[anzahl++] = neu;
  }

  public Object rausnehmen() {

     ...
  }
}
```

Erster Versuch: Urne für Object-Elemente

```
public static void main(String[] args) {

  // Ziehung der Lottozahlen
  //
  UrneObject urneLotto = new UrneObject();

  for(int i = 1; i < 50; i++)
    urneLotto.reinlegen(i);

  Integer[] glueckszahl = new Integer[6];

  for(int i = 0; i < 6; i++)
    glueckszahl[i] = (Integer) urneLotto.rausnehmen();

  java.util.Arrays.sort(glueckszahl);

  System.out.println("\nDie Lottozahlen:");
  System.out.println(
    java.util.Arrays.toString(glueckszahl));
}
```

```
public static void main(String[] args) {

  // DFB Pokal Auslosung
  //
  UrneObject urnePokal = new UrneObject();

  String[] vereine =
    { "Bayern München", "Greuther Fürth", "Werder Bremen",
      "1899 Hoffenheim", "FC Augsburg", "1. FC Köln",
      "VfL Osnabrück", "Schalke 04" };
  for( String v: vereine )
    urnePokal.reinlegen(v);

  System.out.println("\nDie Viertelfinalpartien:");
  for(int i = 0; i < 4; i++)
  {
    String heim = (String) urnePokal.rausnehmen();
    String gast = (String) urnePokal.rausnehmen();
    System.out.println(heim + " - " + gast);
  }
}
```

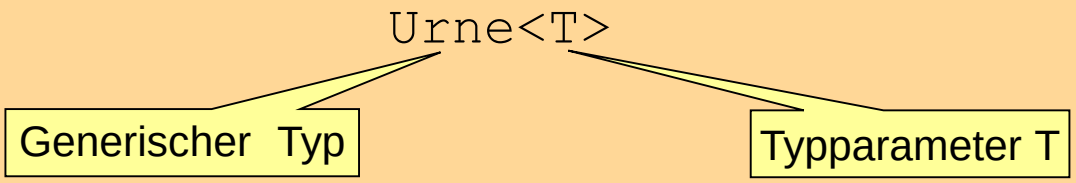
- **Nachteile**
  - **Beliebige** Objekte in Urne
  - Typumwandlung „**Cast**“
  - Gefahr von **Laufzeitfehlern** (`ClassCastException`)

Lösungsidee: Generische Typen

- Unterschiedliche Datentypen aber dieselbe Logik der Operationen? (`reinlegen`, `rausnehmen`)
- Abstrahieren von den zugrunde liegenden Datentypen? (`Integer`, `String`)
- Implementierung der `Urne` unabhängig von diesen Typen
- Einführung eines generischen Typs

Generischer Typ

Datentyp mit der Angabe von Typparametern („*Parametrische Polymorphie*“)



Generische „Urne“ – Erste Idee

```
public class Urne<T> {

  private T[] urne;
  private int anzahl;

  public Urne() {
    urne = new T[100];
    anzahl = 0;
  }

  public void reinlegen(T neu) {
    urne[anzahl++] = neu;
  }

  public T rausnehmen() {
    anzahl--;

    int ziehung =
      (int) Math.floor(anzahl * Math.random());
    T wert = urne[ziehung];

    for(int i = ziehung; i < anzahl-1; i++)
      urne[i] = urne[i+1];

    return wert;
  }
}
```

Cannot create a generic array of T

- Generische Arrays nicht zulässig?
- **Typlöschung!**

Generische „Urne“

- Wir ersetzen den Typ des internen Felds `urne` durch `ArrayList` der Java Collection API

- Auch das Java Collection Framework arbeitet mit generischen Typen

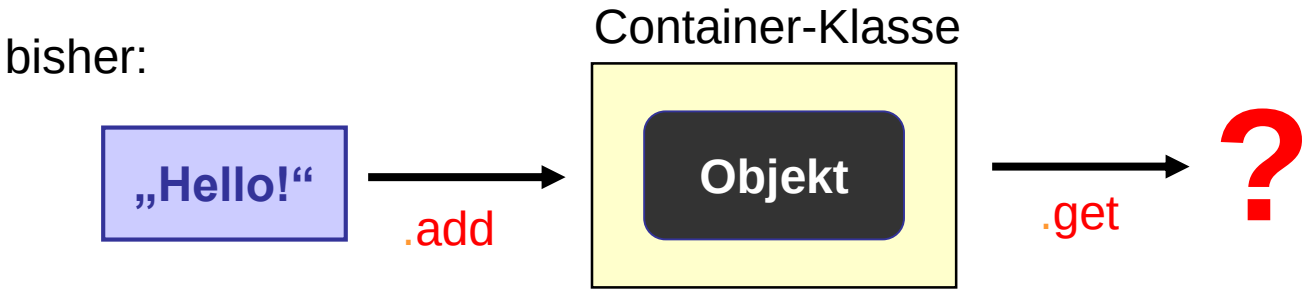
```
public interface List<E>
public class ArrayList<E>
```

- Wir ersetzen die Operationen zum Einfügen, Lesen und Löschen durch entsprechende Methoden

```
public class Urne<T> {

  private java.util.List<T> urne;
  private int anzahl;

  public Urne() {
    urne = new java.util.ArrayList<T>();
    anzahl = 0;
  }

  public void reinlegen(T neu) {
     anzahl++;
     urne.add(neu);
  }

  public T rausnehmen() {
    anzahl--;

    int ziehung =
      (int) Math.floor(anzahl * Math.random());
    T wert = urne.remove(ziehung);

    return wert;
  }
}
```

Generics und Container-Klassen

- Containerklassen in JAVA 1.4 sind generisch benutzbar, aber nicht typsicher (enthalten nur Objekte).
- Der Programmierer ist für die Konsistenz (Casting) selbst verantwortlich.



Vorteil : Code einfach zu programmieren
Nachteil : keine Typsicherheit zur Laufzeit
Code nicht aussagekräftig (List von was?? Kunde, Konto, String ?)

Generics - Beispiel Stack

Implementierung bisher:

```
class Stack {
   List data = new ArrayList();
   Object pop(){
          return data.remove(data.size - 1);
   }
   void push (Object element){
      data.add(element);
   }
}
```

```
class StackClient {
   public static void main(String[] args){
       Stack stack = new Stack();

       stack.push(„abc");
       String s = (String) stack.pop();

   }
}
```

ClassCastException
zur Laufzeit!

Typüberprüfung zur **Laufzeit**

mit Generics:

```
class Stack<T> {
   List<T> data = new ArrayList<T>();
   T pop(){
          return data.remove(data.size() - 1);
   }
   void push (T element){
      data.add(element);
   }
}
```

```
class StackClient {
   public static void main(String[] args){
       Stack<String> stack =
                          new Stack<String>();
       stack.push(„abc");
       String s = stack.pop();

   }
}
```

Compiler-Fehler
zur Übersetzungszeit!

Typüberprüfung zur **Übersetzungszeit** !

Begriffsdefinitionen für Generics

- ***Typ-Variable***
  - Platzhalter für einen konkreten Typen
    Beispiele: `class Stack<T>, class Pair<K,V>`
- ***Generischer Typ***
  - Klasse oder Schnittstelle, die mindestens eine Typ-Variable definiert
    Beispiel: `class Stack<T>`
- ***Parametrisierter Typ***
  - Generische Klasse, die an einen konkreten Typen gebunden wurde
    Beispiel: `Stack<Integer>`
- **Typ-Parameter**
  - Konkreter Typ, an den ein generischer Typ gebunden wird
    Beispiel: `Stack<Integer>`

Typlöschung (Type Erasure)

Parametrisierte Klasse

Generische Klasse

- Java Compiler
  - Erzeugt **eine** generische Klasse zur Laufzeit, die von allen Instanzen gemeinsam benutzt wird
  - **Typinformationen** **`<T>`** werden **entfernt**
  - Typvariablen werden durch **`Object`** ersetzt
  - Einfügen von Typanpassungen (**Cast**) dort, wo Typ nicht korrekt

Beispiel für Typlöschung

Source

```
public class Stack <T> {


    public void push(T t) {
        data.add(t);
    }

    public T pop() {
        return data.remove(data.size());
    }

    private List<T> data = new ArrayList<T>();
}
```

Classfile

```
public class Stack {

    public void push(Object t) {
        data.add(t);
    }

    public Object pop() {
        return (Object)data.remove(data.size());
    }

    private List data = new ArrayList();
}
```

Generische Typen ohne Parameter: Raw-Type

Benutzung von generischen Typen (Aussensicht)

```
Stack<String> stack = new Stack<String>();

stack.push(„abc");              // ok

String s = stack.pop();         // ok – kein CAST nötig → (String) stack.pop();

stack.push(new Long(0));        // nok → Compiler Fehler
```

Aus Kompatibilitätsgründen lassen sich generische Typen auch ohne Parameter benutzen:

```
Stack stack = new Stack();                  // ok (Legacy Code)

stack.push(„abc");                          // ok

String s = (String) stack.pop();            // ok –CAST nötig

stack.push(new Long(0));                    // ok

Stack<String> stringstack = stack;          // Warning: Unchecked Assignment

stack = stringstack;                        // ok (Legacy Code)
```

Raw Type

Dies ist möglich dank **Type Erasure**: Die generischen Deklarationen werden aus der Methodensignatur entfernt.

Folgen der Typlöschung

- Es gibt nur ein Klassen-Objekt für alle Instanziierungen eines generischen Typs

```
C<X>.class == C<Y>.class
```

- Jeder generische Typ kann ohne Warnung in seinen Raw-Type konvertiert werden (Abwärtskompatibilität)

```
C c = new C<X>();
```

- Ein Cast vom Raw-Type in einen generischen Typ erzeugt eine Compilerwarnung

```
C<X> cx = new C(); // Warning
```

- Es sind keine statischen Attribute vom Typ T möglich
- Es sind keine generischen Arrays erlaubt
- Refelektion ist nicht möglich (`getClass(), instanceof`)

```
ArrayList<Integer> al_i = new ArrayList<Integer>();
ArrayList<String>  al_s = new ArrayList<String>();

System.out.println(al_i.getClass() == al_s.getClass());
System.out.println(al_i instanceof ArrayList<Integer>);
```

true

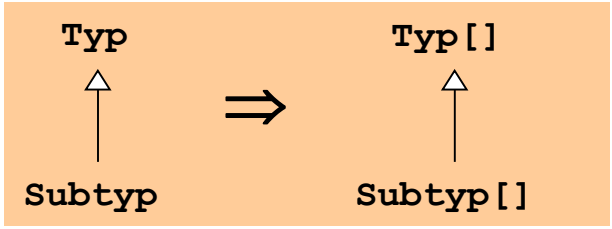
Cannot perform instanceof check against parameterized type ArrayList<Integer>.

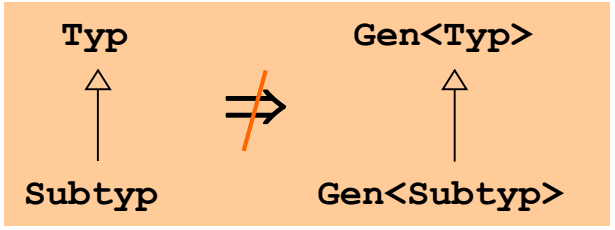
Unterschied von parametrisierten Typen zu Arrays

```
Number  n;
Integer i = new Integer(17);

n = i;

Number[]  arr_n;
Integer[] arr_i = new Integer[10];

arr_n = arr_i;
```

```
Number  n;
Integer i = new Integer(17);

n = i;

ArrayList<Number>  alist_n;
ArrayList<Integer> alist_i = new ArrayList<Integer>();

alist_n = alist_i;
```

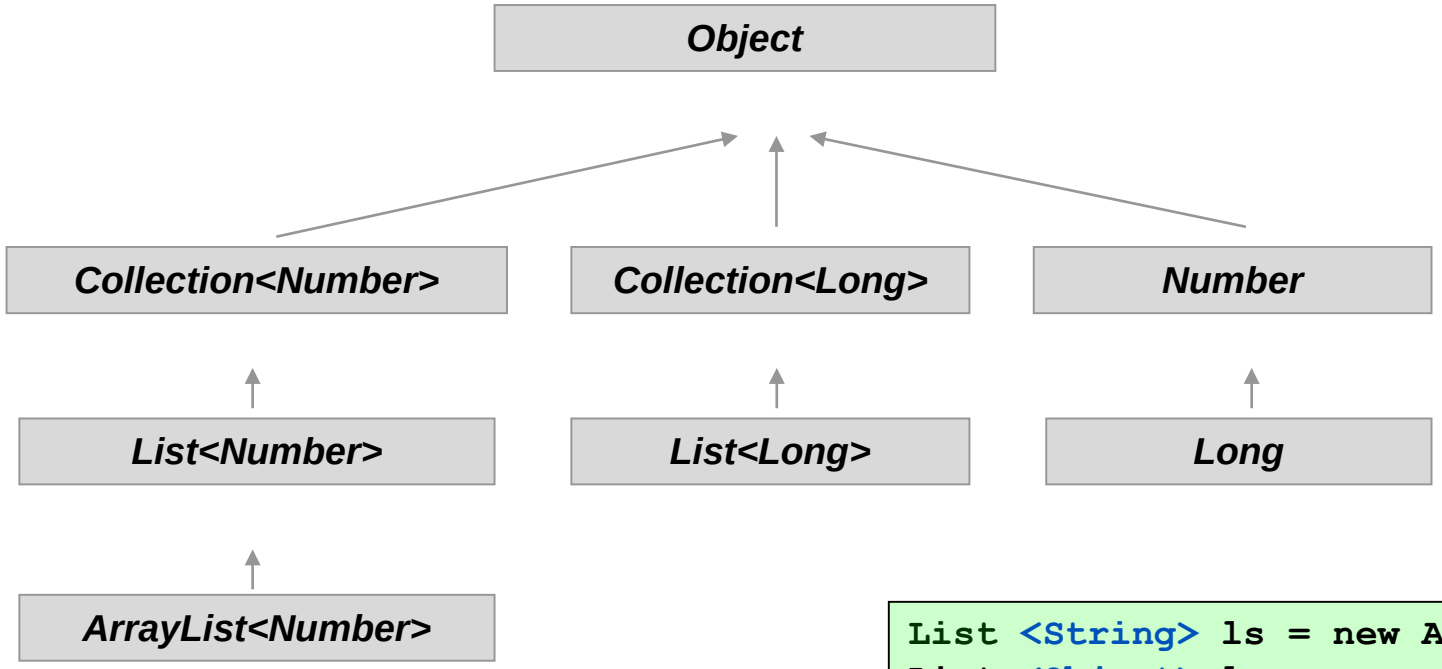
```
Type mismatch:
cannot convert from
ArrayList<Integer> to ArrayList<Number>
```





- Vererbung von Typ-Parameter überträgt sich nicht auf parametrisierte Klasse
  - Widerspricht intuitiver Nutzung
    - **Invariant**
    - Im Vergleich zu Arrays (kovariant)
  - Zuweisung von Objekten der generischen Klasse ist **nicht typsicher,** da sie von JRE nicht erkannt werden
    - daher verboten und Compiler-Fehler

Generics und Vererbung - Typsystem




```
List <String> ls = new ArrayList <String>();
List <Object> lx;
lx = ls;
```

- Die Typen C<X> und C<Y> sind nicht verwandt, selbst wenn X und Y verwandt sind (X extends Y)
  - → Collection<Object> kann nicht Collection<String> referenzieren
  - Die Typen sind NICHT zuweisungskompatibel und können nicht ineinander gecastet werden

Generische Methoden (Polymorphe Methoden)

Generische Methode

Deklaration mit einem oder mehreren Typparametern (ähnlich wie bei Deklaration eines generischen Typs)

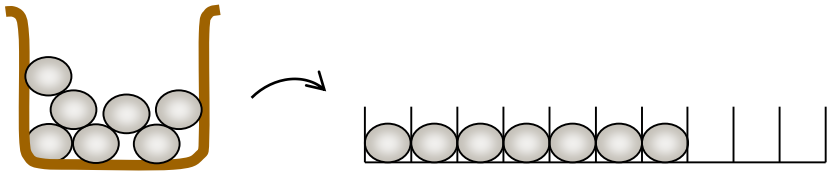
```
static <T> void copy(List<T> dest,
                     List<T> src )
```

Typvariable T

Typparameter

- Gehen bei Objekt- und Klassen-Methoden
- Vorteil
  - Übergabe von Typargumenten nicht nötig
    ➔ **Ableitung durch Java Compiler** auf Basis der aktuellen Argumente
  - **Abhängigkeiten** zwischen den Typen der Argumente oder der Rückgabe können beschrieben werden
- Nachteil
  - Nutzung der Typparameter nur innerhalb einer Methode

Generische Methoden



- Gesucht: eine Methode, welche für beliebige Urnen deren „Inhalt“ in ein Feld kopiert
- Lösung: eine statische generische Methode der Klasse `Urne<T>`

```
public class Urne<T> {

  private java.util.List<T> urne;

  ...

  public static <T> void toArray(Urne<T> u, T[] a) {
    int i = 0;
    for(T x: u.urne)  a[i++] = x;
  }
}
```

- Anwendung

```
Urne<String> urnePokal = new Urne<String>();

String[] myArray = new String[10];
Urne.toArray(urnePokal, myArray);
```

```
Urne<String> urnePokal = new Urne<String>();

Integer[] myArray = new Integer[10];
Urne.toArray(urnePokal, myArray);
```

The method toArray(Urne<T>, T[]) in the type Urne is not applicable for the arguments (Urne<String>, Integer[])

Einschränkungen für Typparameter: Bounds

```
class C1 <T extends C & I1 & I2 … &In>
```

- Bounds schränken Typparameter ein
  - Als Typparameter kann jede Unterklasse von C, welche die Schnittstellen I1 – In implementiert, verwendet werden
  - T kann von einer Klasse abgeleitet sein und beliebig viele Schnittstellen implementieren
  - Angabe der Bounds: Bei der Deklaration des Typparameters

- Die Methoden der Basisklasse / Interfaces sind in T sichtbar
  - Der Compiler prüft Methodenaufrufe von T gegen die Bounds:

```
class SortedList<T extends Comparable<T>> {
        public void add(T t) {
                if (t.compareTo(…)) { }
        }
}
```

Umsetzung von Bounds durch Compiler

Source

```
public class ClonableComparableStack <T extends Cloneable & Comparable> {


    public void push(T t) {
        data.add(t);
    }

    public T pop() {
        return data.remove(data.size());
    }

    private List<T> data = new ArrayList<T> ();
}
```

Classfile

```
public class ClonableComparableStack {

    public void push(Cloneable comparable) {
        data.add(comparable);
    }

    public Cloneable pop() {
        return (Cloneable) data.remove(data.size());
    }

    private List data = new ArrayList();
}
```

Sind Bounds angegeben, so wird der erste angegebene Bound statt `Object` verwendet.
Bei Bedarf wird auf das zweite Bound gecastet.

Generics und Vererbung

```
class Box <T extends Comparable> { …
…
}

class Pair <T extends Comparable, S> extends Box <T> {
…
}

class StringBox extends Box <String> {
…
}
```

- Von generischen Klassen lassen sich andere Klassen wie gewohnt ableiten
- Typvariablen können als Parameter an die Basisklasse weitergegeben werden
- Unterklassen von generischen Klassen sind nicht notwendigerweise generisch.

Collection<Object> ?

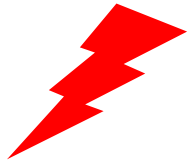
Problem: Schreibe eine Methode, die alle Elemente einer beliebigen Collection ausgibt.

Bisher:

```
public void printCollection (Collection xs){
   Iterator i = xs.iterator();

   while (i.hasNext()) {
      System.out.println(i.next());
   }}
```

Erster Versuch:

```
public void printCollection (Collection <Object> xs){

   for (Object x: xs){
      System.out.println(x);
   }
}
```



Funktioniert nur für Collection<Object>

Lösung:

```
public void printCollection (Collection <?> xs){

   for (Object x: xs){
      System.out.println(x);
   }
}
```

Wildcards

Wildcards: Eine Wildcard bezeichnet eine *Familie von Typen*.

3 Arten von Wildcards:

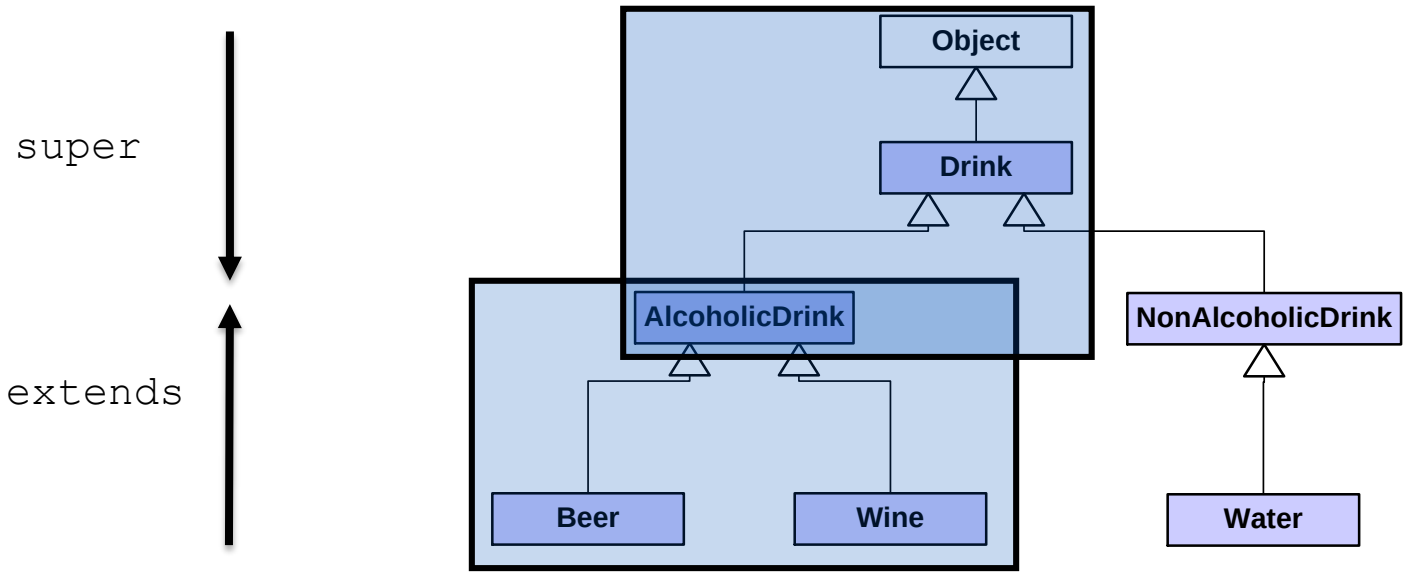
- ? → Menge aller Typen
- ? extends T → Menge aller Typen, die von T abgeleitet sind (Upper Type Bound)
- ? super T → Menge aller Supertypen von T (Lower Type Bound)

- Collection<?> gleicht dem Raw-Type, aber alle Objekte sind vom selben Typ

Praxis: Wildcardtypen sind spezielle generische Typen

- Wildcards findet man i.R. als Argument- und Return-Typ von Methoden; seltener für die Deklaration von Variablen.
- Sie werden verwendet wenn das konkrete Typargument <T> keine Rolle spielt
- Wildcardtypen mit extends sind read-only
- Wildcardtypen mit super sind write-only

Wildcards: Typkompatibilität



- `List<? extends AlcoholicDrink>` ← `List<AlcoholicDrink>`
- `List<? extends AlcoholicDrink>` /← `List<Drink>, List<Object>`
- `List<? extends AlcoholicDrink>` ← `List<Wine>, List<Beer>`

- `List<? super AlcoholicDrink>` ← `List<AlcoholicDrink>`
- `List<? super AlcoholicDrink>` ← `List<Drink>, List<Object>`
- `List<? super AlcoholicDrink>` /← `List<Wine>, List<Beer>`

Beispiel: lower type bound, upper type bound in Kombination

- Java Collection Framework
  - java.util.Collections
  - Methode `copy` zum Kopieren einer Liste

```
public static <T> void copy(List<? super T>  dest,
                            List<? extends T> src )
```

  - Lesen (aus Liste `src`)
    Typparameter ist eine Liste von `T` beziehungsweise Subtypen von `T`
  - Schreiben (in Liste `dest`)
    Typparameter ist eine Liste von `T` beziehungsweise Supertypen von `T`
- PECS – **P**roducer **E**xtends, **C**onsumer **S**uper

Beispiele aus java.util

- Arrays.sort (Bounds mit super)

```
public class Arrays {
       …
      static <T> void sort(T[] ts, Comparator<? super T> comparator) {…}
}
```

- java.util.Collections

```
public interface Collection<E> extends Iterable<E> {

    <T> T[] toArray(T[] ts);

    boolean containsAll(Collection<?> xs);

    boolean addAll(Collection<? extends E> xs);

    boolean removeAll(Collection<?> xs);

}
```

Einschränkungen generischer Typen in JAVA

- Statische Attribute dürfen keine Typvariablen verwenden (`static T myAttribute`)
- Eine Klasse kann nicht von einer Typvariablen erben (`extends T`)
  - Keine Mix-In Klassen möglich (`class X <T> extends T`)
- Eine Klasse kann höchstens eine Interfacespezialisierung implementieren (`Comparable<String>, Comparable<Long>`)
- Aufrufe von `instanceof T` / `T.class` sind verboten
- `new T()` **bzw** `new T[x]` ist aufgrund von Type Erasure nicht möglich
- Typ-Variablen sind in catch-Abschnitten verboten (`catch (T t)`)
- Über eine Wildcard Referenz **(<? extends X>)** kann man Objekt nur lesen, aber nicht verändern.

Zusammenfassung Generics

- Generics schaffen Typsicherheit und ersparen viele Cast-Operationen
- Durch Generics können einige Programmierfehler bereits zur Compile-Zeit erkannt werden, die sonst zum Programmabbruch führen
- Verallgemeinerung von Klassendefinitionen durch Generics ermöglicht die Wiederverwendung derselben Logik/Algorithmus für unterschiedliche Datentypen
- Programme mit Generics sind oft schwieriger zu programmieren und aber meist lesbarer
- Die fortgeschrittene Verwendung von Wildcards und Bounds erfordert sehr viel Übung
- Das Konzept für Generics wurde nachträglich in die Sprache Java integriert und hat Pro und Kontras
- Bei der Verwendung von Containerklassen des JDK werden heute häufig Generics zur Typsicherheit eingesetzt.

Zusammenfassung Generics – Vorsicht

- „Generics“ erst ab **JDK 1.5**
- Legacy Code
  - Übersetzter Code mit „Generics“ **nicht** auf **VM < 5.0**
  - **Typlöschung**: zwar Interoperabilität, aber keine Typinformationen zur Laufzeit
    - Fehler in falschem Legacy Code kaum zu lokalisieren
    - Debuggen mit „checked collection wrapper“ `java.util.Collections`
- „… dann geht es eben mit mehr Gewalt!“
  - Typsichere Programmierung kann man auch (wieder) zerstören
    - Verwendung von sog. Raw-Types
    - „Kaputt Casten“
    - Compiler Warnungen ignorieren

```
ArrayList<Integer> al_i =
    new ArrayList<Integer>();

ArrayList al = (ArrayList) al_i;
al.add("Kein Integer");
```