Programmierung 2

Kapitel 4: **Generics**

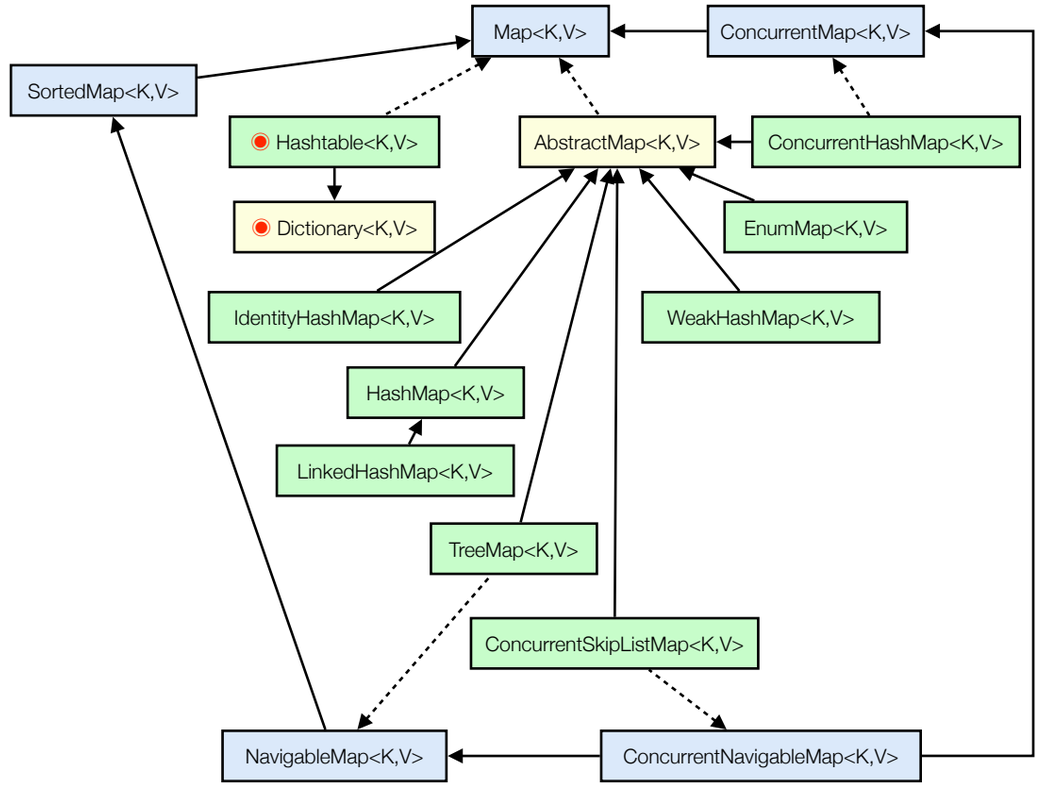
Prof. Dr. Markus Esch
htw saar

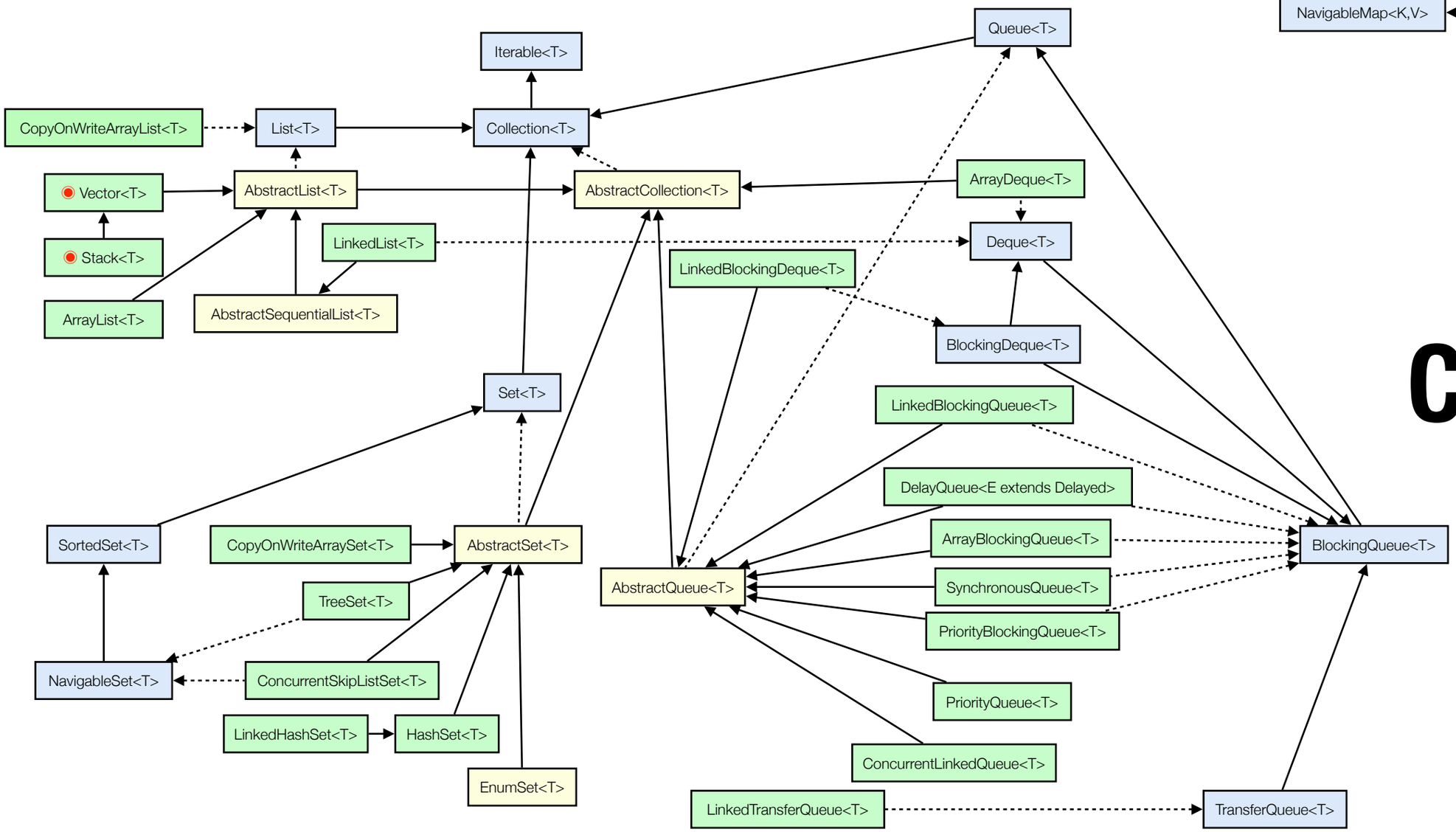
Collections in a **Nutshell**

Java **Collections** Framework

- Sammlung von Klassen und Interfaces, welche Datenstrukturen zur Verwaltung von Datenobjekten implementieren
  - Listen, Queues, Bäume etc.
  - Datencontainer / Datensammlungen
  - inkl. effizienter Algorithmen
- wiederverwendbar
- Reduzierung des Implementierungsaufwands

The **BIG** Picture

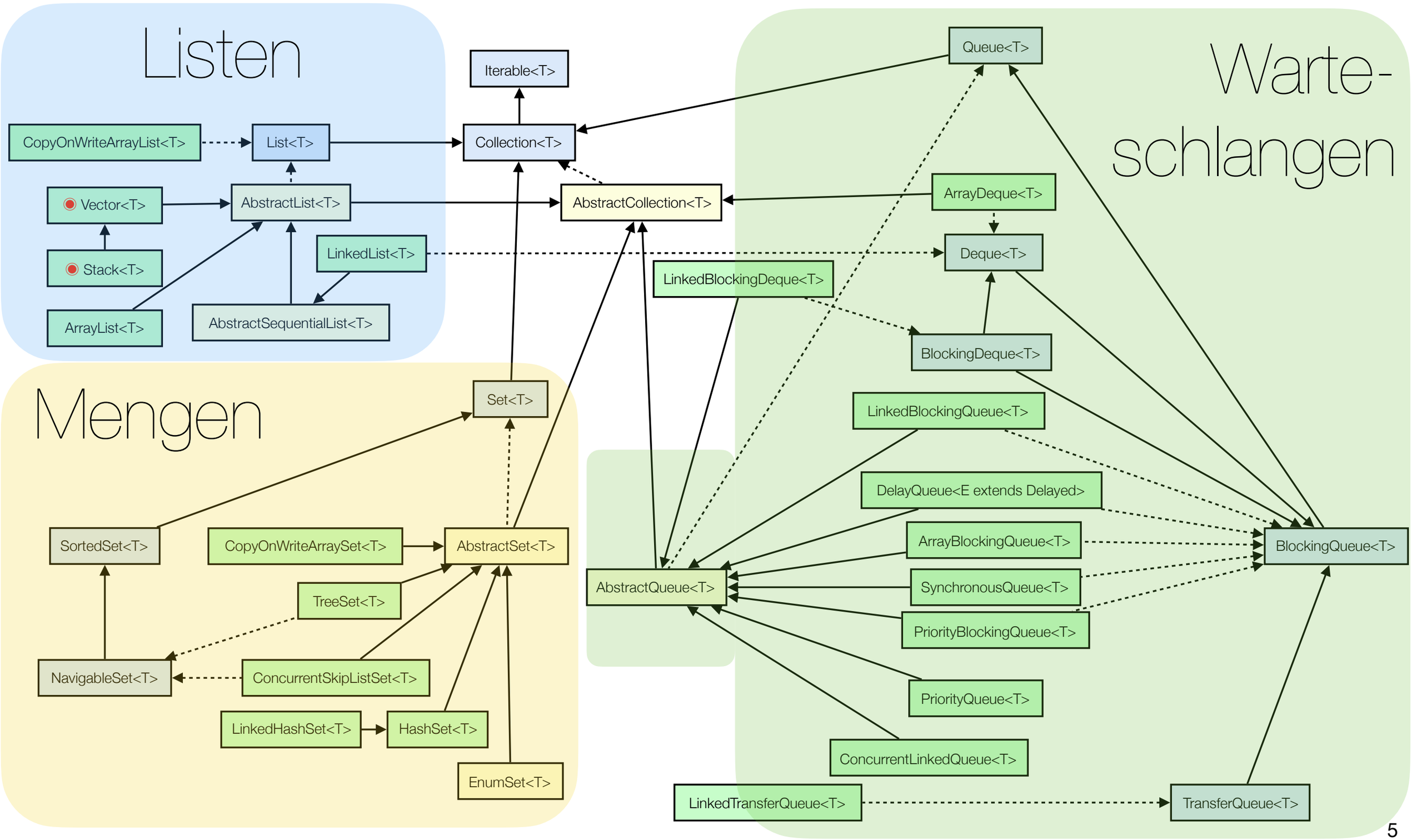




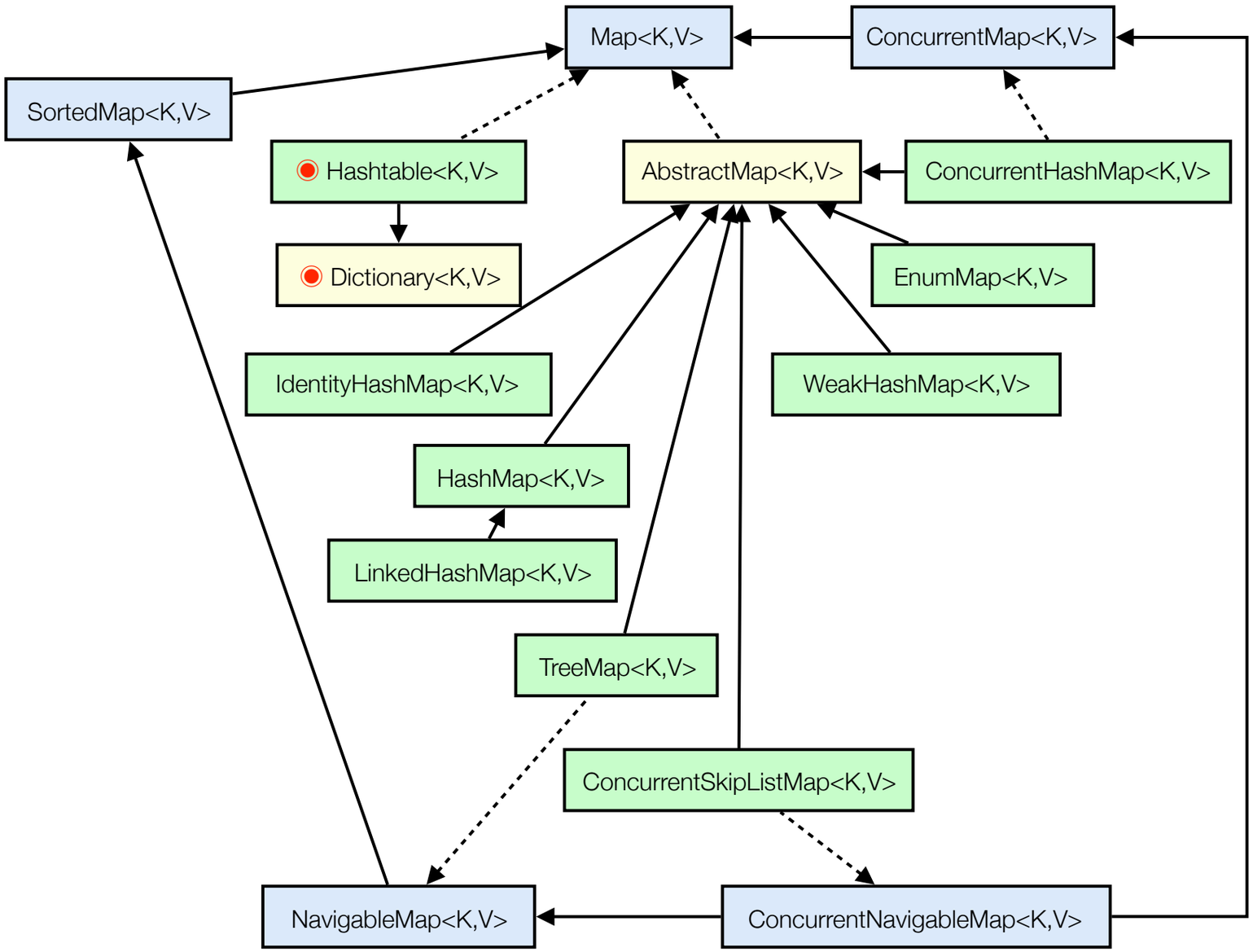
Maps & Collections

Collections

- Collections verwalten Mengen von Elementen



Maps



Key-Value-Stores

Beispiel **List**

```
public class CollectionsInANutshell {
   public static void main(String[] args)
   {
      List al = new ArrayList();
      al.add("Hello");
      al.add("World");
      al.add("!");

      for(int i=0; i<al.size();i++)
      {
         System.out.println(al.get(i));
      }

      al.remove(2);

      for(Object o : al)
      {
         System.out.println(o);
      }
   }
}
```

Beispiel **List**

```
public class CollectionsInANutshell {
   public static void main(String[] args)
   {
      List al = new LinkedList();
      al.add("Hello");
      al.add("World");
      al.add("!");

      for(int i=0; i<al.size();i++)
      {
         System.out.println(al.get(i));
      }

      al.remove(2);

      for(Object o : al)
      {
         System.out.println(o);
      }
   }
}
```

Motivierendes **Beispiel**

```
public class Box {
  private Object item;
  public void set(Object object) { this.item = object; }
  public Object get() { return item; }
}
```

```
Box b = new Box();
b.set(new Integer(42));
Integer i = (Integer)b.get();
String s = (String) b.get();
```

Compiler ✓

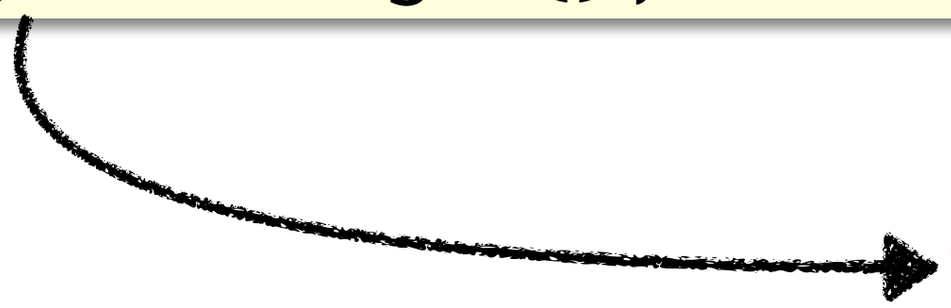
Laufzeit:

ClassCastException

Motivierendes **Beispiel**

```
public class IntegerBox {
   private Integer item;
     public void set(Integer item) { this.item = item; }
     public Integer get() { return this.item; }
}
```

```
IntegerBox ib = new IntegerBox();
ib.set(new Integer(42));
Integer i = ib.get();
String s = ib.get();
```

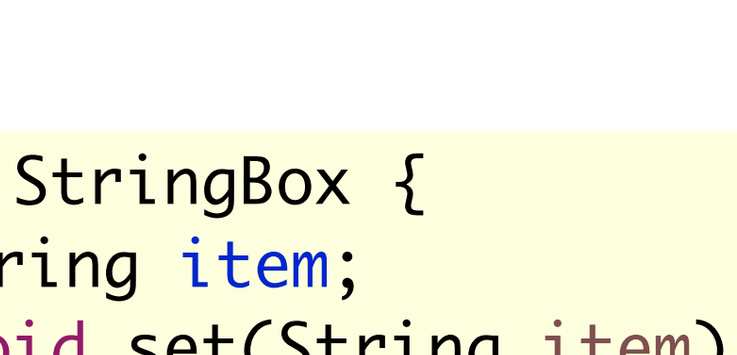


Compiler: Type mismatch:
cannot convert from Integer
to String

Motivierendes **Beispiel**

```
public class StringBox {
   private String item;
     public void set(String item) { this.item = item; }
     public String get() { return this.item; }
}
```

```
StringBox sb = new StringBox();
sb.set("42");
sb.set(new Integer(42));
```



Compiler: The method set(String) in the type StringBox is not applicable for the arguments (Integer)

Motivierendes **Beispiel**

```
public class StringBox {
   private String item;
     public void set(String item) { this.item = item; }
     public String get() { return this.item; }
}
```

```
public class IntegerBox {
   private Integer item;
     public void set(Integer item) { this.item = item; }
     public Integer get() { return this.item; }
}
```

```
public class StringBox {
   private String item;
     public void set(String item) { this.item = item; }
     public String get() { return this.item; }
}
```

```
public class IntegerBox {
   private Integer item;
     public void set(Integer item) { this.item = item; }
     public Integer get() { return this.item; }
}
```

```
public class DoubleBox {
   private Double item;
     public void set(Double item) { this.item = item; }
     public Double get() { return this.item; }
}
```

```
public class BoxBox {
   private Box item;
     public void set(Box item) { this.item = item; }
     public Box get() { return this.item; }
}
```

Motivierendes **Beispiel**

```
LinkedList list = new LinkedList();
list.add(new Integer(42));
list.add("42");
for(int j=0; j<list.size(); j++)
{
   Integer i = (Integer) list.get(j);
   System.out.println(i);
}
```

42

ClassCastException

Generische Programmierung

- Generics seit Java 5
  - Ähnlich zu Templates in C++
- Parametrisierte Typen
  - Klassen und Methoden mit variable Datentypen
  - Festlegung erst bei Instanziierung
- Vorteil
  - verbesserte Typ-Sicherheit

Generische Klassen

- Syntax:
  - ```
    public class name <T1, T2, . . . Tn> { /* . . . */ }
    ```
  - ```
    public interface name <T1, T2, . . . Tn> { /* . . . */ }
    ```
- T1, T2, … : Typ-Parameter oder Typ-Variablen
  - können für ***nicht-primitive*** Typen genutzt werden
    - Klassen, Interfaces, Arrays

Beispiel **GenericBox**

```java
public class GenericBox <T>{
    private T item;
    public void set(T object) { this.item = object; }
    public T get() { return item; }
}
```

```java
public class Box {
    private Object item;
    public void set(Object object) { this.item = object; }
    public Object get() { return item; }
}
```

Generische Typen **instanziieren**

- Deklaration
  - ClassName<Typ> Bezeichner

Typ-Argument

- Instanziierung
  - ClassName<Typ> Bezeichner = new ClassName<Typ>();
  - Seit Java 7 Typ-Deduction durch Target Typing:
    - ClassName<Typ> Bezeichner = new ClassName<>()

Diamond (informel)

Beispiel **GenericBox**

```
public class GenericBox <T>{
   private T item;
   public void set(T object) { this.item = object; }
   public T get() { return item; }
}
```

```
GenericBox<Integer> gb = new GenericBox<Integer>();
```

Beispiel **GenericBox**

```
public class GenericBox <T>{
   private T item;
   public void set(T object) { this.item = object; }
   public T get() { return item; }
}
```

```
GenericBox<Integer> gb = new GenericBox<Integer>();
gb.set(new Integer(42));
String s2 = (String)gb.get();
gb.set("42");
```

The method set(Integer) in the type GenericBox<Integer> is not applicable for he arguments (String)

Cannot cast from Integer to String

Naming **Conventions**

- Typ-Variablen werden durch einen Großbuchstaben bezeichnet
  - erleichtert Lesbarkeit des Codes
- typische Typ-Variablen-Namen:
  - **E** - Element
  - **K** - Key
  - **N** - Number
  - **T** - Type
  - **V** - Value
  - **S**,**U**,**V** etc. - 2nd, 3rd, 4th types

Mehrere Typ-Parameter

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

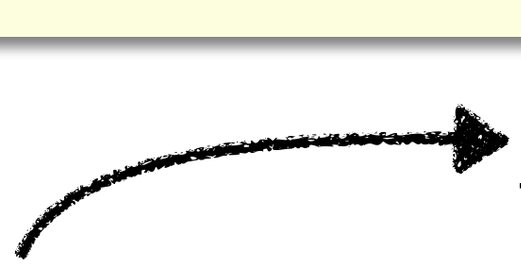
```
public class OrderedPair<K, V> implements Pair<K, V> {
  private K key;
  private V value;
  public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
  }
  public K getKey(){ return key; }
  public V getValue() { return value; }
}
```

```
Pair<String, Integer> p1 = new OrderedPair<>("Even", 42);
Pair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

Raw Typen

- Raw Typ: Ohne Typ-Argument instanziierte generische Klassen
  - Abwärtskompatibilität

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



parametrisierter Typ

```
GenericBox<Integer> gb = new GenericBox<Integer>();
```

Raw Typ

```
GenericBox rawbox = new GenericBox();
```

Raw Typen

- Raw Typ: Ohne Typ-Argument instanziierte generische Klassen
  - Abwärtskompatibilität

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

```
GenericBox<Integer> gb = new GenericBox<Integer>();
```

akzeptiert beliebiges Object

```
GenericBox rawbox = new GenericBox();
rawbox.set(new Integer(42));
Integer i = (Integer)rawbox.get();
String s = (String) rawbox.get();
```

ClassCastException

Raw Typen

```
GenericBox<Integer> intBox = new GenericBox<Integer>();
```

```
GenericBox rawbox = new GenericBox();
```

```
rawbox = intBox;
```

OK!

```
intBox = rawbox;
```

Warning: Type safety, unchecked conversion

```
rawbox.set(42);
```

Warning: unchecked invocation to set(T)

Generische **Methoden**

- Typ-Variablen können auch im Scope einer Methode definiert werde
  - Syntax

```
public <T1, T2, . . . TN> void methodName(){. . .}
```

```
public static <T1, T2, . . . TN> void methodName(){. . .}
```

Typ-Parameter

```
<T1, T2, . . . TN> methodName()
```

Typ-Argumente

Generische Methoden: **Beispiel**

```
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; }
}
```

```
public class Util {
  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: **Beispiel**

```
public class Util {
  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());
    }
}
```

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

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

Bounded Type Parameters

- Häufig ist es sinnvoll den Typ-Parameter einzuschränken
  - nur Subklassen einer bestimmten Oberklasse
  - nur Implementierungen eines bestimmten Interfaces

```
public <T extends Number> void methodName(){. . .}
```

nur Subklassen von Number als Typ-Parameter zulässig

“extends” auch bei Interfaces

Bounded Type Parameters

- Häufig ist es sinnvoll den Typ-Parameter einzuschränken
  - nur Subklassen einer bestimmten Oberklasse
  - nur Implementierungen eines bestimmten Interfaces

```
public <T extends IntSupplier> void methodName(){. . .}
```

funktionales Interface

“extends” auch bei Interfaces

nur Implementierungen von IntSupplier zulässig

Bounded Type Parameters

```
public class GenericBox <T>{
  private T item;
  public void set(T item) { this.item = item; }
  public T get() { return item; }
  public <U extends Number> void inspect(U u){
    System.out.println("T: " + item.getClass().getName());
    System.out.println("U: " + u.getClass().getName());
  }
}
```

```
GenericBox<Integer> integerBox = new GenericBox<Integer>();
integerBox.set(new Integer(10));
```

Bounded Type Parameters

```
public class GenericBox <T>{
  private T item;
  public void set(T item) { this.item = item; }
  public T get() { return item; }
  public <U extends Number> void inspect(U u){
    System.out.println("T: " + item.getClass().getName());
    System.out.println("U: " + u.getClass().getName());
  }
}
```

```
GenericBox<Integer> integerBox = new GenericBox<Integer>();
integerBox.set(new Integer(10));
integerBox.inspect("some text");
```

The method inspect is not applicable for the arguments (String)

Bounded Type Parameters

- Bounded Types erlauben den Aufruf von Methoden für Typ-Variablen

```
public class NaturalNumber<T extends Integer> {
  private T n;
  public NaturalNumber(T n)
  {
    this.n = n; }
  public boolean isEven() {
    return n.intValue() % 2 == 0;
  }
}
```

Methode der Klasse Integer kann aufgerufen werden

Bounded Type Parameters

- Bounded Types erlauben den Aufruf von Methoden für Typ-Variablen

```
public class NaturalNumber<T extends Integer> {
  private T n;
  public NaturalNumber(T n)
  {
    this.n = n; }
  public boolean isEven() {
    return n.intValue() % 2 == 0;
  }
}
```

The method intValue() is undefined for the type T

Bounded Type Parameters

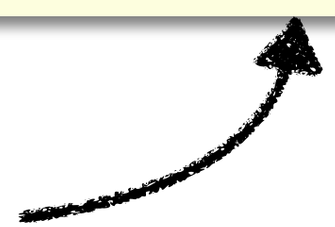
- Ein Typ kann mehreren Einschränkungen unterliegen

```
public class name <T extends A & B & C> { /* . . . */ }
```

- Klassen müssen zuerst genannt werden
  - class A { /* ... */ }
  - interface B { /* ... */ }
  - interface C { /* ... */ }

```
public class name <T extends B & A & C> { /* . . . */ }
```

Compiler-time Error



Bounded Types **Beispiel**

- **Count.java**

Bounded Type Parameters vs. Vererbung

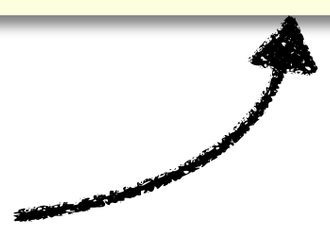
- Ein Typ kann mehreren Einschränkungen unterliegen

```
public class name <T extends A & B & C> { /* . . . */ }
```

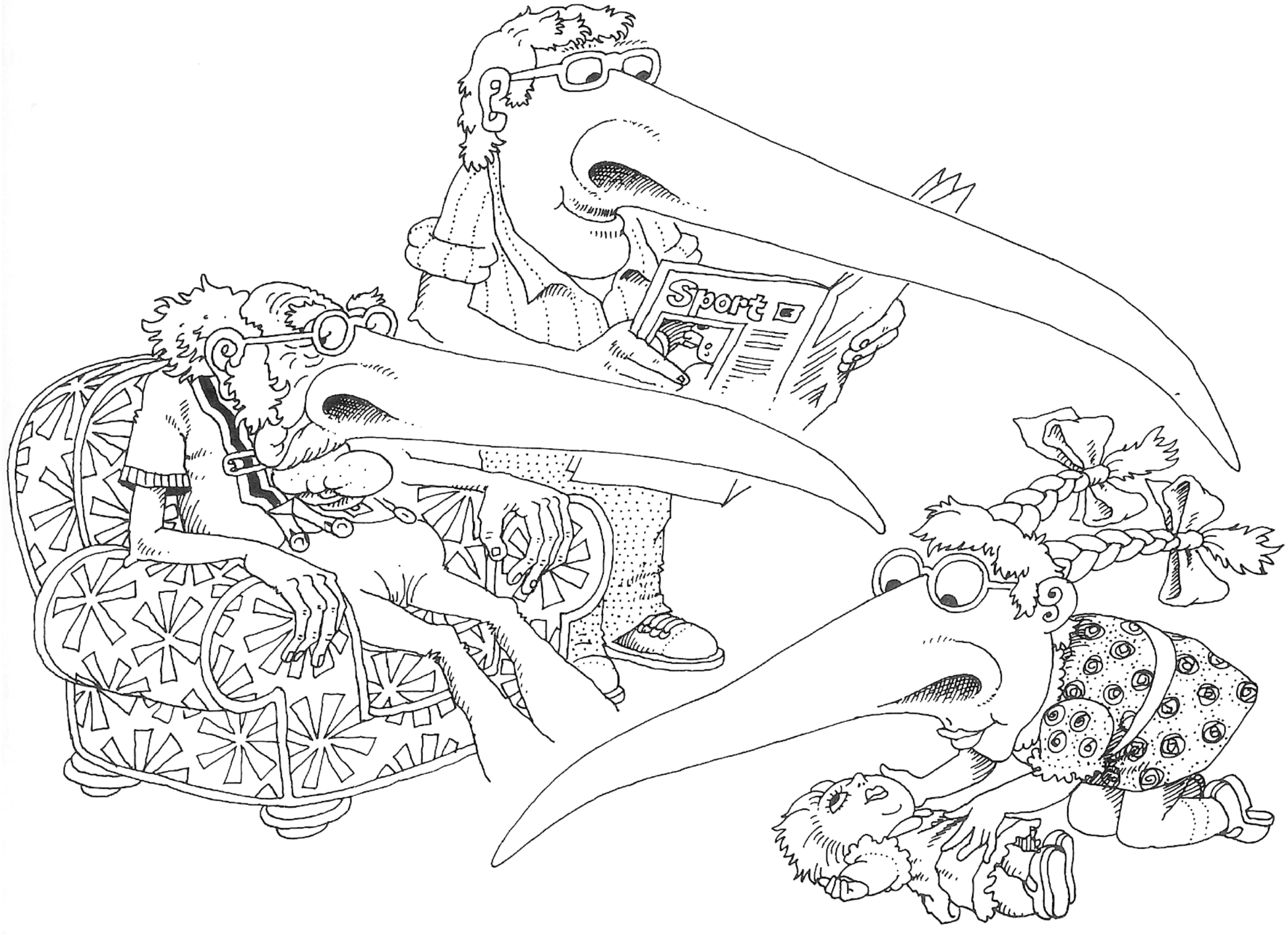
- Klassen müssen zuerst genannt werden
  - class A { /* ... */ }
  - interface B { /* ... */ }
  - interface C { /* ... */ }

```
public class name <T extends B & A & C> { /* . . . */ }
```

Compiler-time Error



Generics und **Vererbung**



Generics und **Vererbung**

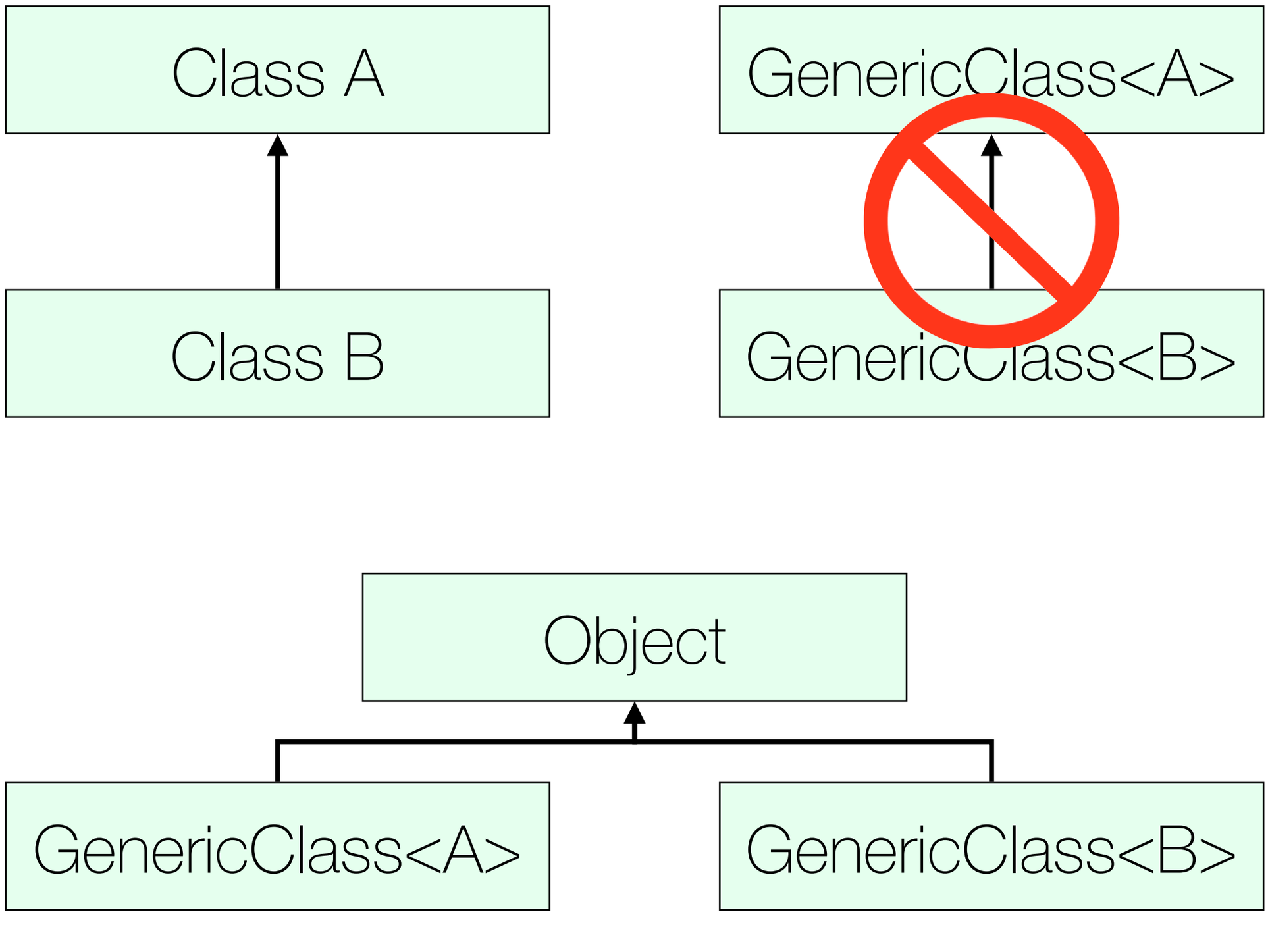
```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;
```

```
public void someMethod(Number n) { /*...*/ }
e.someMethod(new Integer(10));
e.someMethod(new Double(10.1));
```

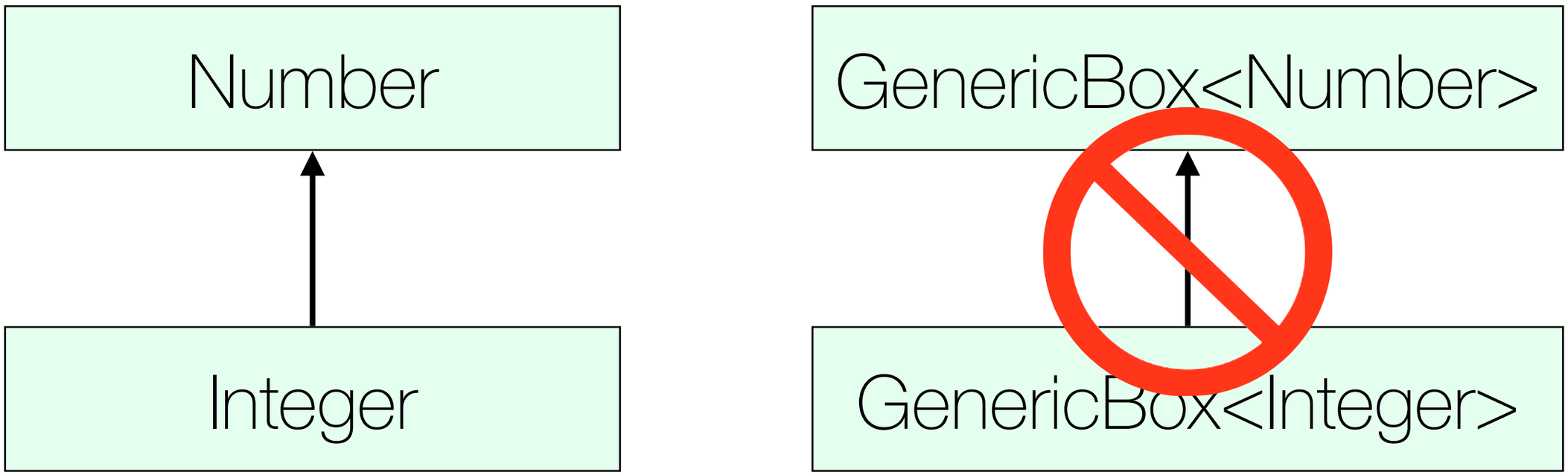
```
GenericBox<Number> box = new GenericBox<Number>();
box.set(new Integer(10));
box.set(new Double(10.1));
```

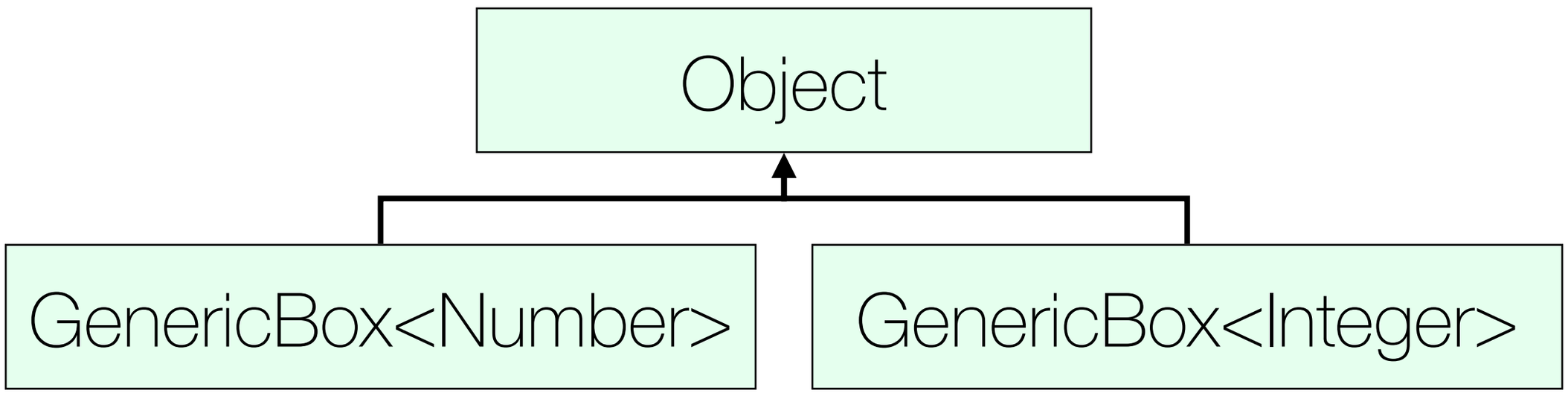
```
public void boxTest(GenericBox<Number> n){/*...*/}
boxTest(new GenericBox<Integer>());
boxTest(new GenericBox<Double>());
```

Generics und **Vererbung**



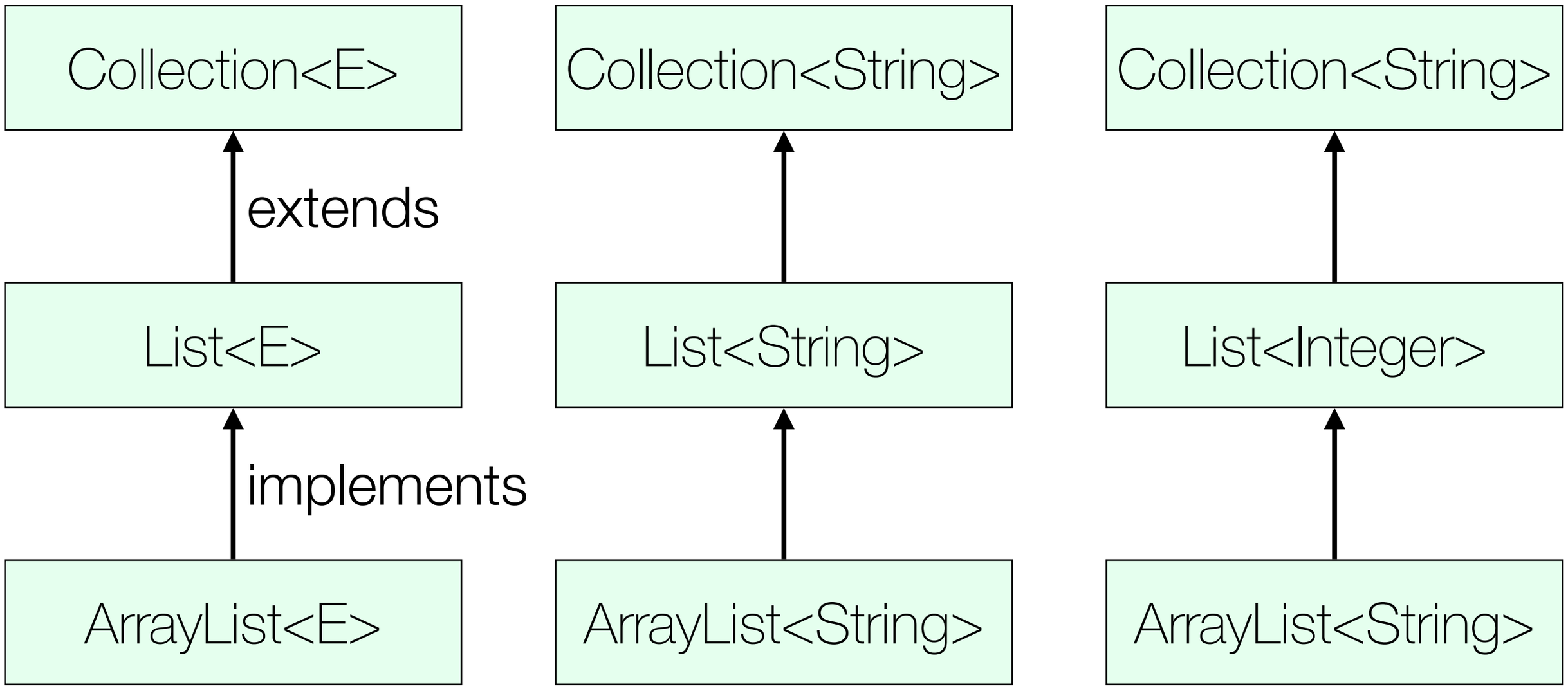
Generics und **Vererbung**





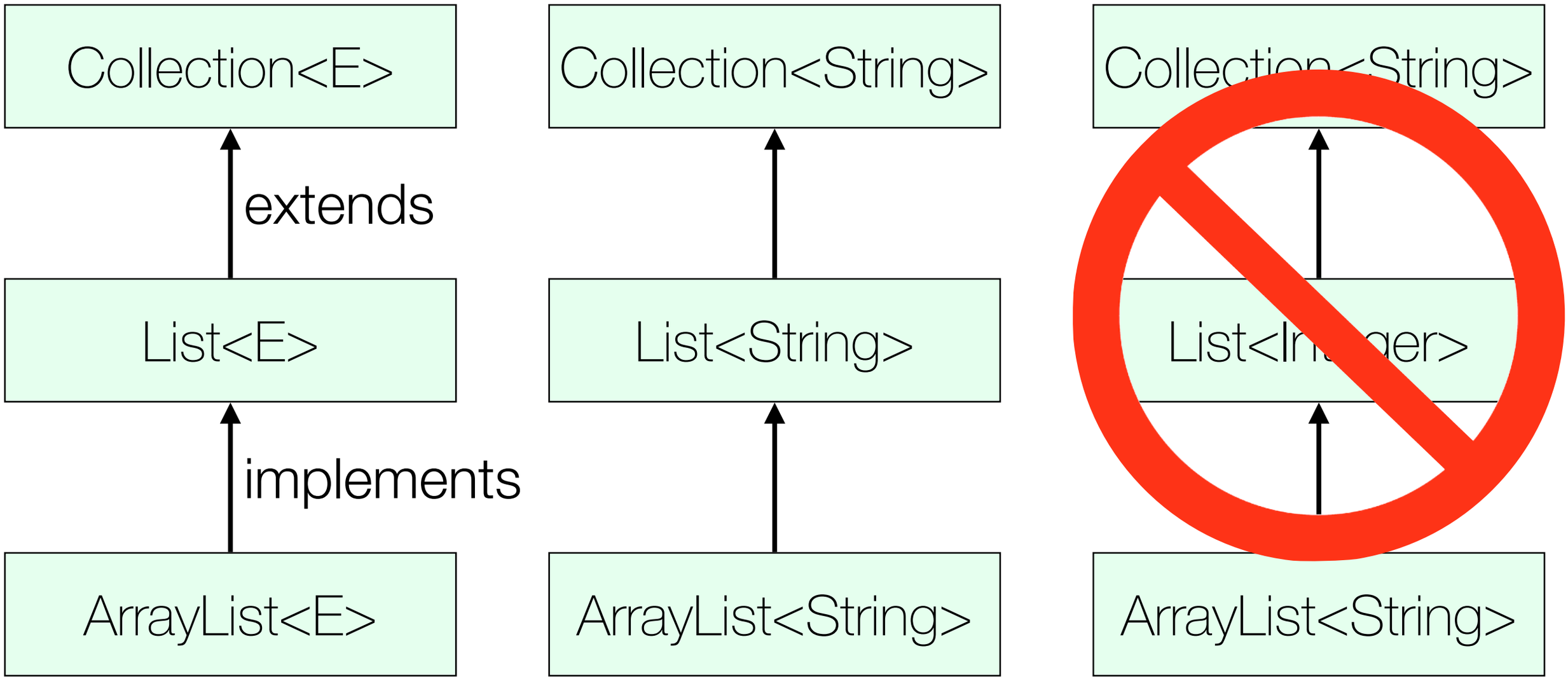
Generics und **Vererbung**

- Vererbung generischer Klassen ist möglich



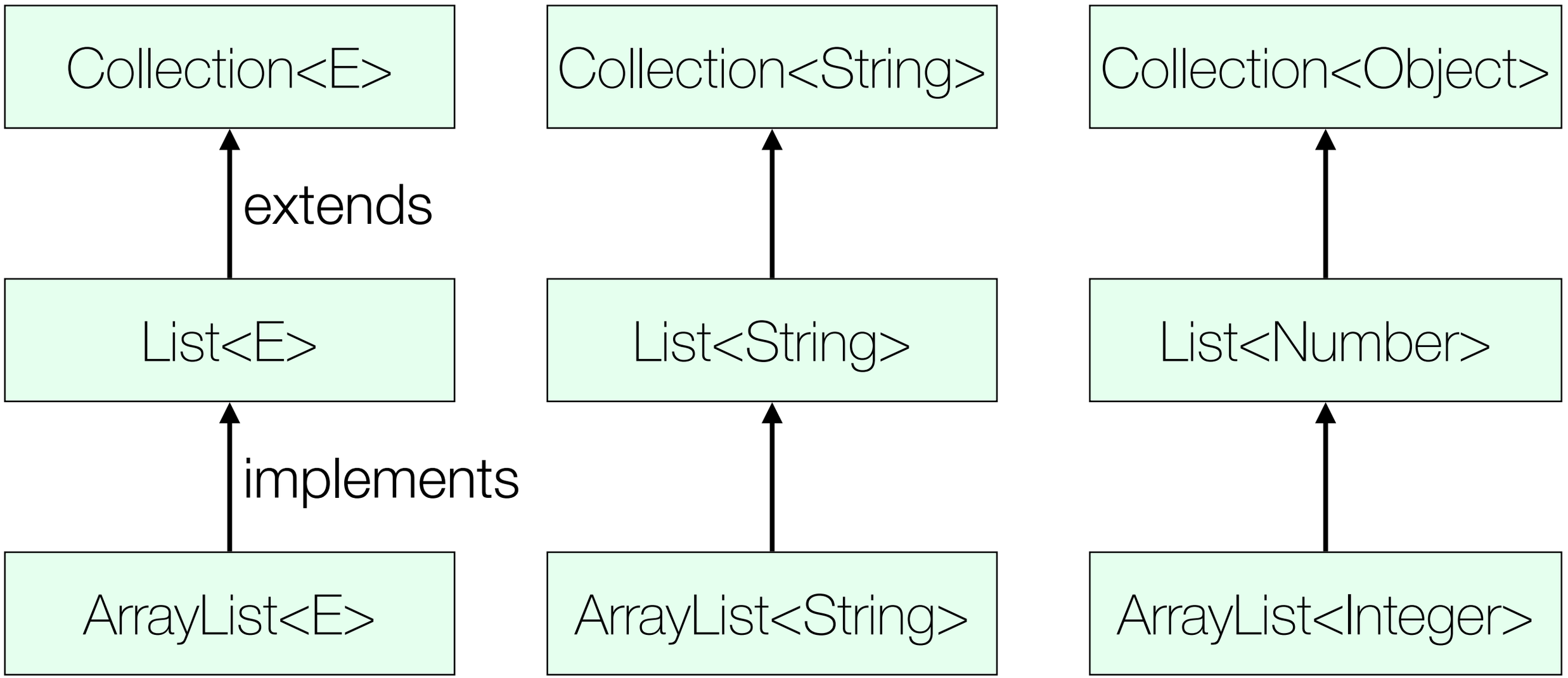
Generics und **Vererbung**

- Vererbung generischer Klassen ist möglich



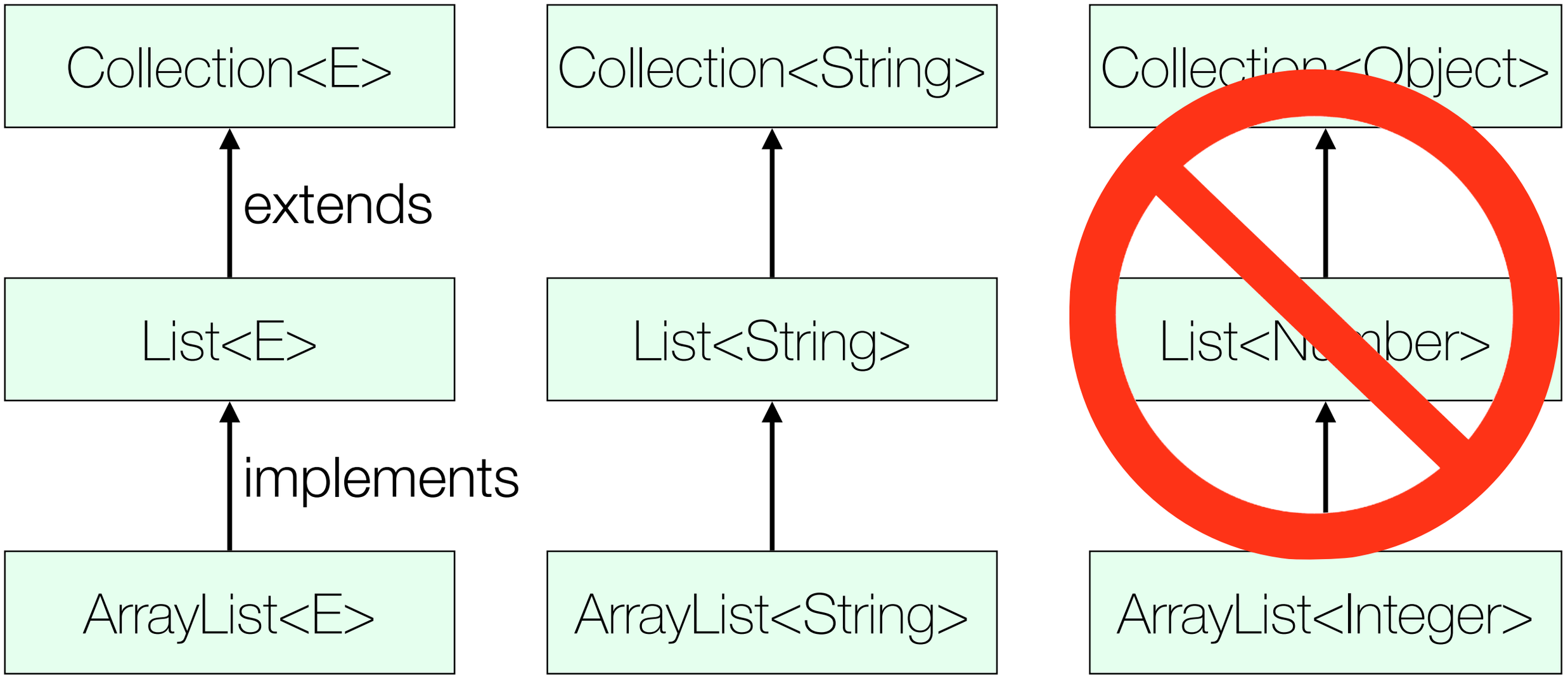
Generics und **Vererbung**

- Vererbung generischer Klassen ist möglich



Generics und **Vererbung**

- Vererbung generischer Klassen ist möglich



Wildcards

Wildcards: Motivation

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (int k = 0; k < c.size(); k++) {
    System.out.println(i.next());
  }
}
```

Pre Java 5

```
List input = Arrays.asList(1, 2, 3);
printCollection(input);
```

```
void printCollectionG(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}
```

Achtung:
Collection<Object>
ist kein Supertype
aller Collections!

```
printCollectionG(new ArrayList<Integer>());
```

not applicable for arguments
(ArrayList<Integer>)

Wildcards: Motivation

```
public <T> void printCollectionG(Collection<T> c){
  for (T e : c) {
    System.out.println(e);
  }
}
```

```
<Object>printCollectionG(new ArrayList<Object>());
```

```
<Integer>printCollectionG(new ArrayList<Integer>());
```

```
<Object>printCollectionG(new ArrayList<Integer>());
```

Parametrisierung mit Subtypen von Object nicht möglich

Parameter-Typ der Collection kann angepasst werden

Wildcards: Motivation

```
public <T extends Object> void printCollectionG(Collection<T> c){
  for (T e : c) {
    System.out.println(e);
  }
}
```

```
<Object>printCollectionG(new ArrayList<Object>());
```

```
<Integer>printCollectionG(new ArrayList<Integer>());
```

```
<Object>printCollectionG(new ArrayList<Integer>());
```

ArrayList<Object>
erwartet!

Lösung:

Wildcards

?

Unbeschränkte Wildcards

- "**?**": erlaubt beliebige Referenztypen

```
public void printCollection(Collection<?> c){
  for (Object e : c) {
    System.out.println(e);
  }
}
```

Supertyp aller Collections

Wildcard erlaubt beliebige Parametrisierung

```
printCollection(new ArrayList<String>());
printCollection(new ArrayList<Integer>());
. . .
```

alle Collections implementieren java.util.Collection<E>

Unbeschränkte Wildcards

- “**?**”: erlaubt beliebige Referenztypen

Supertyp aller Collections

```
public void printCollection(Collection<?> c){
  for (Object e : c) {
    System.out.println(e);
  }
}
```

Wildcard erlaubt beliebige Parametrisierung

```
printCollection(Arrays.asList(1, 2, 3));
printCollection(Arrays.asList("1",“2","3"));
. . .
```

alle Collections implementieren java.util.Collection<E>

Unbeschränkte Wildcards

- "**?**": erlaubt beliebige Referenztypen

```
public void printCollection(Collection<?> c){
  for (Object e : c)
    System.out.println(e);
  }
}
```

Cast zu Object
ist sicher!

Unbeschränkte Wildcards

```
ArrayList<String> strings = new
  ArrayList<String>();
strings.add("42");
List<?> s = strings;
Object o = s.get(0);
String str = s.get(0);
Integer i = s.get(0);
```

Compile time error:
Missmatch, cannot convert
from ? to String (Integer)

Unbeschränkte Wildcards

```
ArrayList<String> strings = new
  ArrayList<String>();
strings.add("42");
List<?> s = strings;
Object o = s.get(0);
String str = (String)s.get(0);
Integer i = (Integer)s.get(0);
```

Runtime error:
ClassCastException

Upper Bounded Wildcards

```
public double sumOfList(List<?> list) {
  //Sum all items in list
}
```

?

Upper Bounded Wildcards

```
public double sumOfList(List<?> list) {
   //Sum all items in list
   for (Object n : list){
      //Sum up
   }
}
```

how?

Upper Bounded Wildcards

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

Methode der Klasse Number

Upper Bounded Wildcard

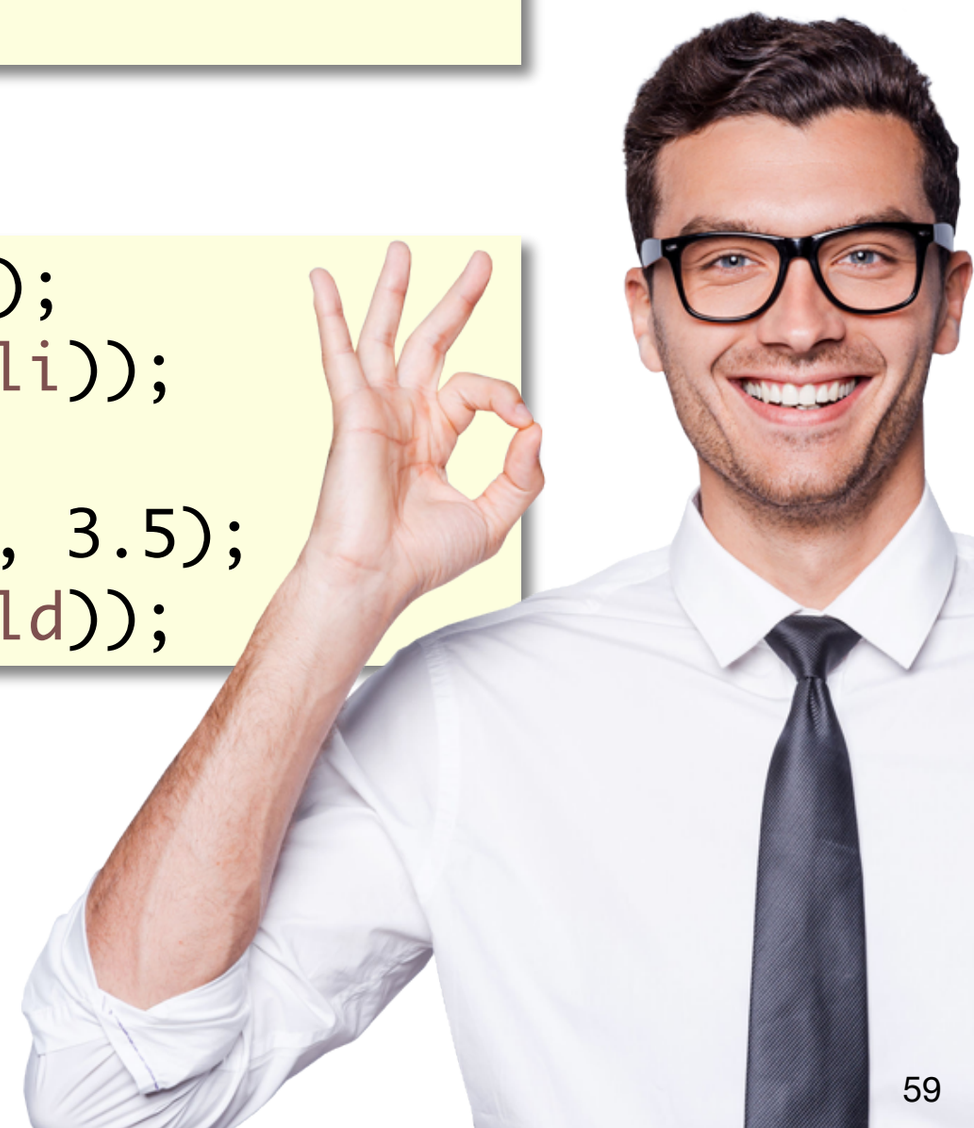
alle Subtypen von Number sind zulässig

Upper Bounded Wildcards

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

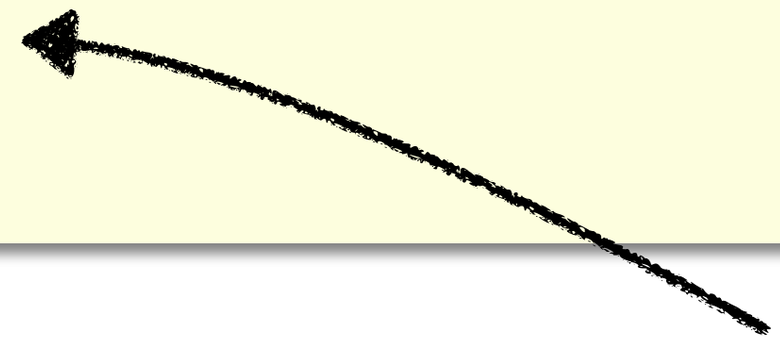
```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));

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



Lower Bounded Wildcards

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



Compile time error: The method add is not applicable for the arguments (int)

Lower Bounded Wildcards

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

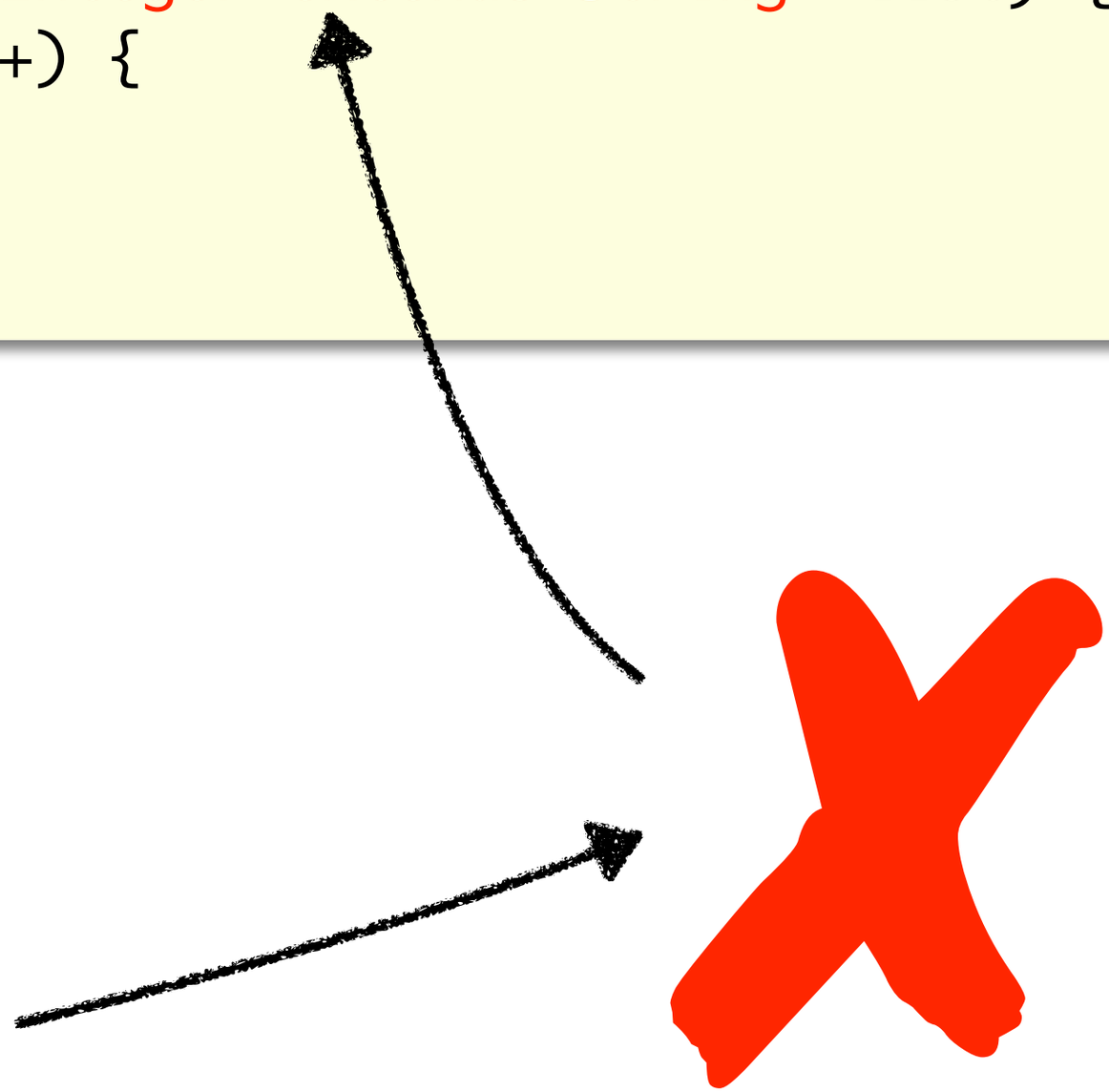
Integer kann zu allen Superklassen gecastet werden

Lower Bounded Wildcard

alle Supertypen von Integer sind zulässig

Lower/Upper Bounded Wildcards

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



Lower Bounded Wildcard

XOR

Upper Bounded Wildcard

Wildcards und **Vererbung**

```
Integer i = new Integer(42);
Number n = i;
```

```
List<Integer> li = new ArrayList<>();
List<Number> ln = li;
```

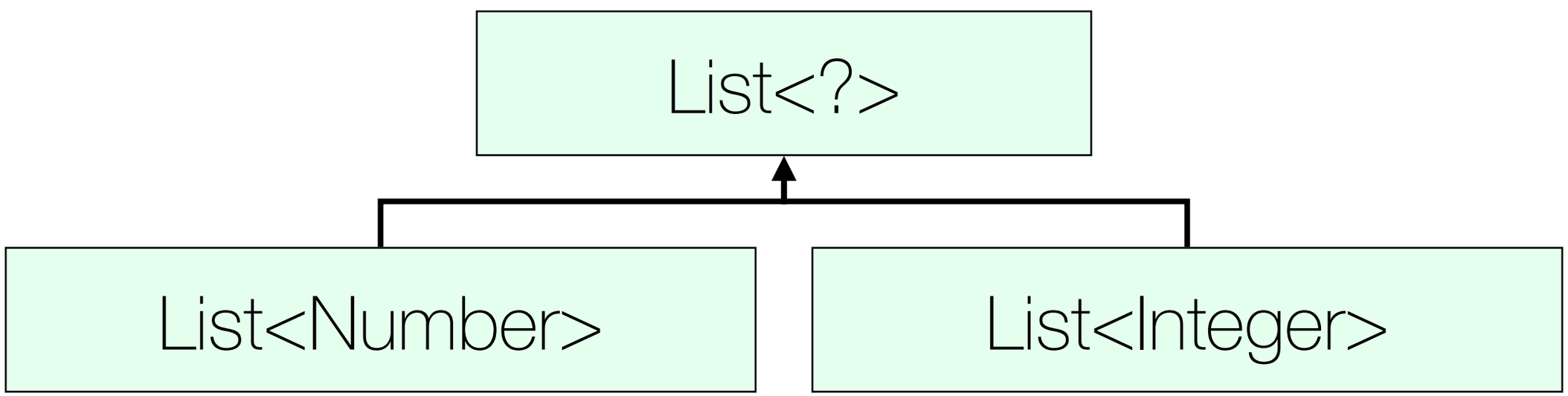
List<Integer> ist kein
Subtype von List<Number>

Wildcards und **Vererbung**

```
Integer i = new Integer(42);
Number n = i;
```

```
List<Integer> li = new ArrayList<>();
List<Number> ln = li;
```

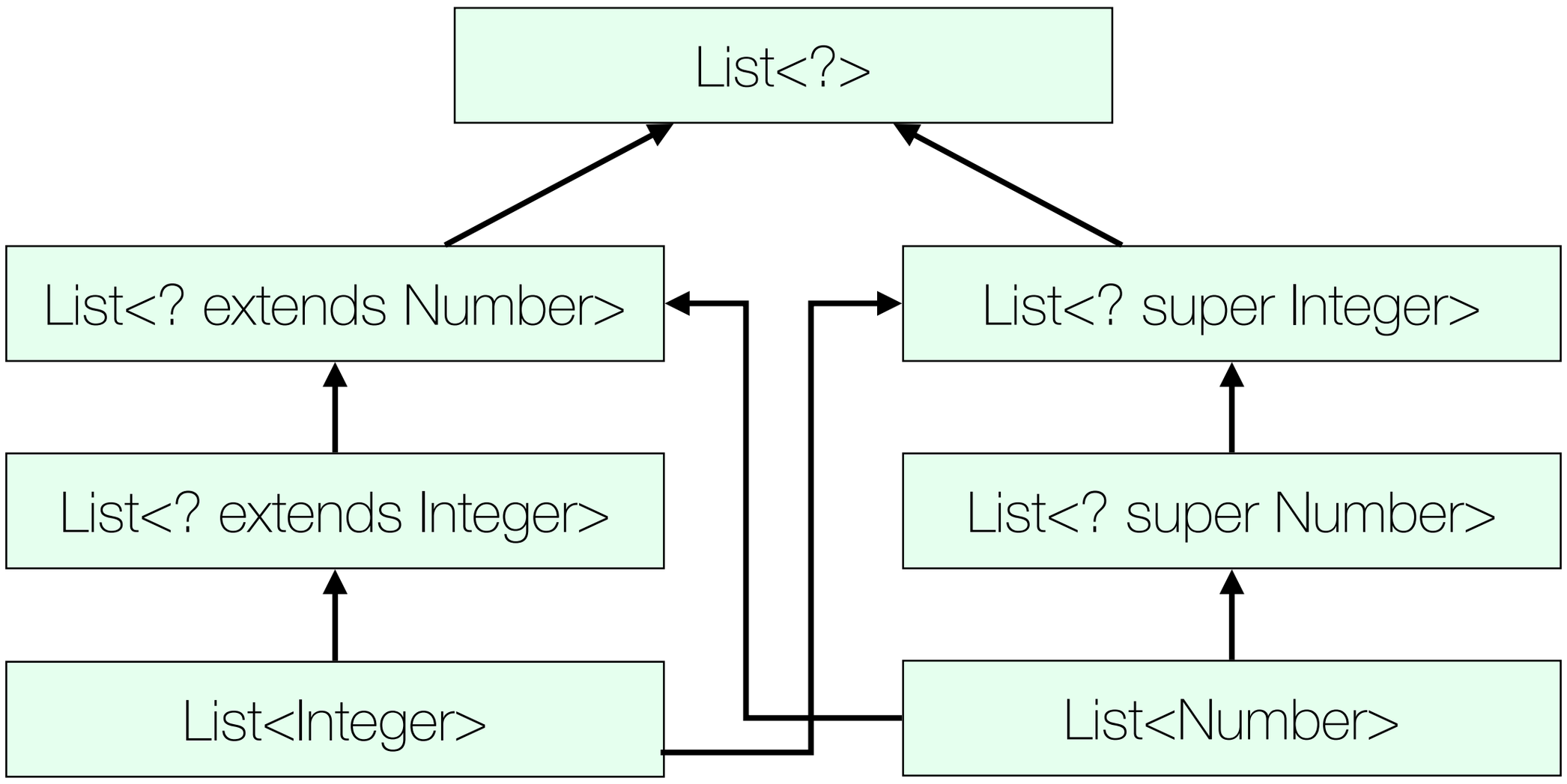
```
List<?> unknownList = li;
unknownList = ln;
```



Wildcards und **Vererbung**

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;
```

✓



Bounded Types vs. **Bounded** Wildcards

```
public class GenericBox <T extends Number>{
   private T item;
   public void set(T object) { this.item = object; }
   public T get() { return item; }
}
```

Einschränkung des Typs für Klassen und Methoden

```
public void printContent(GenericBox<? super Integer> box)
{
   System.out.println(box.get().intValue());
}
```

Einschränkung des Typs bei der Verwendung einer generischen Klasse

Literatur-Hinweise

(1)https://docs.oracle.com/javase/tutorial/java/generics/index.html

(2)http://www.angelikalanger.com/Articles/EffectiveJava/30.GenericsIntro/30.GenericsIntro.html

(3)Maurice Naftalin, Philip Wadler: Java Generics and Collections. O'Reilly, 2006.