

Java 5 "Tiger" Erweiterungen



- Autoboxing und Unboxing
- Erweiterte for-Schleife
- Variable Anzahl Methoden-Argumente und printf
- Enumerationen
- Statische Imports
- Generics
- Annotationen (hier nicht behandelt)

Java 5 bis 11

- Erste Java (1992) Spracherweiterung im Jahr 2004
- Erweiterungen wurden so vorgenommen, dass bestehende Programme weiter funktionieren
- Aber: im JDK z.T. jetzt neue und alte Variante ("deprecated") nebeneinander vorhanden
- Java 6 keine Spracherweiterungen
- Java 7 nur kleine Verfeinerungen bez. Sprache (z.B. case mit Strings);
 - JDK: parallele Verarbeitung à la Thread Pool (+ `java.util.concurrent`)
- Java 8 Lambda Ausdrücke und Stream Bibliothek
 - JDK: parallele Verarbeitung à la Map-Reduce (+ `java.util.streams`)
- Java 9/10 Module
 - "Modularisierung" des JDKs für z.B. Embedded Devices
- Java 11 Neues Lizenzmodel (LTS Versionen, JRE nicht mehr gratis)



Verschiedene neue Konzepte

Autoboxing und Unboxing

- Sehr oft Umwandlung von Wertetyp in Referenztyp und zurück:
int<-> Integer
- vereinfacht, automatisiert

```
Integer i = new Integer(5656);  
int j = i.intValue();  
  
Object o = new Integer(4711);  
int y = (Integer)o.intValue();  
  
void setValue(Object o){  
    ...  
}  
p.setValue(new Integer(34));
```

```
Integer i = 5656;  
int j = i;  
  
Object o = 4711;  
int y = (Integer)o;  
  
void setValue(Object o) {  
    ...  
}  
p.setValue(43);
```

Foreach-Schleife

- Sehr oft Iteration durch alle Elemente einer Collection, Arrays (in aufsteigender Reihenfolge)

```
int[] a = {1,2,4};    int sum=0;
for (int i = 0; i < a.length; i++) {
    sum+=i;
}
```

```
List numbers = new ArrayList(a);
// numbers is a list of Numbers.
Iterator it = numbers.iterator();
while (it.hasNext() ) {
    Integer number = (Integer)
    it.next();
    // Do something with the number...
}
```

```
int[] a = {1,2,4};    int sum=0;
for (int i : a) {
    sum +=i;
}
```

```
List numbers = new
    ArrayList(a);
for (int n : numbers) {

    // Do something with the
    number...
}
```

- Typen, deren Wertebereich einer endlichen, kleinen Menge von Werten entspricht
- bisher als int oder static "Object"-Konstante)

```
public class Color {  
    public static int Red = 1;  
    public static int White = 2;  
    public static int Blue = 3;  
}  
  
int myColor = Color.Red;  
int myColor = 999;    // no compile time error !!!  
  
oder:  
  
public class Color {  
    public static Color Red = new Color(255,0,0);  
}  
  
Color c = Color.Red;
```

```
// The color enumeration
public enum Color {
    Red,
    White,
    Blue
}

Color c = Color.Red;
// Cycling through the values of an enumeration
// (static method values())
for (Color c : Color.values()) {System.out.println(c);}
^
```



Red
White
Blue

Enumerationen in Java sind Klassen

- sie können auch Konstruktor und Methoden haben
 - Vorsicht mit Methoden: es kann zu möglichen Seiteneffekten führen
- Die Enumerationswerte sind vordefinierte Instanzen i.e. Objekte
- Dies im Gegensatz zu fast allen andern Sprachen -> Enumwert = Integer

Variable Methoden Argumente und printf

```
// An example method that takes a  
// variable number of parameters
```

```
// old
```

```
int sum(int[] numbers){  
    int mysum = 0;  
    for (int i=0; i<numbers.length;  
        i++)  
        mysum += numbers[i].;  
    return mysum;  
}
```

```
// Code fragment that calls  
// the sum method  
sum(new int[] {12,13,20});
```

```
// An example method that takes a  
// variable number of parameters
```

```
// New
```

```
int sum (int ... numbers) {  
    int mysum = 0;  
    for (int i: numbers) mysum += i;  
    return mysum;  
}
```

```
// Code fragment that calls the  
// sum method  
sum(12,13,20);
```

Variable Methoden Argumente und printf

- Es gibt neu eine printf Methode in System.out für die formatierte Ausgabe
- Erster String definiert ein Template mit Platzhaltern
 - %d,%i -> Integer
 - %f -> Float
 - %s -> String
 - %n -> New Line

```
// Pre Java 5 print statement  
System.out.println(x + " + " + y + " = " + sum);
```

```
// Using Java 5 printf statement  
System.out.printf("%d + %d = %d%n", x, y, sum);
```

<http://alvinalexander.com/programming/printf-format-cheat-sheet>

Static Imports

```
// x is a number of type double such as 5.345  
double y = Math.sin(x);
```

With static imports in Java 5, you can ask the Java compiler to import only a class's static portions, as shown, for example, in the rewritten code fragment below:

```
// Import declaration along with the other imports  
import static java.lang.Math.sin;  
  
// And then somewhere in the code...  
// "floating functions" arise from the ashes!!  
double y = sin(x);  
  
// Import all static methods from Math  
import static java.lang.Math.*;
```

Java Generics

■ Bisher:

```
List list = new LinkedList();  
list.add(new Integer(99));  
list.add(new String("a"));  
Integer i = (Integer)list.get(1); -->Runtime-Error
```

■ Gefährlich:

- kein Schutz gegen Einfügen von Elementen vom "falschen" Typ
- ClassCastException zur Laufzeit möglich bei falschem Typcast

■ Lösung:

try-catch, Befüllung der List absichern oder sonstige Klammzüge

■ Wünschenswert: Collection Klassen spezialisiert auf bestimmte Elementtypen

■ Java 5

```
List<Integer> list = new LinkedList<Integer>();  
list.add(new Integer(99)); // oder dank Boxing: list.add(99);  
Integer i = list.get(0);    // oder dank Unboxing: int i = list.get(0);  
list.add(new Double(3.1415)); -->Compile-Error
```

■ Vorüberlegung über den Datentyp, den die Collection aufnehmen soll.

■ Hier: Der *Typparameter* ist vom Typ Integer.

- erhöht die Aussagekraft und erleichtert Verständnis des Codes
- Einfügen von anderen Typen werden zur **Compile-Zeit** schon abgelehnt
- Cast beim Lesen kann entfallen, da garantiert keine anderen Typen enthalten sind.

Java Generics im Collection-Framework

```
List<Integer> list = new LinkedList<Integer>();  
list.add(99);
```

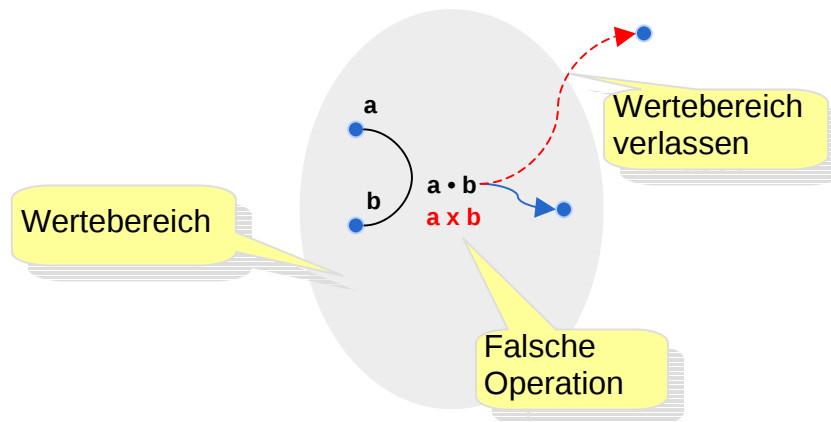
- Generischer Datentypen, List<Integer>, LinkedList<Integer>
- Duden: Generisch = "Die Gattung betreffend", "Gattungs..."
- Man sieht auch "parametrisierter Datentyp" oder "parametrischer Datentyp"
- Regeln sind sehr restriktiv, z.B.
list.add(new Object());
ergibt **Compile-Error**
- Boxing wird unterstützt, d.h. int <-> Integer

- Erweiterung des Java 5 Collection Frameworks
 - erhöht die Effizienz von Collections, macht sie vielseitiger einsetzbar
 - macht deren Einsatz komfortabler, sicherer und aussagekräftiger
 - senkt die Gefahr von Fehlern bei Typprüfungen zur Laufzeit
- Erleichtert so die Implementierung von Datenstrukturen wie Containern
- Möglichkeit zur Erstellung von eigenen Klassenschablonen und Methodenschablonen
- Erleichtert das Schreiben von Algorithmen, die mit verschiedenen Datentypen funktionieren.

Typsicherheit

■ Stellt sicher, dass bei der Programmausführung

- die Datentypen gemäss ihren Definitionen in der benutzten Programmiersprache verwendet werden
- keine Typverletzungen auftreten
 - **definierter Wertebereich** nicht überschritten wird
 - nur **definierte Operationen** angewendet werden



■ Statische Typsicherheit

- Compiler überprüft

■ Dynamische Typsicherheit

- Laufzeitsystem überprüft

■ Strikte/starke Typenüberprüfung/Sicherheit

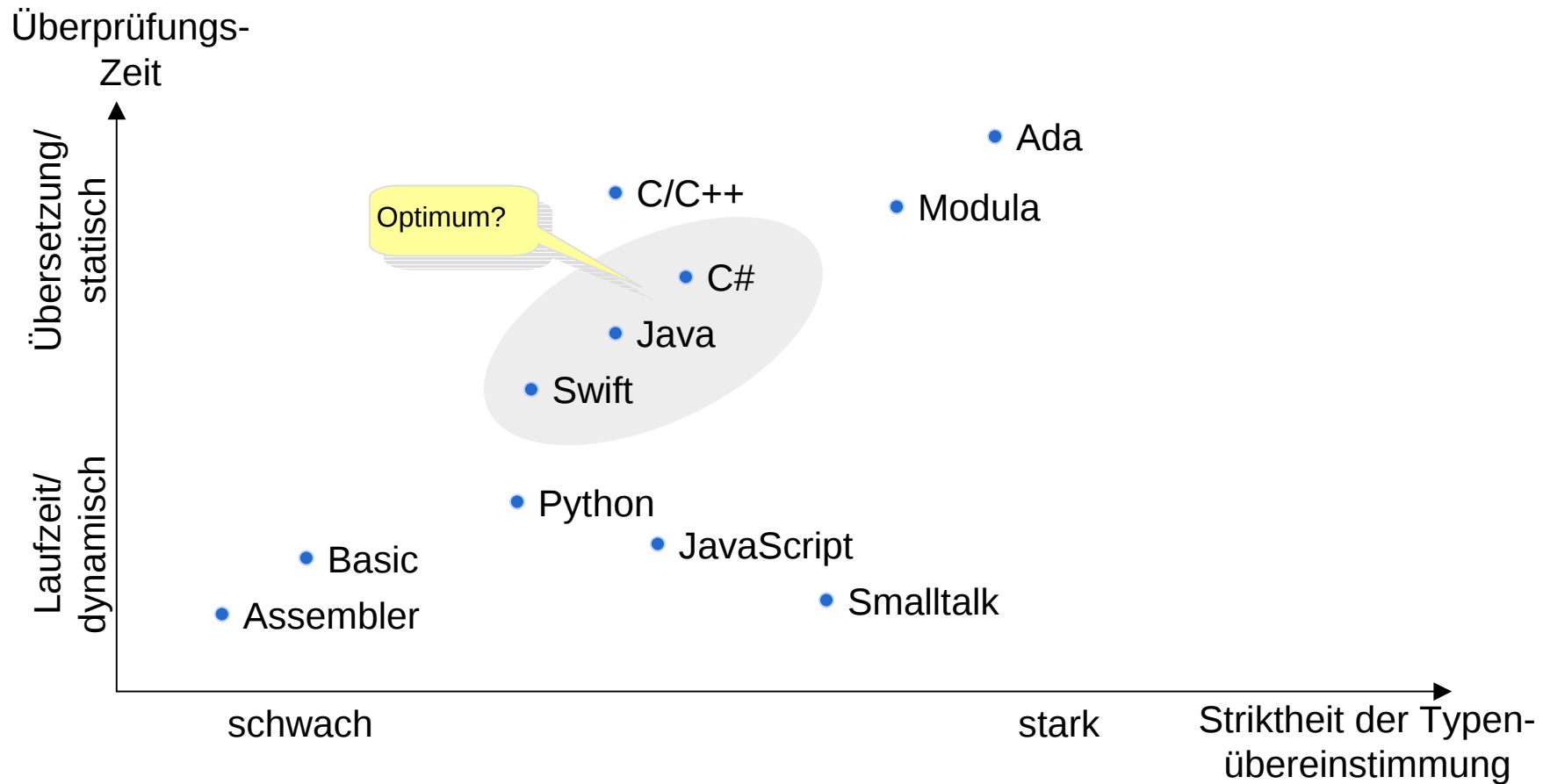
- Verletzungen nicht erlaubt -> Fehler

■ Schwache Typenüberprüfung/Sicherheit

- Verletzungen führen automatisch zu Typumwandlung oder -Erweiterung

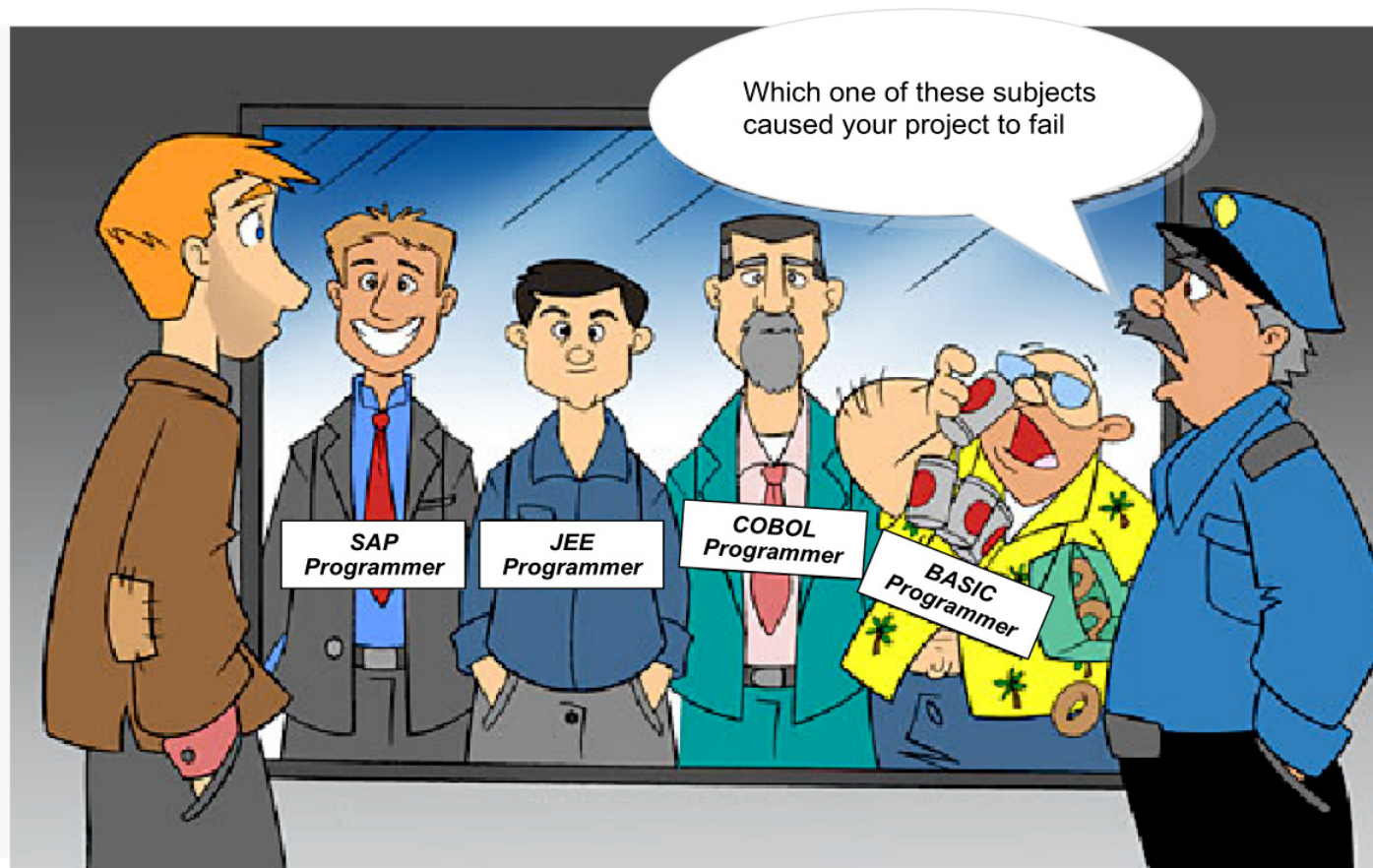
■ Klassifizierung der Programmiersprachen nach Typensystem

- Schwaches Typensystem -> "generische" Programme einfach möglich
- Starkes Typensystem -> Mehr Fehler werden schon während der Übersetzung erkannt



Projekterfolg hängt auch von Sprache ab

- Persönlichkeit des Programmierers und Wahl der Programmiersprache



Entwicklung von generischen Klassen

- Datentypen wurden in die Programmiersprachen eingeführt, um die Anwendung von Programmiersprachen sicherer zu machen.
 - Assembler und frühe C Versionen kannten keine Datentypen-> oft Quelle von Fehlern
- Mit dem Datentyp wird die Menge der Operationen (Operatoren), die auf Werte dieses D.T. angewandt werden können, eingeschränkt (constraint).
 - z.B. $6 * 8$ aber nicht `"hallo" * "world"` oder `int i = "hallo"`
- Es soll aber trotzdem möglich sein, eine Datenstruktur / einen Algorithmus auf Werte verschiedener Datentypen anzuwenden
 - z.B. Datenstrukturen: Liste, Stack, Queue
 - z.B. Algorithmen: Sortieren

Einen Algorithmus, der auf Werte von unterschiedlichen Datentypen angewandt werden kann, nennt man *generisch*.

Generizität bisher (bis Java 1.4)

- Generizität erreicht durch:

- 1) überladen von Methoden

```
int max(int i, int j);  
double max(double d, double e);
```

- 2) Object als Parameter (als Oberklasse aller Klassen)

```
class Box {  
    private Object val;  
    void setValue(Object val ) {  
        this.val = val;  
    }  
    Object getValue() {  
        return val;  
    }  
}  
intBox.setValue(new Integer(32));  
int i = (Integer)intBox.getValue();
```

© Java ist auch eine Insel
Christian Ullenboom

- beim Lesen muss ein Cast zum gewünschten Typ durchgeführt werden
- Es können Werte von beliebigen D.T. eingefügt werden obwohl das u.U. keinen Sinn macht: Laufzeitfehler beim Lesen (TypeCastException)

Generizität bisher (bis Java 1.4)

■ 3) für jeden Datentyp eine eigene Klasse

Container für eine einfache Zahl (Datentyp int)

```
class IntBox {  
    private int val;  
    void setValue( int val ) {  
        this.val = val;  
    }  
    int getValue() {  
        return val;  
    }  
}
```

Das Gleiche für String, Integer, Float, Double etc.

```
class StringBox {  
    private String val;  
    void setValue(String val ) {  
        this.val = val;  
    }  
    String getValue() {  
        return val;  
    }  
}
```

© Java ist auch eine Insel
Christian Ullenboom

Generische Klassen (ab Java 5)

- für den Typ wird lediglich ein Platzhalter, z.B. T, E, V eingesetzt
- der Typ kann später bei der Instanziierung (Variablendeklaration) festgelegt werden: Konkretisierung des Typs

```
class Box<T> {  
    private T val;  
  
    void setValue(T val ) {  
        this.val = val;  
    }  
  
    T getValue() {  
        return val;  
    }  
}
```

Platzhalter durch < >
gekennzeichnet

© Java ist auch eine Insel
Christian Ullenboom

■ Deklarationen:

- `Box<String> box = new Box<String>();`
- `Box<Integer> box = new Box<Integer>();`
- `Box<Point> box = new Box<Point>();`

Bei der
Instanziierung wird
der Typ konkretisiert

- Einschränkung: für T nur Referenztypen erlaubt, keine einfachen Datentypen

Verwendung von generischen Typen

- Es sind keine Typenumwandlungen (Casts) notwendig
- Fehler werden während der Übersetzung erkannt.

```
Box<String> box = new Box<String>();  
box.setValue("hallo");  
String x = box.getValue();
```

Konkretisierung des
Platzhaltertyps

- funktioniert auch mit einfachen Typen dank Boxing

```
Box<Integer> box = new Box<Integer>();  
box.setValue(42);  
int x = box.getValue();  
box.setValue("hallo"); // Compiler Fehler
```

automatisches
Boxing

Generische Methoden

Methoden mit Typparameter

"<T>" vor dem Rückgabebetyp

```
static <T> void foo(T arg) {  
  
}  
foo(4);
```

- Der konkrete Typ muss nicht angegeben sondern wird anhand der Parameter hergeleitet (Type Inference)

```
static <T> T foo(T t) {  
    return t;  
}  
int i = foo(4);  
int i = foo(4.3); --> Compile-Error
```

- Generische Methoden können auch in nicht-generischen Klassen verwendet werden.

Subtyping von generischen Klassen

Erben bei generischen Klassen

- Es kann auf drei Arten von generischen Klassen geerbt werden
- Die erbende Klasse ist generisch, geerbte nicht

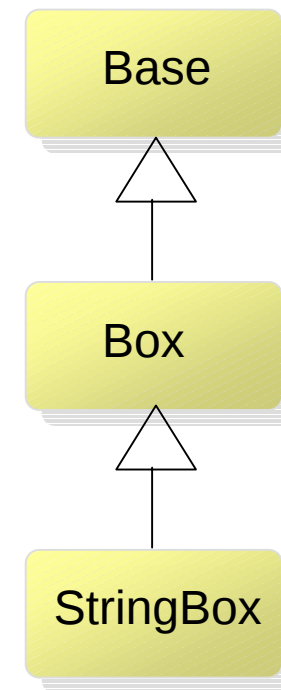
```
class Box<E> extends Base
```

- Die erbende Klasse bleibt weiterhin generisch

```
class Box<E> extends Base<E>
```

- Die erbende Klasse konkretisiert den Typ

```
class StringBox extends Box<String>
```



```
Base base;  
Box box;  
StringBox sbox;  
base = box; //ok  
box = sbox; //ok
```

Generische Interfaces

```
public interface List<E> {  
    public void add(E e);  
    public E get(int i);  
    ...  
}
```

```
public class LinkedList <E> implements List<E> {  
    private E first= null;  
  
    public void add(E e){  
        ...  
    }  
    ...  
}
```

```
List<Integer> list = new LinkedList<Integer>();
```

Mehrere generische Typen

- Mehrere Platzhaltertypen werden einfach durch "," abgetrennt

```
public interface Map <K,V> {  
    public void put(K key, V value);  
    public V get(K key);  
    ...  
}
```

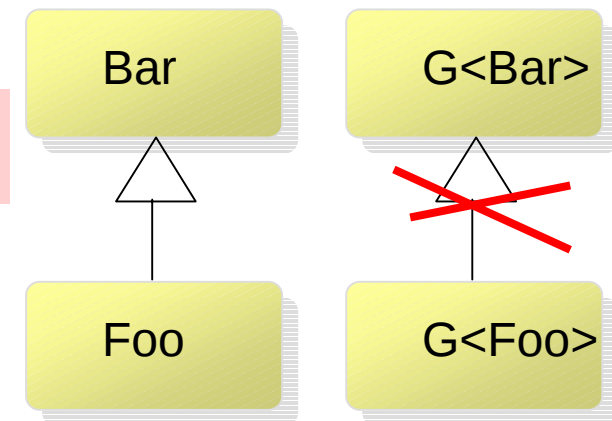
im JDK Object wegen
Abwärtskompatibilität

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls; ok?
```

■ Problem

```
lo.add(new Integer(4)); // unsafe  
String s = ls.get(0); // run time error
```

nein! Compile time error.



Wenn Foo Oberklasse Bar ist und G eine generische Typendeklaration dann ist **G<Foo>** keine Oberklasse von **G<Bar>**.

Übung 1 Generischer ADT

- 1) Definieren Sie das Stack Interface generisch

- 2) Implementieren Sie diese Klasse und beschreiben Sie die Methoden aber noch ohne Implementation

Übung 2 Implementation

- 1) Implementieren Sie die push Methode Ihrer ListStack Methode mittels einer generischen Liste

- 2) Instanzieren Sie eine Stack Klasse von Integer

Wildcards

Klassen mit Wildcards

- Ein bewusstes "Vergessen" der Typinformationen lässt sich durch das Wildcard Zeichen '?' erreichen.
- Es erlaubt, verschiedene Unterklassen zusammenzuführen.
- Wildcards dienen dazu, unterschiedliche parameterisierte Typen zuweisbar und damit in einer Methode benutzbar zu machen.

Nur lokale Variable und Methoden-Parameter können mit Wildcards benutzt werden!

```
Box<Object> b;  
Box<Integer> bI = new Box<Integer>();  
Box<Double> bD = new Box<Double>();  
b = bI;    // --> Compile-Error  
b = bD;    // --> Compile-Error
```

■ Variablendeklaration mit Wildcards

```
Box<?> bw;  
bw = bL;    // ok  
bw = bI;    // ok  
bw = b;     // ok
```

Methoden mit Wildcards

- Eine Methode die alle enthaltenen Elemente einer Collection ausgibt:

früher:

```
public void printCollection(Collection c) {  
    iterator iter = c.iterator();  
    while(iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

Methoden mit Wildcards

- Eine Methode die alle enthaltenen Elemente einer Collection ausgibt:
- naiver Ansatz

Java 5

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

- Funktioniert aber nur mit Collections mit dem Parametertyp <Object>
- List<String> kann ja nicht List<Object> zugewiesen werden

Methoden mit Wildcards

- Eine Methode die alle enthaltenen Elemente einer Collection ausgibt:

Java 5

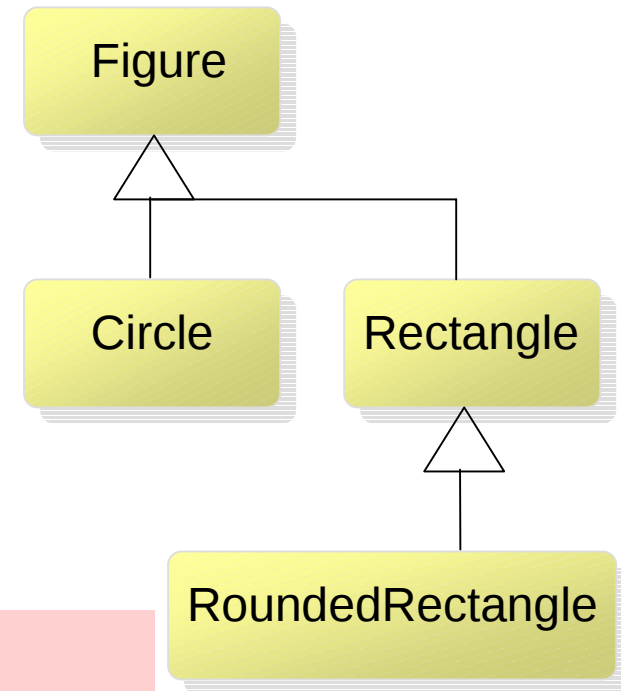
```
public void printCollection(Collection<?> c) {  
    for(Object e:c) {  
        System.out.println(e);  
    }  
}
```

- **Collection<?>** ist der Oberklasse aller Collections

Bounds

Generics, Bounded Wildcards

```
abstract class Figure {  
    public abstract void draw();  
}  
class Circle extends Figure {  
    public void draw() {}  
}  
class Rectangle extends Figure {  
    public void draw() {}  
}  
class RoundedRectangle extends Rectangle  
{  
    public void draw() {}  
}
```



```
public void drawAll(List<Figure> figures) {  
    for (Figure f : figures) f.draw();  
}  
  
List<Circle> lc = new LinkedList<Circle>();  
drawAll(lc); // ok ?
```

nein! Compile time error.

Generics, Bounded Wildcards

```
public void drawAll(List<?> figures) {  
    for (Figure f : figures) f.draw();  
}
```

```
List<Circle> lc = new LinkedList<Circle>();  
drawAll(lc); // ok ?
```

nein! Compile time error.:
draw Methode nicht bekannt

- Problem List<Figure> ist zu restriktiv List<?> zu offen (keine Information über Typ)
- Wir wollen eigentlich ausdrücken: "irgend ein Typ der von Figure erbt"
- **Upper Bound Wildcard**

```
public void drawAll (List<? extends Figure> figures) {  
    for (Figure f : figures) f.draw();  
}  
  
List<Circle> lc = new ArrayList<Circle>();  
drawAll(lc);
```

Bounded Wildcards, Methoden

- Abhängigkeit zwischen zwei Typen in Kombination mit Wildcards

```
class Collections {  
    public static <T> void copy(List<T> dest, List<? extends T> src) { ... }  
}
```

- auch möglich

```
class Collections {  
    public static <T, S extends T> void copy(List<T> dest, List<S> src) {...}  
}
```

WildCards bei Rückgabetyp

- Upper bound wildcards (**extends**) erlauben lesende Funktionen, also Funktionen, die einen parametrisierten Typ zurückgeben.

```
public List<? extends Figure> getFigures () { }
```

Oberklasse aller numerischen Typen

- `java.lang.Number` ist Oberklasse von `Integer`, `Long`, `Double`, ..

- Somit

```
public List<? extends Number> getValueList() { }
```

- Eine Liste von numerischen Werten

```
class Number {  
    double doubleValue() Returns the value of the specified number as a double.  
    float floatValue() Returns the value of the specified number as a float.  
    int intValue() Returns the value of the specified number as an int.  
    long longValue() Returns the value of the specified number as a long.  
    short shortValue() Returns the value of the specified number as a short.  
}
```

Aufgabe 3

- 1) Definieren Sie eine generische **max** Methode (Werte müssen von identischem Typ und vergleichbar sein)

- 2) Definieren Sie eine generische **PriorityQueue** Klasse , so dass der Wert beliebig und die Priorität ein von Comparable abgeleiteter Typ ist

```
public class PriorityQueue  
    public push(Object o, Priority p)
```

Raw Types & Type-Erasure

- Wenn man eine Variable ohne "<>" (wie 1.4) deklariert, dann spricht man von einem Raw Type
- Raw Types und Generic Types sind Zuweisungskompatibel; die statische Typensicherheit geht aber verloren; es werden deshalb vom Compiler Warnungen generiert
- Mittels **SuppressWarnings("unchecked")** lassen sich diese ausschalten

```
Box<String> bs = new Box<String>  
Box raw;    //Raw Type  
  
raw = bs;   // ok  
@SuppressWarnings("unchecked")  
bs = raw;   // unchecked Warning  
bs = (Box<String>)raw;    // unchecked Warning
```

- Java entfernt die Typeninformation vollständig zur Laufzeit
- Entscheid des Java Design Teams zur Implementierung der Aufwärtskompatibilität
- Aus `Box<T>` wird zur Laufzeit `Box<Object>` und
- Störend: Daraus ergeben sich Einschränkungen
 - keine Typenprüfung möglich: `if (e instanceof List<Integer>) ...`
 - Cast sind nicht überprüfbar -> Warnung `E e = (E)o;`
 - kein Instanzierung möglich `E e = new E();`
 - keine Arrays von `E[]` `a = new E[10];`

- Beim Ablauf des Programms kann nicht mehr von z.B. `LinkedList<String>` und `LinkedList<Integer>` unterschieden werden.
Beide haben zur Laufzeit den Typ `LinkedList<Object>`

```
List<String> l1 = new ArrayList<String>();  
List<Integer> l2 = new ArrayList<Integer>();  
System.out.println(l1.getClass() == l2.getClass());
```

- Auch Casts funktionieren nicht wirklich -> Warning von Compiler

```
<T> T badCast(T t, Object o) { return (T) o; // unchecked warning
```

- Wieder mit `@SuppressWarnings`

```
@SuppressWarnings("unchecked")  
<T> T badCast(T t, Object o) { return (T) o; // unchecked warning
```

Type Erasures und Erzeugung von Instanzen

- Erzeugung von Instanzen funktioniert nicht

```
E e = new E(); // not allowed
```

- Lösung: man gebe beim Aufruf noch die Class<T> mit;
- dann kann ich mit newInstance eine Instanz erzeugen

```
<T> T foo(Class<T> clazz, String s) {  
    return (T)clazz.newInstance();  
}
```

```
String s = foo(String.class, "hallo");
```

- Falls Wert nicht verwendet wird (Spezialfall siehe Hashing Vorlesung)

```
E e = (E)new Object();
```

Type Erasures und Arrays

- Eine Array von generischen Typen kann nicht angelegt werden

```
List<T>[] lsa = new List<T>[10]; // not allowed
```

- Wildcard Typ instanzieren ist erlaubt

```
List<?>[] lsa = new List<?>[10]; // ok, array of unbounded type
```

- a) Mit Wildcard Typ erstellen und cast -> warnung

```
List<T>[] lsa = (List<T>[])new List<?>[10]; //
```

- b) mit Object und cast -> warnung (geht, weil auf Referenztyp beschränkt)

```
List<T>[] lsa = (List<T>[])new Object[10]; //
```

- Autoboxing und Unboxing
- Erweiterte for-Schleife
- Variable Anzahl Methoden-Argumente und printf
- Enumerationen
- Statische Imports
- Generics
 - Generische Typen und Methoden erhöhen die Sicherheit, da Typprüfungen zur Laufzeit reduziert werden
 - Viele falsche Typzuweisungen werden vom Compiler abgelehnt
 - Die Aussagekraft der Quelle wird erhöht
 - Es gibt aber Einschränkungen und Ausnahmen, die das Arbeiten mit Java Generics erschweren