

# 13 Polymorphie

## Problem:

- ▶ Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- ▶ Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur nochmal definieren.

## 13.1 Unterklassen-Polymorphie

Eine Funktion, die für mehrere Argumenttypen definiert ist, heißt auch **polymorph**.

### Idee:

Überall wo ein Objekt vom Typ `ClassA` verwendet wird, können wir auch ein Objekt einer Unterklasse von `ClassA` nutzen.

- ▶ Zuweisungen:

```
ClassA a;  
ClassB b = new ClassB();  
a = b; // weise Objekt einer Unterklasse zu
```

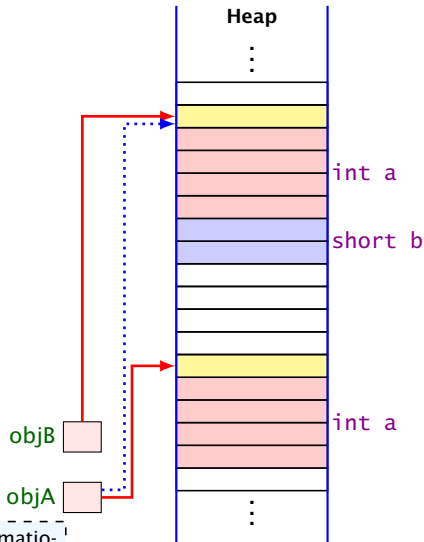
- ▶ Methodenaufrufe:

```
void meth(ClassA a) {};  
ClassB b = new ClassB();  
void bla() {  
    meth(b); // rufe meth mit Objekt von  
            // Unterklasse auf  
}
```

# Was passiert hier eigentlich?

```
public ClassA {  
    int a;  
}  
public ClassB extends ClassA {  
    short b;  
}  
public class Test {  
    public static void main() {  
        ClassA objA = new ClassA();  
        ClassB objB = new ClassB();  
        objA = objB;  
    }  
}
```

Ein Objekt vom Typ **ClassB** ist auch ein Objekt vom Typ **ClassA**.



Die gelben Felder enthalten Verweise auf Typinformationen, die aber für diesen Fall der Zuweisung nicht benötigt werden.

## Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10€ gutschreiben, um die Wirtschaft anzukurbeln:

```
void boostEconomy(BankAccount[] arr) {  
    for (int i=0; i<arr.length, i++) {  
        arr[i].deposit(10);  
    }  
}
```

## Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10€ gutschreiben, um die Wirtschaft anzukurbeln:

Das Array `arr` enthält Objekte unterschiedlicher Typen.

```
void boostEconomy(BankAccount[] arr) {  
    for (int i=0; i<arr.length, i++) {  
        arr[i].deposit(10);  
    }  
}
```

- ▶ Die Methode bekommt ein Array mit allen Konten übergeben.
- ▶ Die einzelnen Elemente des Arrays können `BankAccount`, `CheckingAccount`, `SavingsAccount`, oder `BonusSaverAccount` sein.
- ▶ Es wird jeweils die Methode `deposit` aufgerufen, die in der Klasse `BankAccount` implementiert ist.

Beachte, dass die Unterklassen von `BankAccount` die Methode `deposit` nicht überschreiben.

## Realistisches Beispiel

Die Mafia ist durch einen Hack in den Besitz einer großen Menge von Bankdaten gekommen. Diese gilt es auszunutzen:

```
void exploitHack(BankAccount[] arr) {  
    for (int i=0; i<arr.length, i++) {  
        arr[i].withdraw(10);  
    }  
}
```

- ▶ Hier wird die (spezielle) `withdraw`-Methode des jeweiligen Account-Typs aufgerufen.
- ▶ Die kann der Compiler aber nicht kennen!!!
- ▶ Dynamische Methodenbindung!!!

## Realistisches Beispiel

Die Mafia ist durch einen Hack in den Besitz einer großen Menge von Bankdaten gekommen. Diese gilt es auszunutzen:

```
void exploitHack(BankAccount[] arr) {  
    for (int i=0; i<arr.length, i++) {  
        arr[i].withdraw(10);  
    }  
}
```

- ▶ Hier wird die (spezielle) `withdraw`-Methode des jeweiligen Account-Typs aufgerufen.
- ▶ **Die kann der Compiler aber nicht kennen!!!**
- ▶ **Dynamische Methodenbindung!!!**

# Statischer vs. dynamischer Typ

Der **statische Typ** eines Ausdrucks ist, der Typ, der sich gemäß den Regeln zu Auswertung von Ausdrücken ergibt.

Der **dynamische Typ** eines Referenzausdrucks **e** ist der Typ des wirklichen Objekts auf das **e zur Laufzeit** zeigt.

## Beispiel:

```
SavingsAccount s = new SavingsAccount(89,10,0.2);  
BankAccount b = s;
```

```
s //statischer Typ SavingsAccount  
b //statischer Typ BankAccount
```

```
s //dynamischer Typ SavingsAccount  
b //dynamischer Typ SavingsAccount
```



# Ermittlung der aufgerufenen Methode

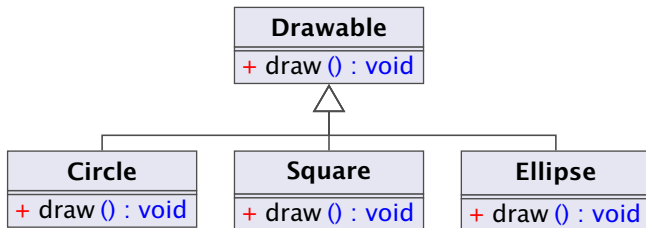
Das Verfahren beschreibt den Vorgang für **Objektmethoden**. Bei statischen Aufrufen, würde die in Schritt 2 gefundene Methode gewählt.

Betrachte einen Aufruf  $e_0.f(e_1, \dots, e_k)$

1. Bestimme die **statischen** Typen  $T_0, \dots, T_k$  der Ausdrücke  $e_0, \dots, e_k$ .
2. Suche in einer **Oberklasse** von  $T_0$  nach einer Methode mit Namen  $f$ , deren Liste von Argumenttypen bestmöglich zu der Liste  $T_1, \dots, T_k$  passt.  
Sei  $S$  **Signatur** dieser rein statisch gefundenen Methode  $f$ .
3. Der **dynamische** Typ  $D$  des Objekts, zu dem sich  $e_0$  auswertet, gehört zu einer Unterklasse von  $T_0$ .
4. Es wird die Methode  $f$  aufgerufen, die Signatur  $S$  hat, und die in der nächsten Oberklasse von  $D$  implementiert wird.

Das Ermitteln der Methode, die am besten **passt**, wurde schon im Kapitel über die Auswertung von Ausdrücken behandelt. Es kommen nur zusätzliche implizite Typecasts hinzu: Ein Cast von einer Unterklasse in eine zugehörige Oberklasse ist immer möglich, und wird vom Compiler als **impliziter Typecast** durchgeführt.

## Weiteres Beispiel



```
1 public class Figure {
2     Drawable[] arr; // contains basic shapes of figure
3     Figure(/* some parameters */) {
4         /* constructor initializes arr */
5     }
6     void draw() {
7         for (int i=0; i<arr.length; ++i) {
8             arr[i].draw();
9         }
10 } }
```

# Die Klasse `Object`

- ▶ Die Klasse `Object` ist eine gemeinsame Oberklasse für **alle** Klassen.
- ▶ Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.

# Die Klasse Object

Einige nützliche Methoden der Klasse `Object`:

- ▶ `String toString()` liefert Darstellung als `String`;
- ▶ `boolean equals(Object obj)` testet auf **Objekt-Identität** oder Referenz-Gleichheit:

```
1 public boolean equals(Object obj) {  
2     return this == obj;  
3 }
```

- ▶ `int hashCode()` liefert Nummer für das Objekt.
- ▶ ...viele weitere **geheimnisvolle Methoden**, die u.a. mit **paralleler Programmausführung** zu tun haben.

**Achtung:** `Object`-Methoden können aber in Unterklassen durch geeignete Methoden überschrieben werden.

# Beispiel

```
1 class PolyA {
2     public String toString() { return "Hello"; }
3 }
4 public class PolyTestA {
5     public static String addWorld(Object x) {
6         return x.toString() + " World!";
7     }
8     public static void main(String[] args) {
9         Object x = new PolyA();
10        System.out.println(addWorld(x));
11    }
12 }
```

liefert: "Hello World!"

- ▶ Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- ▶ Die Klasse `PolyA` ist eine Unterklasse von `Object`.
- ▶ Einer Variable der Klasse `ClassA` kann ein Objekt **jeder Unterklasse** von `ClassA` zugewiesen werden.
- ▶ Darum kann `x` das neue `PolyA`-Objekt aufnehmen.

Die Klasse `PolyA` enthält keinen explizit definierten Konstruktor.

Eine Klasse `ClassA`, die keinen anderen Konstruktor besitzt, erhält einen **Default-Konstruktor** `public ClassA()`.

# Beispiel

```
1 class PolyB {
2     public String greet() { return "Hello"; }
3 }
4 public class PolyTestB {
5     public static void main(String[] args) {
6         Object x = new PolyB();
7         System.out.println(x.greet()+" World!");
8     }
9 }
```

liefert: **Compilerfehler**

Method greet() not found in class java.lang.Object.  
System.out.print(x.greet() + " World!\n");

^

1 error

# Erklärung

- ▶ Die Variable `x` ist als `Object` deklariert.
- ▶ Der Compiler weiß nicht, ob der aktuelle Wert von `x` ein Objekt aus einer Unterklasse ist, in welcher die Objektmethode `greet()` definiert ist.
- ▶ Darum lehnt er dieses Programm ab.



# Methodenaufruf

Der Aufruf einer **statischen** Methode:

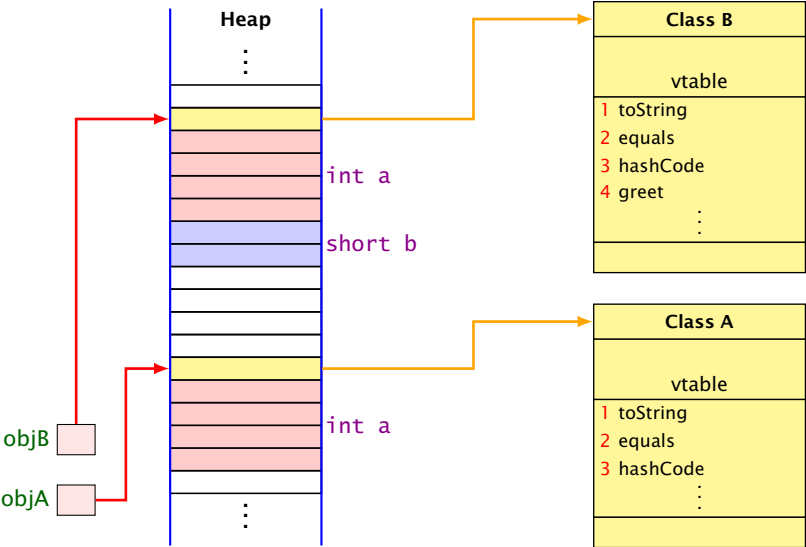
1. Aktuelle Parameter und Rücksprungadresse auf den Stack legen.
2. Zum Code der Funktion springen.

Aufruf einer Objektmethode:

1. Aktuelle Parameter (auch **this**) und Rücksprungadresse auf den Stack legen.
2. **Problem:** Die aufgerufene Funktion ist zur Compilezeit noch nicht bekannt; existiert vielleicht nicht einmal.

Die Funktion `addWorld()` im vorherigen Beispiel könnte schon existieren, bevor die Klasse `Poly` geschrieben wird. Wie kann dann `addWorld()` die richtige `toString()`-Funktion aufrufen?

# Methodenaufruf



## Methodenaufruf

Ein Aufruf `obj.equals()` wird wie folgt verarbeitet.

- Suche die `vtable` des Objekts auf das `obj` zeigt.
- Suche in dieser `vtable` nach dem Index von `equals()`.
- Springe an die dort gespeicherte Sprungadresse.

- ▶ Jede Klasse hat eine Tabelle (`vtable`) mit Methoden, die zu dieser Klasse gehören. Darin wird die Adresse des zugehörigen Codes gespeichert.
- ▶ Ein Aufruf einer Objektmethode (z.B. `equals`) sucht in dieser Tabelle nach der Sprungadresse.
- ▶ Beim **Überschreiben** einer Methode in einer Unterklasse wird dieser Eintrag auf die Sprungadresse der neuen Funktion geändert.
- ▶ **Dynamische Methodenbindung**

### Wichtig

Der Index der Funktionen innerhalb der (`vtable`) ist in jeder abgeleiteten Klasse gleich.

Dies ist nur eine (sehr naheliegende) Möglichkeit dynamische Methodenbindung zu realisieren. D.h. nicht, dass die **JVM** dies genau so umsetzt.

# Beispiel

```
1 class PolyB {
2     public String greet() { return "Hello"; }
3 }
4 public class PolyTestB {
5     public static void main(String[] args) {
6         Object x = new PolyB();
7         System.out.println(x.greet()+" World!");
8     }
9 }
```

liefert: **Compilerfehler**

```
Method greet() not found in class java.lang.Object.
    System.out.print(x.greet() + " World!\n");
```

^

1 error

Benutze einen expliziten **cast** in die entsprechende Unterklasse!

```
class PolyC {
    public String greet() { return "Hello "; }
}
public class PolyTestC {
    public void main(String[] args) {
        Object x = new PolyC();
        if (x instanceof PolyC)
            System.out.println(((PolyC) x).greet()+"World!");
        else
            System.out.print("Cast not possible!\n");
    }
}
```

Java vergisst die Zugehörigkeit zu **B** nicht vollständig. Bei einem Aufruf von Objektmethoden, werden evt. von **B** überschriebene Methoden aufgerufen.

- ▶ Eine Variable **x** einer Klasse **A** kann Objekte **b** aus sämtlichen Unterklassen **B** von **A** aufnehmen.
- ▶ Durch diese Zuweisung vergisst **Java** die Zugehörigkeit zu **B**, da **Java** alle Werte von **x** als Objekte der Klasse **A** behandelt.
- ▶ Mit dem Ausdruck **x instanceof B** können wir zur **Laufzeit** die Klassenzugehörigkeit von **x** testen;
- ▶ Sind wir uns sicher, dass **x** aus der Klasse **B** ist, können wir in diesen Typ **casten**.
- ▶ Ist der aktuelle Wert der Variablen **x** bei dem versuchten Cast tatsächlich ein Objekt (einer Unterklasse) der Klasse **B**, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine **Exception** ausgelöst.

Umwandlungen zwischen primitiven Datentypen (widening/narrowing conversions) erzeugen keine Laufzeitfehler.



# 13 Polymorphie

## Problem:

- ▶ Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- ▶ Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur nochmal definieren.



## Beispiel — Listen

Wir definieren Liste für `Object` anstatt jeweils eine für `Rational`, `BankAccount`, etc.

```
1 public class List {
2     public Object info;
3     public List next;
4     public List(Object x, List l) {
5         info = x;
6         next = l;
7     }
8     public void insert(Object x) {
9         next = new List(x, next);
10    }
11    public void delete() {
12        if (next != null) next = next.next;
13    }
14 // continued...
```

## Beispiel — Listen

```
14     public String toString() {
15         String result = "[" + info;
16         for (List t = next; t != null; t = t.next)
17             result = result + ", " + t.info;
18         return result + "]";
19     }
20     ...
21 } // end of class List
```

- ▶ Die Implementierung funktioniert ganz analog zur Implementierung für `int`.
- ▶ Die `toString()`-Methode ruft implizit die (stets vorhandene) `toString()`-Methode der Listenelemente auf.

# Beispiel — Listen

## Achtung:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = list.info;
5 System.out.println(x);
6 //...
```

liefert einen **Compilerfehler**. Der Variablen `x` dürfen nur Unterklassen von `Poly` zugewiesen werden.

# Beispiel — Listen

## Stattdessen:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = (Poly) list.info;
5 System.out.println(x);
6 //...
```

Das ist hässlich!!! Geht das nicht besser???

## 13.2 Generische Klassen

### Idee:

- ▶ Java verfügt über generische Klassen...
- ▶ Anstatt das Attribut `info` als `Object` zu deklarieren, geben wir der Klasse einen `Typ-Parameter T` für `info` mit!
- ▶ Bei Anlegen eines Objekts der Klasse `List` bestimmen wir, welchen `Typ T` und damit `info` haben soll...

## Beispiel — Listen

```
1 public class List<T> {
2     public T info;
3     public List<T> next;
4     public List (T x, List<T> l) {
5         info = x;
6         next = l;
7     }
8     public void insert(T x) {
9         next = new List<T> (x, next);
10    }
11    public void delete() {
12        if (next != null) next = next.next;
13    }
14    //continued...
```

## Beispiel — Listen

```
15     public static void main (String[] args) {
16         List<Poly> list
17             = new List<Poly> (new Poly(), null);
18         System.out.println(list.info.greet());
19     }
20 } // end of class List
```

- ▶ Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- ▶ Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typparameter `T = Poly`.
- ▶ Deshalb ist `list.info` vom Typ `Poly`.
- ▶ Folglich ruft `list.info.greet()` die entsprechende Methode der Klasse `Poly` auf.

# Bemerkungen

```
class List<T> {  
    public static T info;  
}
```

funktioniert nicht, da `info` kein Objekt-Attribut ist.

- ▶ Die Typ-Parameter der Klasse dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden!!!
- ▶ Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:
  - `A<S,T> extends B<T>` ist erlaubt.
  - `A<S> extends B<S,T>` ist **verboten**.
- ▶ `Poly` ist eine Unterklasse von `Object`; aber `List<Poly>` ist **keine** Unterklasse von `List<Object>`!!!



```
1 List<Poly> l;  
2 // some other code...  
3 Poly p = l.info;
```

## Hauptvorteil von Generics:

Der Compiler garantiert, dass in Zeile 3 `l.info` immer ein `Poly`-Objekt ist (**statische Typsicherheit**).

Dieses kann nicht garantiert werden, wenn für Unterklasse `ClassB` von Elternklasse `ClassA` gelten würde, dass `List<ClassB>` Unterklasse von `List<ClassA>` ist.

# Fallstricke

```
1 List<Animal> a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d; // Compilerfehler  
5 a.insert(new Cat());  
6 myDog = d.info;
```

Der Code ist eindeutig fehlerhaft. Deshalb erlaubt der Compiler die Zuweisung in Zeile 4 nicht.

`List<Dog>` ist keine Unterklasse von `List<Animal>`.

# Fallstricke

Ähnliche Probleme ergeben sich bei der Verwendung von Arrays. Deshalb kann man keine Arrays von generischen Typen erstellen.

```
List<Dog>[] arrOfLists = new List<Dog>[100];
```

liefert: **Compilerfehler**

```
TestGenericArray.java:11: error: generic array creation
    List<Dog>[] arrOfLists = new List<Dog>[100];
                                ^
```

1 error

Wir werden später einen Workaround dafür kennenlernen.

# Generics/Type Erasure/Arrays

Parametrisierte Datentypen sind in Java über Type-Erasure implementiert.

Das heißt, dass für die JVM `List<Poly>` und `List<String>` gleich aussehen und es nur einen Typ `List<Object>` gibt.

Nur der Compiler unterscheidet zwischen `List<Poly>` und `List<String>` und stellt z.B. sicher, dass

```
1 List<Poly> l;  
2 // some other code...  
3 Poly p = l.info;
```

in Zeile 3 immer einen `Poly` zurückliefert.

## Beispiel – Arrays

Die Zuweisung in Zeile 4 ist erlaubt, da Arrays in **Java** kovariant sind.

Der Basistyp des Arrays ist (normalerweise) zur Laufzeit bekannt. Deshalb kann in Zeile 5 eine Typprüfung (zur Laufzeit) stattfinden.

```

1 Animal[] a;
2 Dog[] d = new Dog[100];
3
4 a = d;
5 a[0] = new Cat(); // Laufzeitfehler
6 myDog = d[0];

```

Die Tatsache, dass Zeile 5 zur Laufzeitfehlern führen kann, ist eine Schwäche des Typsystems in **Java**.

### Kovarianz/Kontravarianz/Invarianz

Seien  $A$  und  $B$  Typen und  $f$  eine Typrtransformation.

- ▶ Falls  $A \leq B \Rightarrow f(A) \leq f(B)$  heißt  $f$  **kovariant**.
- ▶ Falls  $A \leq B \Rightarrow f(A) \geq f(B)$  heißt  $f$  **kontravariant**.
- ▶ Falls  $A \leq B$  keine Beziehung zwischen  $f(A)$  und  $f(B)$  impliziert heißt  $f$  **invariant**.

Beispiele:

- ▶  $A \leq B \Rightarrow A[] \leq B[]$  (Arrays sind in **Java** kovariant)
- ▶  $A \leq B$  impliziert keine Beziehung zwischen  $List<A>$  und  $List<B>$
- ▶  $A \leq B \Rightarrow A.meth() \leq B.meth()$  (wenn man eine Methode überschreibt muß der Rückgabebetyp kovariant sein)

# Beispiel – Generische Listen

```
1 List<Animal> a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d; // Compilerfehler  
5 a.insert(new Cat());  
6 myDog = d.info;
```

Zeile 4 erzeugt einen Compilerfehler, da generische Datentypen in **Java invariant** sind.

Zeile 5: hier kann **Java** aufgrund von **Type-Erasure** keinen Laufzeitfehler erzeugen. Für die JVM sehen **List<Cat>** und **List<Dog>** gleich aus, also kann man diese Zuweisung nicht verbieten; außerdem ist die Idee hinter generischen Listen, dass **statische** Typsicherheit garantiert wird, d.h. man möchte keinen Laufzeitfehler erzeugen.

Zeile 6: Eine Hauptvorteil von Generics in Java ist, dass diese Zuweisung nicht zu einem Laufzeitfehler führt; d.h., der Compiler stellt zur Compilezeit sicher, dass dieses funktioniert. Deshalb möchte man hier keinen Laufzeitfehler erzeugen.

# Beispiel – Wildcards

```
1 List<?> a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d;  
5 a.insert(new Cat()); // Compilerfehler  
6 myDog = d.info;
```

`List<?>` steht für eine Liste mit einem beliebigen (aber unbekanntem) Datentyp. Man kann solch einer Liste beliebige Listen der Form `List<T>` zuweisen (für konkrete Werte von `T`). Demnach ist Zeile 4 erlaubt.

Zeile 5: Da der eigentliche Typ der Liste (bei Zugriff über `a`) nicht bekannt ist, können keine Methoden aufgerufen werden, die den parametrisierten Datentyp benutzen (aber z.B. `a.length()` könnte man aufrufen).

# Beispiel – Raw Types

Raw Types sollte man so weit wie möglich vermeiden!

```
1 List a;  
2 List<Dog> d = new List<Dog>(...);  
3  
4 a = d;  
5 a.insert(new Cat()); // Compilerwarnung  
6 myDog = d.info; // Laufzeitfehler
```

`List` ist ein **Raw Type**. Bei der Verwendung dieses Typs wird die Typprüfung des Compilers umgangen. Deshalb ist Zeile 4 erlaubt.

Zeile 5: Da `a` eine Liste mit einem unbekanntem Basistyp ist, ist diese Zuweisung gefährlich und kann nicht vom Compiler auf Typsicherheit geprüft werden.

In Zeile 6 bekommt man den Laufzeitfehler, der durch die Umgehung der Typprüfung entsteht. Wenn man keine Raw-Typen verwendet (und alles kompiliert) ist eine Zuweisung wie `myDog = d.info` **statisch** geprüft.



## Beispiel – Arrays

Die JVM kann in Zeile 6 (unten) keinen Laufzeitfehler erzeugen, da durch Type Erasure auf beiden Seiten der Zuweisung einfach nur eine Liste steht. Stattdessen gibt es einen Compilerfehler in Zeile 2.

```

1 Animal[] a;
2 Dog[] d = new Dog[100];
3
4 a = d;
5 a[0] = new Cat(); // Laufzeitfehler
6 myDog = d[0];

```

```

1 Object[] a;
2 List<Dog>[] d =
3     new List<Dog>[100]; // Compilerfehler!
4
5 a = d;
6 a[0] = myCatList; // Laufzeitfehler???
7 myDog = d[0].info;

```

Man sollte bei gleichzeitiger Verwendung von Generics und Arrays in Java aufpassen; die Kovarianz der Arrays und die Type-Erasure sorgen dafür, dass die statische Typprüfung der Generics nicht mehr funktioniert.

# Bemerkungen

- ▶ Für einen Typ-Parameter `T` kann man auch eine Oberklasse (oder ein Interface) angeben, das `T` auf jeden Fall erfüllen soll...

```
1 class Drawable {
2     void draw() {}
3 }
4 public class DrawableList<E extends Drawable> {
5     E element;
6     DrawableList<E> next;
7     void drawAll() {
8         element.draw();
9         if (next == null) return;
10        else next.drawAll();
11    }
12 }
```

## 13.3 Wrapper-Klassen

### Problem

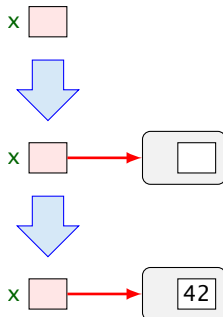
- ▶ Der Datentyp `String` ist eine Klasse;
- ▶ Felder sind Klassen; **aber**
- ▶ **Basistypen** wie `int`, `boolean`, `double` sind keine Klassen!  
(Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.)

### Ausweg

- ▶ Wickle die Werte eines Basis-Typs in ein Objekt ein!  
⇒ **Wrapper-Objekte** aus **Wrapper-Klassen**.

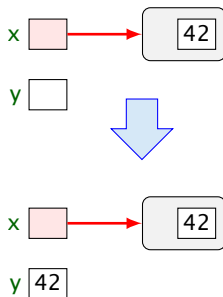
# Beispiel

Die Zuweisung `Integer x = new Integer(42);` bewirkt:



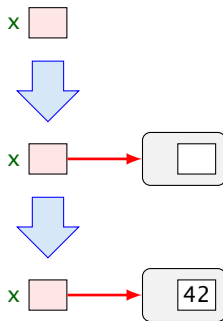
# Beispiel

Eingewickelte Werte können auch wieder ausgewickelt werden.  
Bei Zuweisung `int y = x;` erfolgt **automatische Konvertierung**:



## Beispiel

Umgekehrt wird bei Zuweisung eines `int`-Werts an eine `Integer`-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Man nennt diese Konvertierungen **boxing-** bzw. **unboxing-conversions**

# Nützliches

Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

## Beispiele:

- ▶ `public static int MIN_VALUE = -2147483648;`  
liefert den kleinsten `int`-Wert;
- ▶ `public static int MAX_VALUE = 2147483647;` liefert den größten `int`-Wert;
- ▶ `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem `String`-Objekt `s` die dargestellte Zahl — sofern `s` einen `int`-Wert darstellt.

Andernfalls wird eine `Exception` geworfen.

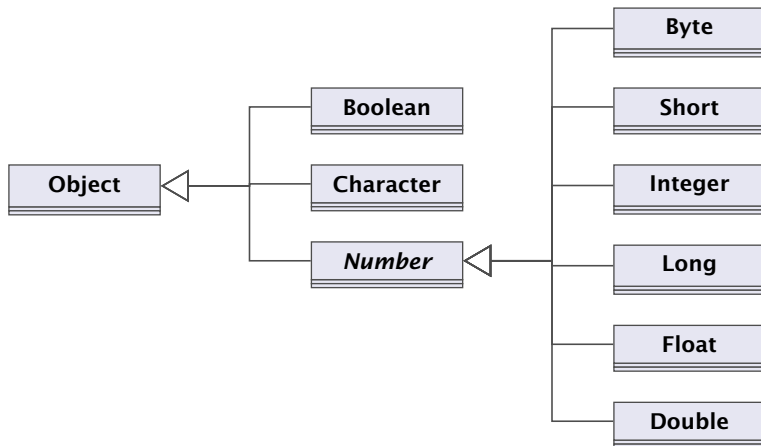
# Bemerkungen

- ▶ Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- ▶ Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- ▶ `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen...



# Wrapper-Klassen



Die Klasse `Number` ist hier in italics geschrieben, da es sich um eine **abstrakte Klasse** handelt.

## Bemerkungen

- ▶ Sämtliche Wrapper-Klassen für Typen `type` (außer `char`) verfügen über
  - ▶ Konstruktoren aus Basiswerten bzw. String-Objekten;
  - ▶ eine statische Methode `type parseType(String s)`;
  - ▶ eine Methode `boolean equals(Object obj)` die auf Gleichheit testet (auch `Character`).
- ▶ Bis auf `Boolean` verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- ▶ `Character` enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln. . .
- ▶ Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- ▶ Diese Klasse ist **↑abstrakt**, d.h. man kann keine `Number`-Objekte anlegen.

# Spezielles

- ▶ `Double` und `Float` enthalten zusätzlich die Konstanten

`NEGATIVE_INFINITY` = `-1.0/0`

`POSITIVE_INFINITY` = `+1.0/0`

`NaN` = `0.0/0`

- ▶ Zusätzlich gibt es die Tests

- ▶ `public static boolean isInfinite(double v);`

- `public static boolean isNaN(double v);`

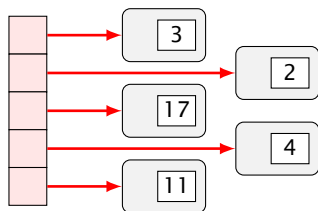
- (analog für `float`)

- ▶ `public boolean isInfinite();`

- `public boolean isNaN();`

mittels derer man auf (Un)Endlichkeit der Werte testen kann.

# Integer vs. Int



Integer[]



int[]

- + **Integers** können in polymorphen Datenstrukturen hausen.
- Sie benötigen mehr als doppelt so viel Platz.
- Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten  
⇒ schlechteres Cache-Verhalten.