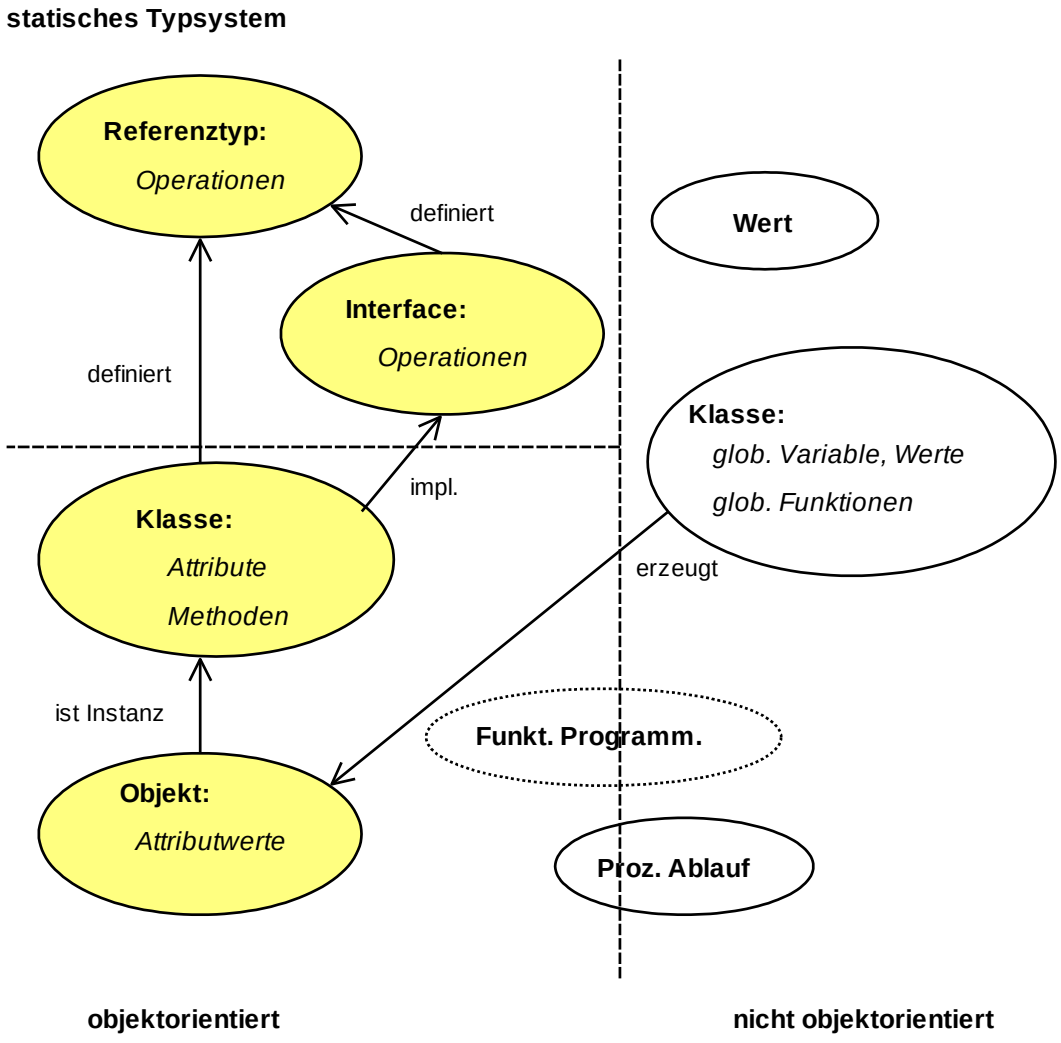
Objekte, Typen und Schnittstellen

- Die unterschiedlichen Aufgaben von Typsystemen
- Abstrakter Datentyp
- Grundbegriffe der Objektorientierung
- Typprüfung bei Objekten
- Objektidentität
- Schnittstellen und Interfaces
- Generalisierung und Spezialisierung
- Zusammenfassung der Syntax

Fokus: Polymorphie und Objektorientierung in Java



Vorbemerkung

Wichtige Besonderheiten der Objektorientierung (in Java):

- besonderes **Typsystem**
- besondere Rolle von **Variablen**
- besondere Funktionsweise des **Methodenaufrufs**
- (Typanpassung ist keine Typumwandlung) (Java)

Java unterscheidet sich von vielen objektorientierten Sprachen durch die Typprüfung im Compiler (statisches Typsystem).

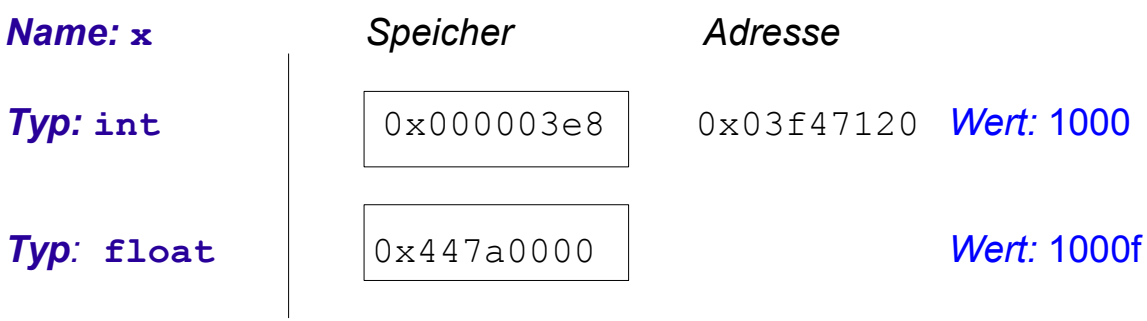
Viele Java-Probleme haben mit der Typprüfung – nicht mit OO zu tun!
(also aufpassen: mal geht es um grundsätzliche, mal um spezielle Dinge!)

Java Datentypen in

Compiler	JVM
boolean	int
byte	
short	
char	
int	
long	long
float	float
double	double
Object	Reference
String	
Stack	
ArrayList	
Integer	

(bei Arrays gibt es ein paar Sonderregeln)

Typsystem – von Neumann Rechner




```
int add(int x, int y) { return x + y; }
add:    pushl %ebp
        movl  %esp, %ebp
        movl  12(%ebp), %eax      // movl = 32 bit Zuweisung
        addl  8(%ebp), %eax       // addl = Addition von 32 bit int
        popl  %ebp
        ret                       // Ergebnis steht in eax
```

- Der Speicher enthält beliebige Daten.
- Die Operationen bestimmen, wie die Daten interpretiert werden.
- Typentscheidungen sind vorher von Programmierer / Compiler getroffen.

Vorteil: maximale Effizienz bei geringem Hardwareaufwand

Nachteile: unsicher, unflexibel, viele Spezialoperationen, …

Typsystem Java Wertdatentypen

```
public static int add(int x, int y) { return x + y; }

public static int add(int, int)
       iload  0     // iload: int Wert auf Stack
       iload  1
       iadd         // iadd: int Addition auf dem Stack
       ireturn      // ireturn: int Rückgabe
```

- Der Speicher enthält beliebige Daten.
- Die Operationen bestimmen, wie die Daten interpretiert werden.
- Typentscheidungen sind vorher von Compiler getroffen.
- Der Java-Befehlssatz enthält redundante Information
- Unterschied zu vN-Rechner: Stackarchitektur anstelle Registerarchitektur

Java Wertdatentypen sind von Neumann Daten

Java Wertdatentypen sind nicht objektorientiert

Allerdings: der ***Bytecode-Verifier*** überprüft vor der Ausführung den Bytecode (Sicherheit)!

Typsystem Objektorientierung (Smalltalk)

```
add: y
    x := x + y
```

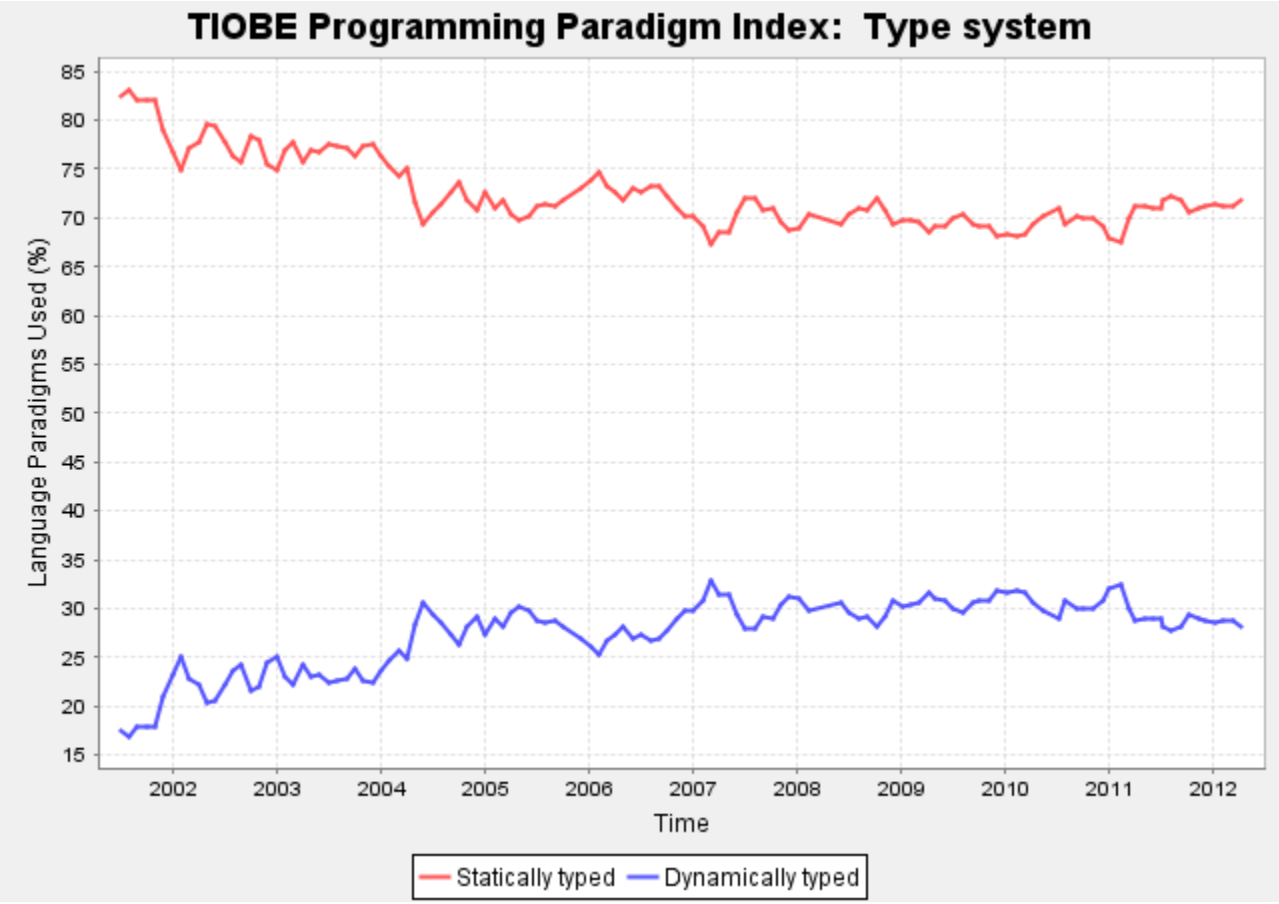
```
add: y
       pushRcvr: 0          // Instvar x
       pushTemp: 0          // temp. Var y
       send: +              // sende + y an x
       popIntoRcvr: 0       // Ergebnis nach x
       returnSelf
```

- Der Speicher enthält Objektreferenzen
- Die Objekte bestimmen, wie die Operationen auszuführen sind
- Der Programmcode enthält keine Typinformation
- Typfehler werden bei der Ausführung (vom Objekt) erkannt

- Referenzdaten haben Typinformation
- Referenzdaten benötigen keine Typangabe
- **Man *kann* auch für Referenzdaten eine Typangabe und Prüfung vorsehen.**

Statisches und Dynamisches Typsystem (70:30)

(Die Daten geben Populariät gemesssen in Suchanfragen wieder)



Wert- und Referenzdatentypen

	Größe	Kodierung	erlaubte Operationen
short	2	2-Kompl.	+ - * /
int	4	2-Kompl.	+ - * /
double	8	IEEE 754	+ - * /
Object	4	Adresse	equals
String	4	Adresse	equals, length, charAt
Stack	4	Adresse	equals, push, pop

Abstrakter Typ: Der Typ der Referenzvariablen bestimmt die Menge der Operationen!

Da alle Referenzen gleich kodiert werden und da jedes Objekt über die Methoden der Klasse `Object` verfügt, kann man beliebige Referenzen in einer Variablen vom Typ `Object` speichern.

Braucht Java für elementare Daten einen Typ?

vielleicht, wenn es die elementaren Daten und ihre Operationen effizient implementieren will

Braucht Objektorientierung einen Typ?

nein, denn zur Laufzeit können die Objekte selbst entscheiden, was geht oder was nicht geht

Warum haben in Java dann Referenzen einen Typ?

Beispiele für Typsysteme

	Variablen-Typ	Laufzeit-Typ	Prüfung	Überein-stimmung	Laufzeit-Wirkung von Cast
Smalltalk	nein	ja	Laufzeit	n.a.	n.a.
C / Java-Werte	ja	nein	Compiler	exakt	Umwandl. der Kodierung
Java-Ref.	**ja**	ja	**Compiler** (Laufzeit)	**Hierarchie**	Prüfung
C-Pointer	ja	nein	(Compiler) (keine)	keine	keine

Smalltalk steht für dynamisch getypte objektorientierte Sprachen und Skriptsprachen.
C/Java-Werte steht für die prozedurale Semantik vom Java und C.
J**ava-Ref.** steht für den objektorientierten Teil von Java.
C-Pointer steht für das unsichere Typsystem der Adressoperationen von C.

Warum haben in Java Referenzen einen Typ?

- der Compiler kann Typfehler feststellen
- der Compiler kann optimieren
- die Dokumentation wird verbessert

- **aber** alles hat seinen Preis:
  - eingeschränkte Wiederverwendbarkeit
  - Schreibaufwand (Rauschen)
  - Typanpassung
  - (sehr) kompliziertes System mit generischen Typen

*Referenzvariablen haben in Java einen **abstrakten Typ** !*

Klasseneigenschaften

Java ist *keine* rein Objektorientierte Sprache! Es vereint (primitive) **Werte** und **Objekte**. Die elementaren Werttypen (**primitives**) sind in die Sprache integriert. Komplexere Arten von Objekten müssen im Programm beschrieben werden. **Objekte** sind die wichtigsten Elemente eines Programms.

Klassenfunktionen und Klassenvariablen *ermöglichen* vollständig prozedural zu programmieren.

Die *eigentliche* Aufgabe einer Klasse ist die Beschreibung und Erzeugung von Objekten,

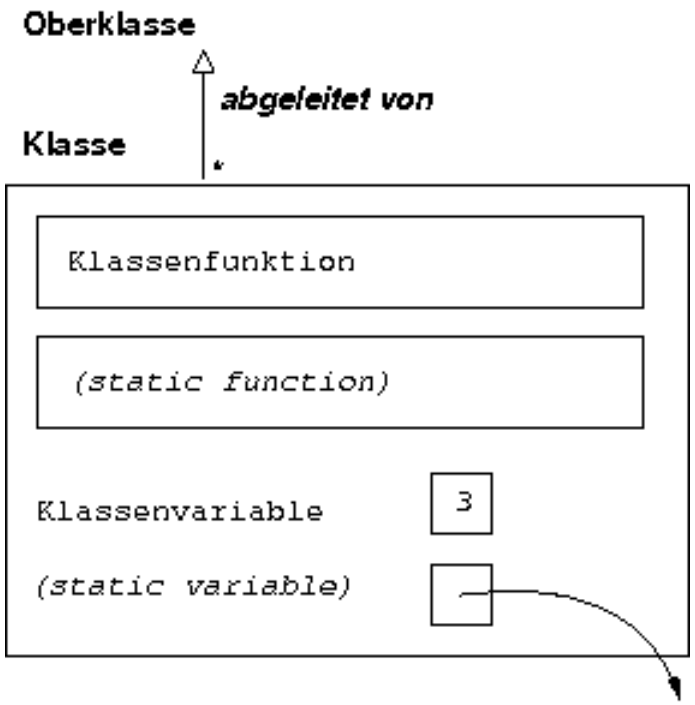
Klassenfunktionen realisieren prozedurale Lösungen (Werte).
Klassenfunktionen können der Erzeugung von Objekten dienen (Fabrikmethoden).

```
public static XyzObject createFromInt(int parameter) {
    return new XyzObject(parameter);
}

XyzObject a = XyzObject.createFromInt(3);
```

Eine Klasse realisiert mittels der Klasseneigenschaften so etwas wie ein einmaliges Objekt mit globalem Namen.
- trotzdem: Java Klassen sind keine Objekte!!

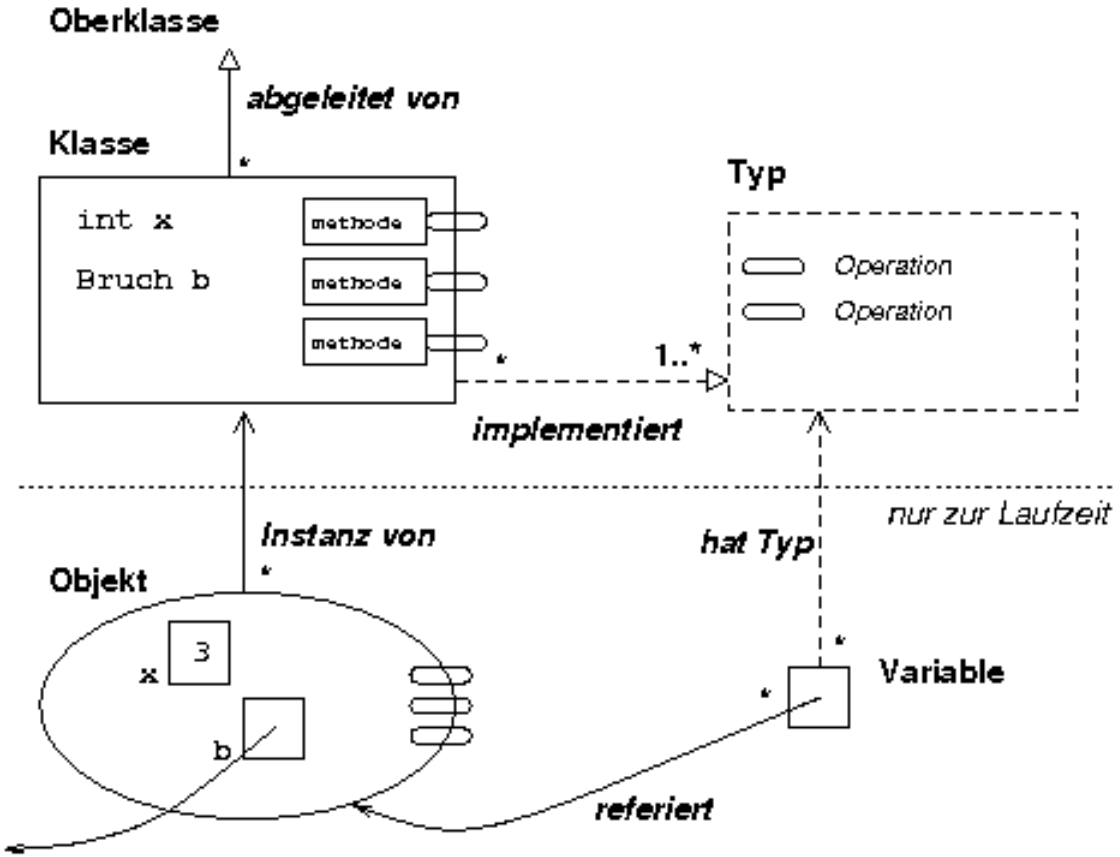
Programmierung mit Klassen (*was Objektorientierung nicht ist)*



Anmerkungen:
Der Unterschied zu C besteht nur darin, dass der Klassenname Bestandteil des Funktions- oder Variablennamens ist. Die „Vererbung“ besteht darin, dass Bestandteile der Oberklasse auch mit dem Namen der Unterklasse angesprochen werden können.

* bedeutet: 0..n Klassen können von einer Klasse abgeleitet sein.

Programmierung mit Objekten (objektorientiert)



Dynamisch getypte Sprachen haben keinen Variablentyp.

Es „funktioniert“ trotzdem, da ja alle nötige Information (zur Laufzeit) da ist.

Anmerkungen:
Zur Übersetzungszeit gibt es keine Variablen und keine Objekte!
Typ = Menge der erlaubten Operationen
Java fordert den Variablentyp zwecks Typprüfung durch den Compiler !
Eine Klasse definiert und implementiert einen Typ.

Sprachverwirrung

- **Prozedur mit Seiteneffekt ohne Rückgabe:**
  subroutine, procedure, function, static method, class function, class method
- **Prozedur mit Seiteneffekt und Rückgabe:**
  procedure, function, static method, class function, class method
- **Funktion ohne Seiteneffekt:**
  pure function, static method
- M**ethode:**
  method, instance method, member function
- **Lokale Variable**
  local variable, temporary variable
- **Instanzvariable:**
  instance variable, member variable, attribute, property, component
- **Globale Variable**:
  global variable, external variable, class variable, static variable, static member

Cast ist nicht gleich Cast

```
intVar = (int) doubleWert
    d2i
doubleVar = intWert
    i2d
intVar = (int) shortWert
    --
StringVar = (String) objectVar
    checkcast String
ObjectVar = (Object) stringVar
    --
```

In der JVM ist **`true`** eine `1`. Warum ist `(`**`int`**`)` **`true`** nicht erlaubt?

Die Spielregeln der Objektorientierung (Schnittstelle**)**

- *„Everything is an object.“* (noch nicht in Java)
- Ein *Objekt* besteht aus *privaten Eigenschaften* und einer Menge von öffentlichen Operationen.
- Der Sender fordert mittels einer *Botschaft* den Empfänger auf, eine Operationen auszuführen.
- Die Menge der *Operationen* eines Objekts, nennt man seinen (abstrakten) *Typ* oder seine *Schnittstelle.*
- Der dynamische Typ eines Objekts ist durch seine Klasse definiert.
- Der statische Typ einer Variablen (Menge erlaubter Botschaften) ist durch ihren Typ (Klasse / Interface) definiert.
- Eine Variable *bezieht sich* (*refers to*) auf ein Objekt.
- Ein Variable speichert nur Referenzen zu Objekten eines *Untertyps*: des Variablentyps. Das Objekt muss über alle Operationen des Variablentyps verfügen.

Die Spielregeln der Objektorientierung (Implementierung**)**

- Die *privaten Eigenschaften* eines Objekts werden beschrieben durch eine durch Menge von *Instanzvariablen* und eine Menge von *Methodenimplementierungen*.
- Eine *Klasse* beschreibt die Implementierung einer Menge von gleichartigen Objekten (*Instanzen*).
- Ein Objekt beschreibt seinen *momentanen Zustand* durch die Werte der Instanzvariablen.
- Ein Objekt reagiert auf eine Botschaft durch die Ausführung einer Methode.
- Ein Programm wird durch einen *Ablauf* (*thread, Ausprache = „thrädd“ nicht „thriet“!*) gesteuert, der zu jedem Zeitpunkt eine Anweisung einer *Methode ausführt.*
- Eine Methode verfügt über *temporäre (lokale) Variable und Parameter* (stack frame).

Typsystem für Objekte = wie geht der Compiler mit Typinformation um?

Der (abstrakte operationale) Typ definiert die Menge der erlaubten Operationen.

Das Typsystem definiert, wie die Programmiersprache den Typ von Werten und Ausdrücken bestimmt und welche Beziehungen zwischen den Typen bestehen,

Für ein Programm, das den Typregeln widerspricht, kann nicht garantiert werden, dass eine Operation wirklich ausgeführt werden kann (dass eine Methode dafür existiert).

Die Typprüfung dient dazu Typfehler zu erkennen und zu melden.

In vielen Programmiersprachen erfolgt die Typprüfung (zum größten Teil) bereits zur Übersetzungszeit (*statische* Typprüfung). Typdeklarationen sind ein gewolltes Element eines statischen Typsystems!

Moderne Compiler können den Variablentyp meist erraten (Typinferenz)!

Aber: Die Typregeln des Compilers müssen beherrscht werden!

Wie verhindert die Typprüfung Fehler?

Die Typprüfung unterdrückt:

- den Aufruf von nicht existierenden Methoden
- den Zugriff zu nicht existierenden Variablen
- die falsche Interpretation von Daten
- den Aufruf zu existierenden Methoden und den Zugriff zu existierenden Variablen, wenn der Compiler (aufgrund der Typdeklaration) nicht genug weiß.

Nochmal: Java verbietet korrekte Aktionen, wenn dem Compiler die nötige Information fehlt!

Wie ermittelt der Compiler den Typ?

- Der Typ von ***Variablen*** und ***Methoden / Funktionen*** ist angegeben: `int x`
- Der Typ von ***Literalen*** ergibt sich von selbst: `3.5, 2f, "abc", {1, 2}`
- Die **Objekterzeugung** legt den Typ fest: `new Bruch(2, 3), new int[5]`
- Der Typ eines ***Ausdrucks*** ergibt sich aus den Typregeln der Programmiersprache: `2 + 3.5`
- Die ***Darstellungsumwandlung*** legt einen neuen Typ fest: `(int) 3.4`
- Die ***Typangabe*** präzisiert die Typinformation: `(String) x`

Basierend auf der Typinformation definiert der Compiler was erlaubt ist (Zuweisung, Funktionsparameter, Methodenaufruf, Operationen)

Problem prozeduraler Programmiersprachen:

- Eine zu einschränkende Prüfung behindert die Programmierung (*Pascal*).
- Eine zu lasche Prüfung findet zu wenig Fehler (*C*).

***Ein* Ausweg:** Typregeln der objektorientierten Programmierung

Typangabe in der Objektorientierung (noch einmal**)**

Der *Typ eines Objekts* oder eines Wertes *bestimmt die Menge aller Operationen*.
Der *Typ einer Variablen* oder eines Ausdrucks *beschreibt eine Teilmenge von Operationen*, die auf dem Wert/Objekt erlaubt sind..

Elementare Datentypen:
Der Typ der Variablen ist **gleich** dem Typ des gespeicherten Wertes.
Jeder Ausdruck hat einen eindeutigen Typ.

Referenzdatentypen:
Der Typ der Variablen ist mit dem Typ des referierten Objekts **verträglich**: **das Objekt „kann“ alles was der Typ der Variablen erlaubt.**
Der genaue Typ eines Ausdrucks ist nicht immer vollständig aus dem Programm erkennbar.

Objekttyp: Was das Objekt ist und was es kann.
Variablentyp / statischer Typ: was man *sicher* über das referierte Objekt weiß.

Dem Compiler sind nur Variablen- und Ausdruckstypen bekannt! Der Compiler kennt keine Objekte!

Es gibt keine Typkonvertierung von Objekten! Die *Typangabe für* einen Ausdrucks durch Cast bezeichnet dem Compiler genauere Information über den Typ des Objekts. Die Laufzeitprüfung überprüft die Verträglichkeit der Cast-Information mit dem Objekttyp (`ClassCastException`).
Der Cast ersetzt die statische durch eine dynamische Typprüfung!

Der Unterschied zwischen (elementarem)Wert und Referenz

Erzeugung:
Werte: durch **Literale** und durch Operationen
Referenzen: durch **New-Ausdruck** (oder durch Array- / String-Literal)
Referenzen zu Wrapper-Objekten: **implizit** aus elementaren Werten (*Autoboxing*)

Zuweisung (`var = Ausdruck`):
Wert: der Ausdruckswert wird in *var* gespeichert.
Referenz: die Referenz des Ausdruckwertes wird in *var* gespeichert

Vergleich in Java (`Ausdruck1 == Ausdruck2`)
Wert: die Werte werden verglichen (**semantische Identität**)
Referenz: die Referenzen werden verglichen (***Pointer-Identität***)

Typumwandlung (cast) (`(Typ) WertAusdruck`)
Wert: der **numerische Wert** wird in einer anderen Form **dargestellt**

Typangabe (cast) (`(Typ) ReferenzAusdruck`)
Referenz: dem Compiler wird ein **genauer Typ** genannt.
Zur Laufzeit wird **geprüft**, ob der angegebene Typ wirklich passt.

Wann sind zwei Referenzen identisch (== oder `equals`)?

Was meinen Sie?

- == beschreibt *Identität* ?
- `equals` beschreibt *Identität* ?
- == beschreibt *Identität* und `equals` beschreibt *Gleichheit* ?
- `equals` gilt nur für Strings?
- `equals` vergleicht immer den *Inhalt*, == die *Adresse*?

Wann sind zwei Referenzen identisch (== oder `equals`)?

Verbreitetes Verständnis:
== beschreibt die Identität zweier Objekte

Was meint Java?

`equals` beschreibt die semantische Identität zweier Objekte

`==` stellt fest, ob es sich um die gleiche Referenz handelt.

Was meinen Sie? (== steht hier für den abstrakten Begriff**)**

Können zwei Objekte identisch sein, wenn sie an verschiedenen Orten gespeichert sind.

```
"hello" == new String("hello")

1 == new Integer(1)

Integer.valueOf(1) == 1

kontoVonHans == kontoVonPaul
```

Wikipedia (deutsch) ist missverständlich!

„In der Informatik ist der Unterschied zwischen identischen Speichern und gleichen Speicherwerten leichter erkennbar: Bezieht sich die Implementierung einer Variablen in Form einer Speicheradresse auf dieselbe Speicherzelle, so ist der Inhalt einer zweiten Referenz auf dieselbe Speicherzelle identisch, in einer anderen Speicherzelle befindet sich nur möglicherweise der gleiche Wert."

Diese Definition gilt bei veränderlichen Objekten! Sie gilt nicht für unveränderliche Objekte (Zahlen, Strings, etc.)

Deutlicher: Zwei Objekte sind identisch, wenn sie zu jeder Zeit in allen beobachtbaren Eigenschaften übereinstimmen. Die Speicheradresse ist keine Eigenschaft des Objekts!

Regeln für Äquivalenzrelationen:

Reflexivität: für alle x gilt x = x

Symmetrie: für alle x und y gilt, wenn x = y, so ist y = x

Transitivität: für alle x, y, z gilt,: wenn x = y und y = z, dann ist x = z

Diese Regeln sind in Java bei der Definition von equals() unbedingt zu beachten.

Sehr, sehr viele equals-Implementierungen sind fehlerhaft!

Wir müssen Gleichheit pro Klasse definieren

```
class Konto: (diese Methode wird als Default geeerbt)
    boolean equals(Object other) { return this == other; }

class Integer:
    boolean equals(Object other) {
        if (this == other) return true;
        if (! other instanceOf Integer) return false;
        return this.value == ((Integer) other).value;
    }
```

Die Klasse `String` vergleicht die Char-Array Variable `value`,
aber nicht die irrelevante Variable `hashCode`.

- Objekte immer mit `equals` vergleichen!
- überlegen, ob eine Klasse ein besonderes `equals` braucht!
- auf Probleme gefasst sein: Vererbung.
- == bei Referenzen in besonderen Fällen benutzen
- `x.equals(null)` macht nur bedingt Sinn (warum?)

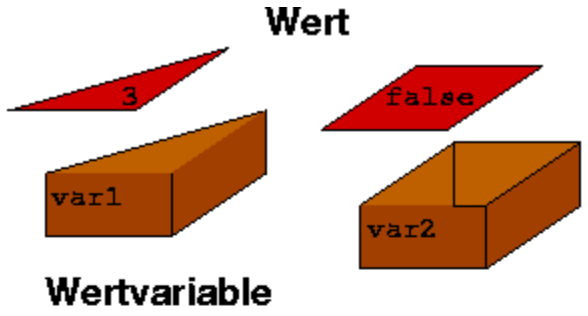
Bildhafter Vergleich von Wert und Objektreferenz

Elementare Typen (**`int`**, **`double`**) werden durch einfache Werte repräsentiert.

Der Typ bestimmt die interne Codierung.

Die Zuweisung ist Kopie des Wertes

Typ der Variablen == Typ des Wertes



Der Typ sagt, was man tun kann.
Der Typ ist festgelegt durch Ausdruck.

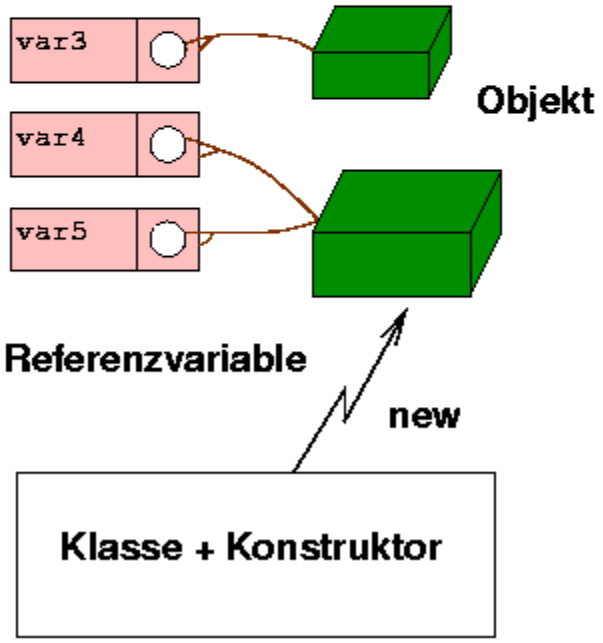
(Die Referenz hat nicht immer einen Namen)

Objekte (und Felder) werden durch **`new`** erzeugt. Ihre Referenz *kann* in einer Variablen gespeichert werden.

Die Zuweisung ist Kopie der Referenz

Der Typ der Variablen ist gleich dem Objekttyp oder ein Obertyp des Objekttyps.

Methoden kennen meist die interne Objektstruktur.



Objekte und Namen

> ***What's in a name? that which we call a rose***
> ***By any other name would smell as sweet.*** (W. Shakespeare, *Romeo & Juliet*)

- Objekte sind reale oder gedachte Dinge, mit denen wir umgehen..
- Objekte haben keinen festen Namen.
- Ein Objekt kann durch verschiedene Typen beschrieben werden.
- Typen können untereinander in Beziehung stehen (Obertyp, Untertyp)
- Die Typangabe ist eine Abstraktion

Ein Objekt kann unterschiedlich benannt werden – und bleibt doch dasselbe:

Hase, Nagetier, Säugetier, Pelztier, Jagdwild

Ein Typname ist eine Vorhersage und ein Versprechen – das von dem Objekt eingelöst werden muss!

Welchen Typ hat ein Objekt?

- Den Typ der Klasse
- Die Obertypen der Klasse:
    - Die Oberklasse und deren Obertypen
    - Ein implementiertes Interface und dessen Obertypen

```
class Klasse extends Oberklasse implements Interface1, Interface2
interface Interface1 extends InterfaceA, InterfaceB
```

Wo spielt der Typ eine Rolle?

- Bei der *Deklaration* von Klassen, Interfaces, Variablen und Methoden
- Bei der *Typanpassung* (Cast)
- Bei der *Prüfung* ob eine *Zuweisung* erlaubt ist
- Bei der *Prüfung* ob ein *Methodenaufruf* erlaubt ist
- Bei der *Auflösung* von (statisch) überladenen Namen

Wo spielt der Typ keine Rolle?

- Bei der objektorientierten Entscheidung welche Methode aufgerufen wird
- Bei der Festlegung, was die Methode tut

Objekt, Typ, Klasse → Interface

```
Typ variable = new Klasse();
```

- **Klassen** beschreiben eine Menge von Objekten, die die gleiche *Struktur* haben.
- **Typen** beschreiben eine Menge von Objekten, die das gleiche *Verhalten* zeigen.
- Eine **Java-Klasse** beschreibt einen Typ **und** die Struktur der Objekte.
- **Java-Interfaces** beschreiben **nur** einen Typ.
- **Variable** werden mit einem **Typ** deklariert.
- Der Compiler prüft die **Verträglichkeit der Typen**.

Anmerkung:

1. Interfaces braucht man nur bei statischer Typprüfung.
2. Sprachen mit Mehrfachvererbung (C++) verwenden anstelle von Interfaces abstrakte Klassen.

Java-Interface = abstrakte Beschreibung eines Typs

Syntax:

[**public**] **interface** <*name*> [**extends** <*Liste von Interfaces*>] **{**
  <*Deklarationen von static final Konstanten*> // praktisches Zugeständnis
  <Deklarationen von Methodenköpfen>
}

Beispiel:

```
public interface Iterator {
    public abstract boolean hasNext();
    public Object next();
    void remove();
}
```

Jede Methode ist automatisch **public** und **abstract.**

Mit Typparameter:

```
public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}
```

Ein paar Interfaces der Javabibliothek

• java.lang.Comparable	natürliche Ordnung
• java.util.Comparator	Ordnungsbeziehung für den momentanen Gebrauch
• java.util.Iterator	Iteration über Ansammlung von Daten
• java.util.Iterable	Verfügt über Iteratoren (für for-each-Schleife)
• java.util.EventListener	Schnittstelle zu GUI-Aktionen
• java.lang.Runnable	deklariert parameterlose Methode
• java.util.Map	assoziativer Speicher
• java.util.List	sequenzielle Datensammlung
• java.util.Set	Menge von Objekten
• java.io.DataInput	binäre Eingabe
• java.io.Serializable	Objekt kann seriell weitergegeben werden (z.B. in einer Datei gespeichert werden)

Man hat ständig mit Interfaces zu tun!

Anwendung von Interfaces (ohne Typparameter)

```
SortedMap m = new TreeMap(); // oder Map m = ...
m.put("Karin", "Düsseldorf");
m.put("Hans", "Köln");
m.put("Lia", "Bonn");
m.put("Karin", "Düsseldorf");
ausgabeAllerDaten(m);
```

```
public void ausgabeAllerDaten(Map map) {
   for (Iterator iter = map.entrySet().iterator(); iter.hasNext();) {
      Map.Entry e = (Map.Entry) iter.next();
      System.out.println(e.getKey() + " wohnt in " + e.getValue());
  }
}
```

Variablen sind durch Interfaces deklariert: Man kann Operationen ausführen.
Klassen braucht man für das Erzeugen von Objekten.
Beachten Sie die Verwendung der **Iterator-Operationen**.

Die fehlenden Typparameter machen Casts nötig!

for-each basiert auf der Anwendung von Interfaces

```
SortedMap<String, String> m = new TreeMap<String, String>();
m.put("Karin", "Düsseldorf");
m.put("Hans", "Köln");
m.put("Lia", "Bonn");
m.put("Karin", "Düsseldorf");
ausgabeVonAllenDaten(m);

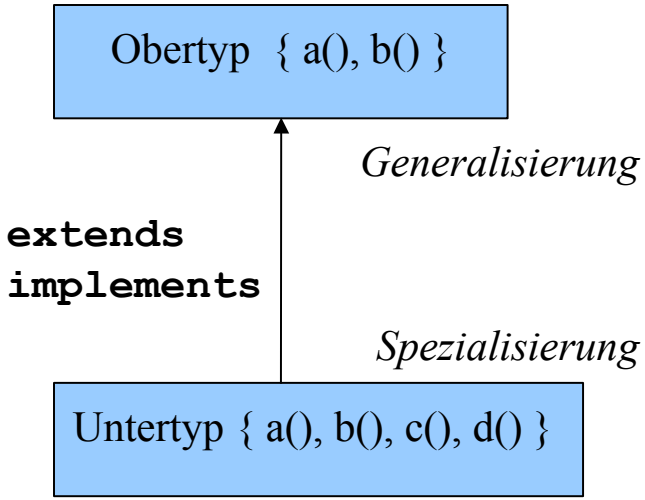
public void ausgabeVonAllenDaten(Map<String, String> map) {
    for (Map.Entry<String, String> e : map.entrySet()) {
        System.out.println(e.getKey() + " wohnt in " + e.getValue());
    }
}
```

Das for-each-Konstrukt beruht auf den Schnittstellen **`Iterator<T>`** und **`Iterable<T>`**.
Der Compiler erzeugt aus dem for-each exakt den Code der vorangehenden Folie.

Darüber hinaus werden durch Typparameter weitere Typinformationen festgelegt.

Typbeziehungen (bildlich)

Der Typ sagt aus, welche Operationen *mindestens* definiert sind.



Variable$_{Obertyp}$ = *Ausdruck*$_{Untertyp}$ ist immer erlaubt. (up cast)

Variable$_{Untertyp}$ = *(Untertyp) Ausdruck*$_{Obertyp}$ *kann* richtig sein (down cast)

Typregeln von Java

Das Grundprinzip der Typregeln ist: ***man darf zuviel verbieten aber nicht zuviel erlauben!***

Das Java-Typsystem verlangt nicht, dass in einer Zuweisung der Typ der linken Seite (Variable) gleich dem Typ der rechten Seite ist (Ausdruck). Statt dessen gilt**:**

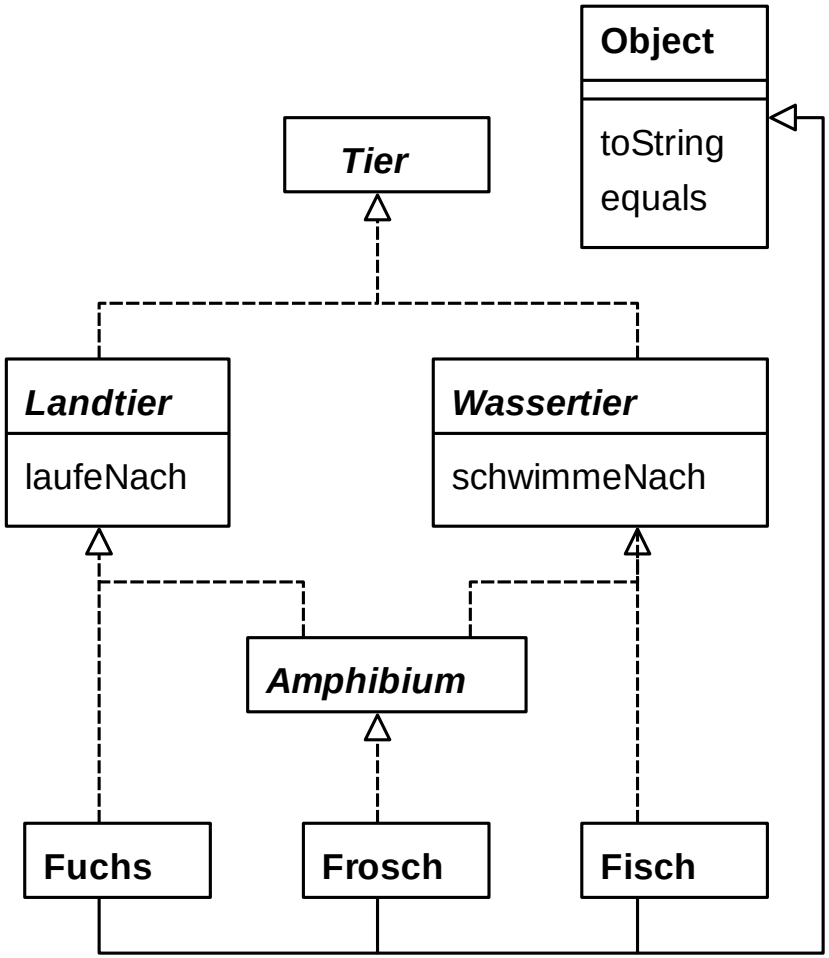
- Ein Typ T1 ist ein **Obertyp** des Typs T2, wenn T2 alle Operationen von T1 enhält. In Java muss diese Beziehung im Klassen-/Interface-Kopf angegeben sein (**`extends`**, **`implements`**).
- Die **Zuweisung** *Var* = *Ausdruck* ist erlaubt, wenn der Typ von *Var* ein Obertyp des Typs von *Ausdruck* ist.
- Der **statische Typ** eines Ausdrucks wird vom Compiler anhand von Deklarationen und Typregeln ermittelt. Er ist immer ein Obertyp des **dynamischen Objekttyps**.
- Mit einer **Typangabe** der Form `(T1)(Ausdruck)` kann der statische Typ des Ausdrucks als Typ T1 angegeben werden. Zur Laufzeit wird geprüft, ob der dynamische Typ ein Untertyp von T1 ist. Das Objekt wird nicht verändert!

Ein Obertyp kennt weniger Operationen als ein Untertyp.

Die Typprüfung stellt sicher, dass niemals eine nicht existierende Methode aufgerufen wird!

Die Auswahl der Methode erfolgt aber erst zur Laufzeit (Objektorientierung, späte Bind.)

Klassendiagramm für eine Typhierarchie (UML)



Interface: ***Tier***
Klasse: **Object**
Methode: toString

Oberklasse: **durchgezogene Linie**
Obertyp: **gestrichelte Linie**

Klassen:
Object, Fuchs, Frosch, Fisch

Interfaces:
Tier, Landtier, Wassertier, Amphibium

Anmerkung: Object hat noch mehr Methoden

Java:
Hierarchie für Klassen (Einfachvererbung)
Hierarchie für Interface (Mehrfachvererbung)
Grund: Mehrfachvererbung für Klassen nicht ganz unproblematisch

Beispiel für Typerweiterung (Deklarationen)

```
interface Tier { }
   => Object, Tier

interface Landtier extends Tier { void laufeNach(Ort x); }
   => Object, Tier, Landtier

interface Wassertier extends Tier { void schwimmeNach(Ort x); }
   => Object, Tier, Wassertier

interface Amphibium extends Landtier, Wassertier { }
   => Object, Tier, Landtier, Wassertier, Amphibium

class Frosch implements Amphibium { . . . }
   => Object, Tier Landtier, Wassertier, Amphibium, Frosch
class Fuchs implements Landtier { . . . }
   => Object, Tier, Landtier, Fuchs
class Fisch implements Wassertier { . . . }
   => Object, Tier, Wassertier, Fisch
```

Frage: Was kann in einer Variablen vom Typ `Landtier` referiert sein?

Beispiel für Typerweiterung (Anwendung)

```
Wassertier[] teichtiere = new Wassertier[3];
    –  // Array = Array-Objekt + Referenztyp !!
teichtiere[0] = new Fisch();
teichtiere[1] = new Frosch();
teichtiere[2] = new Fuchs(); // Fehler!

for (Wassertier tier : teichtiere) {
    tier.schwimmeNach(futterQuelle);
    tier.laufeNach(garten); ???
}
```

Wie ist es mit:

```
    teichtiere[1].laufeNach(garten); ???
```

Richtig:

```
    ((Landtier) teichtiere[1]).laufeNach(garten);
    ((Frosch) teichtiere[1]).laufeNach(garten);
    ((Amphibium) teichtiere[1]).laufeNach(garten);
```

Und was ist mit:

```
    ((Landtier) teichtiere[0]).laufeNach(garten); ???
```

Beispiel für Typerweiterung (Anmerkung)

In der Regel sollte man so allgemein programmieren, dass alle Anwendungsfälle bearbeitet sind, aber keine Ausnahmen vorkommen.

```
for (WasserTier tier : teichtiere) {
    tier[i].schwimmeNach(futterQuelle);
}
```

Manchmal kommt man um Ausnahmen nicht herum. Dann hilft **`instanceof`**:

```
for (Wassertier tier : teichtiere) {
    if (tier instanceof Amphibium) // eine Ausnahme !!
        ((Amphibium) tier).laufeNach(garten);
    else
        tier.schwimmeNach(futterQuelle);
}
```

Frage: wie hätte man durch einen besseren Klassenentwurf die instanceof-Abfrage vermeiden können?

Anwendungsbeispiel: Bruch

```
class Bruch implements Comparable<Bruch> {
    public Bruch(int z, int n) { ... }

    /**
     * @return Summe von this und summand.
     */
    public Bruch addiere(Bruch summand) { ... }

    /**
     * @return >0, wenn this > bruch
     *         ==0, wenn this == bruch
     *          <0, wenn this < bruch
     */
    public int compareTo(Bruch andererBruch) { ... }

    public boolean equals(Object anderesObjekt) { ... }
    ...
}
```

Anmerkung:
Man kann (halbwegs) sinnvoll fragen: „Ist dieses Haus gleich diesem Bruch?“
Es macht (absolut) keinen Sinn zu fragen: „Ist dieses Haus größer als dieser Bruch?“

Welchen Typ haben Ausdrücke? -- Was ist erlaubt?

```
Bruch v = new Bruch(2,1);
new int[10]            ??
Object x = v;
x                      ??
(Comparable)x          ??
v = x;
v.compareTo(x)         ??
x.equals(v)            ??
Bruch[] brueche = new Bruch[3];
Arrays.max(brueche)    ??

Arrays.max(brueche).addiere(new Bruch(1,2));
Bruch b = Arrays.max(brueche);
```

Mit:
```
public class Arrays {
        public static Object max(Object[] array) { ... }
```

Welchen Typ haben Ausdrücke? -- Was ist erlaubt?

	statisch(Compiler)	dynamisch(Laufzeit)
`Bruch v = new Bruch(2,1);`		
`v`	`Bruch`	`Bruch`
`new int[10];`	`int[]`	`int[]`
`Object x = v;`		
`x`	**`Object`**	**`Bruch`**
`(Comparable)x`	**`Comparable`**	**`Bruch`**
`v = (Bruch)x;`		
`v.compareTo(x)`	`int`	`int`
`x.equals(v)`	`boolean`	`boolean`
`Bruch[] brueche = new Bruch[3];`		

Wenn wir diese Methode haben **`static Object max(Object[]):`**

`max(brueche)`	**`Object`**	**`Bruch`**

```
Bruch b = (Bruch) max(brueche);
Bruch c = ((Bruch) max(brueche)).add(new Bruch(1,2));
```

```
// Fehler (das mag der Compiler nicht):
v = x;
Bruch c = max(brueche).add(new Bruch(1,2);
```

Sortieren von Strings mit deutschen Umlauten

```
import java.util.Arrays;      // Klasse mit Arrayfunktionen
import java.util.Comparator;  // Interface
import java.util.Locale;      // länderspezifische Eigenschaften
import java.text.Collator;    // spezielle Vergleiche

Collator cmp = Collator.getInstance(Locale.GERMANY);
// man hätte auch schreiben können: Comparator<Object> cmp = ...
// die Collator-Schreibweise lässt noch mehr Aufrufe zu.

String[] strs = {"üblich" , "Unsinn" , ..., "Ärger", "auch"};
Arrays.sort(strs, cmp);
```

Hinweise:

```
public <T> void Arrays.sort(T[] a, Comparator<? super T> c); //***

public interface Comparator<T> {
    public int compare(T a, T b);
}
```

*//*** stören Sie sich nicht an der Syntax für die Typparameter …*
bedeutet: der Comparator c muss für den Elementtyp des Array a
oder für einen Obertyp davon definiert sein.

Zusammenfassung: Deklaration eines Interface

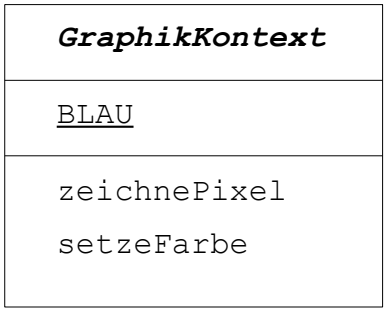
```
public interface GraphikKontext extends Serializable {
    // Konstante
    public static final int BLAU=0x0000FF;

    // Deklaration von Operationen

    public void zeichnePixel(int x, int y);

    public void setzeFarbe(int farbe);
}
```

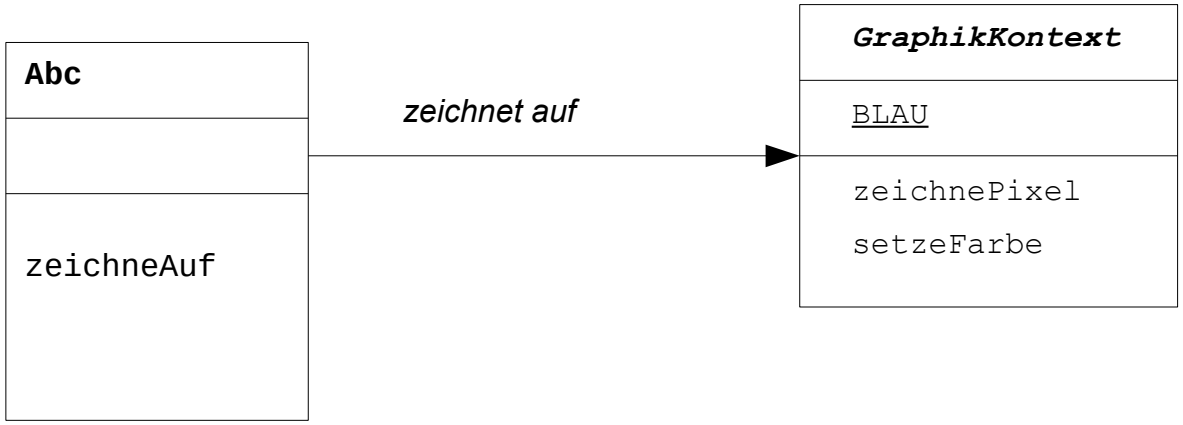
- Ein Interface *kann* andere Interfaces erweitern
- Ein Interface *kann* Konstanten deklarieren
- Ein Interface *kann* Operationen deklarieren



Zusammenfassung: Nutzung eines Interface

```
class Abc {
    public void zeichneAuf(GraphikKontext g) {
        g.setzeFarbe(GraphikKontext.BLAU);
        g.zeichnePixel(17, 18);
    }
}
```

- Der Interfacetyp erlaubt, Operationen aufzurufen ohne die Objektklasse zu kennen
- Methoden „funktionieren“ für unterschiedliche Objekttypen (Polymorphie)



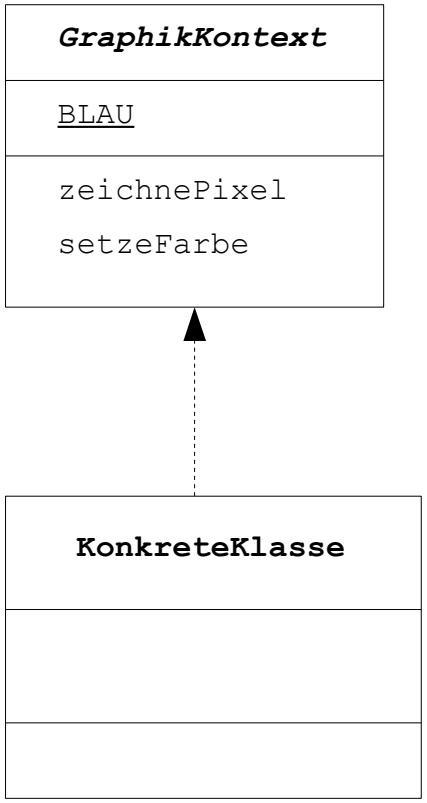
Zusammenfassung: Konkrete Klassen implementieren Schnittstellen

```
class KonkreteKlasse implements
    Comparable<KonkreteKlasse>, GraphikKontext, Serializable {

    public int compareTo(KonkreteKlasse k) { … }
    public void zeichnePixel(int x, int y) { … }

    public void andereMethoden() { … }
}
```

- Alle Operationen müssen implementiert werden
- `Serializable` kennzeichnet Objekte, deklariert keine Methode
- Mehrfach definierte Operationen werden nur einmal implementiert
- Objekte können in Variablen vom Typ einer Schnittstelle referiert werden



Zusammenfassung: Attribute von Methoden, Feldern und Klassen

Sichtbarkeit von Inhalten einer Klasse:

- **`public`** überall
- **`private`** nur in der Klasse
- **`protected`** in Unterklassen und im Paket
- *nichts* im Paket

Unveränderbarkeit von Variablen:

- **`final`** *Variable* unveränderliches Attribut
- **`static final`** *Variable* globale Konstante

Kein Überschreiben von Methoden:

- **`final`** *Methode* kann nicht überschrieben werden
- **`final`** *Klasse* es kann keine weitere Unterklasse definiert werden

Bezug zu Klasse / Instanz:

- **`static`** gehört zur Klasse (kennt kein this-Objekt)
- *nichts* gehört zur Instanz (Objekt)

Vererbung und Typbeziehung:

- **`implements`** *Interfaces* implementiert die angegebenen Schnittstellen
- **`extends`** *Interfaces* erweitert die angegebenen Schnittstellen
- **`extends`** *Klasse* erweitert die angegebene Klasse