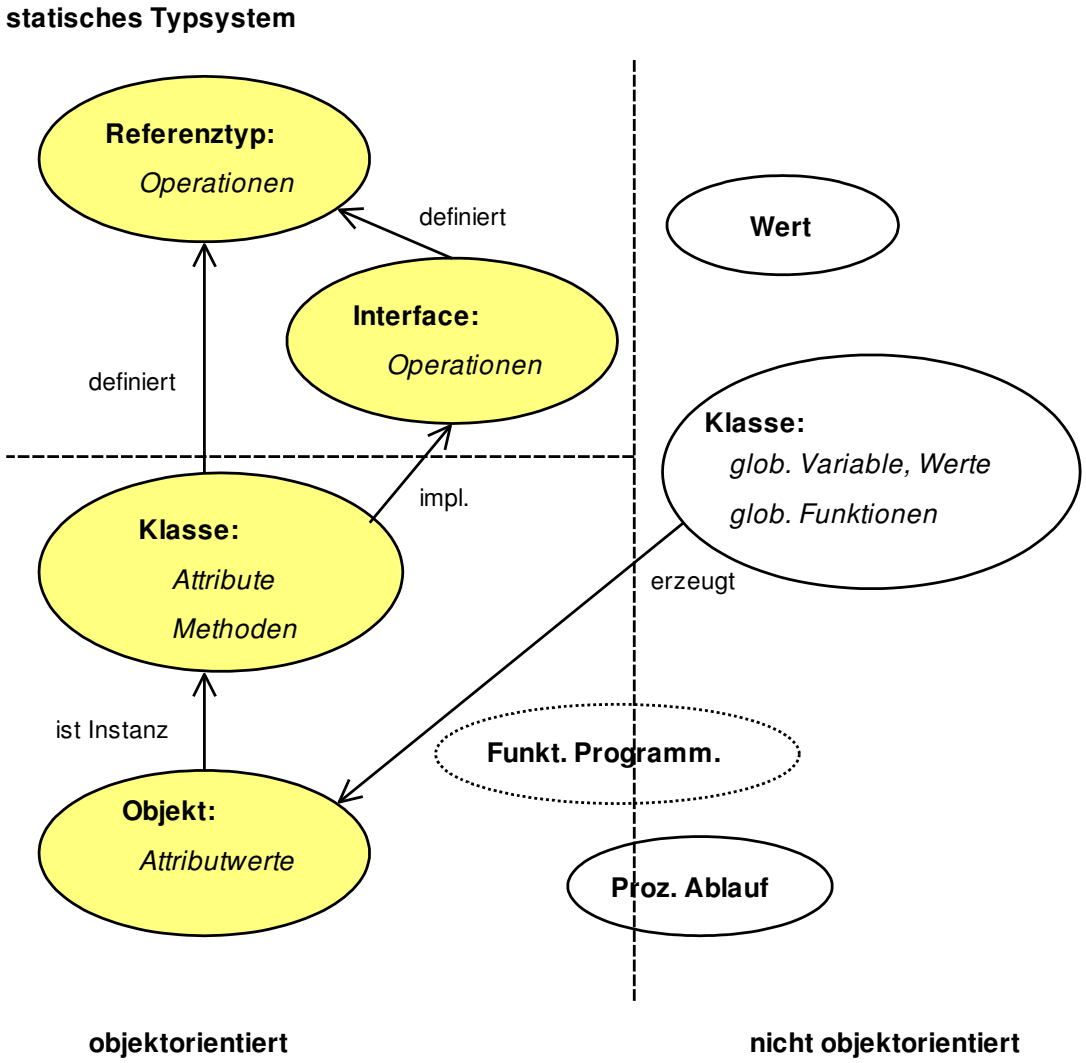
Objektorientierung und Vererbung

- Aufgaben von Klassen
- Die Klasse `Object` und das Grundverhalten von Objekten
- Wertobjekte
- Vererbung
- Abstrakte Klassen
- Beispielanwendung
- Polymorphie
- Innere Klassen / Anonyme Klassen

Fokus: Objektorientierung und Vererbung



Die Aufgabe einer Klasse

- **Jede Klasse hat eine bestimmte Aufgabe**

- **Es gibt verschiedene Anwendungsszenarien:**
  - eine Klasse ist Teil einer bestimmten **Anwendung**
  - eine Klasse soll allgemein zur Verfügung stehen (**Bibliotheksklasse**)
  - Objekte speichern **unveränderliche / veränderliche Information**
  - ein Objekt implementiert ein bestimmtes **Verhalten (***Funktionsobjekte***)**

Das Grundverhalten von allen Objekten (definiert in der Klasse `Object`)

- **`String toString()`** legt fest, wie ein Objekt ausgegeben wird.
- **`boolean equals()`** definiert die Identität zweier Objekte. Es ist immer und für alle Objekte definiert. Jedes Objekt ist ungleich zu **`null`**.
- **`int hashCode()`** liefert eine Objektkennung. Wenn zwei Objekte (gem. `equals`) gleich sind, muss auch die Kennung gleich sein.

Es gibt eine Reihe von Bibliotheksklassen, die diese Methoden aufrufen. Sie funktionieren nur dann richtig, wenn eigene Klassen `equals()` und `hashCode()` **richtig überschreiben**.

(*Randbemerkung es gibt in Java wohl keine 100% eindeutige Definition für equals*!)

- **`myObject instanceof Klasse`** stellt fest, ob `myObject` mit dem Typ von `Klasse` verträglich ist.
- **`(Klasse) myObject`** sagt, dass `myObject` als vom Typ `Klasse` betrachtet werden soll (wird vom Compiler "geglaubt" und zur Laufzeit geprüft).
- **`Class getClass()`** gibt eine Referenz auf das Klassenobjekt des Objekts zurück. (das Klassenobjekt gibt Auskunft über die Eigenschaften der Klasse)

Klassen für unveränderliche (Wert-)Objekte

- **Keine Methode ändert den logischen Zustand.**
- Die Klasse wird **`final`** deklariert, damit keine Ableitung möglich ist.
- Manchmal implementiert die Klasse die Schnittstelle `Serializable`, damit Objekte in Datei/Netz übertragen werden können.
- Die Klasse überschreibt die Methoden `equals` und `hashCode`.
- Bei numerischen und sonstigen geordneten Datentypen wird die Schnittstelle `Comparable` implementiert und die Methode `compareTo` überschrieben. `equals` und `compareTo` dürfen sich nicht widersprechen!
- Meist wird die Methode `toString` überschrieben.

Unveränderliche Objekte erhöhen die Lesbarkeit und die mögliche Effizienz eines Programms (funktionale Programmierung, Nebenläufigkeit).

Ableitung einer Klasse

Syntax: `class` *Name* `extends` *Oberklasse* `{ ... }`

- **Die abgeleitete Klasse definiert einen Untertyp der Oberklasse.**
- Ein Objekt *enthält* alle Instanzvariablen der eigenen Klasse und der Oberklasse.
- Ein Objekt *kennt* alle Methoden der eigenen Klasse und der Oberklasse.
- **Klassenelemente** und **Variablen** gleichen Namens **verdecken** (*overwrite*) diejenigen der Oberklasse.
- **Methoden** gleichen Namens und gleicher Signatur **überschreiben** (*override*) diejenigen der Oberklasse (to override = überstimmen).
- Durch Qualifikation des Zugriffs mit `super`, ist der ausdrückliche Zugriff auf verdeckte und überschriebene Elemente möglich.
- **Der Konstruktor einer Oberklasse wird mit `super(...)` aufgerufen. Dieser Aufruf muss die erste Anweisung im einem Konstruktor sein.**

V*erdecken* wirkt sich auf Entscheidungen des Compilers aus,
Ü*berschreiben* wirkt sich nur auf den Ablauf zur Laufzeit aus.

*Die Vererbung von Klassen vererbt die Schnittstelle **und** die Implementierung einer Klasse.*

Ableitung von einer Oberklasse

```
abstract class SuperClass<T> {                    abstrakt darf fehlen!
    private T x;                                  nicht sichtbar in SubClass
    SuperClass(T x) {
        this.x = x;
    }

    T getX() {
        return x;
    }
}

class SubClass extends SuperClass<String>         Typparameter festgelegt
{   private int alter;
    StringClass(String name, int alter) {
        super(name);                              ruft den Konstruktor
        this.alter = alter;                         von SuperClass auf!
    }                                               (notwendig, wenn Parameter!)

    int getAlter() {
        return alter;
    }
}
```

Bei this und super ist folgendes zu beachten:

`this` referiert das Objekt, an das die Nachricht gesendet wurde (Empfängerobjekt).

`super` steht auch für das Empfängerobjekt. Bei der **Suche** nach Variablen und Methoden wird aber **in der Oberklasse der Klasse der aufrufenden Methode** begonnen.

Beispiel:

```
class C {
    void f() {...}
}
class B extends C {
    void f() { super.f(); ...}
}
class A extends B { ... }
```

Irgendwo:

```
C a = new A();
a.f();  // nicht C.f sondern B.f (späte Bindung)
        // B.f ruft durch super.f C.f auf.
```

Es gibt verschiedene Arten von Typbeziehung und Vererbung

1. **Interfaces** definieren in Java gemeinsames Verhalten verschiedener Objekte, ohne dabei die Implementierung festzulegen. Ein Interface definiert einen allgemeinen **Obertyp** für Objekten verschiedener Klassen. Interfaces können andere Interfaces spezialisieren (erweitern); Klassen können Interfaces implementieren. (Manchmal nennt man dies irreführend „Vererbung“ der Schnittstelle.)
2. **Klassen** definieren den **genauen Typ** und die **vollständige Implementierung** einer Menge von Objekten. Bei der Vererbung wird alles übernommen, was nicht ausdrücklich überschrieben wird. Die Hauptaufgabe einer Klasse ist die Erzeugung von Objekten.
3. **Abstrakte Klassen** definieren einen **Typ** und eine **Teilimplementierung** für mehrere Klassen. Aus abstrakten Klassen kann man keine Objekte erzeugen. „implementierte“ Schnittstellen müssen nicht in der abstrakten Klasse selbst implementiert werden. Es können abstrakte Methoden definiert werden.
4. **Anonyme Klassen** sind namenlos. Ihr Typ ist gleich dem Typ ihrer Oberklasse oder gleich dem angegebenen Interface. Eine anonyme Klasse **überschreibt Methoden**.

Interface, abstrakte Klasse, konkrete Klasse, Vererbung

```
interface Schnittstelle {
    int methode1();              auch eine abstrakte Methode
    int methode2();              == public abstract int methode2();
}

abstract class AbstrakteKlasse implements Schnittstelle {
    protected int variable;      sichtbar in Unterklassen

    protected AbstrakteKlasse(int variable)
    { this.variable = variable; }

    public int methode1() { abstrakteMethode(); return variable; }

    abstract void abstrakteMethode();
}

class KonkreteKlasse extends AbstrakteKlasse {
    KonktreteKlasse(int variable) {
        super(variable);         ruft Konstruktor der Oberklasse
    }

    public int methode2() { ... }      bisher nicht implementiert
    void abstrakteMethode() {  bisher nicht implementiert
         ... }
}
```

Klassen und Schnittstellen

	Interface	Abstrakte Klasse	Konkrete Klasse	Anonyme Klasse
Typ	**ja**	ja	ja	nein
Objekt erzeugen	nein	nein	**ja**	ja
definiert Methoden Variablen	nein	**ja**	**ja**	**ja**
Abstrakte Methoden	**ja**	**ja**	nein	nein

Abstrakte Klasse im Vergleich zu Interface

- abstrakte Methoden erkennt man an dem Schlüsselwort **`abstract`**. Sie haben keinen Methodenkörper. Sie dürfen nur in einem Interface oder einer abstrakten Klasse stehen.
- In einem Interface sind alle Methoden abstrakt.
- Ein Interface kann mehrere Interfaces *erweitern (`extends`)*; eine Klasse kann mehrere Interfaces *implementieren (`implements`)*.
- Eine Klasse kann nur eine einzige (abstrakte) Klasse erweitern (**`extends`**) (Einfachvererbung).
- Interface und (abstrakte) Klassen definieren einen Typ.
- Abstrakte Klassen definieren eine Teilimplementierung.

Die Bedeutung abstrakter Klassen

Beispiel:

Die Schnittstelle `java.util.Collection` definiert mit 13 Methoden die Schnittstellen für beliebige Datenbehälter. Die davon abgeleitete Schnittstelle `java.util.List` dient der Beschreibung sequentiell (in einer Reihenfolge) angeordneter Daten und definiert weitere 10 Methoden. Dazu kommt die Forderung, die Methoden `hashCode`, `equals` und `toString` zu überschreiben. Insgesamt sind also mindestens 26 Methoden zu implementieren!

Diese Aufgabe wird erheblich vereinfacht durch die abstrakten Klassen `AbstractCollection` (14 Methoden) und `AbstractList` (15 Methoden).

Beispiel: Unveränderbares Array

Aufgabe: *Unveränderliche Sicht auf Feldelemente.*

```
public class ImmutableArray<T> extends AbstractList<T> {
    private T[] data;

    public ImmutableArray(T[] array) {
        if (data == null) throw new NullPointerException();
        data = array;

    public T get(int index) {
        return data[index];
    }

    public int size() {
        return data.length;
    }
}
```

Anwendungsbeispiel:

```
String[] chars = {"alfa", "beta", "gamma", "delta", "epsilon"};
List<String> charList = new ImmutableArray<String>(chars);
System.out.println(charList);
```

Beispiel: Frameworks (hier Android)

```
public class MainActivity extends Activity {
    public final static String EXTRA_TAG = "com.exa.app.TAG";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void sendMessage(View view) {
        Intent intent = new Intent(this,
                                   DisplayMessageActivity.class);
        EditText edit = (EditText) findViewById(R.id.edit_message);
        intent.putExtra(EXTRA_TAG, edit.getText().toString());
        startActivity(intent);
    }
}
```

Ein Framework gibt durch Abstrakte Klassen einen Rahmen für eigene Klassen vor:

- Überschriebene Methoden werden von der Umgebung aufgerufen (`onCreate`).
- Alternativ werden Defaultimplementierungen genutzt (`onResume, ...`).
- Weitere Methoden werden bereitgestellt (`setContentView, ...`).

Vorteile der Konkretisierung von Abstrakten Klassen

- Mit wenig Aufwand können besondere Anforderungen realisiert werden.
- Die neue Klasse „erbt“ alle Algorithmen der gesamten Klassenfamilie!
- Die neuen Klassen fügen sich nahtlos in vorhandene Frameworks ein.

Anmerkung:

In der Java Bibliothek gibt es Methoden mit denen wir ein unveränderliches Feld sogar noch einfacher erzeugen können:

In der Klasse `java.util.Arrays`:

```
public static List asList(Object[] array)
```

In der Klasse `java.util.Collections`:

```
public static <T> List<T> unmodifiableList(List<T> list)
```

In unserem Beispiel:

```
List<String> unmodifiableArray =
        Collections.unmodifiableList(Arrays.asList(chars));
```

Interface, Abstrakte Klasse, Konkrete Klasse (1)

- Ein Interface definiert einen Typ, indem es eine Mengen von Operationen (durch abstrakte Methoden) deklariert.
- Ein Interface *kann* auch Konstanten definieren
- Mittels `instanceof` kann man abfragen, ob ein Objekt ein Interface implementiert. Interfaces können daher auch zur Markierung von Objekten genutzt werden.
- Interfaces können vorhandene Interfaces erweitern.
- Interfaces erlauben mehrfache Typerweiterung („Vererbung“).
- Die Implementierung durch eine Klasse garantiert, dass die Elemente der Klasse der Schnittstelle genügen, indem erzwungen wird, dass die zur Schnittstelle gehörenden Methoden implementiert werden.

Beispiel Java-Collection-Framework: Schnittstellen in Kombination mit abstrakten Klassen und konkreten Klassen.

Interface, Abstrakte Klasse, Klasse (2)

- **Von einer als abstrakt gekennzeichneten Klasse lassen sich keine Objekte erzeugen.** Dafür braucht diese Klasse nicht alle implementierten Methoden zu definieren und darf auch abstrakte Methoden enthalten.
- Eine Abstrakte Klasse definiert einen Typ und eine (teilweise) Implementierung.
- Eine Abstrakte Klasse dient dazu, mehreren konkreten Klassen eine gemeinsame Teilimplementierung vorzugeben (*Klassenskelett*).
- Eine Abstrakte Klasse kann nur einfach vererbt werden.
- Eine Abstrakte Klasse *kann* abstrakte Methoden enthalten.

Interface, Abstrakte Klasse, Konkrete Klasse (3)

- Eine konkrete Klasse definiert einen Typ und eine vollständige Implementierung.
- Eine konkrete Klassen dient dazu, Objekte zu erzeugen.
- Man *kann* mit einer konkreten Klasse Variablen und Schnittstellen deklarieren.
- Wenn die konkrete Klasse nicht `final` ist, *kann* man durch Vererbung eine Unterklasse ableiten.

Polymorphie

New Oxford Dictionary:

polymorphism *noun*
the occurrence of something in several different forms, in particular: …
Computing: a feature of a programming language that allows routines to use variables of different types at different times.

poly – viel
morphe – Form (vgl. Morphologie)

Objektorientierung unterstützt Polymorphie dadurch, dass eine Variable (zu verschiedenen Zeiten) Referenzen auf Objekte unterschiedlicher Klassen speichern kann.

Die dynamische Polymorphie wird in der Objektorientierung implementiert durch späte Bindung.

Bei Typparametern spricht man oft von statischer Polymorphie.

Statische Polymorphie

```
class Stack<T>{
    Object[] d = new Object[100];
    void push(T x) {
        d[top++] = x;
    }
    T pop() {
        return (T) d[--top];
    }
}

Stack<Integer> intStack = new Stack<Integer>();
Stack<String> stringStack = new Stack<String>();
```

- **Die statische Polymorphie der Typparameter unterstützt Wiederverwendung.**
- **Die dynamische Polymorphie der Objektorientierung erlaubt flexible Programmierung .**

Polymorphie (1) -- *Typstruktur*

```
interface Typ {
    void tueWas(int n);
}

class Klasse1 implements Typ {
   void tueWas(int n) { … }
   ...
}

class Klasse2 implements Typ { // und/oder extends Klasse1
    viod tueWas(int n) { … }
    …
}
```

Klasse1 und **Klasse2** sind Untertypen von **Typ**. Alles andere ist hier egal.

Polymorphie (2) -- *polymorphe Variablen / Methoden*

„man kann eine Methode mit Argumenten unterschiedlichen Typs aufrufen“
„eine Operation wird je nach Objektklasse durch eine andere Methode ausgeführt“

```
public class Test3 {
    void teste(Typ t) {
        t.tueWas(2);      // tueWas aus Klasse1 oder aus Klasse2
        t.tueWas(1);
        t.tueWas(2);
    }

    public static void main(String[] argv) {
        Test3 tst = new Test3();
        tst.teste(new Klasse1());
        tst.teste(new Klasse2());
    }
}
```

`t` kann Referenzen zu Objekten speichern, deren Klasse den Typ `Typ` implementiert. Es wird jeweils die passende Methode `tueWas` aufgerufen.

Es ist dabei egal ob `Typ` durch ein Interface oder durch eine Klasse definiert ist.

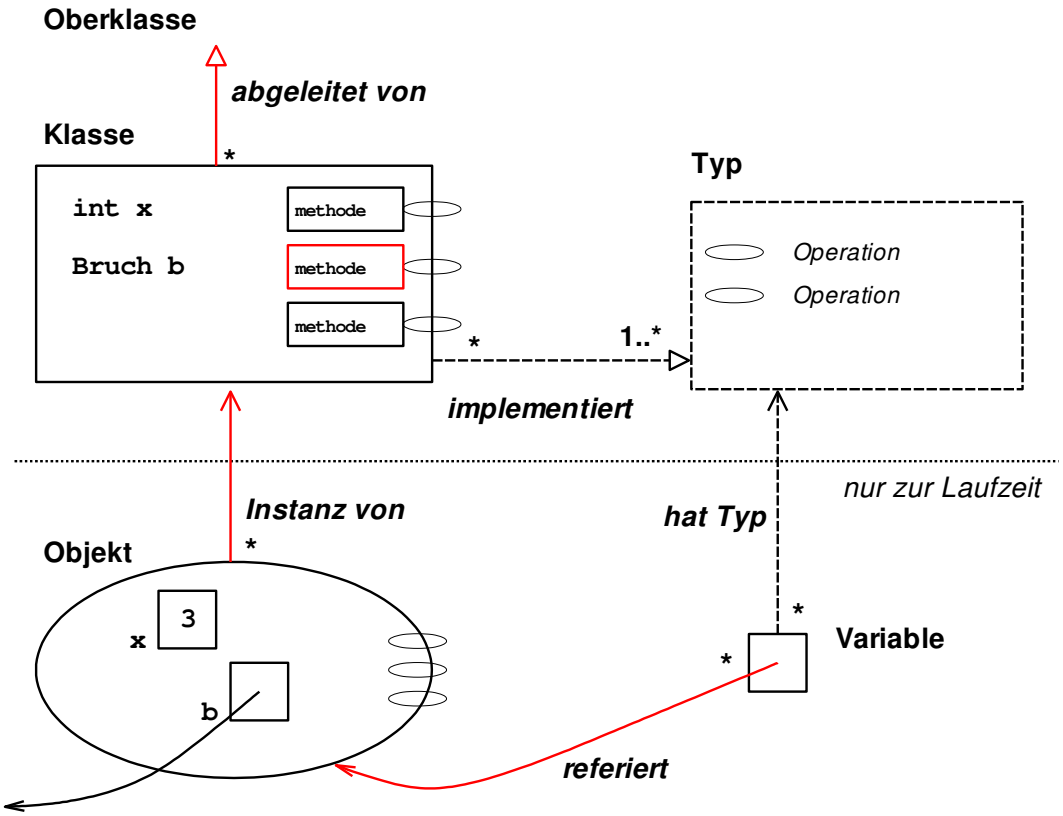
Polymorphie (3)
polymorphe Datenstrukturen

„eine Datenstruktur kann Objekte unterschiedlichen Typs enthalten“
„eine Operation auf einem Element einer Datenstruktur ruft die passende Methode auf“

```
Typ[] t = new Typ[10];
t[0] = new Klasse1();
t[1] = new Klasse2();
...
for (Typ x : t)
    x.tueWas(2);

Object[] obj = new Object[100];    // Object ist der allgemeinste Typ!
obj[0] = new Klasse1();
obj[1] = new Klasse2();
obj[2] = "Hallo";
...
obj[0].tueWas(2);                  //??? Compiler-Fehler !
((Typ) obj[0]).tueWas(1);
((Klasse1)obj[0]).tueWas(2);
((Klasse2)obj[0]).tueWas(2);       ... ClassCastException
```

Klasse, Typ, Variable, Objekt und späte Bindung



Methodenbindung: ***verknüpft den den Aufruf mit der passenden Methode***

`variable.methode()` → `Klasse.methode() { ...}`

Frühe Bindung: Methodenbindung durch den Compiler
Späte Bindung: Methodenbindung beim Aufruf (bei der Programmausführung)

Vererbung und späte Bindung (1)

Übersetzungszeit	*Laufzeit*
(Laden der Classfiles)	
früh	*spät*
`StatischerTyp var`	`= new KonstruktorEinerKlasse();`
`var.methode();`	Welches Objekt ist in `var` ?
	Zu welcher Klasse gehört dieses Objekt?
	Welche Methode wird aufgerufen?
	Bindung
`var.privateMethode();`	frühe Bindung
`super.methode();`	frühe Bindung
`var.methode();`	**späte Bindung erforderlich**

Objektorientierung resultiert aus der späten Bindung durch das Objekt

Vererbung und späte Bindung (2)

```
class Oben {
    int var = 7;
    void methode(int x) { ... }
}
class Unten extends Oben {
    int var = 8;                    // verdeckt Oben.var
    void methode(int x) { ... }     // überschreibt Oben.methode(int)
    void methode(String x) { ... }// neue Methode
}
Unten unten = new Unten(); Oben oben = unten;
oben.var                            ??
unten.var                           ??
oben.methode(3);                    ??
unten.methode(3);                   ??
oben.methode("hallo");              ??
```

Vererbung und späte Bindung (2)

```
class Oben {
    int var = 7;
    void methode(int x) { ... }
}
class Unten extends Oben {
    int var = 8;                        // verdeckt Oben.var
    void methode(int x) { ... }         // überschreibt Oben.methode(int)
    void methode(String x) { ... }// neue Methode
}
Unten unten = new Unten(); Oben oben = unten;
oben.var                                steht für var aus Oben (Wert = 7)
unten.var                               steht für var aus Unten (Wert = 8)
oben.methode(3);                        Unten.methode (späte Bindung)
unten.methode(3);                       Unten.methode (späte Bindung)
oben.methode("hallo");                  Fehler!
```

Für Variablen und für Klassenfunktionen gibt es keine späte Bindung!

Aufgaben von Compiler und Laufzeitsystem

	C	Java	Dyn. Sprache
Syntax	Compiler	Compiler	Compiler
Typprüfung	Compiler	**Compiler** (Laufzeit)	Laufzeit
Optimierung	Compiler	(Laufzeit)*	
Codegener.	Compiler	Compiler Laufzeit*	Compiler
Bindung	Compiler Linker	**Laufzeit**	Laufzeit
Ausführung	Laufzeit	Laufzeit	Laufzeit

* Hotspot VM mit Generierung von optimiertem native Code
  Die Java-VM unterstützt effizient lang laufende Server-Anwendungen
  Dalvik/Android: der Compiler optimiert zwecks effizienter Ausführung von Apps

Funktions-/Methodenaufrufe der JVM

Aufruf von	JVM-Befehl	Bindung	objektorientiert?
Klassenfunktion	invokestatic	früh	nein
Konstruktor, private Methode, super-Methode	invokespecial	früh	ja
Methode per Klasse	invokevirtual	spät	ja
Methode per Interface	invokeinterface	spät	ja
ungetypt	invokedynamic	spät	ja

Geschachtelte Klasse / Innere Klasse

```
class Outer {
    // geschachtelte Klasse = von außen nicht immer sichtbar
    // Name = Outer.Nested
    private static class Nested {  // muss nicht private sein!
        // ganz normale Klasse, Mame = Outer.Nested
    }

    // Innere Klasse: kennt das äußere Objekt!
    private int variable = 0;

    class Inner {            // Name = Outer.Inner
       void getVariable() {
           return variable; // oder Outer.this.variable
       }
    }

    void methode() {
        final int var = 3;
        class InnerOfMethode {
            // kann auf lokale final-Variablen zugreifen
        }
    }
}
```

Anonyme Klasse

Eine Anonyme Klasse ist eine Innere Klasse ohne Namen.
Damit sie verwendbar ist, muss bei der Definition ein Typ angegeben werden (Interface oder Oberklasse).

```
String[] strings = {...}; // ein Array von Strings

// absteigend sortieren
Arrays.sort(strings, new Comparator<String>() {
    public int compare(String a, String b) {
        return b.compareTo(a);
    } });
```

```
Comparator<String>() :  Angabe des Typs
                        (bei einer Klasse auch
                        Konstruktor-Argumente)

compare              :  Sinn und Zweck des Objekts ist
                        eine eigene Methode compare zu
                        Haben

compareTo            :  wir benutzen compareTo um zwei
                        Strings zu vergleichen (aber umgekehrt)
```

Innere und anonyme Klassen

- **Eingebettete Klasse:** Klasse in Klasse (Name: `Aussen.Innen`)
- **static class**: eingebette Klasse kennt kein umfassendes Objekt
- **Innere Klasse:** Innere Klasse kann auf Variablen des umfassenden Objekts zugreifen.
- **Anonyme Klasse:** Innere Klasse ohne Namen.
- **Bedeutung von static class:** Verstecken von Hilfsklassen
- **Bedeutung von anonymer Klasse:** Implementierung von Closures (Verwendung: GUI-Programmierung, Sortieren, Funktionen höherer Ordnung)
- **Anonyme Funktion (Lambda-Ausdruck) anstelle anonymer Klassen:** Funktionsobjekt mit Kenntnis der Umgebung von freien Variablen (aus Funktionaler Programmierung) – ab Java 8 auch endlich in Java

  Java 8: `Arrays.sort(strings, (x,y) -> y.compareTo(x));`

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