Wo sind wir?

- Java-Umgebung
- Lexikale Konventionen
- Datentypen
- Kontrollstrukturen
- Ausdrücke
- **Klassen, Pakete, Schnittstellen**
- JVM
- Exceptions
- Java Klassenbibliotheken
- Ein-/Ausgabe
- Collections
- Threads
- Applets, Sicherheit
- Grafik
- Beans
- Integrierte Entwicklungsumgebungen

Übersicht

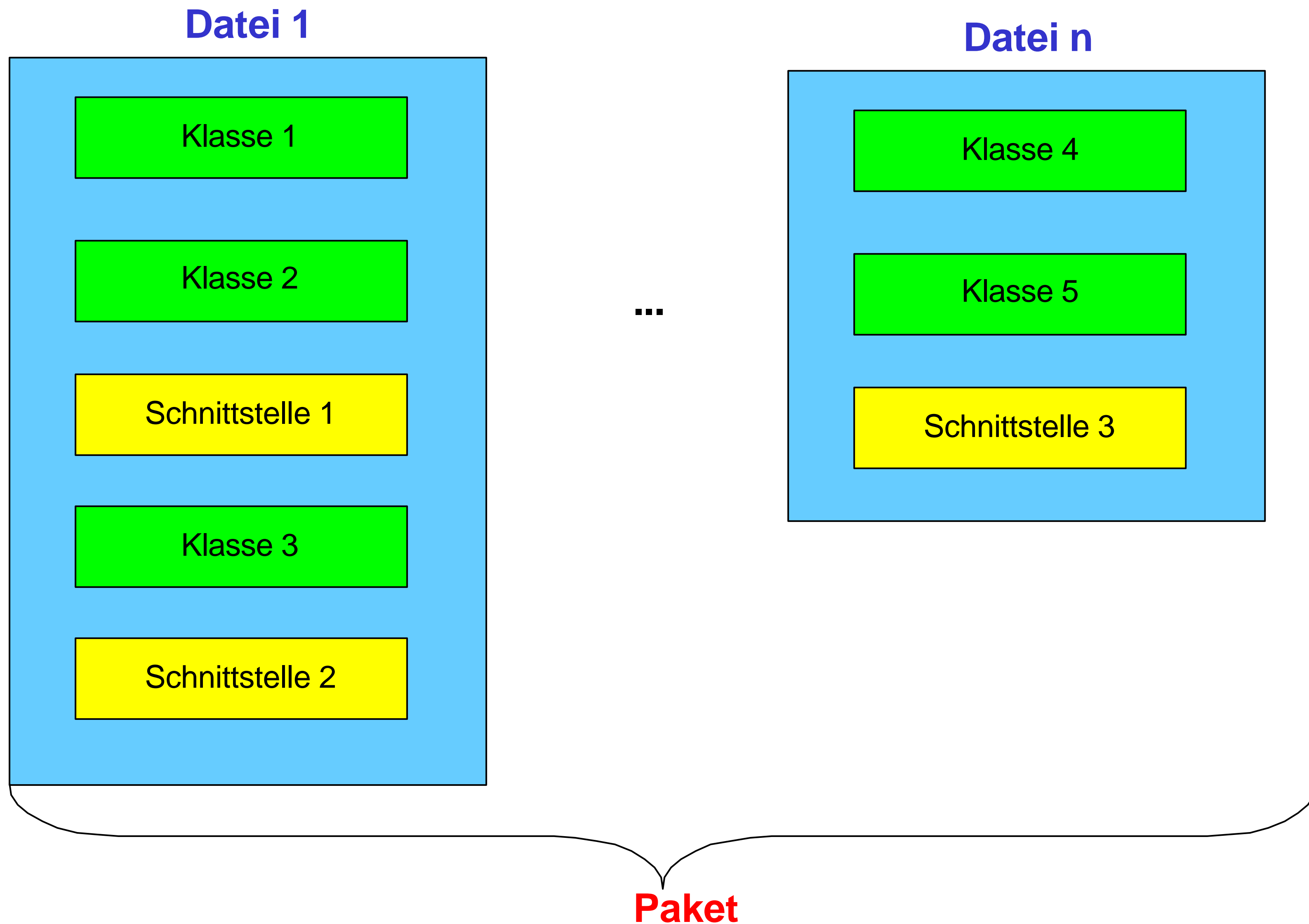
- Klassen sind mit abstrakten Datentypen vergleichbar und enthalten Datenfelder (Variablen) und Methoden. Klassen sind die vom System bereitgestellten oder selbst geschaffenen Schablonen für Objekte.

- Schnittstellen sind Zusammenstellungen von Methodenköpfen und eventuell Konstanten. Schnittstellen können von Klassen implementiert werden, die dann für jeden Methodenkopf der Schnittstelle eine konkrete Implementierung der Methode haben müssen. Eine Schnittstelle definiert also quasi ein Protokoll zur Kommunikation (Trennung von Spezifikation und Implementierung).

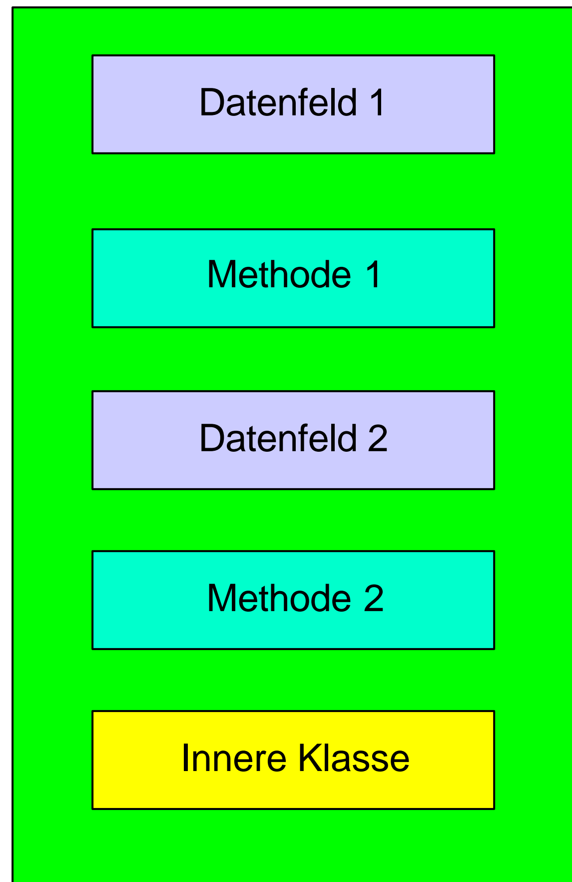
- Pakete dienen als Organisationseinheit, um mehrere Klassen, Schnittstellen oder Pakete sinnvoll zusammenfassen zu können. Man kann ein Paket auch als Klassenbibliothek auffassen. Weiterhin bilden Pakete einen abgeschlossenen Namensraum.

Klassen und Pakete spielen auch eine wichtige Rolle, den Zugriff auf Datenfelder und Methoden zu reglementieren (Nur Methoden aus meinem Paket dürfen meine Methode x() aufrufen und auf meine Variable y zugreifen).

Organisation von Programmteilen



Organisation von Programmteilen

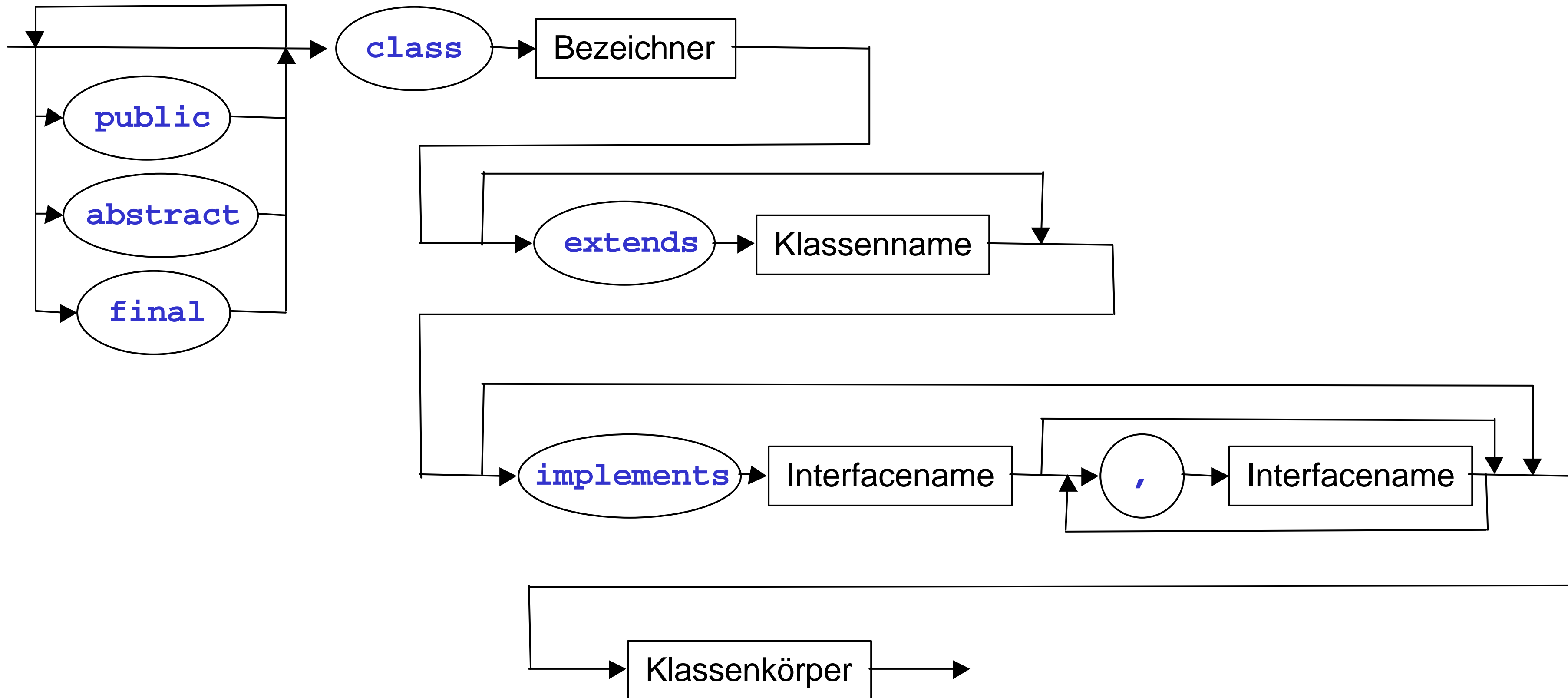


Wo sind wir im Detail bei KPS?

- **Aufbau von Klassen** (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Klasse

Klassendefinition

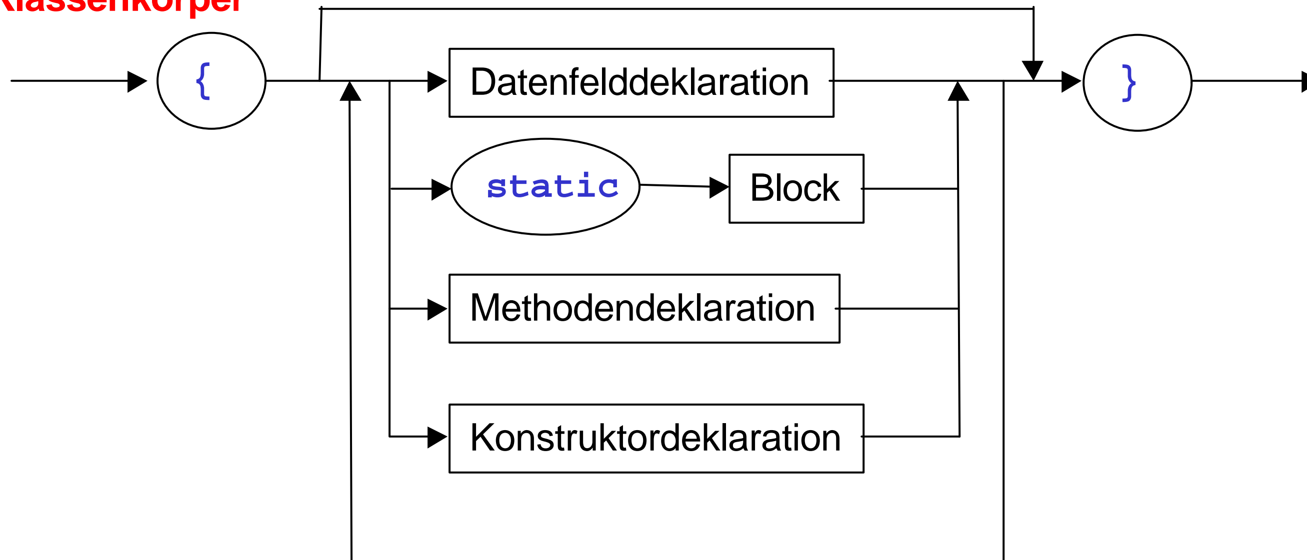


Mit einer Klassendeklaration spezifiziert man einen neuen Referenztypen und beschreibt dessen Implementierung (Datenfelder und Methode; vgl. abstrakter Datentyp).

Beispiele

```
// einfache Klasse
class MeineKlasse1 { /* Klassenkörper */ }

// abgeleitete Klasse
class MeineKlasse2 extends MeineKlasse1 { /* Klassenkörper */ }

// Klasse, die 2 Interfaces implementiert
class MeineKlase3 implements Interface1, Interface2 { /* Klassenkörper */ }

// Klasse mit Modifikatoren
public final class MeineKlasse4 { /* Klassenkörper */ }
```

Klassenkörper



Ein Klassenkörper (oder -rumpf) besteht typischerweise aus einer Anzahl von Datenfelddeklarationen und Methodendeklarationen.

Beispiel:

```
class MeineKlasse {
    int x;                          // Datenfelddeklaration
    MeineKlasse()  { x = 1; }       // Konstruktor (ähnlich einer Methode)
    void methode() { x = 2; }       // Methode
}
```

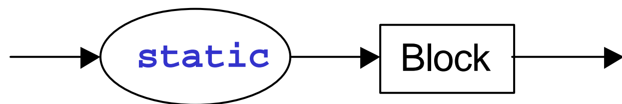
Übersicht Klasse / Klassenkörper

- Datenfelder (Variablen) speichern den internen Zustand eines Objektes. Beispiel: Eine Klasse Konto besitzt Datenfelder Kontonummer, Kontostand usw. Ein Objekt dieser Klasse hat für das Datenfeld Kontonummer einen konkreten Wert.
  Es gibt Klassenvariablen und Instanzenvariablen. Klassenvariablen existieren pro Klasse nur ein mal, in ihnen können also Objekte gemeinsame Informationen halten.

- Methoden implementieren die Algorithmen. Es gibt wiederum Klassenmethoden und Instanzenmethoden. Klassenmethoden existieren (fast) immer, also unabhängig von einem Objekt. Instanzenmethoden sind nur im Zusammenhang mit einem Objekt aufrufbar.

- Über Modifikatoren bei der Deklaration lassen sich z.B. Zugriffsrechte für Datenfelder und Methoden angeben. Beispiel: (1) Methode ist global bekannt, kann aus jeder Methode (auch aus anderen Klassen) ausgerufen werden. (2) Methode kann nur von Methoden innerhalb dieser Klasse aufgerufen werden.

Klassenbezogene Initialisierer



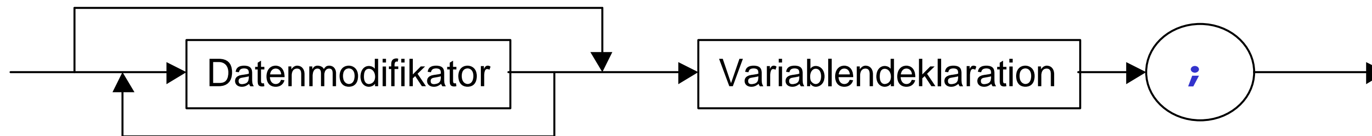
In jeder Klasse kann man beliebig viele klassenbezogene Initialisierer definieren. Diese werden genau ein mal, wenn die Klasse initialisiert wird (beim Laden der Klasse durch die JVM; später), in der Reihenfolge ihres Auftretens ausgeführt.

Beipiel:

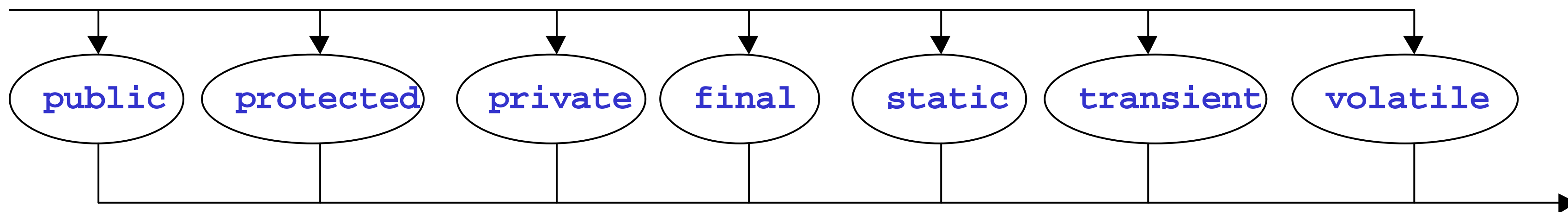
```
class MeineKlasse {
    static long startzeit;
    static {
        // ermittle die aktuelle Zeit (in msec seit 1970)
        startzeit = System.currentTimeMillis();
        System.out.println("Gestartet um " + startzeit);
    }
    static void sonstwas() { System.out.println("sonstwas"); }
}

class Test {
    public static void main(String[] args) {
        MeineKlasse.sonstwas();
    }
}
```

Datenfelder

Datenfelddeklaration



Datenmodifikator



Beispiele:

```
class MeineKlasse {
    int a;                                    // Typ und Variablenname
    protected volatile int b,c;               // 2xModifikator, 2 Variablen
    int[] d;                                  // Feldtyp
    static int e=4, f, g=6;                   // 3 Variablen, z.T. mit Initialisierung
    private static double[] = {1.0, 2.0};     // 2xModifikator, Felddimensionierung
}
```

Variablen

Drei Arten von Variablen sind hier interessant:

- Datenfelddeklaration in Form einer Klassenvariablen: Werden für jede Klasse einmal angelegt (mit `static` deklariert). Der Zugriff kann innerhalb der Klassen über den Namen des Datenfeldes erfolgen, von außerhalb der Klasse über Klassenname.Datenfeldname (falls über Zugriffsrechte erlaubt).
- Datenfelddeklaration in Form einer Instanzenvariablen: Werden für jede Instanz einer Klasse einmal angelegt (ohne `static` deklariert).
- Lokale Variablen werden bei Eintritt in einen Block angelegt. Methodenparameter sind spezielle lokale Variablen.

Beispiel:

```
class MeineKlasse {
    static int x;                     // Klassenvariable x
    int y;                            // Instanzenvariable y
    void meineMethode(int a) {        // Methodenparameter a
        int b;                        // lokale Variable b
    }
}
```

Beispiel

```
class Konto {
    int naechsteFreieNummer = 0;              // Klassenvariable
    int meineNummer;                          // Instanzenvariable

    Konto(String inhaber) {
        meineNummer = ++naechsteFreieNummer;          // neue Nummer vergeben
    }
}


class Bank {
    public static void main(String[] args) {
        Konto kontoSchmitz = new Konto("Willi Schmitz");    // Konto 1
        Konto kontoMeier = new Konto("Marlene Meier");      // Konto 2
    }
}
```

Die Klassenvariable `naechsteFreieNummer` wird benötigt, um über alle Konto-Objekte hinweg die nächste noch nicht vergebene Kontonummer zu speichern.

Initialisierung

- Klassenvariablen werden genau ein mal, bei der Initialisierung der Klasse, mit einem Wert initialisiert (Default-Wert oder expliziter Wert). Dies kann entweder in einem Zuweisungsausdruck bei der Deklaration oder in einem klassenbezogenen Initialisierer erfolgen.

- Instanzenvariablen werden mit jeder Instanziierung eines Objektes neu erzeugt und damit auch neu initialisiert (Default-Wert oder expliziter Wert).

- Nur wenn ein expliziter Initialisierungsausdruck angegeben wird, wird eine lokale Variable mit einem Wert initialisiert, nämlich an dem Punkt, wenn die Variable erzeugt wird. Ansonsten hat eine lokale Variable einen undefinierten Wert!

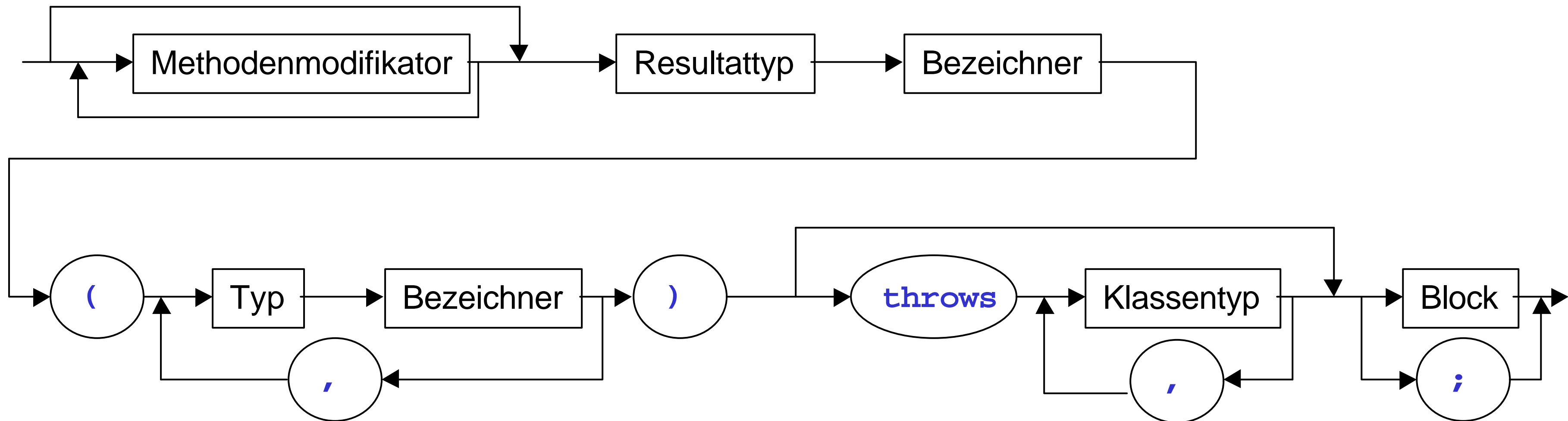
Beispiel

```
class MeineKlasse {
    static int a = 5, b;         // Klassenvariablen
    int c = 4711, d;             // Instanzenvariablen

    MeineKlasse() {
        int i = 5, j;            // lokale Variable
    }
}
```

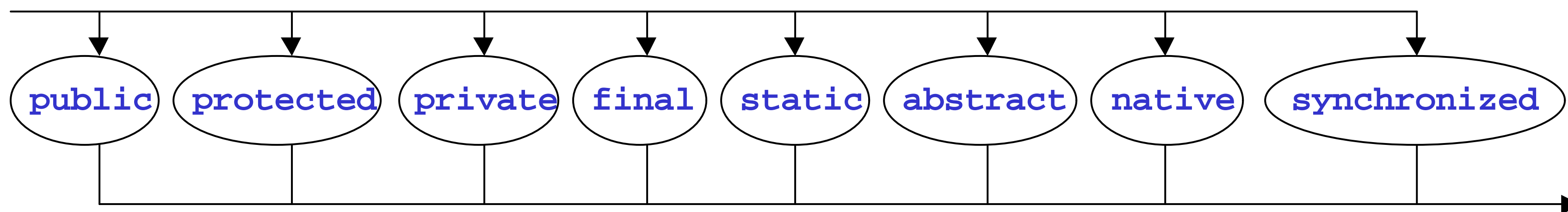
- a wird bei der Initialisierung der Klasse mit dem Wert 5 belegt (explizit).
- b wird bei der Initialisierung der Klasse mit dem Wert 0 belegt (implizit).
- c wird mit jedem neuen Objekt neu erzeugt und mit dem Wert 4711 belegt (explizit).
- d wird mit jedem neuen Objekt neu erzeugt und mit dem Wert 0 belegt (explizit).
- i wird mit jedem Aufruf des Konstruktors erzeugt und mit dem Wert 5 belegt (explizit).
- j wird mit jedem Aufruf des Konstruktors erzeugt und hat keinen definierten Wert.

Methoden

Methodendeklaration



Methodenmodifikator



Eine Methode hat eine Semikolon als Rumpf genau dann, wenn die Methode als `abstract` oder `native` deklariert wird.

Beispiele

```
class MeineKlasse {

    /* Methode ohne Rückgabewert (Resultattyp void), ohne formale Parameter
       und leerem Block
    */
    void methode1() { }


    // Modifikator final, Resultattyp int, einen Parameter
    final int verschluesseln(String str) { ... }


    // Modifikator private, Resultattyp int[], einen Parameter
    private int[] methode2(int[] a) { ... }


    // 2 Modifikatoren, Resultattyp DeineKlasse, 2 Parameter
    private static DeineKlasse methode3(int i, DeineKlasse[] a) {...}


    // Resultattyp int[][], einen Parameter, kann Exception auslösen
    int[][] methode4(int [][] a) throws ArrayIndexOutOfBoundsException { ... }

}
```

Signatur einer Methode, Überladen

Die Signatur einer Methode besteht aus dem Namen der Methode und der Anzahl und den Typen ihrer formalen Parameter. Weder die Namen der formalen Parameter, Modifikatoren oder throws-Angaben gehören zur Signatur.

In einer Klasse darf es nicht zwei Methoden mit der gleichen Signatur geben. Existieren zu einem Methodennamen einer Instanzenmethode mehrere Signaturen (auch ererbt möglich), so spricht man von Überladen dieses Methodennamens.

Beispiel:

```
class MeineKlasse {
    int methode1(int i, float j) { ... }             // neue Methode

    float methode1(int i, float j) { ... }           // erlaubt

    private int methode1(int j, float x) { ... }     // Fehler wg 1

                                                     // Fehler wg 1
    int methode1(int f, float g) throws ArrayIndexOutOfBoundsException { ... }

    int methode2(int i, float j) { ... }             // erlaubt
}
```

Klassen- und Instanzenmethoden

- Hat eine Methode den Modifikator static, so ist diese Methode eine Klassenmethode, die ohne Referenz auf ein Objekt aufgerufen wird. Solche Methoden existieren bereits mit dem Laden der Klasse durch die JVM (später) und können explizit über Klassenname.Methodename() jederzeit aufgerufen werden (siehe auch Zugriffsrechte). Innerhalb der Klasse reicht Methodenname().
  Klassenmethoden haben nur Zugriff auf Klassenvariablen und Klassenmethoden, außer man legt über new() in einer Klassenmethode eine neue Instanz an und spricht über diese Instanzenvariablen oder Instanzenmethoden an.

- Hat eine Methode keinen Modifikator static, so ist dies eine Instanzenmethode, die nur über ein Objekt (Instanz) dieser Klasse aufgerufen werden kann.

Klassen- und Instanzenmethoden

```
class MeineKlasse {
    MeineKlasse() {}

    // Klassenmethode wegen static
    static int klassenMethode() {
        System.out.println("Klassenmethode");
    }

    // Instanzenmethode wegen fehlendem static
    int instanzenMethode() {
        System.out.println("Instanzenmethode");
    }
}
```

```
class Test {
    public static void main(String[] args) {        // main ist auch static!

        MeineKlasse.klassenMethode();       // Aufruf Klassenmethode erlaubt

        MeineKlasse o = new MeineKlasse();
        // Instanzenmethode nur über Objektinstanz erlaubt
        o.instanzenMethode();
    }
}
```

Klassenmethoden und Klassenvariablen

```
class MeineKlasse {
    static int x;                              // Klassenvariable
    int y;                                     // Instanzenvariable

    static void klassenMethode() {             // Klassenmethode wegen static
        System.out.println("Klassenmethode");
    }

    void instanzenMethode() {                  // Instanzenmethode
        System.out.println("Instanzenmethode");
    }

    public static void main(String[] args) {
        x = 5;                                 // erlaubt
        y = 6;                                 // Fehler: aus Klassenmethode

        klassenMethode();                      // erlaubt
        MeineKlasse.klassenMethode();          // erlaubt
        instanzenMethode();                    // Fehler: aus Klassenmethode

        MeineKlasse m = new MeineKlasse();     // neue Instanz anlegen
        m.y = 7;                               // erlaubt
        m.instanzenMethode();                  // erlaubt
    }
}
```

Konstruktoren

Konstruktoren sind ähnlich den Methoden, nur mit den folgenden Einschränkungen:

- Ein Konstruktor hat den Namen der Klasse als Methodenname
- Ein Konstruktor hat keinen Ergebnistyp.
- Als Modifikatoren sind nur `public`, `private` und `protected` erlaubt.

Ebenso wie bei Methoden kann es auch gleichzeitig Konstruktoren mit unterschiedlicher Signatur geben (Überladen).

Enthält eine Klasse keine explizite Konstruktordeklaration (und nur dann), so wird implizit ein parameterloser Konstruktor definiert, der lediglich den parameterlosen Konstruktor der Oberklasse aufruft.

Beispiel

```
class Punkt {

    int x, y;                          // Koordinaten des Punktes

    Punkt() {                          // Punkt mit Default-Koordinaten erzeugen
        x = y = 0;
    }

    Punkt(int x, int y) {              // Punkt mit vorgegebene Koordinaten erzeugen
       this.x = x;
       this.y = y;
    }
}
```

```
class K1 {
    // Kein expliziter Konstruktor, also wird Konstruktor von Object aufgerufen
    int methode1() { ... }
}

class K2 extends K1 {
    // Kein expliziter Konstruktor, also wird Konstruktor von K1 aufgerufen
    int methode2() { ... }
}
```

Konstruktor ruft Konstruktor auf

Ein Konstruktor kann in seiner ersten Anweisung einen anderen Konstruktor der gleichen Klasse aufrufen (der eine andere Signatur haben muss). Sinnvoll kann dies sein, wenn man einen "Hauptkonstruktor" hat und weitere Konstruktoren anbieten will.

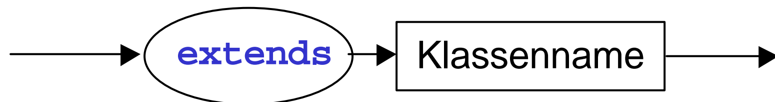
Beispiel:

```
class Punkt {
    float x,y;

    // Hauptkonstruktor
    Punkt(float xpos, float ypos) { this.x = xpos; this.y = ypos; }

    // Nebenkonstruktoren rufen Hauptkonstruktor auf
    Punkt(int xpos, int ypos) { this((float)xpos, (float(ypos); }
    Punkt(double xpos, double ypos) { this((float)xpos, (float)ypos); }
    ...
}
```

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- **Vererbung** (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Vererbung



Wird bei einer Klassendefinition einer Klasse K1 zusätzlich `extends K2` angegeben, so ist K1 eine direkte Unterklasse von K2 ist, K2 ist die direkte Oberklasse von K1. Es kann nur eine direkte Oberklasse geben.

Der Klassenname K2 muss entweder im aktuellen Paket existieren oder man muss Zugriff auf das betreffende Paket (später) haben und die Klasse dort als `public` deklariert sein.

Existiert in einer Klassendefinition keine `extends`-Angabe, so hat diese Klasse die Klasse `Object` als direkte Oberklasse.

Alle Variablen und Methoden der Oberklasse werden in der Unterklasse geerbt. Durch Überschreiben in der Unterklasse und über Modifikatoren in der Oberklasse kann man die Sichtbarkeit von Variablen und Methoden jedoch einschränken (später).

Beispiel

```
class MeineKlasse {
    static int x = 4711;
    private static int y;
    int z;
}

class DeineKlasse extends MeineKlasse {
    static double x = 3.1415;

    public static void main(String[] args) {
        DeineKlasse o = new DeineKlasse();
        x = 31415;          // möglich
        y = 2714;           // Übersetzungsfehler wegen private in Oberklasse
        t.z = 64136;        // möglich
        t.ausgabe();
    }

    static void ausgabe() {
        System.out.println(x + " " + super.x);
    }
}
```

Was wird ausgegeben (ohne Zeile mit Übersetzungsfehler)?

```
3.1415 4711
```

Überschreiben

Wir hatten das Überladen von Methoden kennen gelernt: Methode gleichen Namens aber unterschiedlicher Signatur.

Leitet man eine Klasse K2 von einer Klasse K1 hat, die eine Instanzenmethode mit der Signatur x hat, so kann man in der Klasse K2 ebenfalls eine Instanzenmethode mit der Signatur x anlegen, man überschreibt damit die Methode in K1.

Beispiel:

```
class K1 {
   void methode(int i) {
      System.out.println("in K1");
   }
}

class K2 extends K1 {
   void methode(int i) {
      System.out.println("in K2");
      super.methode(i);
   }
}
```

```
class K3 {
    public static void main(String[] args) {
        K2 o = new K2();
        o.methode(5);     // aus K2
    }
}
```

Beispiel

```
class KFZ {
    int ladevermoegen = 0;
    void motorStarten() { System.out.prinln("brumm brumm"); }
}

class Motorrad extends KFZ {
    boolean kickstarter;
    Motorrad(boolean kickstarter) { this.kickstarter = kickstarter; }
    void motorstarten() { System.out.println(kickstarter ? "kick" : "drück"); }
}

class PKW extends KFZ {
    int ladevermoegen = 400;
    void klimaAn() { System.out.println("angenehme Temperatur"); }
}

class LKW extends KFZ {
    int ladevermoegen = 36000;
}

class Kipper extends LKW {
    int ladevermoegen = 16000;
    void kippen() { System.out.println("kipp"); }
}
```

Frage: Welche Methoden und Datenfelder kennen Objekte der einzelnen Klassen?

Methodenaufruf und Instanzenvariablen

Erfolgt ein Methodenaufruf, so wird zur Laufzeit (dynamisch) jeweils die richtige Methode gesucht.
"Richtig" bedeutet, dass vom Objekt beginnend die Vererbungshierarchie aufwärts (von Unterklasse zur Oberklasse) gegangen wird, bis die erste Methode mit der gesuchten Signatur gefunden wird.
"Jeweils" bedeutet, dass bei inneren Methodenaufrufen (in einem Methodenaufruf ein weiterer Methodenaufruf) auch wieder vom Ursprungsobjekt ausgegangen wird!

Bei Instanzenvariablen sucht dagegen der Compiler schon (statisch) die Variable aus (Regeln zum Gültigkeitsbereich und Sichtbarkeit von Variablen; später).

Bei Typumwandlungen von Referenztypen kommen wir noch einmal auf dieses Thema zurück.

Beispiel

```
class KFZ {
    int ladevermoegen = 0;
    public String bezeichnung() { return "KFZ"; }
    public String toString() {return bezeichnung() + ", Laden=" + ladevermoegen;}
}

class LKW extends KFZ {
    int ladevermoegen = 36000;
    public String bezeichnung() { return "LKW"; }
}

class Kipper extends LKW {
    int ladevermoegen = 16000;
    public String bezeichnung() { return "Kipper"; }
}

public class Test {
    public static void main(String[] args) {
        Kipper kipper = new Kipper();
        System.out.println(kipper.toString());
    }
}
```

statisch

dynamisch

dynamisch

Ausgabe:

```
Kipper, Laden=0
```

Bindung

Die Verknüpfung eines Methodenaufrufs mit einen Methodenrumpf nennt man Bindung.

Java nutzt für Instanzenmethoden die dynamische Bindung (dynamic binding, late binding), wie es eben beschrieben wurde. Im Zusammenhang mit Typumwandlungen wird der Vorteil dieser Bindungsmethode ersichtlich.

Für Klassenmethoden wird die statische Bindung (static binding, early binding), wo der Compiler und Linker zur Übersetzungs- und Bindungszeit eindeutig festlegt, welche Methode bzw. Methodenrumpf für einen Methodenaufruf genommen wird.

Andere Programmiersprachen wie z.B. C kennen nur die statische Bindung bei Methodenaufrufen.

Die Klasse Object

Die Klasse `Object` ist die "Mutter aller Klassen", von ihr sind alle weiteren Klasen abgeleitet. Das heißt auch, dass alle Methoden der Klasse `Object` in allen anderen Klassen bekannt sind (wo sie aber überschrieben werden können, wenn nicht als `final` deklariert).

Hier einige wichtige Methoden der Klasse `Object`:

```
public final Class getClass();     // zugehöriges Klassenobjekt
public String toString();          // textuelle Repräsentation
public boolean equals(Object obj); // liefert true für "gleiche" Objekte
public int hashCode();             // für Hashtabellen
protected void finalize();         // wird vor Löschen aufgerufen
```

Für eine vollständige Beschreibung siehe API-Dokumentation.

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- **Wrapperklassen**
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Wrapper-Klassen

Zu allen einfachen Typen gibt es sogenannte Wrapper-Klassen mit Namen `Boolean, Byte, Short, Integer, Long, Character, Float, Double`.

Wrapper-Klassen haben mehrere Funktionen:

- Mit diesen Klassen kann man Objekte und Referenzen darauf erzeugen. Anstatt die Zahl 3 (Typ `int`) direkt zu speichern, kann man nun ein Objekt (Typ `Integer`) erzeugen, das die Zahl 3 enthält. Man kann z.B. mehrere Referenzvariablen auf dieses Objekt verweisen lassen. Man kann allerdings den Wert solch eines Objektes nachträglich nicht mehr verändern!

- In den Wrapper-Klassen sind Konstanten und Methoden zu den Typen definiert, z.B. der maximal darstellbare Wert dieses Typs, das Umwandeln der String-Darstellung eines Wertes in einen int-Wert.

Details zu allen Wrapper-Klassen in der API-Dokumentation auf der WWW-Seite zur Veranstaltung.

Beispiel: Wrapper-Klasse Integer

```
public final class Integer extends Number implements Comparable {
// (Klassen-) Konstanten
    public static final int MAX_VALUE;     // 2147483647
    public static final int MIN_VALUE;     // -2147483648
    public static final Class TYPE;

// Klassenmethoden
    public static int parseInt(String s) throws NumberFormatException;
    public static int parseInt(String s, int radix) throws Number...;
    public static Integer valueOf(String s) throws NumberFormatException;
    public static Integer valueOf(String s, int radix ) throws Number...;
    public static String toString(int i);
    public static String toString(int i, int radix);
        ...

// Konstruktoren
    public Integer(int value);
    public Integer(Strings s) throws NumberFormatException;

// Instanzenmethoden
    public int compareTo(Integer anotherInteger);
    public byte byteValue();
    public double doubleValue();
    public float floatValue();
    ...
}
```

Beispiel 1

```
// In Kommandozeile Werte übergeben

class Test {

    public static void main(String[] args) {

        int i, wert1, wert2;
        float wert2;

        if(args.length != 2)
            System.out.println("Bitte 2 Argumente übergeben");

        else {
           // Strings der Kommandozeile umwandeln in Werte
           wert1 = Integer.parseInt(args[0]);
           wert2 = Integer.parseInt(args[1]);

           // Mit den beiden Werten rechnen
           System.out.println(wert1 + "*" + wert2 + "=" + (wert1*wert2));
        }
    }

}
```

Beispiel 2

```
class WrapperTestInteger {
    public static void main(String [] args) {

        Integer i1 = new Integer(3);    // Konstruktor
        Integer i2 = i1;                // Referenz
                                       // Klassenmethode
        Integer i3 = new Integer(Integer.parseInt("31"));

                                        // Konstanten
        System.out.println("Bereich:" + Integer.MIN_VALUE
                           + " bis " + Integer.MAX_VALUE);
        System.out.println(i1 + " " + i2 + " " + i3);
    }
}
```

Liefert zum String "31" die Zahl 31

Ausgabe:

```
C:\java\>  java WrapperTestInteger
Bereich:-2147483648 bis 2147483647
3 3 31
```

Beispiel: Wrapper-Klasse Character

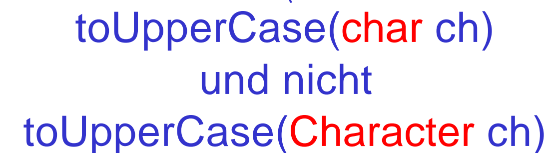
```
public final class Character extends Number implements Comparable {
// (Klassen-) Konstanten
    public static final char MAX_VALUE;    // '\uffff'
    public static final char MIN_VALUE;    // '\0'
    public static final Class TYPE;
    ....

// Klassenmethoden
    public static boolean isDigit(char ch);
    public static boolean isLetter(char ch);
    public static boolean isLetterOrDigit(char ch);
    public static boolean isLowerCase(char ch);
    public static boolean isUpperCase(char ch);
    public static char toLowerCase(char ch);
    public static char toUpperCase(char ch);
    ...

// Konstruktoren
    public Character(char value);

// Instanzenmethoden
    public char charValue();
    ...
}
```

Beispielanwendung

```
class WrapperTestCharacter {
    public static void main(String [] args) {

        Character c1 = new Character('a');    // Konstruktor
        char c2 = c1.charValue();             // Instanzenmethode

        System.out.println("c1=" + c1);
        System.out.println("c2=" + c2);
        System.out.println(c1 + " ist Großbuchstabe: "
                        + Character.isUpperCase(c1.charValue()));
        System.out.println(c1 + " als Großbuchstabe: "
                        + Character.toUpperCase(c1.charValue()));
    }
}
```

Ausgabe:

```
C:\java\>  java WrapperTestCharacter
c1=a
c2=a
a ist Großbuchstabe: false
a als Großbuchstabe: A
```



Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- **Geschachtelte Klassen**
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Geschachtelte Klasse

Innerhalb einer Klasse kann man weitere geschachtelte (innere) Klassen definieren. Diese geschachtelten Klassen dienen dazu, "Hilfstypen" zu implementieren, die nur in einer (der umfassenden) Klasse benötigt werden. Da die innere Klasse nur in der umfassenden Klasse bekannt ist, wird somit ein Überfluten des Namensraums verhindert.

Es gibt vier verschiedene Arten von inneren Klassen (je nachdem, wo sie definiert sind):

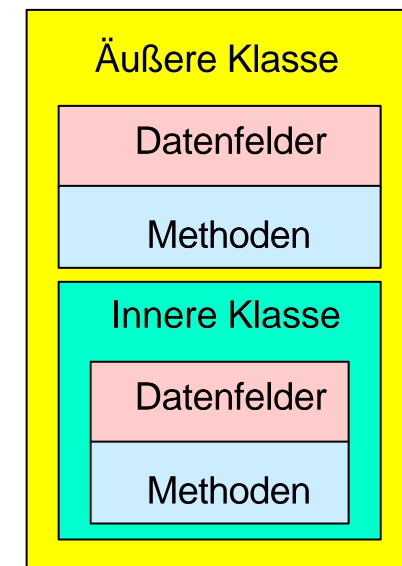
- Elementklassen
- Lokale Klassen
- Anonyme Klassen
- Statisch geschachtelte Klasse

Geschachtelte Klasse: Elementklassen

Elementklassen können, analog wie Instanzenvariablen und -methoden, nur im Zusammenhang mit einem Objekt der umfassenden Klasse existieren. Für Elementklassen gilt der gleiche Zugriffsschutz wie für Datenfelder und Methoden.

Beispiel:

```
public class Aussen {
    int x;
    Aussen() { x = 5; }
    public void methode1(int z) { x = z; }

    public class Innen {
        int y;
        public void methode2(int z) {y = z; x = z; }
    }
}

public class Test {
    public static void main(String[] args) {
        Aussen a = new Aussen();
        a.methode1(31415);
        Aussen.Innen b = a.new Innen();
        b.methode2(4711);
    }
}
```



Geschachtelte Klasse: Statische Klasse

Statische Kassen (mit dem Schlüsselwort `static` versehen) brauchen im Gegensatz zu Elementklassen kein Objekt der äußeren Klasse, um ein Objekt der inneren Klasse zu erzeugen. Der einzige Unterschied zu einer äußeren Klasse ist der Umstand, dass die innere Klasse nur über den Namen der äußeren Klasse angesprochen wird.

Beispiel:

```
public class Aussen {
    ...
    public static class Innen {
        ...
    }
}

public class Test {
    Aussen a = new Aussen();
    Aussen.Innen b = new Aussen.Innen(); // unabhängig von letzter Zeile möglich
}
```

Geschachtelte Klasse: Lokale und anonyme Klassen

- Lokale Klasse: Innerhalb eines Blocks eine neue Klasse
  Beispiel:
```
class Aussen {
    void methode() {
        class Innen {
            ...
        }
        Innen a = new Innen();
    }
}
```

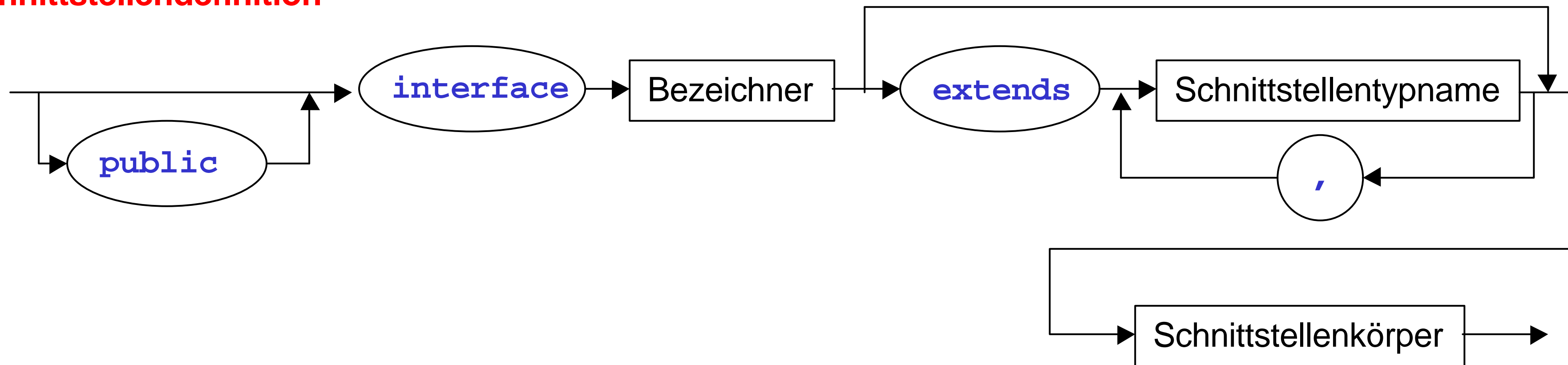
- Anonyme Klasse: Klassen ohne Namen, von denen bei der Klassendefinition ein Objekt schon erzeugt wird.

Diese beiden Fälle von inneren Klassen werden selten genutzt!

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- **Schnittstellen**
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Schnittstelle



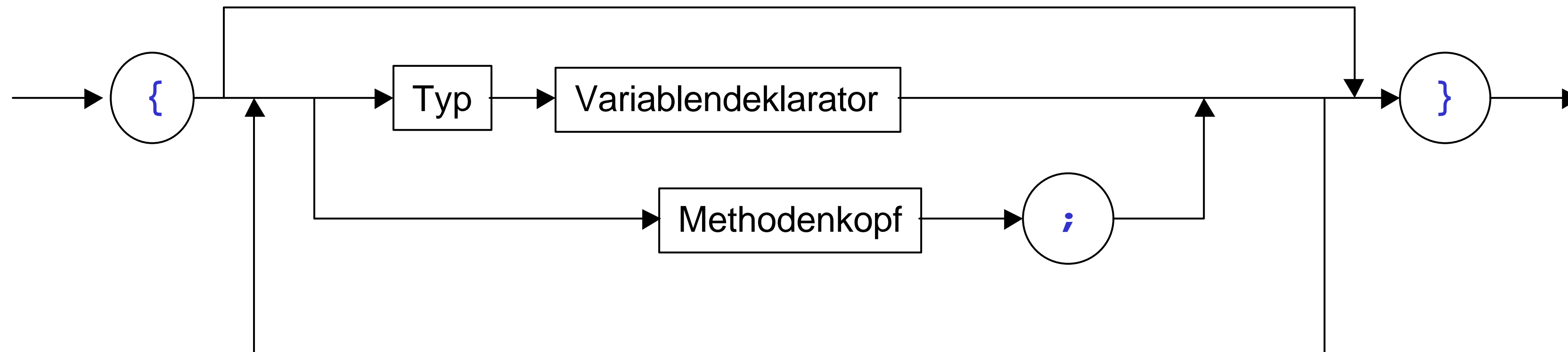
Schnittstellen dienen der Spezifikation eines Protokolls in Form von abstrakten Methoden. Eine oder mehrere Klassen können eine Schnittstelle implementieren, indem sie konkrete Methoden für die in der Schnittstelle bezeichneten abstrakten Methoden angeben.

Eine Schnittstellendeklaration führt einen neuen Referenztyp mit Konstanten und abstrakten Methoden ein.

Im Gegensatz zu Klassen ist bei Schnittstellen Mehrfachvererbung zugelassen!

Schnittstelle

Schnittstellenkörper



Jede Deklaration eines Datenfeldes ist implizit `public static final` und muss im Variablendeklarator einen Initialisierungsausdruck besitzen (Wert der Konstanten). Der Initialisierungsausdruck muss nicht konstant sein!

Alle Methoden sind implizit als `abstract public` deklariert. Genau wie bei Klassen können auch bei Schnittstellen mehrere Methoden mit gleichem Namen aber unterschiedlicher Signatur deklariert werden (Überladen) und genauso Methoden aus Oberschnittstellen überschrieben werden.

Beispiel

```
interface Punkt {
    int NULLPUNKT_X = 0;          // Konstante (implizit final)
    int NULLPUNKT_Y = 0;          // Konstante (implizit final)

    void punktVerschieben(int delta_x, int delta_y);
}
```

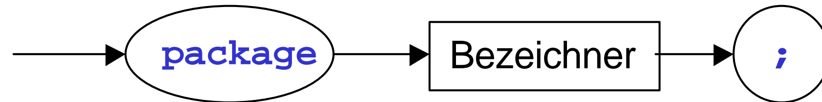
```
interface Linie {
    void linieVerschieben(int delta_x, int delta_y);
}
```

```
class GrafikKlassen implements Punkt, Linie {
    int ursprung_x, ursprung_y;

    // konkrete Implementierung der abstrakten Methode aus Punkt
    public void punktVerschieben(int delta_x, int delta_y) {
        ursprung_x += delta_x;
        ursprung_y += delta_y;
    }

    // konkrete Implementierung der abstrakten Methode aus Linie
    public void linieVerschieben(int delta_x, delta_y) {
        ursprung_x += delta_x;
        ursprung_y += delta_y;
    }
}
```

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- **Pakete**
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Paketdeklaration



Pakete dienen dazu, die Software eines Projektes in größere inhaltlich zusammengehörige Bereiche mit eigenem Namen einzuteilen (siehe Java API).

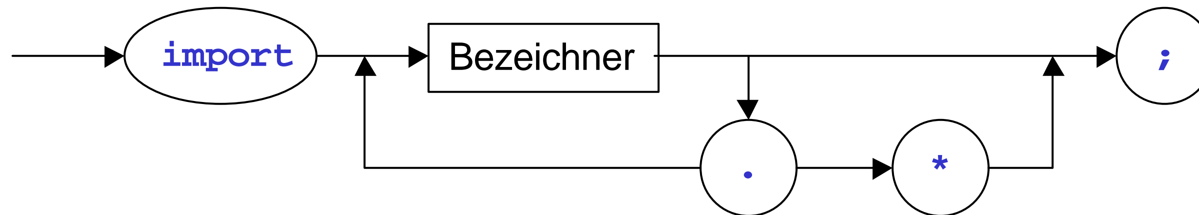
Ein Paket kann aus mehreren Quelldateien bestehen. Jede Quelldatei muss als erste Anweisung in der Datei eine Paketdeklaration haben. Nur Kommentare sind vorher erlaubt. Mehr als eine Paketdeklaration ist pro Datei nicht erlaubt.

Enthält die Datei eine Klasse mit dem Modifier `public`, so muss der Dateiname gleich dem Namen der Klasse mit Endung `.java` bestehen.
Folglich kann es auch nicht mehr als eine public-Klasse in einer Java-Datei geben.
Es kann in einer Datei aber beliebig viele Klassen ohne den Modifier `public` geben. Dies sind dann "Hilfsklassen", die nur innerhalb des Pakets (inkl. den anderen Quelldateien des Pakets) genutzt werden können.

Für Quelldateien ohne public-Klasse kann der Name beliebig gewählt werden.

Paketnutzung

Importdeklaration



Will man Klassen (inkl. Methoden und Datenfelder) aus einem Paket in einem anderen Paket nutzen, muss man das Paket dort erst über eine `import`-Deklaration zugreifbar machen. Es gibt zwei Möglichkeiten:

- Genau eine Klasse eines Pakets zugänglich machen.
  Beispiel:
  ```
  import GrafikPaket.Linie;
  ```

- Alle Klassen eines Pakets zugänglich machen.
  Beispiel:
  ```
  import GrafikPaket.*;
  ```

Das Paket java.lang wird automatisch von jedem Java-Compiler importiert!

Zugriff auf Paketkomponenten

Möchte man in einer Klasse A auf eine Klasse B im Paket P zugreifen, so geschieht dies (nach einer import-Deklaration) durch die Syntax:

P.B

d.h. durch Voranstellen des Paketnamens und eines Punktes.

Ist der Name B eindeutig, so kann man auch (nach der import-Deklaration) nur B angeben.

Um Namenskonflikte bei Paketnamen zu vermeiden (Firma X, Firma Y haben beide ein Paket A), gibt es folgende Übereinkunft für Pakete, die man extern zugänglich machen möchte:

- Paketnamen werden komplett aus Kleinbuchstaben gebildet.
- Dem Paketnamen wird in umgekehrter Reihenfolge ein (in diesem Zusammenhang sinnvoller) Internet-Domain-Name vorangestellt.
  Beispiel:
  `de.fh-bonn-rhein-sieg.inf.meintollespaket`
  für das Paket `meintollespaket` in der Domain `inf.fh-bonn-rhein-sieg.de`

Beispiel

```
package grafikpaket;

class Punkt {               // Hilfsklasse (kein public)
    float x, y;
    Punkt(float xpos, float ypos) { x = xpos; y = ypos; }
}

public class Linie {        // öffentliche Klasse
    Punkt startpunkt, endpunkt;
    public Linie(float x1, float y1, float x2, float y2) {
        startpunkt = new Punkt(x1, y1);
        endpunkt = new Punkt(x2, y2);
    }
}
```

GrafikPaket/Linie.java

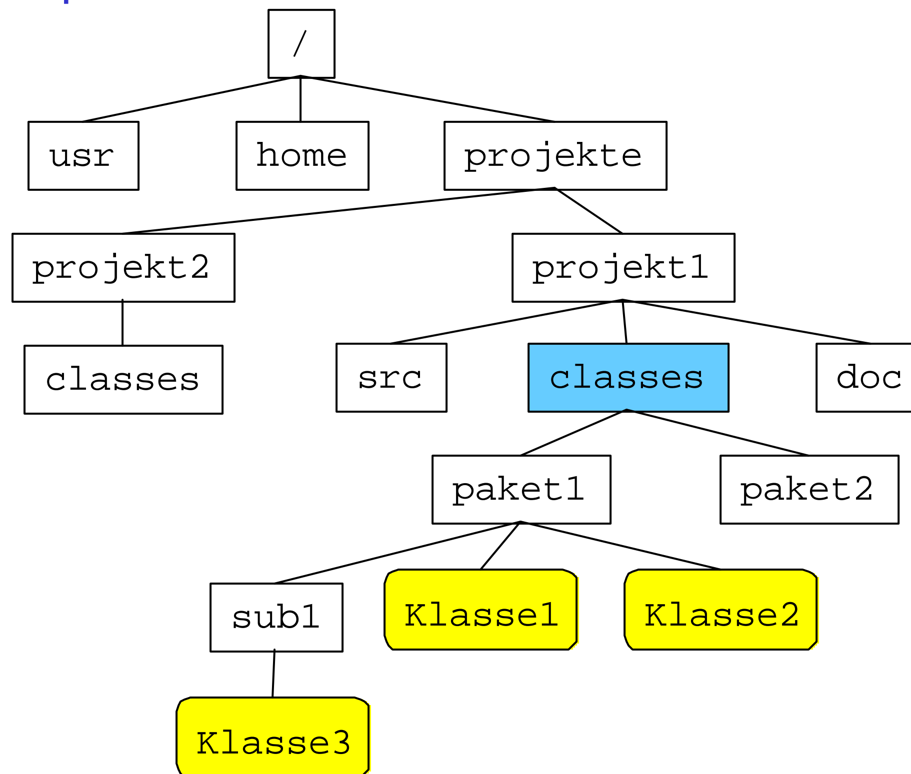
```
import grafikpaket.Linie;

public class Test {
    public static void main(String[] args) {
        grafikpaket.Linie l1 =
                   new grafikpaket.Linie(0.0f, 0.0f, 1.0f, 1.0f);
        Linie l2 = new Linie(0.0f, 0.0f, 2.0f, 2.0f);
    }
}
```

Test.java

Organisation von Paketen

Die Paketstruktur wird normalerweise 1:1 auf eine Dateistruktur übertragen, indem jedem Paket ein Verzeichnis zugeordnet wird, das den gleichen Namen trägt wie das Paket, und die Klassen als Dateien dort abgelegt sind.

Beispiel:



classes ist Wurzel des Paketbaums für projekt1

```
import paket1.Klasse1;
import paket1.Klasse2;

import paket1.sub1.Klasse3;
```

Organisation von Paketen

Um beim Übersetzen und Starten einer Anwendung die Wurzel eines Paketbaums (oder mehrerer) bekannt zu machen, dient der Klassenpfad (class path). Üblicherweise kann man den Klassenpfad beim Starten des Java-Compilers und -Interpretierers über eine Option angeben oder auch über eine Umgebungsvariable `CLASSPATH` setzen.

Beispiel:

```
prompt> export CLASSPATH=/projekte/projekt1/classes:/projekte/projekt2/classes
```

Der Java-Compiler und -Interpretierer such z.B. eine Klasse `paket1.Klasse1`, indem mit obigem Classpath nacheinander gesucht wird:

```
/projekte/projekt1/classes/paket1.Klasse1
/projekte/projekt2/classes/paket1.Klasse1
```

Seit neueren JDK-Versionen sind das aktuelle Verzeichnis und das Java-Verzeichnis (wo das Java JDK installiert ist), implizit im Suchpfad enthalten.

Organisation von Paketen

Arbeitet man, ohne den Klassenpfad explizit gesetzt zu haben, so übersetzt und startet man relativ von der Wurzel des Paketsbaums.

Beispiel (angenommen, auch die Quelldateien befinden sich im Verzeichnisbaum unter classes):

```
prompt> cd /projekte/projekt1/classes
prompt> javac paket1/Klasse1.java
prompt> javac paket1/Klasse2.java
prompt> javac paket1/sub1/Klasse3.java

prompt> java paket1/Klasse1
```

Anonymes Paket

Gibt man in einer Datei keine Paketdeklaration an, so gehört diese Übersetzungseinheit zu einem anonymen Paket. Zu diesem anonymen Paket gehören alle Klassen aller Dateien in diesem Verzeichnis, die keine Paketdeklaration enthalten.

Für kleinere Testanwendungen ist dies wesentlicher einfacher zu handhaben.

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- **Gültigkeitsbereich, Sichtbarkeit, Lebensdauer**
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Gültigkeitsbereich von Deklarationen

Der Gültigkeitsbereich einer Deklaration gibt an, in welchen Programmteilen auf die deklarierte Einheit mit einem einfachen Namen zugegriffen werden kann.

Die Möglichkeit zum Zugriff kann durch Zugriffskontrolle (über Modifier) weiter eingeschränkt werden.

Beispiel:

```
class MeineKlasse {
    public static void main(String[] args) {
        int i = 5;

        {
            int j = 3;

            i = j;
        }
    }
}
```

Gültigkeitsbereich von i

Gültigkeitsbereich von j

Regeln zum Gültigkeitsbereich von Namen

Typ	Gültigkeitsbereich
Paket	Systemumgebung legt dies fest (CLASSPATH usw.)
import-Deklaration	Übersetzungseinheit (Datei)
Klassendefinition, Schnittstellendef.	Alle Klassen und Schnittstellen des Pakets (auch in anderen Dateien)
Attribut in Klasse, Schnittstelle	Gesamte Klasse / Schnittstelle
Methoden- und Konstruktorparameter	Rumpf der Methode / des Konstruktors
Lokale Variable in Block	Block ab dem Deklarationspunkt
Variable in for-Anweisung	for-Anweisung ab dem Deklarationspunkt
Parameter in catch-Konstrukt	catch-Block

Sichtbarkeit

Die Sichtbarkeit einer Variablen besagt, dass diese Variable an einer Stelle im Programm gültig ist und über ihren einfachen Namen zugegriffen werden kann.

Eine Variable kann gültig sein, aber durch eine andere Variable gleichen Namens "verdeckt" sein.

Beispiel:

```
class MeineKlasse {
    static int x = 4;                      // Klassenvariable x

    static void meineMethode(int y) {
        int x = y;                         // Klassenvariable wird verdeckt
        System.out.println(x);             // Lokale Variable x (=4711)
        System.out.println(MeineKlasse.x); // Klassenvariable x (=4)
    }

    public static void main(String[] args) {
        meineMethode(4711);
        System.out.println(x);             // Klassenvariable x (=4)
    }
}
```

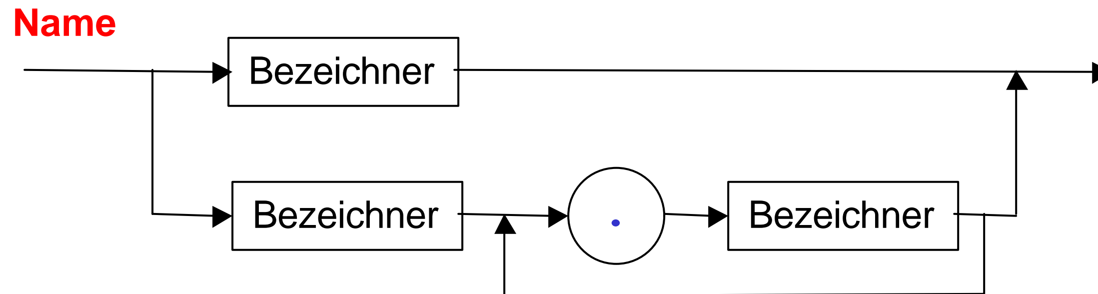
Lebensdauer

Die Lebensdauer einer Variablen oder eines Objektes ist die Zeitspanne, in der die Variable oder das Objekt existiert und Speicher belegt.
Beispiel:

```
class MeineKlasse {

    static int x;                    // Klassenvariable x
    int y;                           // Instanzenvariable y

    int meineMethode(int a) {
        int b = 2 * a;               // lokale Variable b
        return b;
    }
}
```

- Klassenvariablen (x) existieren vom Laden der Klasse bis zum Entladen
- Instanzenvariablen (y) werden mit dem Erzeugen eines Objektes der Klasse MeineKlasse erzeugt und existieren solange, solange dieses Objekt existiert.
- Lokale Variablen (b) existieren, solange der Block aktiv ist (Aufruf der Methode (genauer: vom Deklarationspunkt im Block) bis zur return-Anweisung). Einfache Implementierung über Stack möglich!

Einfache und qualifizierte Namen



Eine Name wird verwendet, um auf eine zuvor deklarierte Einheit (Paket, Typ, Methode, Variable) Bezug nehmen zu können. Namen sind entweder einfache Namen oder qualifizierte Namen (z.B. über Pakete, Klassen).

Beispiel:

```
class MeineKlasse {
    static int x;

    public static void main(String [] args) {
        x = 4711;                         // einfacher Name (var)
        MeineKlasse.x = 31415;            // qualifizierter Name (Klasse.var)
        x = java.lang.Integer.MAX_VALUE;  // qualifizierter Name
                                          // (Paket.Klasse.var)
    }
}
```

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- **Modifikatoren** (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Übersicht Modifikatoren (1)

Durch die Angabe von Modifikatoren lassen sich Eigenschaften (z.B. Zugriffsrechte) von Datenfeldern, Methoden, Klassen u.a. steuern (später genauer).

- `final`
  Eine als final deklarierte Variable, Methode oder Klasse kann nicht mehr verändert/überschrieben werden.
- `private, protected, public`
  Steuerung von Zugriffsrechten
- `static`
  Klassenvariablen, Klassenmethoden, geschachtelte Klassen, Schnittstellen
- `abstract`
  Kennzeichnung abstrakter Methoden, Klassen oder Schnittstellen

Übersicht Modifikatoren (2)

- `volatile`
  Datenfeld kann von extern modifiziert werden (z.B. bei Multi-Threading; später)

- `synchronized`
  Wechselseitiger Ausschluss bei Methoden und Blöcken (Multi-Threading; später)

- `transient`
  Datenfeld ist nicht serialisierbar (nicht weiter betrachtet)

- `native`
  Methode ist in einer anderen Sprache implementiert (z.B. C; nicht weiter betrachtet)

- strictfp
  Bei Klassen: Alle Methoden der Klasse sind strictfp.
  Bei Methoden: Fließkommaberechnungen müssen strikt nach IEEE 754 Norm ausgeführt werden.

Übersicht Modifikatoren (3)

	Datenfeld	Methode	Konstruktor	Klasse	Schnittstelle
abstract		■		■	■
final	■	■		■	
native		■			
private	■	■	■		
protected	■	■	■		
public	■	■	■	■	■
static	■	■		■	■
synchronized		■			
transient	■				
volatile	■				
strictfp		■	■	■	■

Mögliche Kombination

`final`: Konstante Variablen

Manchmal kann es Sinn machen, einer Konstanten einen symbolischen Namen zu geben. Beispiel: π, e.

Mit dem Modifikator `final` kann man jedes Datenfeld (keine lokale Variablen!) als konstant definieren. Ihr Wert, der bei der Deklaration angegeben werden muss, lässt sich anschließend nicht mehr ändern (ansonsten Compiler-Fehler).

Beispiel:

```
final float PI = = 3.1415;
```

Wichtig:

Bei Referenzvariablen kann man die Referenz, d.h. den Zeiger, konstant machen (zeigt immer auf das gleiche Objekt), aber es gibt keine Möglichkeit, das Objekt selbst konstant zu deklarieren.

Beispiel:

```
/* o zeigt anschließend immer auf das gleiche Objekt, aber in diesem
   Objekt (z.B. Instanz einer Klasse) lassen sich Werte (Datenfelder)
   verändern.
*/
final MeineKlasse o = new MeineKlasse();
o.variable = 7;           // ändern eines Datenfeldes
```

`final`: Konstante Methoden und Klassen

Gibt man bei einer Klasse oder Methode das Schlüsselwort `final` an, so kann diese Klasse nicht vererbt bzw. diese Methode in abgeleiteten Klassen nicht überschrieben werden, was z.B. aus Sicherheitsgründen nützlich sein kann. Ist die Klasse als final markiert, so sind alle Methode und Datenfelder automatisch final.

Beispiel:

```
class MeineKlasse {
    final String verschluesseln(String text) { ... }
}

class DeineKlasse extends MeineKlasse {
    // hier kann verschluesseln( ) nicht überschrieben werden!
}
```

Zugriffsrechte

- Keine Angabe (Default):
  Klasse und Schnittstellen: nur Klassen/Schnittstellen des gleichen Pakets
  Methoden und Datenfelder: Zugriff aus allen Klassen des gleichen Pakets

- `public`
  Klassen und Schnittstellen: überall
  Datenfelder und Methoden: Zugriff von überall

- `protected` (nur Datenfelder und Methoden)
  Zugriff aus gleichem Paket und zusätzlich aus Unterklassen in anderen Paketen auf die ererbten Elemente (Datenfelder und Methoden)

- `private` (nur Datenfelder und Methoden)
  Zugriff nur innerhalb der Klasse

Auf alle Datenfelder und Methoden kann innerhalb einer Klasse zugegriffen werden.
Datenfelder und Methoden einer Schnittstelle sind immer implizit `public`.

Übersicht Zugriffsrechte auf Datenelemente / Methoden

Hat Zugriff auf Datenelemente in	Default	`private`	`protected`	`public`
Klasse selbst	ja	ja	ja	ja
Subklasse, gleiches Paket	ja	nein	ja	ja
Andere Klasse, gleiches Paket	ja	nein	ja	ja
Subklasse, anderes Paket	nein	nein	ja	ja
Andere Klasse, anderes Paket	nein	nein	nein	ja

Beispiel

```
package paket1;
class K1 {          // K1 muss nicht public sein, weil K2 im gleichen Paket
              int default_var = 1;
    private   int private_var = 2;
    protected int protected_var = 3;
    public    int public_var = 4;
}
```

```
package paket1;
class K2 {
    void methode() {
        K1 o = new K1();
        int i = o.default_var + o.private_var + o.protected_var + o.public_var;
    }
}
```

Zugriff erlaubt

Zugriff nicht erlaubt

Beispiel

```
package paket1;
public class K1 { // K1 muss hier public sein!
              int default_var = 1;
    private   int private_var = 2;
    protected int protected_var = 3;
    public    int public_var = 4;
}
```

Zugriff erlaubt
Zugriff nicht erlaubt

```
package paket2;
import paket1.K1;
class K3 {
    void methode() {
        K1 o = new K1();
        int i = o.default_var + o.private_var + o.protected_var + o.public_var;
}}
```

```
package paket2;
import paket1.K1;
class K4 extends K1 {
    void methode() {
        K1 o = new K1();
        int i = o.default_var + o.private_var + o.protected_var + o.public_var;
        int j = super.default_var + super.private_var + super.protected_var
                + super.public_var;        // Zugriff auf ererbten Teil!
}}
```

abstract

Eine Methode, Klasse oder Schnittstelle kann als `abstract` markiert werden, was bedeutet, dass diese Methode, Klasse, Schnittstelle unvollständig ist. Unvollständige Klassen können nicht instanziiert werden, sie können nur durch Unterklassen erweitert werden (extends ...). Eine Unterklasse kann dann eine abstrakte Methode implementieren.

Eine Klasse besitzt abstrakte Methoden (und muss dann selber als abstract deklariert werden), wenn eine der drei Bedingungen gilt:

- In ihr ist eine abstrakte Methode deklariert.
- Sie erbt eine abstrakte Methode aus ihrer direkten Oberklasse.
- Eine unmittelbare Oberschnittstelle der Klasse deklariert oder erbt eine Methode (wg. Schnittstelle dann abstract), und die Klasse deklariert oder erbt keine Methode, die diese Methode implementiert.

Abstrakte Methode (und Klassen) sind sinnvoll, um Forderungen an eine konkrete Implementierung zu stellen:

"Wenn du die Klasse Punkt nutzen willst, musst du eine Methode action() zur Verfügung stellen, die diese vorgegebene Signatur hat!"

Beispiel

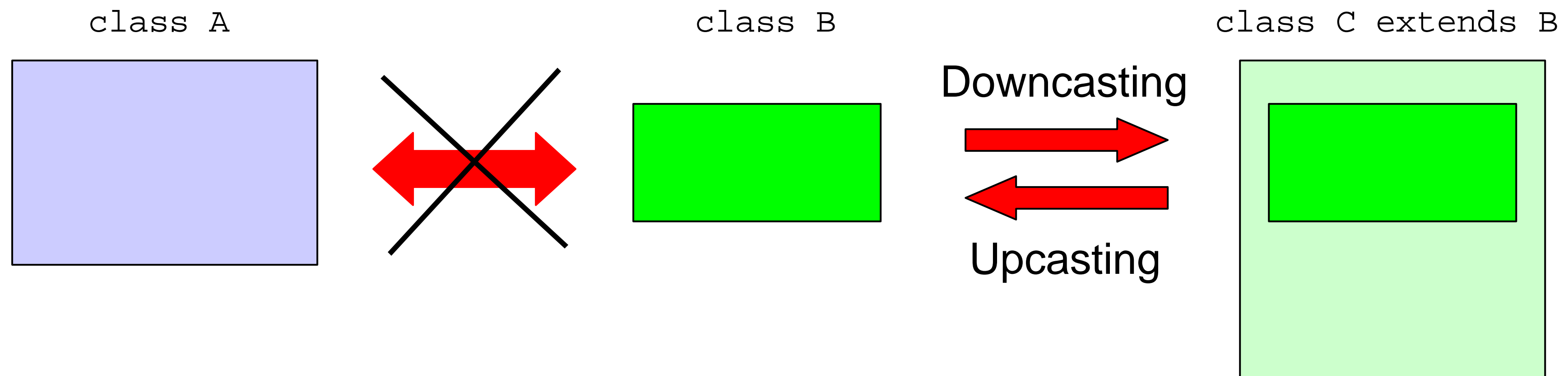
```
/* Hintergedanke bei der Klasse Punkt:
   Ich biete den Rahmen für eine Klasse Punkt. Wer mich benutzen will, muss
   allerdings eine Methode action() zur Verfügung stellen, weil ich nicht
   weiß, man beim Zeichnen noch zusätzlich gemacht werden soll (action).
*/
abstract class Punkt {    // kann wg. Abstract nicht instanziiert werden
    void zeichne() {
        ...
        action();
    }

    // abstrakte Methode, muss von Unterklasse implementiert werden.
    abstract void action();
}


// Die Klasse muss als abstract deklariert werden, weil sie die abstrakte
// Klasse Punkt erweitert, aber nicht action implementiert.
abstract class BunterPunkt extends Punkt {
    int x;
}

// Hier soll die Klasse Punkt genutzt werden.
class MeinPunkt extends Punkt {
    void action() { ... }         // hier ist die Implementierung
}
```

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- **Typumwandlungen bei Referenztypen** (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- Methodenbindung (2)

Typumwandlungen bei Referenztypen

Genau wie es bei einfachen Typen Typumwandlungen gibt, gibt es auch bei Referenztypen Umwandlungen von einem Referenztypen in einen anderen Referenztypen, die wie bei den einfachen Typen entweder implizit oder explizit (cast) stattfinden können. Nicht alle Umwandlungen sind erlaubt (und würden Sinn machen).



Upcasting bei Referenztypen

In Java ist es immer erlaubt, ein Objekt einer Unterklasse U in eine Objekt einer Oberklasse O zu verwandeln (Upcast, erweiternde Umwandlung). Dies ist einfach einzusehen, wenn man bedenkt, dass eine Instanz von U alle Komponenten und Methoden von O enthält (eventuell jedoch überschrieben). Ein Upcast kann sogar implizit stattfinden. Überall, wo ein Objekt einer Klasse verlangt ist, kann auch ein Objekt einer Unterklasse stehen.

Beispiel:

```
class Oberklasse {
    static void methode(Oberklasse o) { ... }       // erwartet Typ Oberklasse
}

class Unterklasse extends Oberklasse {
}

class Test {
    public static void main(String[] args) {
        Unterklasse u = new Unterklasse();
        Oberklasse.methode(u);     // übergeben wird u mit Typ Unterklasse
                                   // implizite Umwandlung nach Typ Oberklasse
        Oberklasse o = u;          // ebenfalls implizite Umwandlung
    }
}
```

Upcasting bei Referenztypen

- Von Klassentyp S nach Klassentyp T, wenn S Unterklasse von T ist.
- Von Klassentyp S nach Schnittstellentyp K, wenn S K implementiert.
- Von Nulltyp in jeden Klassen-, Schnittstellen- oder Feldtypen.
- Von Schnittstellentyp J in Schnittstellentyp K, wenn J eine Unterschnittstelle von K ist.
- Von Schnittstellentyp oder Feldtyp zu Object
- Von Feldtyp zu Cloneable (später)
- Von Feldtyp SC[ ] nach Feldtyp TC [ ], wenn SC und TC Referenztypen sind und es einen Upcast von SC nach TC gibt.

Upcasting von Referenztypen ist immer möglich. Das resultierende Objekt verhält sich wie ein Objekt der Oberklasse.

Beispiele

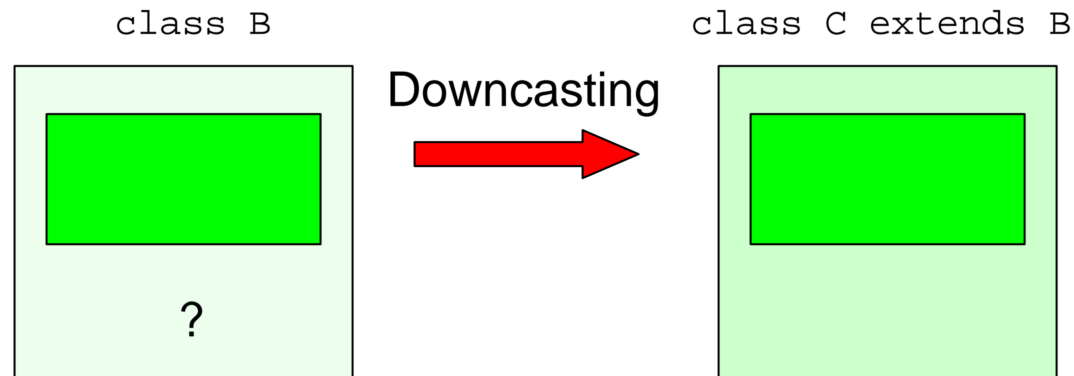
```
class OKlasse {
    int x = 31415;
}

class UKlasse extends OKlasse {
    int x = 4711;
}

class Test {
    public static void main(String[] args) {

                                            // alle Umwandlungen implizit
        UKlasse unter = new UKlasse();
        OKlasse ober = unter;               // Unter- nach Oberklasse

        unter = null;                       // Nulltyp in jeden Referenztypen

        UKlasse[] feld_u = new UKlasse[10];
        OKlasse[] feld_o = feld_u;          // Feld nach kompatibles Feld

    }
}
```

Beispiel

```
class KFZ {
    int ladevermoegen = 0;
    public String bezeichnung() { return "KFZ"; }
    public String toString() {return bezeichnung() + ", Laden=" + ladevermoegen;}
}

class LKW extends KFZ {
    int ladevermoegen = 36000;
    public String bezeichnung() { return "LKW"; }
}

class Kipper extends LKW {
    int ladevermoegen = 16000;
    public String bezeichnung() { return "Kipper"; }
}

public class Test {
    public static void main(String[] args) {
        Kipper kipper = new Kipper();
        System.out.println(kipper.toString());              // Kipper, Laden=0
        System.out.println(kipper.ladevermoegen);           // Upcast, 16000
        System.out.println(((LKW)kipper).ladevermoegen);    // Upcast, 36000
        System.out.println(((KFZ)kipper).ladevermoegen);    // Upcast, 0
    }
}
```

Downcasting bei Referenztypen

Bei Downcasts (einengende Umwandlungen) von einem Typ O zu einem Typ U ist es komplizierter. Hier muss zur Laufzeit festgestellt werden, ob der Wert eine Instanz vom gewünschten Typ U ist und nur zur Zeit der Teil "sichtbar" ist, der zu O gehört. Ist dies nicht der Fall, ist die Umwandlung nicht erlaubt.




```
class O { ... }
class U1 extends O { ... }
class U2 extends O { ... }
class Test {
    public static void main(String[] args) {
        O o = new U1();          // Downcast
        U1 u1 = (U1) o;          // Upcast kein Problem
        U2 u2 = (U2) o;          // Upcast erzeugt zur Laufzeit(!) Fehler
    }
}
```

Downcasting bei Referenztypen

- Von Klassentyp S nach Klassentyp T, wenn S Oberklasse von T ist.
- Von Klassentyp S nach Schnittstellentyp K, wenn S nicht final und K nicht implementiert.
- Von `Object` in einen Feldtypen oder Schnittstellentypen.
- Von Schnittstellentyp J in Klassentyp T, wenn T nicht final ist.
- Von Schnittstellentyp J in finalen Klassentyp K, wenn T J implementiert.
- Von Schnittstellentyp J in Schnittstellentyp K, falls J keine Unterschnittstelle von K ist und es keine Methode in beiden gibt mit gleicher Signatur aber unterschiedlichem Rückgabetyp.
- Von Feldtyp SC[ ] nach Feldtyp TC [ ], wenn SC und TC Referenztypen sind und es einen Downcast von SC nach TC gibt.

Downcasting von Referenztypen ist nur mit einer expliziter cast-Operation möglich und muss zur Laufzeit auf Korrektheit überprüft werden (löst im Fehlerfall eine `ClassCastException` aus).

Beispiel

```
class KFZ {
    int ladevermoegen = 0;
    public String bezeichnung() { return "KFZ"; }
    public String toString() {return bezeichnung() + ", Laden=" + ladevermoegen;}
}

class LKW extends KFZ {
    int ladevermoegen = 36000;
    public String bezeichnung() { return "LKW"; }
}

class Kipper extends LKW {
    int ladevermoegen = 16000;
    public String bezeichnung() { return "Kipper"; }
}

public class Test {
    public static void main(String[] args) {
        KFZ kfz = new Kipper();                                  // Upcast
        System.out.println(kfz.toString());                      // Kipper, Laden=0
        System.out.println(kfz.ladevermoegen);                   // Downcast, 0
        System.out.println(((LKW)kfz).ladevermoegen);            // Downcast, 36000
        System.out.println(((Kipper)kfz).ladevermoegen);         // Downcast, 16000
    }
}
```

Übersicht Cast-Operationen bei Referenztypen

```
class A { ... }
class B extends A { ... }
class C extends B { ... }
```

Zulässige implizite Upcasts:

```
B b = new B();
A a = b;

C c = new C();
B b = c;
A a = c;
```

Zulässige explizite Downcasts:

```
A a = new B();
B b = (B) a;

A a = new C();
B b = (B) a;
C c = (C) a;
```

Implite Upcasts implizieren auch die Möglichkeit für explizite Upcasts.
(explizit = nur mit expliziter cast-Operation möglich)

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- **Zugriff auf verdeckte Variablen**
- Methodenbindung (2)

Zugriff auf verdeckte Instanzenvariablen

```
class Oberklasse {
    int x = 4711;
}

class Unterklasse extends Oberklasse {
    int x = 31415;

    // Konstruktor
    Unterklasse() {
        // 2 Möglichkeitem zum Zugriff auf x der Unterklasse
        System.out.println("x Unterklasse: " + x);
        System.out.println("x Unterklasse: " + this.x);

        // 2 Möglichkeiten zum Zugriff auf x der Oberklasse
        System.out.println("x Oberklasse: " + super.x);
        System.out.println("x Oberklasse: " + ((Oberklasse)this).x);
    }
}

class Test {
    // Aufruf des Konstruktors
    Unterklasse u = new Unterklasse();
}
```

Zugriff auf verdeckte Instanzenvariablen

```
class Oberklasse                            { int x = 4711;   int y = 12;}
class Mittelklasse extends Oberklasse       { int x = 2714;              }
class Unterklasse extends Mittelklasse { int x = 31415;  int y = 20;
    // Konstruktor
    Unterklasse() {
        // 2 Möglichkeitem zum Zugriff auf x der Unterklasse
        System.out.println("x Unterklasse: " + x);
        System.out.println("x Unterklasse: " + this.x);

        // 2 Möglichkeiten zum Zugriff auf x der Mittelklasse
        System.out.println("x Oberklasse: " + super.x);
        System.out.println("x Oberklasse: " + ((Mittelklasse)this).x);

        // 1 Möglichkeit zum Zugriff auf x der Oberklasse
        System.out.println("x Oberklasse: " + ((Oberklasse)this).x);

        // 3 Möglichkeiten zum Zugriff auf y von Oberklasse
        // weil Mittelklasse kein y besitzt, wird nächstes in Hierarchie genommen
        System.out.println("y Oberklasse: " + ((Oberklasse)this).y);
        System.out.println("y Oberklasse: " + super.y);
        System.out.println("y Oberklasse: " + ((Mittelklasse)this).y);
    }
}

class Test { Unterklasse u = new Unterklasse(); }
```

Zugriff auf verdeckte Klassenvariablen

```
class Oberklasse                          { static int x = 4711;}
class Mittelklasse extends Oberklasse     { static int x = 2714;              }
class Unterklasse extends Mittelklasse { static int x = 31415;
    // Konstruktor
    Unterklasse() {
        // 3 Möglichkeitem zum Zugriff auf x der Unterklasse
        System.out.println("x Unterklasse: " + x);
        System.out.println("x Unterklasse: " + this.x);
        System.out.println("x Unterklasse: " + Unterklasse.x);

        // 3 Möglichkeiten zum Zugriff auf x der Mittelklasse
        System.out.println("x Oberklasse: " + super.x);
        System.out.println("x Oberklasse: " + ((Mittelklasse)this).x);
        System.out.println("x Oberklasse: " + Mittelklasse.x);

        // 2 Möglichkeiten zum Zugriff auf x der Oberklasse
        System.out.println("x Oberklasse: " + ((Oberklasse)this).x);
        System.out.println("x Oberklasse: " + Oberklasse.x);

        // analog bei fehlendem y in Mittelklasse
    }
}

class Test { Unterklasse u = new Unterklasse(); }
```

Wo sind wir im Detail bei KPS?

- Aufbau von Klassen (Datenfelder, Methoden, Konstruktoren, Klassen- und Instanzenvariablen/methoden)
- Vererbung (Überschreiben von Methoden, Methodenbindung (1))
- Wrapperklassen
- Geschachtelte Klassen
- Schnittstellen
- Pakete
- Gültigkeitsbereich, Sichtbarkeit, Lebensdauer
- Modifikatoren (u.a. Zugriffsrechte)
- Typumwandlungen bei Referenztypen (Upcast, Downcast)
- Zugriff auf verdeckte Variablen
- **Methodenbindung (2)**

Methodenbindung wieder betrachtet

```
class KFZ              { public String bezeichnung() { return "KFZ"; } }
class LKW extends KFZ  { public String bezeichnung() { return "LKW"; } }

public class Test {
    public static void main(String[] args) {
        KFZ kfz1 = new KFZ();
        System.out.println(kfz1.bezeichnung());
        KFZ kfz2 = new LKW();
        System.out.println(kfz2.bezeichnung());
    }
}
```

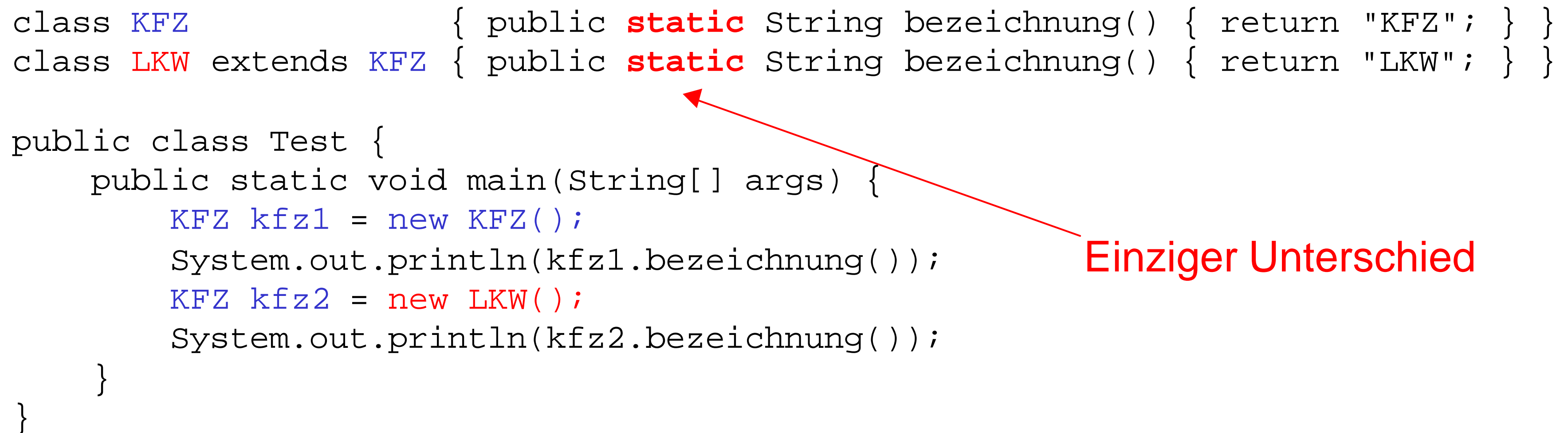
Ausgabe:

```
KFZ
LKW
```

Instanzenmethoden werden dynamisch gebunden (zur Laufzeit bestimmt).

Motivation: Instanzenmethoden sind an den Vererbungsprozess gekoppelt. Man möchte, dass überall wo ein Objekt der Oberklasse erlaubt ist, auch ein Objekt der Unterklasse stehen kann (gleich mehr).

Methodenbindung wieder betrachtet



```
class KFZ              { public static String bezeichnung() { return "KFZ"; } }
class LKW extends KFZ { public static String bezeichnung() { return "LKW"; } }

public class Test {
    public static void main(String[] args) {
        KFZ kfz1 = new KFZ();
        System.out.println(kfz1.bezeichnung());
        KFZ kfz2 = new LKW();
        System.out.println(kfz2.bezeichnung());
    }
}
```

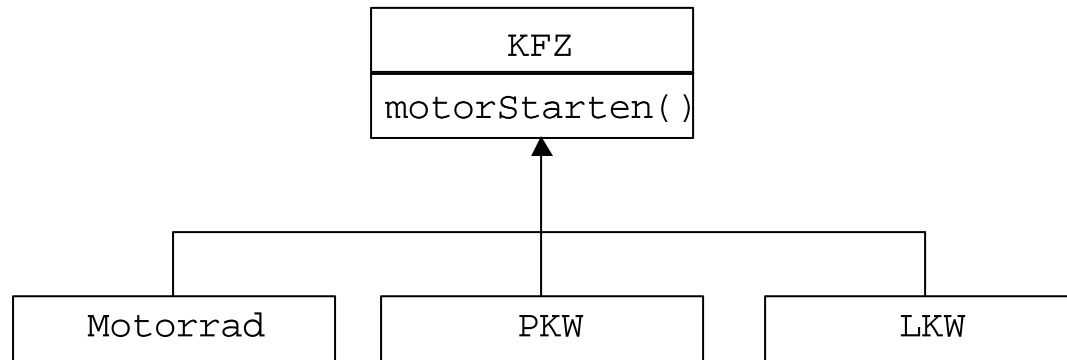

Ausgabe:

```
KFZ
KFZ
```

Klassenmethoden (Modifikator `static`) und Methoden mit den Modifikatoren `private` und `final` werden statisch gebunden (zur Übersetzungszeit bestimmt). Der Compiler schaut sich den Typ der Referenz an und bestimmt darüber die Methode.

Motivation: Klassenmethoden gehören zu genau einer (dieser) Klasse.

Vererbung und Typumwandlung sinnvoll einsetzen



Eine wesentliche Idee hinter der Objektorientierten Programmierung ist die, dass man sich für eine Basisklasse ein mal Gedanken macht und implementiert und anschließend diese Klasse in weiteren Klassen wiederverwendet. In den abgeleiteten Klassen muss man nur noch den Unterschied zur Basisklasse implementieren. Der Code der Basisklasse bleibt unverändert.

Über Upcasts kann man dann sinnvoll die Polymorphie ausnutzen:

- Erzeugen von konkreten Objekten der abgeleiteten Unterklassen (Die Objekte der abgeleiteten Klassen möchte man ja haben!)
- Upcast zu einem Objekt der Oberklasse
- Verwenden der umgewandelten Objekte der Oberklasse

Beispiel

```
class KFZ {
    public String bezeichnung() { return "KFZ"; }
    public String toString() { return bezeichnung(); }
}

class Motorrad extends KFZ {
    public String bezeichnung() { return "Motorrad"; }
}

class PKW extends KFZ {
    public String bezeichnung() { return "PKW"; }
}

class LKW extends KFZ {
    public String bezeichnung() { return "LKW"; }
}

public class Test {
    public static void main(String[] args) {
        KFZ[] kfz = new KFZ[3];
        kfz[0] = new Motorrad();
        kfz[1] = new PKW();
        kfz[2] = new LKW();
        for(int i=0; i<kfz.length; i++)
            System.out.println(kfz[i].toString());
    }
}
```

bezeichnung() ist objektspezifisch.

toString() ist allen Objekten bekannt.

Alle Objekte werden einheitlich behandelt.

instanceof-Operator

Dieser Operator, der einen boolschen Wert liefert, wird eingesetzt um zur Laufzeit festzustellen, ob eine Referenz einer Oberklasse vom Typ einer bestimmten Unterklasse ist, um z.B. auf diese Unterklasse zu casten. Die Referenz könnte ja einen beliebigen Typ aller Unterklassen haben und bei falschem Casting eine `ClassCastException` auslösen.

```
class Oberklasse { ... }
class Unterklasse1 extends Oberklasse { ... }
class Unterklasse2 extends Oberklasse { ... }

class Test {
   ...
   Oberklasse o = new Unterklasse1();      // Upcast
   methode(o);
   ...
   void methode(Oberklasse o) {
       if(o instanceof Unterklasse1) Unterklasse1 u1 = (Unterklasse1)o; ...
}
```