



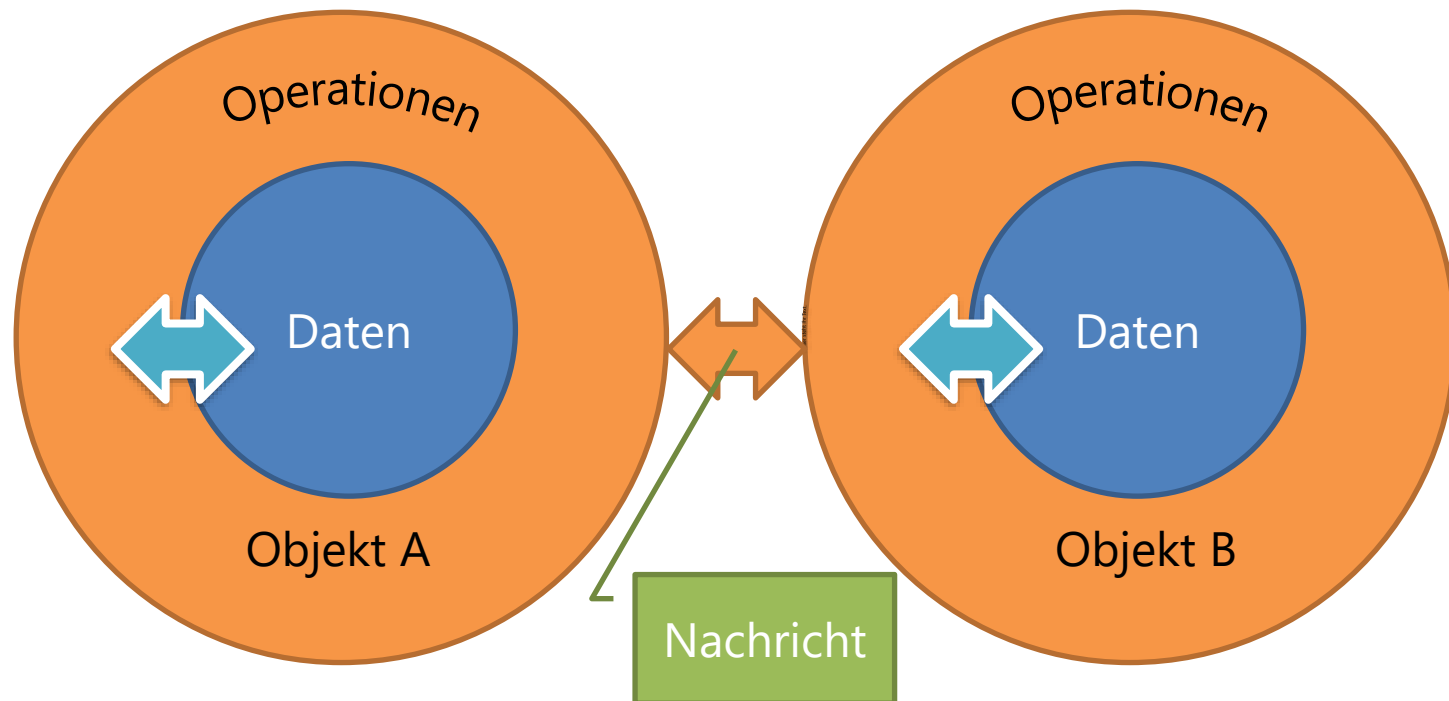
5. Objektorientierung I – Klassen und Objekte

ARINKO[®]

- Die nächsten 6 Folien wiederholen nochmals die wichtigsten Grundbegriffe:
 - Objekt
 - Nachricht/Operation/Methode
 - Attribut
 - Klasse
 - Typ

- Objekte repräsentieren Dinge einer fachlichen „Domain“ (dt. Anwendungsbereich), die Informationen in Form von Daten halten und deren Interaktion/Verhalten von Interesse ist.
- Beispiele von Objekten:
 - „greifbare“ (konkrete) Gegenstände wie Vertrag, Smartphone, Auto
 - „nicht greifbare“ (abstrakte) Dinge wie Konzern, Wahl, SMS
- Man erkennt, dass Objekte hauptsächlich durch Hauptworte charakterisiert werden.
- Objekte:
 - kapseln Daten und Operationen
 - haben einen Zustand und können diesen verändern
 - haben „Verhalten“
 - senden und empfangen Nachrichten
 - haben eine Identität

- Interaktion zwischen Objekten erfolgen mit Nachrichten.
- Ein Objekt A sendet eine Nachricht an Objekt B.
- Objekt B verarbeitet die Nachricht, indem es die passende Operation ausführt und ggf. eine Antwort zurückliefert.
- Objekt A wartet so lange, bis es die Antwort von Objekt B erhalten hat bzw. bis die Operation von Objekt B fertig ist.



Attribute (engl. „attributes“ oder gerne auch „fields“):

- Informationen, Daten eines Objekts
- können unveränderlich (konstant) oder variabel sein

Methoden (engl. „methods“):

- Operationen eines Objekts
- können den Objektzustand verändern oder in Abhängigkeit von diesem agieren
- können Nachrichten/Daten (= Parameter) von ihrem Aufrufer empfangen
- können Antworten (= Rückgabewerte) an diesen zurückliefern

- Gleichartige Objekte werden identifiziert, deren gemeinsame Attribute und Methoden werden exemplarisch in einer „Klasse“ beschrieben
- Eine Klasse enthält also die Definitionen von Attributen und Methoden für zu erstellende Objekte.
- Objekte werden dann aus den passenden Klassen erzeugt. Eine Klasse ist somit eine Art „Bauvorschrift“ für Objekte.
- Die Java-Programmentwicklung besteht nahezu ausschließlich aus der Entwicklung von Klassen!

- Objekte entstehen aus einer Klasse durch „Instanziierung“
 - Objekte können daher auch alternativ als „Instanz einer Klasse“ bezeichnet werden (im Deutschen ist der Begriff Instanz leider anderweitig [juristisch] belegt, was zu Missverständnissen führen kann)
 - Objekte können nur erzeugt werden, wenn es eine passende Klasse gibt.
 - Umkehrschluss: jedes Objekt ist Instanz einer Klasse.
-
- Klassen entstehen zur Entwicklungszeit (Programmierung)
 - Objekte entstehen zur Laufzeit
-
- Alle Objekte einer Klasse kennen dieselben Methoden
 - Jedes Objekt einer Klasse unterscheidet sich in seinem Zustand von anderen Objekten seiner Klasse (unterschiedliche Attributsbelegung)

- Jede Klasse definiert einen eigenen „Typ“.
- Objekte einer Klasse haben alle denselben Typ, den der Klasse.
- Der Java-Compiler und das Laufzeitsystem überprüfen Typzugehörigkeiten und verweigern bei Unstimmigkeiten die Compilierung bzw. brechen mit einem Laufzeitfehler ab.
- Java ist (bis Java 7) eine streng-typisierte Programmiersprache.
- Ab Java 8 wird in den sog. Lambda-Ausdrücken eine relativ große Freiheit gewährt.

Zusammengefasst kann man also sagen:

- Eine Klasse beschreibt schablonenhaft gleichartige Objekte durch Angabe von
 - Attributen (Daten), deren Belegung den Zustand der Objekte darstellen
 - Methoden (Operationen), die auf die Objekte angewendet werden können und die mit den jeweiligen Attributen arbeiten dürfen
 - Konstruktoren, die für die Erzeugung von Objekten gebraucht werden
- Objekte werden aus Klassen erzeugt durch Instanziierung

Klassenkopf

```
public class Konto  
{
```

```
    private String kontonummer;  
    private double zinssatz;  
    private int kontostand;
```

Attribute

```
    public Konto(String n, double z, int k)  
    {  
        kontonummer = n;  
        zinssatz = z;  
        kontostand = k;  
    }
```

Konstruktor

```
    public String getKontonummer()  
    {  
        return kontonummer;  
    }
```

Klassenrumpf

Methoden

...hier steht noch mehr... ausgelassen

```
    public void setKontostand(int stand)  
    {  
        kontostand = stand;  
    }
```

```
}
```

- Jedes Objekt einer Klasse hat dieselben Attribute wie ein anderes Objekt derselben Klasse.
- Jedes Objekt einer Klasse kann seine Attribute anders belegen.
- Diese unterschiedliche Belegung stellt den **Zustand** eines Objektes dar.
- Attribute sind Variablen (die durchaus aber "final", d.h. nur einmalig belegbar sein können)
- Datenkapselung: nur Methoden eines Objektes sollten den Zustand des Objektes ändern können.
- Auf Attribute sollte von außen nicht zugegriffen werden dürfen.
- → gewöhnen Sie sich an, automatisch `private` vor Attribute zu schreiben.

```
public class Konto
{
    private String kontonummer;
    private double zinssatz;
    private int kontostand;
    ...
}
```



Attribute

- Konstruktoren werden zur Erzeugung von Objekten benötigt.
- Ihre Aufgabe ist es, ein fachlich "sauberes", initialisiertes Objekt zu hinterlassen, bei dem alle Attribute einen definierten Zustand erhalten haben.
- **Achtung:** man kann in Java durchaus so codieren, dass Attribute mal belegt, mal vergessen werden – achten Sie darauf, dass Ihre Konstruktoren immer alle Attribute bedenken.
- Konstruktoren sind spezielle Methoden mit folgenden spezifischen Eigenschaften:
 - sie haben als Namen den Namen der Klasse (und beginnen somit mit einem Großbuchstaben im Gegensatz zu Methodennamen)
 - sie haben keinen Rückgabetyt (auch nicht `void`)
 - sie werden ausschließlich zur Objekterzeugung aufgerufen, und das implizit

```
public class Konto
{
    private String kontonummer;
    private double zinssatz;
    private int kontostand;

    public Konto(String n, double z, int k)
    {
        kontonummer = n;
        zinssatz = z;
        kontostand = k;
    }
}
```



Konstruktor

...hier steht noch mehr... ausgelassen

}

```
public class Konto
{
    private String kontonummer;
    private double zinssatz;
    private int kontostand;

    public Konto(String n, double z, int k)
    {
        kontonummer = n;
        zinssatz = z;
        kontostand = k;
    }

    public Konto(String n, double z)
    {
        kontonummer = n;
        zinssatz = z;
    }

    ...hier steht noch mehr... ausgelassen

}
```



Konstruktoren

Was ist hier nicht gut gelöst?

- Damit man keine Probleme mit unvollständigen Konstruktoren oder dupliziertem Code hat, kann man aus einem Konstruktor einen anderen Konstruktor aufrufen.
- Hier kann man Default-Werte mitgeben.
- Der Aufruf eines anderen Konstruktors
 - kann nur aus einem Konstruktor erfolgen
 - und muss die erste Anweisung sein

```
public class Konto
{
    public Konto(String n, double z, int k)
    {
        kontonummer = n;
        zinssatz = z;
        kontostand = k;
    }

    public Konto(String n, double z)
    {
        this(n, z, 0);
    }
    ...
}
```

Aufruf des oberen Konstruktors

- Wenn man in einer Klasse keinen Konstruktor angibt, dann wird vom Compiler ein Konstruktor eingefügt.
- Dieser Konstruktor ist parameterlos.
- Man spricht dann vom sog. "Default-Konstruktor".
- Hinweis: sehr häufig wird (auch in der Literatur) der irrige Umkehrschluss verbreitet, ein parameterloser Konstruktor wäre ein Default-Konstruktor.

Korrekt ist: Jeder Default-Konstruktor ist parameterlos, nicht jeder parameterlose Konstruktor ist aber ein Default-Konstruktor!

- Diese Konstruktion sorgt dafür, dass in jeder Klasse mindestens ein Konstruktor vorhanden ist. Technisch Gebrauch macht davon der Ableitungsmechanismus (späteres Kapitel).

- Attributsdeklarationen können Initialisierer enthalten.
- Diese werden nach der Objekterzeugung, aber noch vor dem Konstruktoraufruf angewendet.

```
private final String kontonummer;  
private double      zinssatz = 0.01;  
private int         kontostand = 0;  
  
public Konto(String n, double z, int k)  
{  
    kontonummer = n;  
    zinssatz = z;  
    kontostand = k;  
}
```

Belegung des final Attributs

- Attribute können mit dem Modifizierer `final` ausgezeichnet werden. Sie werden dadurch unveränderlich.
- `final` Attribute haben oft einen Initialisierer.
- Wenn `final` Attribute keinen Initialisierer haben, müssen sie spätestens im Konstruktor mit einem Wert belegt werden.

- Code, der in jedem Fall bei der Konstruktion eines Objektes ausgeführt werden soll, kann man in einen Initialisierungsblock legen.
- Initialisierungsblöcke werden nach den Initialisierern für Attribute, aber unmittelbar vor dem Konstruktor ausgeführt.
- Falls es mehrere gibt, werden sie von oben nach unten ausgeführt.

```
public class KlasseMitInitialisierungsblock
{
    /*
     * Initialisierungsblock
     */
    {
        // hier Code
    }
}
```

Nochmals zusammengefasst die Abläufe bei der Konstruktion eines Objektes:

1. der nötige Speicherplatz wird reserviert
 2. die Attribute werden mit Standardwerten (Nullwerte bzw. false) belegt
 3. der Konstruktor wird aufgerufen
 4. ist `this()` die erste Anweisung, wird der spezifizierte andere Konstruktor aufgerufen
 5. BEVOR die erste "richtige" Codezeile des Konstruktors abgearbeitet wird, werden für Attribute, die einen Initialisierer haben, dessen Wert zugewiesen.
 6. BEVOR dann die erste Zeile des Konstruktors abgearbeitet wird, werden die Initialisierungsblöcke der Reihe nach ausgeführt.
 7. dann wird der Code des Konstruktors ausgeführt.
- Der Konstruktor hat also sozusagen "das letzte Wort".
 - Hat eine Klasse mehrere Konstruktoren, dann entscheidet die Parameterliste, welcher der überladenen Konstruktoren aufgerufen wird.

- Da es für die Erzeugung von Objekten viele Baustellen (Initialisierer, Initialisierungsblöcke, mehrere Konstruktoren) gibt, empfiehlt sich folgende Vorgehensweise für guten Code bei der Objektkonstruktion:
 - Benutzen Sie Initialisierer selten und am besten nur für `final` Attribute.
 - Verzichten Sie auf im Code verteilte Initialisierungsblöcke.
 - Arbeiten Sie nur im Konstruktor.
 - Gestalten Sie Ihre Konstruktoren so, dass nur ein einziger Konstruktor die Initialisierungsarbeit durchführt und die anderen Konstruktoren diesem zuarbeiten (Vermeidung von Redundanz).

```
public Konto(String kontonummer, double zinssatz, int kontostand)
{
    this.kontonummer = kontonummer;
    this.zinssatz = zinssatz;
    this.kontostand = kontostand;
}

public Konto(String kontonummer, double zinssatz)
{
    this(kontonummer, zinssatz, 0);
}
```

- Die "Entsorgung" von Objekten erfolgt im Java automatisch.
- Verantwortlich dafür ist der sog. "Garbage Collector".
- Objekte *können* zerstört werden, wenn sie nicht mehr referenziert werden.
- Es gibt eine im Format festgeschriebene Methode `finalize()`, die von Klassen implementiert werden könnte, wenn man kurz vor der Destruktion noch Aufräumarbeiten durchführen will.
- Da man aber nicht sicher sein kann, ob und wann der Garbage Collector ausgeführt wird, ist es best practice, diese nicht zu nutzen.

- In Methoden finden sich Anweisungen für abgeschlossene Funktionalitäten. Beispiele: einen Namen ändern, eine Abrechnung durchführen, eine Bestellposition hinzufügen...
 - Methoden arbeiten auf dem jeweiligen Objekt und können auf den dem Objekt zugeordneten Satz von Attributen zugreifen.
 - Methoden können Parameter besitzen oder sie sind parameterlos.
 - Methoden können einen Rückgabewert liefern oder sie tun das nicht.
 - Sind für eine Methode Parameter spezifiziert, dann muss der Aufrufcode Werte für die Parameter angeben.
-
- Man sagt: Methoden bestimmen das Verhalten der Objekte (im Gegensatz zu Attributen, die den Zustand des Objektes bestimmen).
 - Objekte, die aus der gleichen Klasse gebildet wurden, zeigen gleiches Verhalten.

- Hier drei schon bekannte Beispiele für Methoden:

```
// ohne Parameter, aber mit Rückgabewert
public int getKontostand()
{
    return kontostand;
}

public double getZinssatz()
{
    return zinssatz;
}

// ohne Rückgabewert, aber mit Parameter
public void setKontostand(int stand)
{
    kontostand = stand;
}
```

- Eine Methode des selben Objekts kann einfach durch Nennung ihres Namens aufgerufen werden:

```
// ohne Rückgabewert und ohne Parameter
// Diese Methode arbeitet also allein auf dem Objektzustand
// und verändert ihn
public void verarbeiteZinsen()
{
    // Beachten:
    // das ist kein guter Code, weil Verluste auftreten
    double neuerStand = kontostand * (1 + zinssatz);
    setKontostand((int) neuerStand);
}
```


- Bei der Deklaration einer Methode oder eines Konstruktors wird eine Parameterliste definiert.
- Beim konkreten Aufruf des jeweiligen Elements muss für jeden Parameter ein zum deklarierten Typ passender Wert angegeben werden.

```
// Cast auf int war nötig, weil so deklariert  
setKontostand((int) neuerStand);
```

```
Konto konto1 = new Konto("DE68-0815-4711", 0.01, 0);  
Konto konto2 = new Konto("DE68-4711-0815", 0.005);
```

- Es gibt verschiedene Arten der Parameterübergabe (nette Teildisziplin des Compilerbaus), u.a. call-by-value, call-by-reference, call-by-name
- Kurzfassung:

call-by-value:

es wird eine Kopie des Wertes übergeben, dieser Wert verhält sich im aufgerufenen Code somit wie eine lokale Variable.

call-by-reference:

es wird ein Zeiger auf den Wert übergeben, der aufgerufene Code kann über diesen Zeiger den Aufrufer manipulieren.

call-by-name:

kurz gesagt eine Art Makro, das quasi textuell im aufgerufenen Code den übergebenen Parameter namentlich einsetzt.

- **Java verwendet ausschließlich call-by-value.**
- Auch wenn viele Internetseiten und einige Bücher hier anderes behaupten.
- Selbst Referenzen auf Objekte werden als Wertkopien übergeben. Sie können die Objekte mit Hilfe der Referenz ändern (das ist so gewollt), aber niemals die Referenz selbst.

- Lokale Variablen (dazu zählen in gewisser Weise auch Parameter) gelten vom Kopf des deklarierenden Elements bis zur schließenden Klammer.

```
public void geltung(int a) // ab hier gilt a
{
    int b; // ab hier b
    if (a > 0)
    {
        int c = a; // ab hier c
    }
    // hier verschwindet c
    // a und b sind noch sichtbar
}
// jetzt gibt es auch kein a und b mehr
```

- Mit return wird die Ausführung einer Methode oder eines Konstruktors beendet.
- Parameterlose Methoden/Konstrukturen müssen kein return codieren, dann endet der Code mit der schließenden Klammer und kehrt zum Aufrufer zurück.
- Deklariert man bei einer Methode einen Rückgabebetyp, dann
 - muss es (mindestens) eine return-Anweisung zwingend geben und
 - deren Argument muss zum deklarierten Rückgabebetyp kompatibel sein.

```
public int auszahlen(int betrag)
{
    if (betrag <= kontostand)
    {
        kontostand = kontostand - betrag;
        return kontostand;
    }

    return 0;
}
```

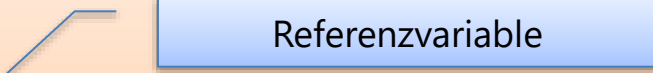
- Objekte werden zur Laufzeit (aka dynamisch) erzeugt.
- Bei der Erzeugung erhält man eine Art Zeiger auf die Speicheradresse des Objektes.
- Man spricht in Java von **Referenzen**
- Eine Referenz hat Typ und Wert:
 - Typ ist der Typ des referenzierten Objektes (der Klassentyp)
 - Wert ist die (nicht sichtbare) Speicheradresse
- Eine Referenz erhält man in den meisten Fällen
 - Direkt durch den Operator new
 - Indirekt, z.B. als Rückgabewert einer Methode
- Referenzen können in Referenzvariablen abgelegt werden.
- In Referenzvariablen dürfen nur Referenzen desselben* Typs abgelegt werden.

```
public class KontoTest
{
    public static void main()
    {
        Konto konto1 = new Konto("DE68-0815-4711", 0.01, 0);
    }
}
```

Referenzvariable

Objekterzeugung

- Zugriff auf Elemente (Attribute/Methoden) eines Objekts erfolgt mittels des Zugriffsoperators . (Punkt).
- Voraussetzung für den Zugriff von außen ist eine hinreichende Sichtbarkeit/Zugreifbarkeit des Elements.
- Linker Operand: Referenz.
- Rechter Operand: Objektelement.



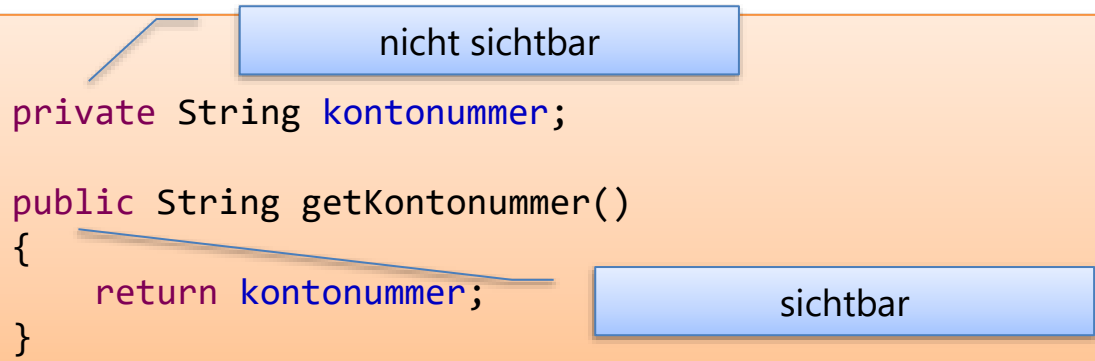
```
Konto konto1 = new Konto("DE68-0815-4711", 0.01, 0);
```

```
konto1.setKontostand(5200);
```



Zugriff auf Methode

- Elemente, die von außerhalb einer Klasse zugreifbar sein sollen, werden mit dem Modifizierer `public` versehen.
- Elemente, die nur innerhalb einer Klasse zugreifbar sein sollen, werden mit dem Modifizierer `private` versehen.
- Das Fehlen eines Modifizierers bedeutet eine andere Sichtbarkeit (später...)



```
private String kontonummer;
```

```
public String getKontonummer()  
{  
    return kontonummer;  
}
```

- In der Objektorientierung existiert immer ein "Kontext".
- Das bedeutet u.a., dass Methoden immer den Kontext eines zugehörigen Objektes haben.
- Methoden werden immer über eine Referenz (explizit oder implizit) an einem Objekt aufgerufen.
- Das Objekt, an dem die Methode aufgerufen wird, ist dann das **aktuelle Objekt**.
- Die Methode operiert auf dem aktuellen Objekt.
- Alle unqualifizierten Zugriffe auf Attribute und andere Methoden arbeiten ebenfalls auf dem selben, aktuellen Objekt.

```
public void verarbeiteZinsen()
{
    // Beachten:
    // das ist kein guter Code, weil verluste auftreten
    double neuerStand = kontostand * (1 + zinssatz);
    setKontostand((int) neuerStand);
}
```

Zugriff auf ein Attribut

Zugriff auf eine Methode

- In Konstruktoren und Methoden kann es sinnvoll sein, das aktuelle Objekt explizit zu benennen, z.B.
 - bei Namensgleichheit von Attributen und Parametern (Parameter und lokale Variablen überdecken namensgleiche Attribute)
 - um dem Leser explizit klarzumachen, woran etwas "hängt"
- In den objektorientierten Sprachen hat das aktuelle Objekt viele Namen, z.B. "me", "self" oder eben wie in Java **this**.

Namensgleiche Parameter

```
public Konto(String kontonummer, double zinssatz, int kontostand)
{
    this.kontonummer = kontonummer;
    this.zinssatz = zinssatz;
    this.kontostand = kontostand;
}

public void verarbeiteZinsen()
{
    // Beachten:
    // das ist kein guter Code, weil Verluste auftreten
    double neuerStand = this.kontostand * (1 + this.zinssatz);
    this.setKontostand((int) neuerStand);
}
```

Visualisierung des Kontexts

- Eine Klasse kann mehrere Konstruktoren besitzen oder mehrere Methoden mit dem selben Namen.
- Diese müssen sich durch die Parameterliste unterscheiden.
- Rückgabetypen würde für Konstruktoren gar nicht zutreffen und unterscheiden auch Methoden nicht.
- Das Phänomen wird als "overloading" (Überladen) bezeichnet.

```
public int auszahlen(int betrag)
{
    if (betrag <= this.kontostand)
    {
        this.kontostand = this.kontostand - betrag;
        return this.kontostand;
    }

    return 0;
}

public int auszahlen()
{
    return this.auszahlen(this.kontostand);
}
```

gleich benannt, andere Parameter

Zugriff auf die "andere" Methode,
um Redundanzen zu vermeiden



6. Objektorientierung I – UML

ARINKO[®]

- Die "Unified Modeling Language" (UML) ist eine graphische Modellierungssprache (inkl. Metamodell) für objektorientierte Softwaresysteme.
 - Als "Unified Method Language" gestartet, hatte man den Ehrgeiz, den objektorientierten Entwurf zu systematisieren, aber schnell erkannt, dass es bereits an vereinheitlichten Notationen fehlt.
 - Somit ist die UML keine *Entwicklungsmethodik*.
 - Ursprünglich von G. Booch, I. Jacobson und J. Rumbaugh, allesamt "Gurus" der Objektorientierung, nach deren Zusammentreffen bei der Fa. Rational (heute IBM) als Kompromiss ihrer bestehenden Notationen entwickelt.
 - Seit 1997 als Standard unter der Obhut der Object Management Group (OMG).
 - Das System ist sehr umfangreich und mächtig, v.a. durch 4 Metaebenen, die es erlauben, quasi eigene graphische "Dialekte" zu konstruieren.
 - Wir werden uns hier auf wichtige Basics konzentrieren.
-
- Die Anstrengungen, eine vereinheitlichte Methodik zur Softwareentwicklung zu haben, wurden dennoch innerhalb der Fa. Rational weitergetrieben und haben als Output den sog. "Rational Unified Process" (RUP)
[https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf]

- Ohne Details (in Übersichten gerne verwendet):

Person

- Mit Details in 3 separaten "compartments" (Abschnitten):

Person
-vorname : String -nachname : String
<<Konstruktor>> +Person(vorname : String, nachname : String) <<Methoden>> +gibVorname() : String +aendereNachname(neuerNachname : String) +gibNachname() : String

- Objekte werden a) ähnlich wie Klassen aber b) seltener dargestellt.
- Da durch die Kenntnis der Klasse bereits die Struktur feststeht, interessiert man sich hier ausschließlich für die Belegung der Attribute (Objektzustand).

<u>: Person</u>
vorname = "Dagobert" nachname = "Duck"

- Vereinfachte Darstellungsweise für Attribute:

```
[Sichtbarkeit] name [: Typ] [= Vorgabewert] [{Eigenschaftswert*}]
```

- Darstellungsweise für Methoden:

```
[Sichtbarkeit] name [{Parameter}] [: Rückgabetyp] [{Eigenschaftswert*}]
```

- Aufbau von Parametern (hier für Java vereinfacht):

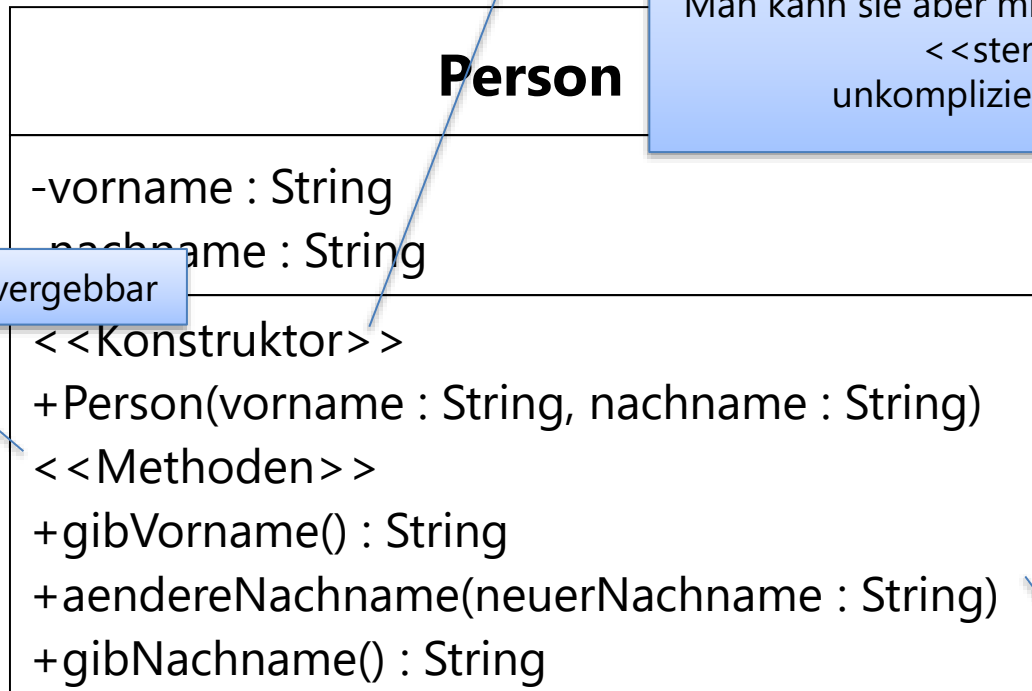
```
name : Typ [{Eigenschaftswert*}]
```

- Sichtbarkeiten (zunächst vereinfacht)

```
+ für public  
- für private
```

- Eigenschaftswerte werden selten benutzt, möglich wären u.a.

```
ordered    (Die Daten sind geordnet bzw. werden geordnet zurückgegeben)  
read-only  (das Attribut kann nur gelesen werden/die Methode verändert nichts)
```



Stereotypen sind frei vergebbar

Konstrukturen existieren nicht in UML.
Man kann sie aber mit Hilfe von Stereotypen
`<<stereotyp>>`
unkompliziert qualifizieren

UML kennt den Pseudotyp `void` nicht.
Wenn es keinen Rückgabewert gibt, lässt
man diesen einfach weg

