Objekte und Klassen

- Eigene Definition von Klassen
- Methoden
- Konstruktoren
- Instanzvariablen
- UML Notation
- Gleichheit von Objekten
- Der spezielle Wert `null`-

Definition von Klassen

Wir wollen Bankkontos verwalten.

Ein Bankkonto enthält einen Kontostand.

(außerdem noch Name, Überziehungslimit, etc. pp.)

Der Kontostand kann

- abgerufen werden
- erhöht werden durch Einzahlung
- erniedrigt werden durch Abhebung

Verwendung von Bankkonten

Wir möchten eine Klasse von Bankkonten, die man etwa so verwenden kann:

```
public class BankkontoTest {
    public static void main(String[] args) {
        Bankkonto matthiasGiro = new Bankkonto();
        Bankkonto johannasSpar = new Bankkonto();
        double zinsSatz = 1.25;

        johannasSpar.einzahlen(2000.00);
        matthiasGiro.einzahlen(30000.00);
        matthiasGiro.abheben(10000.00);
        johannasSpar.einzahlen(
            johannasSpar.getKontostand() * zinsSatz / 100.0
```

Verwendung von Bankkonten

```
System.out.println("Johannas Sparkonto: " +
                                johannasSpar.getKontostand());
        System.out.println("Matthias Girokonto: " +
                                matthiasGiro.getKontostand());
    }
}
```

Ausgabe wäre hier:

```
Johannas Sparkonto: 2025.0
Matthias Girokonto: 20000.0
```

Wie kann man so eine Klasse schreiben?

Die Klasse `Bankkonto`

Kommt in eine Datei `Bankkonto.java`.

Darin steht

```
public class Bankkonto {
    Deklaration der Daten
    Definition der Methoden
    Definition der Konstruktoren }
```

Die Einträge können in beliebiger Reihenfolge stehen.

Deklaration der Daten

Die Daten eines Objekts bestehen aus Variablen, den *Instanzvariablen*, die in der Klasse deklariert werden.

Beim Bankkonto brauchen wir nur eine Instanzvariable vom Typ `double`.

Wir schreiben also unter *Deklaration der Daten*:

```
private double kontostand;
```

Damit wird gesagt, dass jedes Objekt der Klasse `Bankkonto` sich einen Double-Wert “merken” kann.

Die Qualifikation `private` besagt, dass dieser Wert von außen nicht direkt einsehbar ist. Nur auf dem Umweg über Methodenaufrufe.

Instanzvariablen als `public`

Man kann Instanzvariablen auch `public` deklarieren. Dann kann man ihren Wert von außen abrufen. Etwa

```
Rectangle r = Rectangle(10,10,30,50);
r.height;
```

Es ist im allgemeinen schlecht, Instanzvariablen `public` zu deklarieren.

Grund: interne Repräsentation der Daten kann dann nicht mehr verändert werden.

Beispiel: Kontostand als Integer (in Cents).

Rectangles

Ein `Rectangle` besteht aus den Koordinaten der linken oberen Ecke, sowie Breite und Höhe, jeweils als `int`.

Wie sehen die Instanzvariablen der Klasse `Rectangle` aus?

Antwort

```
public int x;
public int y;
public int width;
public int height;
```

Wie gesagt, `public` sollte für Instanzvariablen eigentlich vermieden werden.

Methodendefinitionen

Die Methode `einzahlen` definieren wir durch

```
public void einzahlen(double betrag) {
    kontostand = kontostand + betrag;
}
```

Das bedeutet im einzelnen:

- Die Methode kann von außen aufgerufen werden (`public`)
- Der Aufruf erfolgt mit einem Parameter vom Typ `double`
- Der Aufruf liefert keinen Wert zurück (`void`)
- Beim Aufruf werden die Statements zwischen den `{ }` ausgeführt. Hier

  ```
  kontostand = kontostand + betrag;
  ```

- Dabei kann sowohl auf die Instanzvariablen, wie auch auf die Parameter zugegriffen werden.

Methodendefinitionen

Die Methode `abheben` definiert man durch

```
public void abheben(double betrag) {
    kontostand = kontostand - betrag;
}
```

Rectangle

Ein Objekt der Klasse `Rectangle` kann man so verschieben:

```
r.translate(34,12)
```

Wie sieht die Deklaration der Methode `translate` aus?

Antwort

```
public void translate(int dx, int dy) {
...
}
```

Und wie sieht der Methodenrumpf (method body) aus?

Antwort

```
public void translate(int dx, int dy) {
   x = x + dx;
   y = y + dy;
}
```

Methodendefinitionen

Die Methode `getKontostand` liefert einen Double-Wert zurück.

```
public double getKontostand() {
    return kontostand;
}
```

Die Deklaration `public double getKontostand()` besagt, dass die Methode keine Parameter erwartet, aber einen `double` zurückliefert.

Das `return` Statement legt den zurückgegebenen Wert fest.

Wie würden wir die Methode `union` der Klasse `Rectangle` deklarieren, die das kleinste achsenparallele Rechteck ausgibt, welches das gegebene Rechteck und ein weiteres umfasst.

Antwort

```
public Rectangle union(Rectangle r){
...
}
```

und wie würden wir sie implementieren?

Antwort

```
Rectangle union(Rectangle r) {
     int resx, resy, reswidth, resheight;

     if (x <= r.x)
         resx = x;
     else
         resx = r.x;
     if (y <= r.y)
         resy = y;
     else
         resy = r.y;
     if (x + width >= r.x + r.width)
         reswidth = x + width - resx;
     else
         reswidth = r.x + r.width - resx;
     if (y + height >= r.y + r.height)
```

Antwort

```
            resheight = y + height - resy;
        else
            resheight = r.y + r.height - resy;
        return new Rectangle(resx, resy, reswidth, resheight
}
```

Beachte hier:

- Im Methodenrumpf kann man jederzeit neue Variablen deklarieren.
- Bei Objekterzeugung das `new` nicht vergessen.

Konstruktoren

Ein neues Bankkonto erzeugt man mit

```
new Bankkonto()
```

Die Instanzvariable `kontostand` ist dann automatisch mit Null vorbesetzt.

Besser ist es, das Verhalten von "`new Bankkonto()`" selber zu definieren.

Dazu schreibt man in die Klasse ein oder mehrere Konstruktordefinitionen.

Zum Beispiel:

```
Bankkonto() {
   kontostand = 0.0;
}
```

Konstruktoren

Man kann auch noch zusätzlich schreiben:

```
Bankkonto(double betrag) {
   kontostand = betrag;
}
```

Schreibt man nun `b = new Bankkonto()` so wird eins mit Kontostand `0.0` erzeugt.

Schreibt man dagegen `b = new Bankkonto(134.0)` so wird eins mit Kontostand `134.0` erzeugt.

Welcher Konstruktor zur Ausführung kommt, richtet sich nach Anzahl und Typen der Parameter.

Dies (ein und derselbe Name bezeichnet mehrere Funktionen) bezeichnet man als *overloading*.

Rectangles

Ein `Rectangle` kann man auch so erzeugen:

```
Point p;
Rectangle b = new Rectangle(p);
```

gemeint ist das Rechteck mit linker oberer Ecke `p` und Breite, Höhe beide Null.

Wie ist das in der Klasse `Rectangle` definiert?

Antwort

```
Rectangle(Point p) {
    x = p.x;
    y = p.y;
    width = 0;
    height = 0;
}
```

UML-Notation

UML = Unified Modelling Language.

Grafische Veranschaulichung von Vorgängen bei der Softwareentwicklung:

- Objektdiagramme: visualisieren Speicherinhalt
- Klassendiagramme: visualisieren Klassen und ihre Beziehungen
- Use-case Diagramme: visualisieren Verwendungsszenarien
- Statecharts: visualisieren Zustandsübergänge
- ...
- Klassen als dreigeteilte Rechtecke (Name, *public* Instanzvariablen, Methoden).
  - "*has-a*"-Beziehung als Pfeil mit ausgefüllter Dreiecksspitze. (Realisierung in Java durch Instanzvariablen)
  - "*is-a*"-Beziehung als Pfeil mit hohler Dreiecksspitze. (Realisierung in Java kommt später)

UML-Notation

- Objekte als zweigeteilte Rechtecke: Unterstrichener Klassenname, Instanzvariablen mit Werten.
  - Pfeile mit ausgefüllter Dreiecksspitze (→▶) für Verweise.

Details: siehe [Horstmann] und Tafelbeispiele.

Definition von Klassen: Formale Wiederholung

```
public class NameDerKlasse {
    Instanzvariablendeklarationen
    Methodendefinitionen
    Konstruktordefinitionen }
```

Die drei Komponenten einer Klassendefinition können in beliebiger Reihenfolge und auch vermischt angegeben werden.

Deklaration einer Instanzvariablen

Die Deklaration einer Instanzvariablen hat die Form

`private` *typ variablenName* `;`

Hier ist *variablenName* der Name der deklarierten Variablen. und *typ* ist der Typ der Instanzvariablen.

Der Typ kann entweder ein Basistyp sein

`double, int, char, boolean, ...`

...oder eine vordefinierte Klasse

`String, Point, Rectangle, Random, ...`

...oder eine selbstdefinierte Klasse

`Bankkonto, Dreieck, ...`

. Das bedeutet, dass die Instanzvariable nur Werte dieses Typs enthalten darf.

Deklaration einer Instanzvariablen

Im Gegenzug ist es möglich, die Instanzvariable mit Operationen dieses Typs zu verarbeiten.

Ist etwa `x` vom Typ `String`, dann darf man `x.length()` schreiben.

Definition einer Methode

`public` *ergebnisTyp methodenName* `(`typ_1 par_1`,` typ_2 par_2`, ...,` typ_n par_n`)` {

methodenRumpf

}

- *methodenName* ist der Name der Methode. Ist *e* ein Objekt der Klasse, dann ruft man die Methode so auf: *e*`.`*methodenName*`(`...*)*
- Die Variablen $par_1, \ldots, par_n$ sind die Parameter. Beim Methodenaufruf muss man konkrete Werte für die Parameter bereitstellen.
- Die Typen $typ_1, \ldots, typ_n$ sind die Typen der Parameter (Basistypen oder Klassen).

  Beim Methodenaufruf müssen die konkreten Parameter den angekündigten Typ haben.
- Der Typ *ergebnisTyp* ist der Typ des Ergebnisses des Methodenaufrufs. Ist er `void`, so wird kein Ergebnis zurückgegeben.

Der Methodenrumpf

- Der Methodenrumpf kommt zur Ausführung, wenn die Methode bei einem Objekt der Klasse aufgerufen wird.
- Im Methodenrumpf sind die Instanzvariablen, sowie die Parameter verfügbar. Außerdem möglicherweise deklarierte lokale Variablen.
- Der aktuelle Wert der Instanzvariablen ist bestimmt durch das Objekt, bei dem die Methode aufgerufen wird.
- Der aktuelle Wert der Parameter ergibt sich aus den konkreten Parametern, mit denen die Methode aufgerufen wird.
- Man kann sagen, dass die Instanzvariablen Teile eines *impliziten Parameters* sind.
- Hat die Methode einen von `void` verschiedenen Rückgabewert, so muss der Rumpf ein entsprechendes `return`-Statement enthalten.

Das `return`-Statement

Das `return`-Statement hat die Form:

`return` e`;`

wobei e ein Ausdruck ist.

Der Ausdruck e muss als Typ den Ergebnistyp der Methode, in der das `return` Statement vorkommt, haben.

Gelangt der Kontrollfluss (*vulgo* “der Computer”) an das Statement

`return` e`;`

so wird e ausgewertet und die Methodenabarbeitung beendet.
Rückgabewert der Methode ist dann der Wert von e.

Beispiele

```
public void einzahlen(double betrag) {
    kontostand = kontostand + betrag;
}

public void abheben(double betrag) {
    kontostand = kontostand - betrag;
}

public boolean equals(Bankkonto konto) {
    return kontostand == konto.getKontostand();
}

public double getKontostand() {
    return kontostand;
}
```

Überweisen

```
public void ueberweisen(double betrag, Bankkonto empfaenger)
    kontostand = kontostand - betrag;
    empfaenger.einzahlen(betrag);
    }
```

auch richtig, evtl. sogar besser:

```
public void ueberweisen(double betrag, Bankkonto empfaenger)
    this.abheben(betrag);
    empfaenger.einzahlen(betrag);
}
```

Der Ausdruck `this` bezeichnet das Objekt, bei dem die Methode aufgerufen wird.

Man darf `this` auch weglassen:

```
abheben(betrag);
```

`this` bei Instanzvariablen

Statt `kontostand` kann man auch `this.kontostand` schreiben.

Das ist nützlich, wenn man eine lokale Variable desselben Namens wie eine Instanzvariable deklariert hat.

Letztere würde nämlich die Instanzvariable unsichtbar machen.

```
public double getKontostand() {
    String kontostand = "Abrakadabra";
    return this.kontostand;
}
```

Wer will, kann immer `this` schreiben, um auf Instanzvariablen zuzugreifen.

Seiteneffekte

Verändert eine Methode die Instanzvariablen von anderen Objekten als `this` so liegt ein Seiteneffekt vor.

Beispiel: die Methode `ueberweisen` verändert die Instanzvariable des zweiten Parameters.

Seiteneffekte sollten so wenig wie möglich verwendet werden und auf jeden Fall nur da, wo wirklich sinnvoll.

Tastatureingaben und Bildschirmausgaben sind auch Seiteneffekte. Auch deren Verwendung innerhalb von Methoden sollte minimiert werden.

Schlechtes Beispiel: `getKontostand()` so implementiert, dass gleichzeitig auf den Bildschirm geschrieben wird.

Beispiel: Verzinsung

```
public double zinsenVorhersagen(int jahre, double zinssatz)
   double stand = this.kontostand;

   for (int i = 1; i <= jahre; i++) {
      stand = stand + zinssatz / 100. * stand;
   }
   return stand;
}
```

Drei Arten von Variablen

Es gibt drei Arten von Variablen:

- Instanzvariablen (z.B. `kontostand`)
- Parameter (z.B. `betrag`)
- Lokale Variablen (z.B. `stand`, `i`)

Jede Variable wird irgendwann mit ihrem Typ deklariert.

Jede Variable hat eine Lebensdauer, während der sie im Speicher repräsentiert ist. Während der Lebensdauer kann auf ihren Wert zugegriffen werden und ihr Wert verändert werden (durch `=`).

Jede Variable hat auch einen Sichtbarkeitsbereich im Programm. Man darf sie nur innerhalb ihres Sichtbarkeitsbereichs verwenden.

Die Sichtbarkeitsbereiche stellen sicher, dass beim Ablauf des Programms nie versucht wird, auf eine Variable zuzugreifen, deren Lebensdauer bereits um ist.

Instanzvariablen

- *Instanzvariablen* werden in einer Klasse deklariert; mit jedem (durch `new` erzeugten) Objekt der Klasse wird ein neuer Satz dieser Instanzvariablen ins Leben gerufen.
- Sie werden gemäß dem Konstruktor oder “per default” initialisiert.
- Die Lebensdauer einer Instanzvariablen ist gleich der Lebensdauer des Objekts, welches sie enthält.
- Eine Instanzvariable ist sichtbar in allen Methodenrümpfen ihrer Klasse. (Ausnahme: Überdeckung durch gleichlautende Variablen. Abhilfe: `this`).
- Mit `public` gekennzeichnete Instanzvariablen sind auch ausserhalb sichtbar. Nur in Bibliotheksklassen.

Parameter

- *Parameter* werden in der Parameterliste einer Methodendeklaration deklariert.
- Bei jedem Aufruf ihrer Methode werden sie ins Leben gerufen und mit den konkreten Parametern des Aufrufs initialisiert.
- Ihre Lebensdauer endet mit dem Verlassen der Methode.
- Parameter sind nur sichtbar im Rumpf ihrer Methode. (Ausnahme: Überdeckung durch gleichlautende lokale Variablen. Abhilfe: hier keine)

Lokale Variablen

- *lokale Variablen* werden innerhalb eines Blocks (Methodenrumpf, Block-Statement) deklariert und müssen explizit initialisiert werden.
- Sie werden bei der Abarbeitung ihrer Deklaration ins Leben gerufen; ihre Lebensdauer endet mit dem Verlassen des Blocks in dem sie deklariert wurden.
- Lokale Variablen sind nur in ihrem Block sichtbar. Ausnahme: Überdeckung.

```
int i = 10;
{
  int i = 8;
}
System.out.println(i); // druckt 10
```

Konstruktoren (Wdh.)

Eine Konstruktordefinition hat die Form

NameDerKlasse`(` typ_1 par_1`,` typ_2 par_2`, ...,` typ_n par_n`)` $\{$

konstruktorRumpf

$\}$

der Konstruktor wird in der Form

`new` *NameDerKlasse*`(` $e_1, \ldots, e_n$ `)`

aufgerufen.

Das ist ein *Ausdruck* dessen Wert ein frisches Objekt der Klasse *nameDerKlasse* ist.

Konstruktorauswertung

Die Instanzvariablen des neu erzeugten Objekts werden wie folgt initialisiert:

- Zunächst werden sie mit den Defaultwerten (0 für Zahlen, `null` für Objekte) besetzt.
- Dann werden die Ausdrücke $e_1, \ldots, e_n$ ausgewertet und ihre Werte den Parametern $par_1, \ldots, par_n$ zugewiesen.

  Wie immer müssen die Typen stimmen.
- Schließlich wird der Block *konstruktorRumpf* ausgeführt.

  Typischerweise hat das eine Zuweisung zu den Instanzvariablen in Abhängigkeit von den Parametern zur Folge.

Beispiel

```
Bankkonto(double betrag) {
    kontostand = betrag + 5.0;
}
```

Der Ausdruck `new Bankkonto(`e`)` hat als Wert einen Verweis uaf ein frisches Objekt der Klasse `Bankkonto`, dessen Instanzvariable `kontostand` initialisiert wurde mit dem Wert des Ausdrucks e plus 5.0.

Overloading

Man kann mehrere Konstruktoren in einer Klasse definieren.

Welcher zur Ausführung kommt richtet sich nach Anzahl und Typ der konkreten Parameter beim Aufruf.

Ebenso kann es mehrere Methodendefinitionen mit demselben Methodennamen geben.

Diese Möglichkeit bezeichnet man als Overloading (auch dt. Überladen).

Beispiele

```
    public void einzahlen(int betragEuro, int betragCent) {
        kontostand = kontostand + betragEuro + betragCent /
    }

    Bankkonto() {
        kontostand = 5.0;
    }
```

Jetzt kann man z.B. schreiben:

```
Bankkonto johannaSpar = new Bankkonto(1000.0);
Bankkonto matthiasGiro = new Bankkonto();

johannaSpar.einzahlen(1000.10);
matthiasGiro.einzahlen(39, 95);
```

Statische Methoden

Man kann in einer Klasse eine Methodendefinition mit dem Qualifikator `static` versehen.

Man hat dann innerhalb der Methode keinen Zugriff auf die Instanzvariablen.

Auf der anderen Seite kann man die Methode dann aufrufen ohne Bezug auf ein bestimmtes Objekt.

Damit man weiß, welche Methode gemeint ist, muss man aber den Klassennamen angeben.

Eine in Klasse *NameDerKlasse* definierte statische Methode *nameDerMethode* ruft man also so auf:

$$\textit{NameDerKlasse}\,.\,\textit{nameDerMethode}\,(\,e_1, \ldots, e_n\,)$$

Statische Methoden sind vergleichbar den *Funktionen* und *Prozeduren* von Pascal, C, etc.

Man benutzt sie z.B. um mathematische Funktionen zu realisieren.

Beispiel

Man könnte in die Klasse Bankkonto eine statische Methode `markNachEuro` schreiben, die DM-Beträge in € konvertiert:

```
public static double markNachEuro(double markBetrag) {
    return markBetrag / 1.9558; // oder was auch immer
}
```

Man kann dann zum Beispiel schreiben:

```
johannaSpar.einzahlen(Bankkonto.markNachEuro(500.0));
```

Man kann natürlich nicht schreiben `Bankkonto.einzahlen(...)`.
Welchen Wert sollte denn dann `kontostand` haben?

`johannaSpar.markNachEuro(...)` ist erlaubt, aber komisch.

“Bibliotheksklassen”

Oft hat man in einer Klasse nur statische Methoden.

So eine Klasse braucht dann keine Instanzvariablen zu haben.

Sie dient einfach der Gruppierung verwandter Operationen.

Zum Beispiel könnten wir eine Klasse `Numeric` schreiben, die numerische Verfahren enthält und insbesondere eine statische Methode für ungefähre Gleichheit.

```
public static approxEqual(double x, double y) {
   final EPSILON = 1E-12;
   return Math.abs(x-y) <= EPSILON
```