Einführung in die Programmierung mit Java

Teil 3: Objekte und Klassen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

26. Oktober 2017

Teil 3: Objekte und Klassen

Was bisher geschah

Nach drei Vorlesungen und einer Übung erwarten wir nun, dass Sie (mit Nachschlagen im Skript) Folgendes können:

- Java Programm schreiben und ausführen, welches 2–3 Zeilen beliebigen Text ausgibt und konstante Zahlen addieren kann.
- Grobes Verständnis der Java Basistypen `int`, `double`, `boolean` und Klasse `String`: Wie initialisiert man Variablen dieser Typen; sowie Kenntnis einer handvoll Operationen dazu.
- Speicherbereich „Stack“ ist eine Reihe von benannten Schubfächern. In den Fächern liegen Werte von Basistypen oder Verweise (Pfeile) auf „Objekte“.
- Alle „Objekte“ liegen in einem Speicherbereich, den wir „Heap“ nennen. Objekte sind „Boxen“ mit Klassennamen (Typ) und benannte Schubfächer darin (wie beim Stack).
- Ein `if-else` Statement erkennen und „lesen“ können.

Definition von eigenen Klassen

Fortlaufendes Beispiel des Kapitels

Wir wollen Bankkontos verwalten.

Ein Bankkonto enthält einen Kontostand

außerdem vielleicht noch Name des Besitzers, usw.

Der Kontostand kann

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

In der Standarbibliothek gibt es dafür keine passende Klasse, also schreiben wir uns diese einfach selbst!

Beispiel: Verwendung von Bankkonten

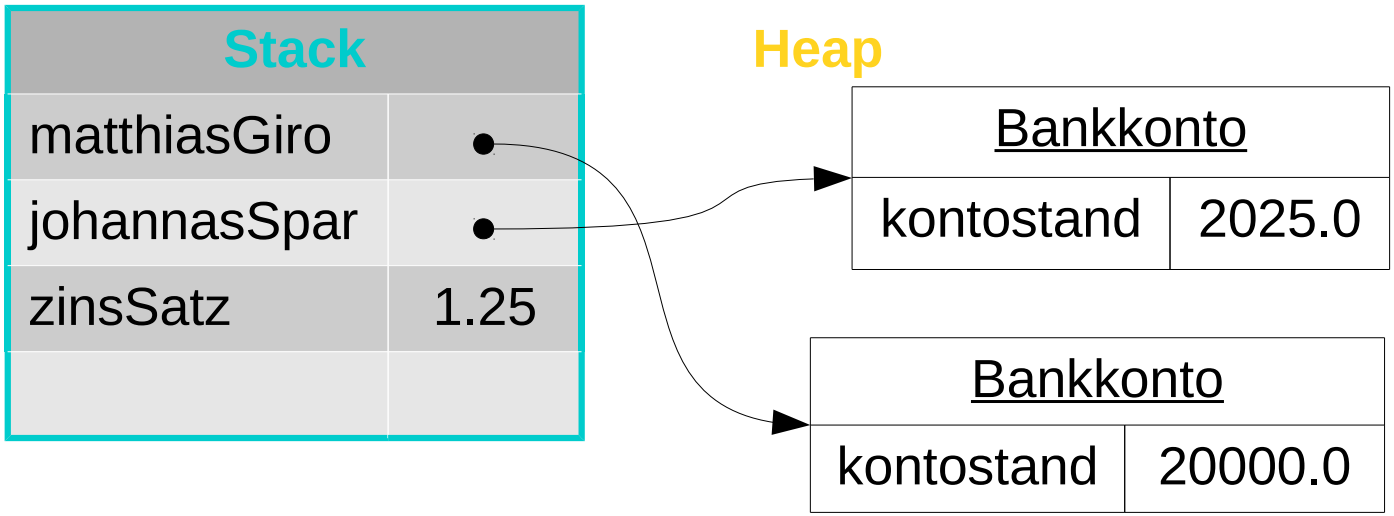
```
public class BankkontoTest {
    public static void main(String[] args) {
        Bankkonto matthiasGiro = new Bankkonto();
        Bankkonto johannasSpar = new Bankkonto();
        double zinsSatz = 1.25;

        matthiasGiro.einzahlen(30000.00);
        johannasSpar.einzahlen(2000.00);
        matthiasGiro.abheben(10000.00);
        johannasSpar.einzahlen(
            johannasSpar.getKontostand()*zinsSatz/100.0);
        System.out.println("Johannas Sparkonto: " +
                           johannasSpar.getKontostand());
        System.out.println("Matthias Girokonto: " +
                           matthiasGiro.getKontostand());
    }
}
```

Ausgabe:

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

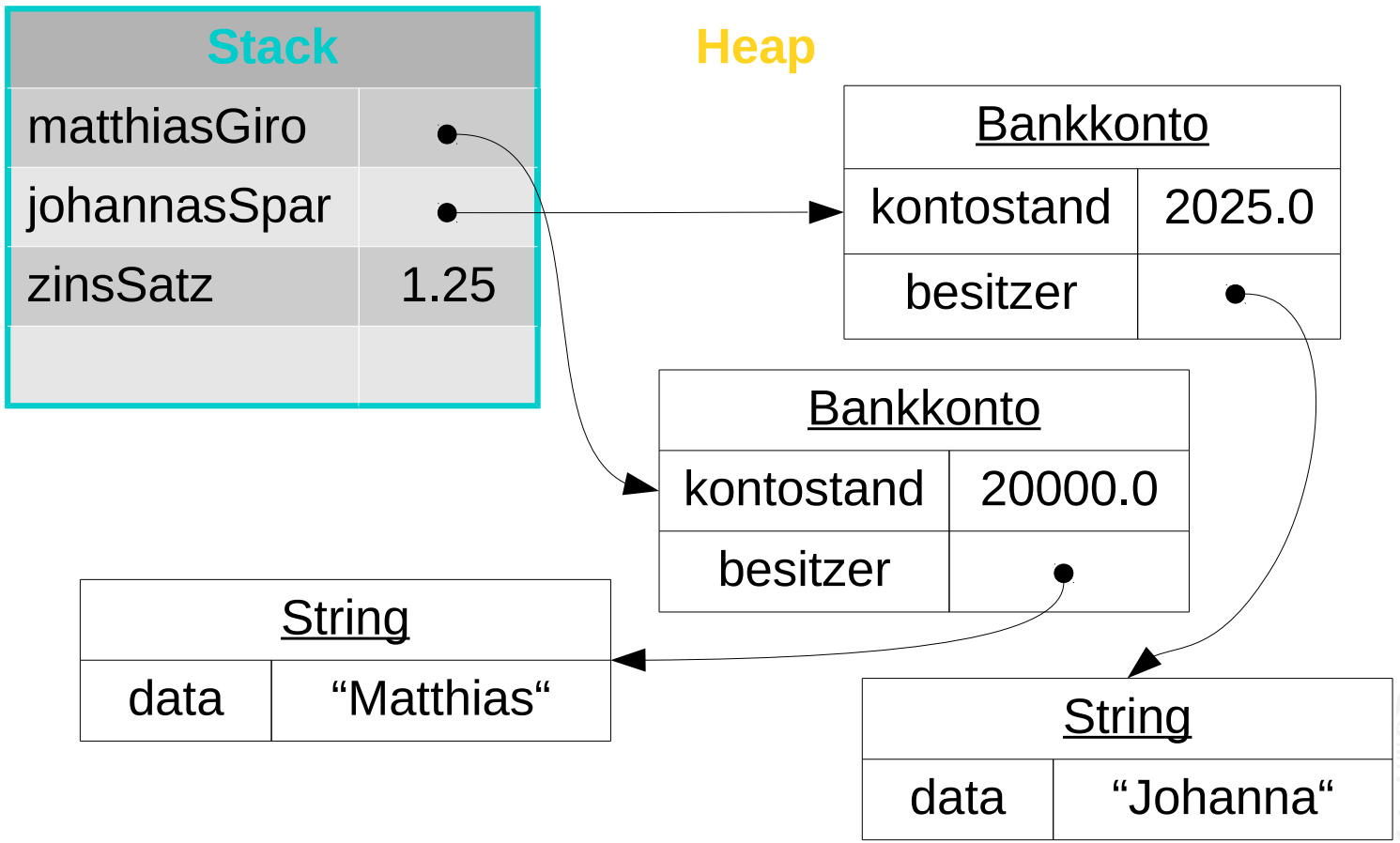
Beispiel: Objektdiagramm

Nach Ende der Ausführung des Programmes von Folie 3.5 hätten wir folgendes Objektdiagramm:



Beispiel: Objektdiagramm II

Objekte der Klasse `Bankkonto` könnten vielleicht auch noch den Namen des Besitzers als `String` speichern, so dass sich ungefähr folgendes Bild ergeben könnte:



Eigene Klassen definieren

Eigene Klassen-Definition folgen immer diesem Muster:

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

Dies muss in einer Datei gespeichert werden, deren Namen mit dem Klassennamen übereinstimmt.

Da hier der **Klassenname** *Bankkonto* gewählt wurde, müssen wir die Definition also in einer Datei `Bankkonto.java` speichern.

Die drei Einträge dürfen in beliebiger Reihenfolge stehen, sogar vermischt, doch obige Konvention kann die Lesbarkeit erleichtern.

Deklaration der Daten

Die im Objekt gespeicherten Daten bestehen aus Variablen, den **Instanzvariablen**, die in der Klasse deklariert werden.

Im Bankkonto-Beispiel brauchen wir (zunächst) nur eine Instanzvariable vom Typ `double`.
Wir schreiben also bei *Deklaration der zu speichernden 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 Aufrufe von Methoden dieser Klasse können darauf zugreifen. Methoden können solche Werte aber nach aussen reichen.

Instanzvariablen als public

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

Beispiel: Klasse `Rectangle` definiert `height` als `public`.

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

Aber es ist meist schlecht, Instanzvariablen `public` zu deklarieren!

Grund: Veränderungen der Deklaration in späteren Versionen sind dann problematisch.
Beispiel: Kontostand als `int` (in Cents) anstatt `double`?!

Gute **Modularisierung** erhöht die Wartbarkeit des Codes!

Beispiel: Rectangle

Ein `Rectangle` besteht aus den Koordinaten der linken oberen Ecke, sowie Breite und Höhe, jeweils als `int`.
Wie sieht die Datendeklaration in der Klasse `Rectangle` aus?

Beispiel: Rectangle

Ein `Rectangle` besteht aus den Koordinaten der linken oberen Ecke, sowie Breite und Höhe, jeweils als `int`.
Wie sieht die Datendeklaration in der Klasse `Rectangle` aus?

Antwort

```
public class Rectangle {
    // Deklaration der Instanzvariablen:
    public int x;
    public int y;
    public int width;
    public int height;

    ...
}
```

Für Instanzvariablen sollte `public` eigentlich vermieden werden!

Methodendefinitionen

Die **Methodendefinition** für `einzahlen` sieht so aus:

```
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** des Typs `double`
  $\Rightarrow$ Methoden dürfen beliebig viele Parameter deklarieren.
- Dieser Aufruf hier liefert *keinen* Wert zurück: `void`
  $\Rightarrow$ Eine Methode kann maximal einen Wert zurückgeben.
- Aufruf führt die Statements zwischen `{ }` aus.
  Im Beispiel hier also:
  `kontostand = kontostand + betrag;`
  $\Rightarrow$ Dabei kann neben den Parametern auch auf alle Instanzvariablen der Klasse zugegriffen werden!

Beispiel Methodendefinition

Die Methode `abheben` definiert man analog durch

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

Diese Methode darf also von jedem aufgerufen werden;
liefert keinen Ergebniswert zurück;
sondern reduziert den Wert der Instanzvariable `kontostand` um den Parameter `betrag`, dessen Wert beim Aufruf der Methode festgelegt wird.

Beispiel: Rectangle

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

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

Wie sieht die *Definition* der Methode `translate` aus?

Beispiel: Rectangle

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

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

Wie sieht die *Definition* der Methode `translate` aus?
Und wie sieht der *Methodenrumpf* (method body) aus?

Antwort

```
public void translate(int dx, int dy) {
  // Rumpf der Methode,
  // also Abfolge von Statements
}
```

Beispiel: Rectangle

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

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

Wie sieht die *Definition* der Methode `translate` aus?
Und wie sieht der *Methodenrumpf* (method body) aus?

Antwort

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

Rückgabewerte

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

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

Die Deklaration `public double getKontostand()` besagt, dass diese Methode keine Parameter erwartet, aber einen Wert des Typs `double` zurückliefert.

Methoden mit Rückgabewert dürfen auch Parameter haben.

Das `return` Statement legt den zurückgegebenen Wert fest.

Das `return` Statement beendet den Methodenaufruf.
Es *muss* immer als letztes Statement folgen,
falls der Rückgabewert ungleich `void` ist.

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*.

Beispiel: Rectangle.union

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

Beispiel: Rectangle.union

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

Antwort

```
public Rectangle union(Rectangle r){

  // ... Rumpf der Methode ...

}
```

Wie würden wir den Rumpf dieser Methode implementieren?

Beispiel: Rectangle.union

```
Rectangle union(Rectangle r) {
 int resx, resy, reswidth, resheight; // lokale Variablen
 resx = Math.min(x, r.x);
 resy = Math.min(y, 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) {
          resheight =   y +   height - resy;
 } else { resheight = r.y + r.height - resy; }
 return new Rectangle(resx, resy, reswidth, resheight);
}
```

Beachte:

- Im Methodenrumpf darf man jederzeit **lokale Variablen** deklarieren.
- Bei Objekterzeugung das `new` nicht vergessen.

Beispiel: Rectangle.union

```
Rectangle union(Rectangle r) {
 int resx, resy, reswidth, resheight; // lokale Variablen
 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)
     resheight = y + height - resy;
 else resheight = r.y + r.height - resy;
 Rectangle res = new Rectangle(resx, resy, reswidth, resheight);
 return res;
}
```

Beachte:

- Im Methodenrumpf darf man jederzeit **lokale Variablen** deklarieren.
- Bei Objekterzeugung das `new` nicht vergessen.

Vier Arten von Variablen

Es gibt vier Arten von Variablen:

- **Lokale Variablen** (z.B. `resheight`, `resx`)
- **Parameter** (z.B. `r`)
- **Instanzvariablen** (z.B. `height`, `x`)
- **Klassenvariablen**

Jede Variable wird irgendwann mit ihrem Typ **deklariert**.

Jede Variable hat während der Abarbeitung des Programms eine **Lebensspanne**, während der sie im Speicher repräsentiert ist. Während dieser Zeit kann auf ihren Wert zugegriffen werden und ihr Wert verändert werden (z.B. mit Zuweisung `=`).

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 außerhalb ihrer Lebensspanne zuzugreifen. Sichtbarkeit $\subseteq$ Lebensspanne

Lokale Variablen

Lokale Variablen werden innerhalb eines Blocks (Methodenrumpf, Block-Statement) deklariert und müssen explizit initialisiert werden.

Beispiel:

```
{
  ...
  String name;        // Deklaration
  name = "Steffen";   // Initialisierung
  ...
}
```

Lebensspanne Beginnt bei Abarbeitung ihrer Deklaration; ihre Lebensspanne endet mit dem Verlassen des Blocks in dem sie deklariert wurden.

Initialisierung Immer explizite Zuweisung mit =

Sichtbarkeit Lokale Variablen sind nur in ihrem Block sichtbar.

Lokale Variablen

Lokale Variablen werden innerhalb eines Blocks (Methodenrumpf, Block-Statement) deklariert und müssen explizit initialisiert werden.

Beispiel:

```
{
  ...
  String name = "Steffen"; // Dekl. mit Init.

  ...
}
```

Lebensspanne Beginnt bei Abarbeitung ihrer Deklaration; ihre Lebensspanne endet mit dem Verlassen des Blocks in dem sie deklariert wurden.

Initialisierung Immer explizite Zuweisung mit =

Sichtbarkeit Lokale Variablen sind nur in ihrem Block sichtbar.

Parameter

Parameter werden in der Parameterliste einer Methodendefinition deklariert.

Beispiel:

```
public void abheben(double betrag) {...}
```

Lebensspanne

Lebensspanne beginnt mit jedem Aufruf ihrer Methode und endet sofort mit jedem Verlassen der Methode.

Initialisierung

Bei jedem Aufruf ihrer Methode mit den konkreten Argumenten.

Sichtbarkeit

Nur sichtbar im Rumpf ihrer Methode.

Instanzvariablen

Instanzvariablen werden in einer Klasse deklariert und gehören jeweils zu einem Objekt dieser Klasse.

Beispiel

```
public class MyClass {
  private Rectangle rechteck;
  private int x = 42; //explizit initialisiert
  ...
}
```

Lebensspanne Mit jedem durch `new` erzeugten Objekt der Klasse wird ein neuer Satz aller Instanzvariablen der Klasse ins Leben gerufen. Die Lebensspanne ist identisch zur Lebensspanne des Objekts, welches sie enthält.

Initialisierung Immer bei der Erzeugung des umschliessenden Objekts: explizit, per Default oder gemäß **Konstruktor**.

Sichtbarkeit Sichtbar in allen Methodenrümpfen ihrer Klasse.

`public` Instanzvariablen sind auch ausserhalb sichtbar.

Ausnahme: Überschattung durch gleichlautende Variablen

Klassenvariablen

Klassenvariablen werden in einer Klasse deklariert und gehören zur Klasse, unabhängig von Objekten.

Beispiel

```
public class MyClass {
  private static Rectangle rechteck;
  private static int x = 42;
  ...
}
```

Lebensspanne

Gesamter Programmablauf.

Initialisierung

Immer explizit notwendig.

Sichtbarkeit

Sichtbar in allen Methodenrümpfen ihrer Klasse.

`public` Klassenvariablen sind auch ausserhalb sichtbar.

Ausnahme: Überschattung durch gleichlautende Variablen

Überschattung

Lokale Variablen und Parameter dürfen Instanzvariablen und Klassenvariablen mit gleichem Namen **überschatten**:

```
public class ShadowDemo {
    double betrag;

    public void abheben(double input) {
        ...
        int betrag = 42;  //(*)
        ...
}   }
```

Vor der mit `(*)` kommentierten Zeile ist die Instanzvariable `betrag` des Typs `double` sichtbar; danach ist diese überschattet. Stattdessen ist ab dann die davon *unabhängige lokale Variable* `betrag` des Typs `int` sichtbar. Der Typ darf auch gleich bleiben.

$\Rightarrow$ Überschattung sollte man vermeiden!

Temporäre Überschattung

```
public class ShadowDemo {
  double betrag;
  public void abheben(double input) {
    ...                              // (A)
    if (betrag > 0) {
        ...                          // (B)
        String betrag = "42";
        ...                          // (C)
      }
    ...                              // (D)
} }
```

- Die Überschattung wirkt nur in (C), der Lebensspanne der lokalen Variablen `betrag` des Typs `String`.
- In Abschnitt (D) hat `betrag` wieder den Wert des Typs `double`, der am Ende von Abschnitt (B) galt.

Sichtbarkeit und Lebensspanne müssen nicht identisch sein!

Impliziter Parameter this

In Rumpf einer Methode ist der **implizite Parameter** `this` sichtbar. Er zeigt auf das Objekt, dessen Methode aufgerufen wurde. Also ist der Typ von `this` immer die Klasse, in der die Methode definiert ist. Ausnahme: statische Methoden (später)

Das ist unter anderem auch nützlich, wenn eine lokale Variable oder ein Parameter eine Instanzvariable desselben Namens überschattet:

```
public class Bankkonto {
    private double kontostand;
    ...

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

Beispiel: Rectangle.union mit this

Man kann auch immer `this` schreiben, um auf Instanzvariablen zuzugreifen, um versehentliche Verwechslungen mit lokalen Variablen oder Parametern zu verhindern:

```
Rectangle union(Rectangle r) {
 int resx, resy, reswidth, resheight; // lokale Variablen
 resx = Math.min(x, r.x);
 resy = Math.min(y, 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) {
          resheight =   y +   height - resy;
 } else { resheight = r.y + r.height - resy; }
 return new Rectangle(resx, resy, reswidth, resheight);
}
```

Beispiel: Rectangle.union mit this

Man kann auch immer `this` schreiben, um auf Instanzvariablen zuzugreifen, um versehentliche Verwechslungen mit lokalen Variablen oder Parametern zu verhindern:

```
Rectangle union(Rectangle other) {
 int resx, resy, reswidth, resheight; // lokale Variablen
 resx = Math.min(this.x, other.x);
 resy = Math.min(this.y, other.y);
 if (this.x + this.width >= other.x + other.width) {
          reswidth  =  this.x +  this.width - resx;
 } else { reswidth  = other.x + other.width - resx; }
 if (this.y + this.height >= other.y + other.height) {
          resheight =  this.y +  this.height - resy;
 } else { resheight = other.y + other.height - resy; }
 return new Rectangle(resx, resy, reswidth, resheight);
}
```

Konstruktoren

Ein neues Bankkonto erzeugt man mit

```
new Bankkonto()
```

Die Instanzvariable `kontostand` wird **automatisch** mit 0 initialisiert.

Besser ist es, das Verhalten von "`new Bankkonto()`" selber zu definieren. Dazu deklariert man in der Klasse ein oder mehrere **Konstruktoren**. Dies sind Methoden, welche den gleichen Namen wie die Klasse tragen. Der Rückgabewert wird nicht explizit angegeben – dieser ist ja immer das neue Objekt!

Beispiel Konstruktor

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

Konstruktoren

Man kann auch noch *zusätzlich* schreiben:

```
Bankkonto(double kontostand) {
    this.kontostand = kontostand;
}
```

- `b = new Bankkonto()` erzeugt ein Bankkonto mit Kontostand `0.0` erzeugt.
- `b = new Bankkonto(134.0)` erzeugt ein Bankkonto mit Kontostand `134.0` erzeugt.

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

Das ein und derselbe Name mehrere Funktionen mit unterschiedlichem Typen referenziert, bezeichnet man als **overloading**. Dies ist auch bei normalen Methoden erlaubt.

Beispiel: Rectangle

Ein `Rectangle` kann man auch so erzeugen:

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

Erzeugt Rechteck mit linker oberer Ecke `p` und Breite & Höhe $= 0$.
Wie ist das in der Klasse `Rectangle` definiert?

Beispiel: Rectangle

Ein `Rectangle` kann man auch so erzeugen:

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

Erzeugt Rechteck mit linker oberer Ecke `p` und Breite & Höhe $= 0$.
Wie ist das in der Klasse `Rectangle` definiert?

Antwort

Durch einen zusätzlichen Konstruktor:

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

Beispiel: Overloading Methoden

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

public void einzahlen(int betragEuro, int betragCent) {
    kontostand = kontostand + betragEuro + betragCent / 100.0;
}
```

Damit kann man dann z.B. schreiben:

```
Bankkonto johannaSpar  = new Bankkonto(100.0);
Bankkonto matthiasGiro = new Bankkonto(100.0);

johannaSpar.einzahlen(12.34);    // =  112.34
matthiasGiro.einzahlen(39.95);   // =  139.95
```

Vergleichen von Objekten

Identische Objekte

Objekte kann man mit == vergleichen. Dabei werden aber nur die Speicheradressen verglichen, es wird also nur geprüft, ob es sich wirklich um zwei Referenzen auf dasselbe Objekt handelt.

„dasselbe“ versus *„das Gleiche“*

Inhaltliche Gleichheit

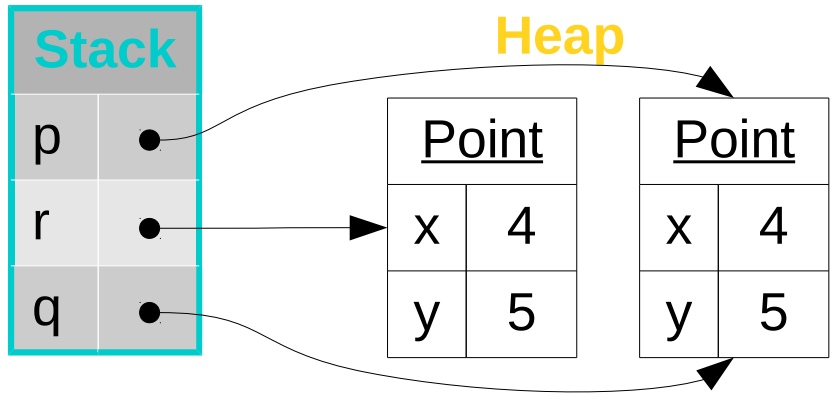
Die meisten Objekte bieten eine Methode `equals` an, welche die Werte der Instanzvariablen vergleicht, oder eine andere sinnvolle semantische Gleichheit definiert.
Dies hängt von der Definition der Methode `equals` ab.

Beispiel: Objekte vergleichen

Die Klasse `Point` repräsentiert Punkte in der (zwei dimensionalen) Ebene, speichert also zwei Werte des Typs `int`

```
Point p = new Point(1,3);
Point r = new Point(4,5);
Point q = p;
q.translate(3,2);
System.out.println(p==q);
System.out.println(q==r);
System.out.println(p.equals(q));
System.out.println(q.equals(r));
```

Was kommt heraus?



Beispiel: Objekte vergleichen

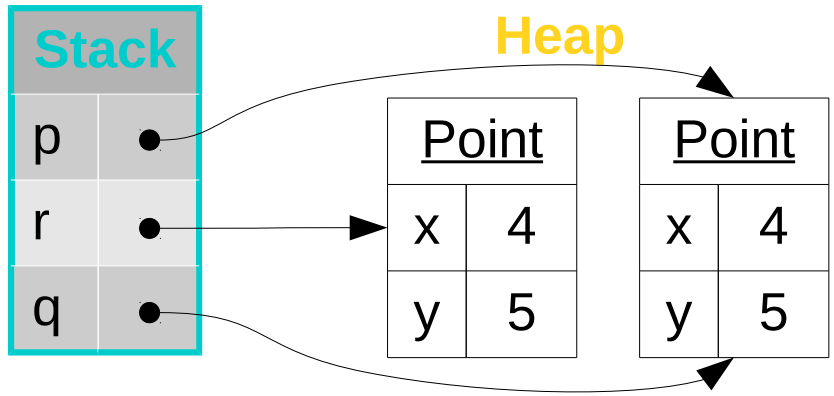
Die Klasse `Point` repräsentiert Punkte in der (zwei dimensionalen) Ebene, speichert also zwei Werte des Typs `int`

```
Point p = new Point(1,3);
Point r = new Point(4,5);
Point q = p;
q.translate(3,2);
System.out.println(p==q);
System.out.println(q==r);
System.out.println(p.equals(q));
System.out.println(q.equals(r));
```

Was kommt heraus?

Ausgabe:

```
true   // dasselbe
false  // nicht dasselbe
true   // gleich
true   // gleich
```



Der spezielle Wert null

Jede Klasse definiert einen Typ von Objekten (`Bankkonto`, `String`, usw.). Ein Typ ist eine Menge von Werten.
Werte eines Objekttyps sind...

- Verweise auf (Speicheradressen von) Objekten der entsprechenden Klasse im Heap; oder
- der spezielle Wert `null`.

Ist der Wert eines Objektausdrucks `null`, so führen darauf ausgeführte Methodenaufrufe zu Laufzeitfehlern!

Alle Variablen von Objekttypen werden mit `null` initialisiert, falls keine explizite Initialisierung angegeben wurde.

Mit der Operationen `==` kann man testen, ob eine Variable eines Objekttyps gleich `null` ist.

Beispiel: null-Test

```
public void ueberweisen(double betrag, Bankkonto empfaenger) {
    kontostand = kontostand - betrag;
    if (empfaenger == null)
        /* Fehlerbehandlung */
    else
       empfaenger.einzahlen(betrag);
    }
```

Beispiel: null-Test

```
public void ueberweisen(double betrag, Bankkonto empfaenger) {
    kontostand = kontostand - betrag;
    if (empfaenger == null)
        /* Fehlerbehandlung */
    else
       empfaenger.einzahlen(betrag);
    }
```

Achtung Seiteneffekt

Diese Methode verändert die Instanzvariable von einem anderem `Bankkonto`-Objekt!

Normalerweise sollte eine Methode nur das Objekt verändern, mit dem die Methode aufgerufen wurde, also `this`

Seiteneffekte

Verändert eine Methode die Instanzvariablen von anderen Objekten als `this` so spricht man in Java von einem **Seiteneffekt**.

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

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

Seiteneffekte erschweren die Lesbarkeit/Verständlichkeit eines Programmes, da man das Vorhandensein von Seiteneffekten nicht an der Signatur der Methode erkennen kann.

Man muss darauf hoffen, dass ein Programmierer alle möglicherweise auftretenden Seiteneffekte in der Dokumentation erwähnt. Good Luck! ;)

Seiteneffekte

Es gibt daher auch Programmieransätze, welche Seiteneffekte weitestgehend verbieten möchten. siehe ProMo, Semester 2

Man kann auch sagen, dass eine simple Zuweisung wie z.B. `x=x+1` den Seiteneffekt hat, die Variable `x` zu verändern.
⇒ Alle Zustand-verändernden Statements haben einen Seiteneffekt!

Lehnt man dies ab, so werden Variablen also nur noch initialisiert, aber danach niemals mehr verändert – wie in der Mathematik!
Dies hat unter Anderem den Vorteil, dass Programme lokal verständlich werden, da man nicht mehr an einen Zustand denken muss.

Auch in Java ist *inzwischen* die Ansicht weit verbreitet, dass Instanzvariablen möglichst nicht verändert werden sollen, sondern stattdessen neue Objekte zurückgegeben werden sollen.

Java Kompiler ist inzwischen darauf optimiert worden

Seiteneffekte

Auch **Ein-/Ausgabe** (engl. **I/O**), also z.B. Lesen/Schreiben von Dateien, Netzwerkverbindungen, Konsole (also Tastatureingaben und Bildschirmausgaben) kann man als Seiteneffekte betrachten!

Auch deren Verwendung innerhalb von Methoden sollte möglichst minimiert werden.
Es ist z.B. oft sinnvoll, alle Ein-/Ausgabe Operationen an einer zentralen Stelle im Programm zu bündeln.

In Sprachen, welche Seiteneffekte verbieten, tut man sich mit notwendigem I/O entsprechend schwerer.
In Java haben wir die Freiheit, selbst zu entscheiden, wo wir Seiteneffekt für angemessen halten.

Statische Methoden

Methodendefinition können den Qualifikator `static` tragen. Diese Methoden haben keinen Zugriff auf die Instanzvariablen und keinen impliziten Parameter `this`.

Dafür kann man die Methode ohne Bezug auf ein bestimmtes Objekt aufrufen. Stattdessen gibt man den Klassennamen an.

Beispiel: statische Methoden

```
int i = Math.max(42,7);
```

`max` ist eine statische Methode der Klasse `Math`
Statische Methoden benutzt man z.B. um mathematische *Funktionen* zu realisieren (auch *Prozeduren* in Pascal, C, etc.)

Statische Methoden — 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;
}
```

Dann kann man zum Beispiel schreiben:

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

Achtung:
Man kann natürlich nicht schreiben `Bankkonto.einzahlen(...)`
Denn welchen Wert sollte denn dann `kontostand` haben?

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

Statische "Bibliotheksklassen"

Manche Klassen besitzen nur statische Methoden und haben dann keine Instanzvariablen. Solche Klassen dienen einfach der Gruppierung verwandter Operationen.

Beispiel

Wir könnten eine Klasse `Numeric` schreiben, welche numerische Verfahren enthält und insbesondere eine statische Methode für ungefähre Gleichheit.

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

Die Klasse `Math` ist ein Beispiel einer solchen statischen Klasse.

Statische Variablen

Deklarationen von Klassenvariablen ähneln Deklarationen von Instanzvariablen, haben aber zusätzlich den Qualifikator `static`:

```
public class MyClass {
    private static int myNumber = 42;
}
```

Klassenvariablen existieren nur *einmal* für die gesamte Klasse! Alle Objekte der Klasse können diese *eine* Variable lesen und verändern. Ein sinnvoller Einsatz solcher Klassenvariablen sind Konstanten:

```
public class Math {
    public static final double PI = 3.1415;
    public static final double E  = 2.7182;
    ...
}
```

Verwendung ähnelt statischen Methoden, z.B.:

```
int x = 2 * Math.Pi + 3;
```

Beispiel: statische Klassenvariable

```
public class Bankkonto {
  /** Der Kontostand. */
    private double kontostand;
  /** Der Wert eines EUR in USD. */
    private static double dollarkurs;

  /** Setzen von {@link #dollarkurs dollarkurs} */
    public static void setDollarkurs(double kurs) {
        dollarkurs = kurs;
    }
  /** Abfragen des Kontostands in USD. @return den
      Kontostand in USD gemaess @see dollarkurs.  */
    public double getbalanceinUSD() {
        return kontostand * dollarkurs;
    }
```

Javadoc erzeugen mit `javadoc -author -version -private`

Methoden unserer Klasse Bankkonto

```
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;
}

boolean istUeberzogen() {
    return kontostand < 0.0;
}
```

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);
}
```

Bei Methodenaufrufe mit `this` kann man es auch weglassen, wenn man will:

```
    abheben(betrag);
```

Instanzvariablendeklaration

Die Deklaration einer Instanzvariablen in einer Klasse hat die Form

```
private typ variablenName ;
```

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

Der Typ kann sein:

- ein Basistyp, z.B. `double`, `int`, `char`, `boolean`,...
- eine vordefinierte Klasse, z.B. `String`, `Point`, `Rectangle`, ...
- eine selbstdefinierte Klasse, z.B. `Bankkonto`, `Dreieck`, ...

Instanzvariablen dürfen nur Werte dieses Typs enthalten und dürfen nur mit Operationen dieses Typs bearbeitet werden.

Klassenvariablendeklaration

Die Deklaration einer Klassenvariable hat die Form

```
private static typ variablenName ;
```

Syntaktischer Unterschied zur Instanzvariablendeklaration ist das Schlüsselwort `static`.

Merkmale Klassenvariable

- Alle Objekte dieser Klasse greifen auf die gleiche Variable zu!
  Vorsicht vor Seiteneffekte!
- Darf auch in statischen Methoden verwendet werden.

Merkmale Instanzvariable zum Vergleich:

- Jedes Objekt trägt seinen eigenen Satz aller Instanzvariablen.
- Dürfen *nicht* in statischen Methoden verwendet werden — sind ja nur in Objekten gespeichert

Methodendefinition

```
public ergTyp methName ( typ1 par1, ... typn parn ) {
    methRumpf
}
```

- *methName* ist der Name der Methode. Ist *e* ein Objekt der Klasse, dann ruft man die Methode so auf:

  $$e.methName(\ldots)$$

- 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 *ergTyp* 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 des *impliziten Parameters* `this` sind.
- Hat die Methode einen von `void` verschiedenen Ergebnistyp, so *muss* der Rumpf ein entsprechendes `return`-Statement enthalten.

Konstruktoren Definition

Eine Konstruktordefinition hat die Form

NameDerKlasse (typ_1 par_1 , typ_2 par_2 , ... typ_n par_n) {
    konstruktorRumpf
}

der Konstruktor wird aufgerufen in der Form

`new` *NameDerKlasse*(e_1 , ... , e_n)

Dies 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 $par_1, \ldots, par_n$ des Konstruktors 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: Konstruktorauswertung

Gegeben sei folgende Konstruktordefinition:

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

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

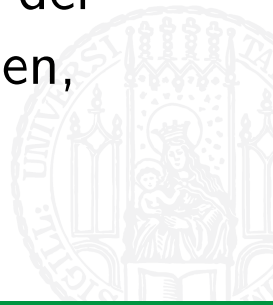
Lebensspanne von Objekten

Alle Objekte werden durch einen Aufruf von `new` erzeugt.
Dabei wird ein Objekt im Heap erstellt; Instanzvariablen werden initialisiert; ein Konstruktor wird ausgeführt.

Frage: Wann wird das Objekt wieder gelöscht?

Antwort: Wenn es nicht mehr gebraucht wird!
Java kümmert sich für uns automatisch darum.
Stichwort **Garbage Collection**, behandeln wir nicht tiefer.

Bemerkung: In vielen anderen Programmiersprachen muss sich der Programmierer selbst darum kümmern. Dies kann Vorteile bieten, aber auch eine große Fehlerquelle sein!



Zusammenfassung Klassen

- Klassen beinhalten Deklaration von Instanzvariablen, Methoden, Konstruktoren.
- Klassen dienen der Definition gleichartiger Objekte
- Von einer Klasse gibt es im allgemeinen mehrere Objekte
- Konstruktoren definieren die Initialisierung der Instanzvariablen bei der Erzeugung neuer Objekte der Klasse.
- Jede Variable hat eine Lebensspanne und einen Sichtbarkeitsbereich im Programmtext.
- Die vier Arten der Variablen sind: Instanzvariablen, Klassenvariablen, Parameter und lokale Variablen.
- Overloading bezeichnet das Vorhandensein unterschiedlicher Methoden und/oder Konstruktoren desselben Namens, aber mit unterschiedlichen Signaturen.

Zusammenfassung Objekte

- Ein Objekt hat seine Klasse als Typ
- Die Werte eines Klassen-Typs umfasst neben ihren Objekten auch den speziellen Wert `null`.
- Verweise auf Objekte werden mit `==` auf Identität verglichen ("dasselbe Objekt")
- Um Objekte anhand ihrer Instanzvariablen zu vergleichen, kann man eine `equals` Methode definieren.
- Jedes Objekte speichert seine eigenen Instanzvariablen
- Alle Objekte einer Klasse teilen sich alle statischen Klassenvariablen der Klasse
- Methoden werden von einem Objekt aus aufgerufen; im Rumpf der Methode is dieses Objekt mit `this` ansprechbar
- Statische Methoden (`static`) werden dagegen über den Klassennamen angesprochen