Refactoring

[Code bearbeiten]

Der folgende Quelltext hilft dir zwar nicht dabei, die Lottozahlen vorauszusagen, aber zumindest dabei, deinen Lottoschein auszufüllen, allerdings relativ unschön in der **`main`**-Methode. Ändere den Quelltext so, dass du eine neue Methode **`druckeLottoschein()`** zum Drucken des Lottoscheins erstellst, die die sieben zu tippenden Zahlen als Parameter bekommt.

```
public static void main(String[] args) {
  int zahl1 = Integer.parseInt(args[0]),
    zahl2 = Integer.parseInt(args[1]),
    zahl3 = Integer.parseInt(args[2]),
    zahl4 = Integer.parseInt(args[3]),
    zahl5 = Integer.parseInt(args[4]),
    zahl6 = Integer.parseInt(args[5]),
    zahl7 = Integer.parseInt(args[6]);
  for(int i = 1; i <= 49; i++) {
  if (i == zahl1 || i == zahl2 || i == zahl3 || i == ↵
  zahl4 || i == zahl5 || i == zahl6 || i == zahl7) {
      System.out.print("|x|");
    } else {
      System.out.print("|_|");
    }
    if (i % 7 == 0) {
      System.out.println("");
    }
  }
}
```

Achtung, Spoiler! Ab hier kommt die Lösung.

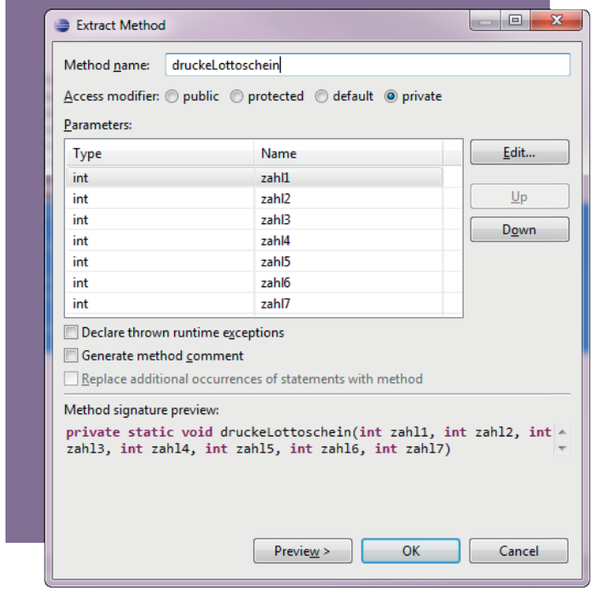
Wie zu erwarten, kann dir Eclipse auch hier wieder ganz gut helfen. Markiere einfach die Zeilen, die du in eine neue Methode packen möchtest und gehe dann auf **Refactor • Extract Method**.


```
public class Main {

    public static void main(String[] args) {
        int zahl1 = Integer.parseInt(args[0]),
            zahl2 = Integer.parseInt(args[1]),
            zahl3 = Integer.parseInt(args[2]),
            zahl4 = Integer.parseInt(args[3]),
            zahl5 = Integer.parseInt(args[4]),
            zahl6 = Integer.parseInt(args[5]),
            zahl7 = Integer.parseInt(args[6]);
        for (int i = 1; i <= 49; i++) {
            if (i == zahl1 || i == zahl2 || i == zahl3 || i == zahl4 || i == zahl5 || i == zah
                System.out.print("|x|");
            } else {
                System.out.print("|_|");
            }
            if (i % 7 == 0) {
                System.out.println("");
            }
        }
    }
}
```

Markiere nur den Teil, den du in eine neue Methode auslagern möchtest.

Im anschließenden Dialogfenster sagst du, welchen Namen die neue Methode haben soll. Außerdem ist Eclipse so klug, die notwendigen Parameter für die neue Methode zu erkennen. Denen kannst du übrigens auch neue Namen geben. Wenn du damit fertig bist, klick auf „**OK**“, und schwuppdiwupp ist die neue Methode fertig.



Hier kannst du unter anderem noch beeinflussen, welche Namen die neue Methode und deren Parameter haben sollen.

```java
    public static void main(String[] args) {
        int zahl1 = Integer.parseInt(args[0]),
            zahl2 = Integer.parseInt(args[1]),
            zahl3 = Integer.parseInt(args[2]),
            zahl4 = Integer.parseInt(args[3]),
            zahl5 = Integer.parseInt(args[4]),
            zahl6 = Integer.parseInt(args[5]),
            zahl7 = Integer.parseInt(args[6]);
        druckeLottoschein(zahl1, zahl2, zahl3, zahl4, zahl5, zahl6, zahl7);
    }

    private static void druckeLottoschein(int zahl1, int zahl2, int zahl3,
            int zahl4, int zahl5, int zahl6, int zahl7) {
        for (int i = 1; i <= 49; i++) {
            if (i == zahl1 || i == zahl2 || i == zahl3 || i == zahl4 || i == zahl5 || i == zah
                System.out.print("|x|");
            } else {
                System.out.print("|_|");
            }
            if (i % 7 == 0) {
                System.out.println("");
            }
        }
    }
}
```

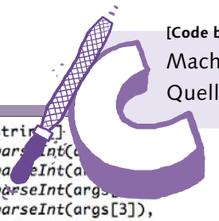
Voilà, die neue Methode ist fertig!

[Notiz]

Diese Art und Weise, bestehenden Quelltext neu zu strukturieren, nennt man auch **Refactoring**. Ziel ist dabei in der Regel, den Quelltext **lesbarer, leichter wiederverwendbar und erweiterbar** zu gestalten. Ein erster Anfang sind **kurze, übersichtliche Methoden**, die einen speziellen Zweck erfüllen. Ein wirklich gutes Buch zu diesem Thema ist „Refactoring: Improving the Design of Existing Code“ von Martin Fowler. Du kannst es ja mal durchblättern, wenn der WoW-Server down ist.

[Notiz]

Eclipse ist nicht die einzige IDE, die solche tollen Refactoring-Tricks drauf hat. Netbeans, IntelliJ und JBuilder können das natürlich auch.



[Code bearbeiten]

Mach direkt noch zwei Refactorings, so dass du den folgenden Quelltext rausbekommst:

```java
    public static void main(Stri[] args) {
        int zahl1 = Integer.pars Int(a
            zahl2 = Integer.pa eInt(a
            zahl3 = Integer.parseInt(args
            zahl4 = Integer.parseInt(args[3]),
            zahl5 = Integer.parseInt(args[4]),
            zahl6 = Integer.parseInt(args[5]),
            zahl7 = Integer.parseInt(args[6]);
        druckeLottoschein(zahl1, zahl2, zahl3, zahl4, zahl5, zahl6, zahl7);
    }

    private static void druckeLottoschein(int zahl1, int zahl2, int zahl3,
            int zahl4, int zahl5, int zahl6, int zahl7) {
        for (int i = 1; i <= 49; i++) {
            druckeLottoscheinKästchen(zahl1, zahl2, zahl3, zahl4, zahl5, zahl6, zahl7, i);
            testeUndDruckeNeueZeile(i);
        }
    }

    private static void testeUndDruckeNeueZeile(int i) {
        if (i % 7 == 0) {
            System.out.println("");
        }
    }

    private static void druckeLottoscheinKästchen(int zahl1, int zahl2,
            int zahl3, int zahl4, int zahl5, int zahl6, int zahl7, int i) {
        if (i == zahl1 || i == zahl2 || i == zahl3 || i == zahl4 || i == zahl5 || i == zahl6
            System.out.print("|x|");
        } else {
            System.out.print("|_|");
        }
    }
```

[Notiz]

Eine weitere Refactoring-Regel sagt übrigens auch, dass man Methoden mit vielen Parametern vermeiden sollte. In der Tat sind unsere Methoden `druckeLottoschein()` und `druckeLottoscheinKaestchen()` mit sieben bzw. acht Parametern schon **weit über dem vertretbaren Maß**.
Wenn man nämlich so viele Parameter braucht und sie **inhaltlich zusammenhängen**, fasst man sie besser zu einem Objekt zusammen. Du bist kurz davor, zu lernen, wie das geht.
Alternativ gibt es seit Java 5 auch die sogenannte **Varargs**-Schreibweise, bei der du die Anzahl der Parameter nicht fest definierst, sondern variabel hältst. Das geht ganz einfach mit drei Punkten hinter dem Datentyp: `druckeLottoschein(int ... zahlen) {}` bedeutet, dass du der Methode beliebig viele `int`-Parameter übergeben kannst. Innerhalb der Methode kannst du dann mit `zahlen.length` herausfinden, wie viele Parameter übergeben wurden, und mit `zahlen[0]`, `zahlen[1]` und so weiter auf die übergebenen Werte zugreifen.

Sauberer Quelltext

Du hast jetzt nicht nur die Wiederverwendbarkeit deines Quelltextes erhöht, sondern auch die Lesbarkeit. Guck dir mal diesen Quelltext-Schnipsel an, ist der nicht gut lesbar (die Parameter hab ich mal weggelassen, um Tinte zu sparen)?

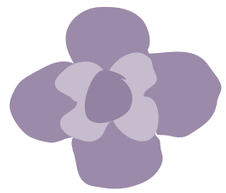

```
private static void druckeLottoschein(…) {
  for(int i=0; i<49; i++) {
    testeUndDruckeNeueZeile(i);
    druckeLottoscheinKaestchen(…);
  }
}
```

[Erledigt!]
Weil wir eben aussagekräftige Methodennamen genommen haben, kann man jetzt den Quelltext auch **besser verstehen**. Man sagt auch, dein **Quelltext ist jetzt sauberer** als vorher, der Google-Suchbegriff dafür heißt **Clean Code.**

[Notiz]
Der Begriff Clean Code wurde von Robert C. Martin in seinem Buch „Clean Code: A Handbook of Agile Software Craftsmanship" geprägt. In diesem Buch sind allerhand Regeln aufgeführt, um den Quelltext möglichst **sauber** und **verständlich** zu halten. Eine davon sagt zum Beispiel, man sollte aussagekräftige Methodennamen wählen, eine andere, die Länge einer Methode (sprich die Anzahl der Anweisungen) sollte möglichst gering gehalten werden. Ein weiteres gutes Buch für deinen Nachttisch.

Puh, jetzt muss ich nicht nur die Wohnung, sondern auch noch den Quelltext sauber halten. Und ich dachte …

… Programmieren wäre ein Spaß? Ist es auch. Aber nur, wenn die Programme **übersichtlich strukturiert** sind. Es gibt fast nichts Schlimmeres, als den dreckigen Quelltext von jemand anders aufräumen zu müssen. Oder würdest du gerne eine Messi-Wohnung putzen?



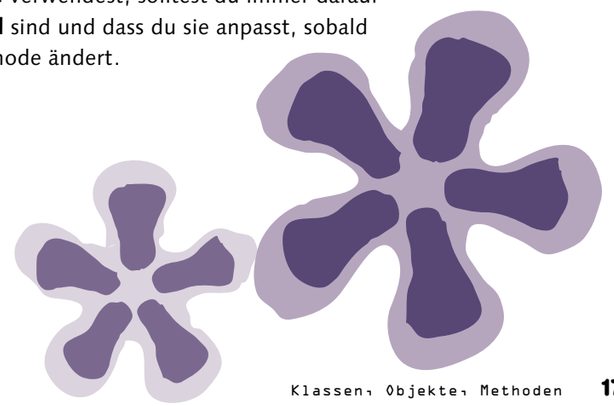
Methodenkommentare

Wenn dir mal kein aussagekräftiger Name einfällt, der wirklich beschreibt, was deine Methode so alles macht, kannst du auch Methodenkommentare verwenden. Ein Methodenkommentar steht vor der Methodendeklaration, fängt mit **/**** an und hört mit ***/** auf, zum Beispiel so:

```
/**
 * Diese Methode berechnet das Maximum für zwei Zahlen.
 *
 * @param zahl1          Die erste Zahl.
 * @param zahl2          Die zweite Zahl.
 * @return               Die größere der beiden Zahlen.
 */
public static int max(int zahl1, int zahl2) {
  return zahl1 > zahl2 ? zahl1 : zahl2;
}
```

Innerhalb eines Methodenkommentars stehen dir verschiedene Schlüsselwörter (sogenannte Tags) zur Verfügung. Über **`@param`** zum Beispiel kannst du die Parameter einer Methode beschreiben, mit **`@return`** den Rückgabewert. Und wenn du diese Tags immer brav verwendest, kannst du dir mithilfe von javadoc, einem weiteren JDK-Tool, eine praktische HTML-Dokumentation für deine Klassen generieren lassen. Das Tool erkennt dann anhand dieser Tags, worauf sich deine Kommentare genau beziehen. Das gucken wir uns noch an, in Kapitel 13.

[Achtung]
Wenn du **Kommentare** verwendest, solltest du immer darauf achten, dass sie **aktuell** sind und dass du sie anpasst, sobald sich die Logik der Methode ändert.



Kann ich das zurückgeben?

Bisher haben wir nur Methoden ohne Rückgabewert (also **`void`**-Methoden) implementiert. Toll wäre es doch, wenn Methoden nicht nur irgendetwas tun, sondern auch etwas **zurückgeben** könnten. Welchen **Typ** eine Methode zurückgeben soll, sagst du in der Methodendeklaration an der Stelle, an der wir bisher immer **`void`** verwendet haben. Welcher konkrete **Wert** zurückgegeben werden soll, sagst du dann innerhalb der Methode mit dem Schlüsselwort **`return`**:

```
public static double *1 berechneSchuhgroesse(double fusslaenge) {
  return *2 (fusslaenge + 1.5) / 0.666; *3
}
```

*1 Der **Rückgabetyp** der Methode ist **`double`**.

*2 Das, was nach dem **`return`** kommt, ist der wirkliche **Rückgabewert** der Methode.

*3 Klingt teuflisch, ist aber so: Die Schuhgröße wird berechnet, indem zur Länge des (längeren) Fußes noch 1,5 cm hinzugerechnet werden und das Ganze dann durch 0,666 geteilt wird.

[Zettel]
Du kannst **`return`** auch in **`void`**-Methoden verwenden, um aus der Methode herauszuspringen.

`return` kann auch an mehreren Stellen innerhalb einer Methode vorkommen, so wie hier:

```
private static String sageWetterVoraus(int monat, String land) {
  if("Deutschland".equals(land)) {
    return "Absolut nicht vorhersehbar.";
  } else if(monat >= 5 && monat <= 9) {
    return "Wahrscheinlich ganz gut.";
  }
  return "Regen mit Graupel.";
}
```

Wenn du deinen Quelltext aber schön sauber und übersichtlich halten möchtest, verwende **`return`** nur an einer Stelle – **Stichwort „Single Entry, Single Exit“.**

Zurück zu den Nachrichten

[Einfache Aufgabe]
Ändere den obigen Quelltext zur Wettervorhersage so, dass nur an einer Stelle (und zwar ganz am Ende) **`return`** verwendet wird.

Warte, das krieg ich hin …

```
private static String sageWetterVoraus(int monat, String land) {
  String vorhersage = "Regen mit Graupel.";*1
  if("Deutschland".equals(land)) {
    vorhersage = "Absolut nicht vorhersehbar.";*2
  } else if(monat >= 5 && monat <= 9) {
    vorhersage = "Wahrscheinlich ganz gut.";*2
  }
  return vorhersage;*3
}
```

*1 Ich brauche einfach eine **`String`**-Variable, …

*2 … der kann ich dann je nach Wetter einen anderen Wert zuweisen …

*3 … und ganz zum Schluss zurückgeben.

[Belohnung/Lösung]
Prima, dann guck mal aus dem Fenster.
`if` Regen, spiel World of Warcraft,
`else` nichts wie raus an die frische Luft!

Auf dem Weg zum Java-Ninja – Klassen und Objekte

Jetzt machen wir aus dir einen OOP-Profi: von Methoden zu Klassen und Objekten.

Du hast bisher ja schon jede Menge Klassen erstellt. Und sei es nur als Container für die `main`-Methode oder wie bisher in diesem Kapitel als **Container für statische Methoden**. Das ist aber nur die eine Seite der Medaille, denn eigentlich können Klassen viel mehr sein. Bisher hast du die eigentlichen Stärken von Java nämlich noch gar nicht ausgenutzt, sondern mehr oder weniger „nur" prozedural programmiert.

[Hintergrundinfo]

Bei der **prozeduralen Programmierung** wird eine Problemstellung in mehrere **Unterprobleme** unterteilt und jedes Unterproblem in einer **Funktion** formuliert. Ein prozedural programmiertes Programm besteht also aus verschiedenen Funktionen, die im optimalen Fall so programmiert sind, dass sie jeweils genau ein Problem lösen und Unterprobleme über den Aufruf von anderen (Unter-)Funktionen lösen. Also vom Gedanken her schon mal ganz gut, da eben Funktionen wiederverwendet werden können.
Mit Java lernst du aber eine Sprache, die wie gemacht ist für die **objektorientierte Programmierung**. Da verwendest du nicht nur Funktionen wieder, sondern ganze Objekte.

[Achtung]

Wenn du Java programmieren kannst, heißt das noch lange nicht, dass du auch objektorientiert programmieren kannst. Bisher hast du noch nicht objektorientiert gearbeitet, zumindest nicht bewusst.

Indem du deine Probleme zerteilst und Daten und Funktionen zusammen kapselst.

Wenn du komplexe Probleme zerlegst, wirst du bald merken, dass Datenstrukturen und die Operationen darauf zusammengehören. Wie du ein Element aus einer Liste korrekt löschen kannst, hängt vom Aufbau der Liste ab. Wie zum Beispiel ein Spieleheld eine Stufe aufsteigen kann, hängt davon ab, welche Eigenschaften und insbesondere Stufen

für ihn vorgesehen sind. Wie du Inkonsistenzen in deinen Daten vermeidest, hängt davon ab, was „konsistent" für diese Daten bedeutet und so weiter.
Da ist es toll, wenn beides implementiert wird, also Daten und Funktionen zusammen, und wenn jemand, der die Datenstruktur benutzen will, Funktionen wie „Löschen", „Belohnung kassieren" oder „Hinzufügen" aufrufen kann, ohne sich über die Interna Gedanken zu machen. Also wird **beides zusammen gekapselt.**

[Begriffsdefinition]

Kapselung bedeutet, dass dem Nutzer eines Objekts nur solche Methoden und Datenfelder zugänglich gemacht werden, die für ihn von Bedeutung sind.
Erinnerst du dich noch an den `StringBuilder`? Da war es dir ja auch egal, wie der intern dynamisch seine Größe ändert. Hauptsache, du kannst mit ihm dynamisch eine Zeichenkette zusammenbauen.
Und genau dafür stellt `StringBuilder` entsprechende Methoden zur Verfügung.

[Zettel]

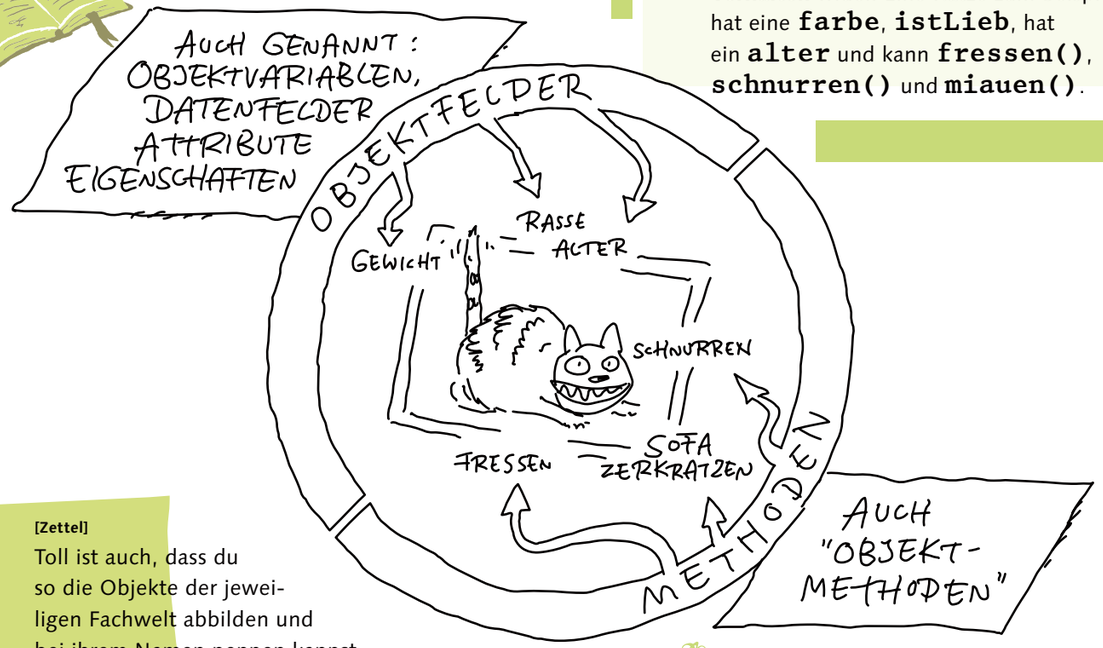
Deine bisherigen Klassen hatten nur statische Methoden. Die nennt man auch **Klassenmethoden**. Daneben gibt es noch **Klassenfelder** bzw. **Klassenvariablen**, bei denen steht wie bei Klassenmethoden auch das Wort `static` davor. Klassenfelder und Klassenmethoden kann man ansprechen bzw. aufrufen, sobald man die Klasse zur Verfügung hat. Klassenmethoden hast du ja in den letzten Abschnitten bis zum Exzess erstellt und Klassenfelder sogar schon in Kapitel 1 verwendet (ich sage nur Hexentexte).

In Java gehört jedes Objekt zu einer **Klasse**. Genauer gesagt, ist die Klasse der Datentyp des Objekts. Eine Klasse `Held` würde also definieren, durch welche Eigenschaften ein Held beschrieben werden kann (zum Beispiel Name, Level und Ausrüstung).
Ein konkreter Spieleheld, sagen wir Morcks auf Level 5 mit drei Schwertern, wäre ein **Objekt der Klasse `Held`**, oder noch genauer eine **Objektinstanz** der Klasse.

[Begriffsdefinition]

Das englische **class** wird im Deutschen mit **Klasse** übersetzt, ist aber eher im Sinne von **Klassifizierung** oder **Kategorie** gemeint. Klassen dienen dazu, zu definieren, wie die Objekte aussehen sollen. Vereinfacht gesagt sind Klassen so etwas wie eigene (beliebig komplexe) Datentypen, die du als Programmierer selbst definieren kannst. Kurz: Eine Klasse stellt sozusagen die Schablone oder den Bauplan für Objekte dar.

[Begriffsdefinition]
Methoden und Datenfelder, die nicht `static` sind, nennt man auch **Objektmethoden** (oder objektbezogene Methoden) und **Objektfelder bzw. Objektvariablen**. Um Objektfelder und Objektmethoden zu verwenden, reicht dir nicht die Klasse, du brauchst zuerst ein Objekt. Bevor wir aber ein Objekt erstellen können, brauchen wir die Klasse. Verwirrt? Keine Angst, nicht mehr lange. Ein Objekt zu einer Klasse besteht aus Daten und Funktionen. Die Daten heißen **Felder** oder **Attribute** oder **Eigenschaften** oder eben **Objektvariablen**, und die Funktionen heißen **Methoden** bzw. **Objektmethoden**.



[Zettel]
Toll ist auch, dass du so die Objekte der jeweiligen Fachwelt abbilden und bei ihrem Namen nennen kannst. Wenn du mit deinem Auftraggeber über seine Dinge sprichst, zum Beispiel mit einem Games-Manager über „Helden" und nicht über „record p2s" oder so, erleichtert das die Verständigung. Du und der Kunde, ihr sprecht eine **gemeinsame Sprache**.

[Ablage]
Die Belegung der Objektvariablen stellt den **Zustand** eines Objekts dar. Die Objektmethoden spiegeln sein **Verhalten** wider. Eine Katze zum Beispiel hat eine `farbe`, `istLieb`, hat ein `alter` und kann `fressen()`, `schnurren()` und `miauen()`.

Bevor du ein Objekt erstellen kannst, brauchst du erstmal eine ordentliche Klasse:

```
package de.galileocomputing.schroedinger.java.kapitel05;
public class FotoApparat {                    *1
  int megaPixel;                              *2
  double displayGroesse;                      *2
  boolean bildStabilisiert;                   *2
  String marke;                               *2
  int brennweiteMin;                          *2
  int brennweiteMax;                          *2

  public void machFoto() {                    *3
    System.out.println("Klick ");
  }
}
```

*1 Eine Klasse definierst du mit dem Schlüsselwort `class`. Das weißt du schon seit Kapitel 1, und daran ändert sich auch jetzt nichts. Und du weißt auch, dass die Datei genauso wie die Klasse heißen muss, also `FotoApparat.java`.

*2 Das sind die Datenfelder. Das können sowohl primitive Datentypen als auch **Referenztypen** sein. `String`-Variablen zum Beispiel wären Referenztypen.

*3 Hier definieren wir eine Methode. Beachte, dass wir diesmal das Schlüsselwort `static` weglassen. Das heißt, das hier ist keine Klassenmethode, sondern eine **Objektmethode**.

Und jetzt wird's mal Zeit, dass du ein `FotoApparat`-Objekt erstellst:
Das machst du über Konstruktoren, und die hast du ja bereits kennengelernt.

Ja, aber die hast du mich ja nie benutzen lassen, `new String()` und `new Integer()` waren ja die absoluten NoGos.

Ja, stimmt genau. Das waren zwei der Ausnahmen, bei denen du keine Konstruktoren verwenden solltest, weil beide Male der interne Caching-Mechanismus für Strings bzw. Ganzzahlen ausgehebelt würde. Das bedeutet aber nicht, dass du niemals Konstruktoren verwenden darfst. Bei deinen eigenen Klassen darfst du das natürlich schon.
Jetzt hast du eine Klasse mit mehreren Objektvariablen und einer Objektmethode, die du aber nicht mehr so wie eben (in einem statischen Kontext) aufrufen kannst. Du brauchst erst ein Objekt vom Typ `FotoApparat`. Am besten, du nimmst eine separate Klasse, in der du wie bisher eine `main`-Methode implementierst: In der kannst du dann das Objekt erzeugen.

```
public class FotoShooting { *1
  public static void main(String[] args) {
*2 FotoApparat *3 fotoApparat *4 = new *5 FotoApparat(); *6
    fotoApparat.machFoto(); *7
  }
}
```

*1 **FotoShooting** ist unsere Klasse mit **main**-Methode.

*2 Das ist hier die wichtige Zeile, in der das Objekt angelegt wird. Endlich! Gucken wir uns mal genau an, was hier passiert:

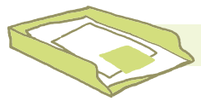
*3 Ganz links steht der Typ des Objekts, …

*4 … gefolgt vom Variablennamen, …

*5 … hinter dem Gleichheitszeichen geht es weiter: Mit **new** sagst du, dass jetzt wirklich das neue Objekt angelegt werden soll, …

*6 … und hinter dem **new** kommt dann der Konstruktoraufruf.

*7 Und jetzt, wo du das Objekt hast, kannst du auch die **Objektmethode** aufrufen. Bitte lächeln!



[Ablage]
Du hättest natürlich auch in der Klasse **FotoApparat** eine **main**-Methode erstellen können. Aber besser, du hältst Klassen, mit denen du etwas modellierst, frei davon und lagerst das Starten des Programms in eine Extraklasse aus.

Nur nochmal, um sicherzugehen: Objektmethoden kannst du ohne Objekt nicht aufrufen. So etwas hier kannst du also nicht machen:

```
public static void main(String[] args) {
  FotoApparat.machFoto(); *1 X
}
```

*1 Das gibt einen Compilerfehler, weil **machFoto() nicht statisch** ist. Du brauchst eine Objektinstanz, um diese Methode aufrufen zu können!

Kapselung

So schlank und schön unsere Klasse **FotoApparat** auch ist, sie verstößt gegen das Prinzip der Kapselung. Die Datenfelder der Klasse sind von außerhalb nämlich direkt zugänglich:

```
fotoApparat.bildStabilisiert = true;
fotoApparat.displayGroesse = 7.5;
fotoApparat.marke = "SoNie";
fotoApparat.megaPixel = 10;
```

Und *das* solltest du verhindern!

Warum das? Und wie kann ich's verhindern?

Moment, das sind ja gleich zwei Fragen.

Warum du es verhindern solltest: Der wichtigste Grund ist, dass dieser unbefugte Zugriff auf die Datenfelder dazu führen könnte, dass das Objekt einen **ungültigen Zustand** bekommt. Zum Beispiel so:

```
fotoApparat.megaPixel = -10; *1
```

*1 Das wäre nicht nur eine extrem schlechte Kamera, sondern auch extrem schlechter Programmierstil.

Jetzt zu deiner zweiten Frage: Wie kannst du **verhindern**, dass jemand von außen direkt auf die Objektvariablen zugreifen kann? Das ist recht einfach, du **versteckst** die Objektvariablen, und zwar indem du sie **private** machst:

```
private String marke;
private int megaPixel;
private double displayGroesse;
private boolean bildStabilisiert;
```

Jetzt kann niemand von außerhalb des Objekts auf die Werte zugreifen.

[Zettel]
Ach übrigens: Wird eine Objektvariable nicht direkt initialisiert, bekommt sie einen Standardwert: **null** für Variablen, die auf Objekttypen basieren, und die aus Kapitel 2 bekannten Werte für primitive Datentypen.

Setter und Getter

Gar nicht von außen zugreifen zu können, ist aber oft nicht wirklich wünschenswert. Manchmal möchtest du den Zustand eines Objekts auch ändern können, zum Beispiel wenn die Katze nicht mehr lieb ist.
Die Lösung ist recht einfach: Du erstellst einfach **Methoden**, die deine Datenfelder ändern (oder zurückgeben), so wie du das möchtest, und diese Methoden machst du dann öffentlich. Diese Art der Methoden nennt man umgangssprachlich **Setter** bzw. **Getter**, weil sie einen Wert setzen (set) oder zurückgeben (get).
Als Methodennamen wählt man daher `setWert()` bzw. `getWert()`, wobei „Wert" durch den Namen des Datenfeldes ersetzt wird. Also zum Beispiel so:

```
public void setMegaPixel(int megaPixel) { *1
  this.megaPixel = megaPixel; *2
}
public int getMegaPixel() { *3
  return megaPixel; *4
}
```

*1 So sieht eine Setter-Methode aus.

*2 Und das macht eine Setter-Methode typischerweise: den Wert zuweisen.

*3 So sieht eine Getter-Methode aus.

*4 Und die gibt typischerweise den entsprechenden Wert zurück.

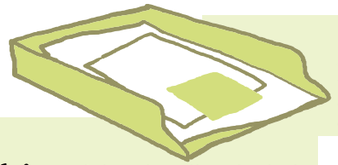
[Belohnung/Lösung]
Jetzt sind die Daten in deinem Objekt ordentlich **gekapselt** und können nur über Methoden geändert werden.

Moment mal, was hast du denn da gemacht? `this.megaPixel`? Was soll denn das `this` da? Das kann ich ja noch nicht mal aussprechen!

Egal, wie du es **aussprichst**, „siss", „sthsis" oder „dis", mit `this` **sprichst du das aktuelle Objekt** an. Du hast sicherlich gemerkt, dass wir eben im Setter zwei Variablen mit dem gleichen Namen (`megaPixel`) hatten: einmal den Parameter und einmal die Objektvariable. Damit der Compiler weiß, welche Variable du jeweils meinst, kannst du Objektvariablen referenzieren, indem du das `this` davor schreibst.

[Hintergrundinfo]
Das hat auch den Vorteil, dass du dir nicht immer einen neuen Namen für deine Parameter ausdenken musst, sowas ist also auch Programmierschule von gestern:

```
public void setMegaPixel(int pMegaPixel) {
  megaPixel = pMegaPixel;
}
```



[Ablage]
`pMegaPixel` (wobei das p für Parameter steht) oder ähnliche Namen sind ein weiteres Beispiel für **schlechte Variablennamen**.

[Ablage]
Weil sich `this` auf das aktuelle Objekt bezieht, kannst du es auch nur in **Objektmethoden** und in **Konstruktoren** verwenden. In Klassenmethoden geht das nicht, weil du dort kein Objekt zur Verfügung hast. Das wäre in etwa so, als ob du in der Fotoapparat-Fabrik mit der Fotoapparat-Schablone versuchen würdest, Fotos zu machen oder statt des Plätzchens die Ausstechform verputzen wolltest.

[Zettel]
Bei den Gettern kannst du auf das `this` verzichten. Findet der Compiler keine lokale Variable mit entsprechendem Namen, guckt er nämlich als Nächstes in die umgebende Klasse und sucht nach einer Objektvariablen mit dem Namen.

```
private int megaPixel;
public void setMegaPixel(int megaPixel) {
  this.megaPixel = megaPixel; *1
}
public int getMegaPixel() {
  return megaPixel; *2
}
```

*1 Hier musst du sagen, welches der beiden `megaPixel` du meinst.

*2 Hier gibt's nur ein `megaPixel`. Das heißt, du kannst das `this` auch weglassen, denn der Compiler findet selbstständig die Objektvariable.

Bei Booleschen Objektvariablen kannst du statt **get** auch **is** für den Getter nehmen, zum Beispiel so:

```
public boolean isBildStabilisiert() {
  return bildStabilisiert;
}
```

Die Setter-Methoden bleiben aber gleich, beginnen also mit **set**, nicht etwa mit **setIs**:

```
public void setBildStabilisiert(boolean bildStabilisiert) {
  this.bildStabilisiert = bildStabilisiert;
}
```

[Zettel]
Ich weiche bei den Settern und Gettern zwar auf der einen Seite ein bisschen von der Regel ab, keine denglischen Methodennamen zu verwenden, auf der anderen Seite ist aber durch den sogenannten **JavaBeans-Standard** auch festgelegt, immer set/get/is bei Settern und Gettern zu verwenden.

Aber zurück zu der Frage, warum Setter und Getter von Vorteil sind:
Du kannst zum Beispiel in der Setter-Methode die **Parameter validieren**, also überprüfen, ob der übergebene Parameter überhaupt gültig ist.

```
public void setMegaPixel(int megaPixel) {
  if(megaPixel >= 1 && megaPixel <= 20) { *2
    this.megaPixel = megaPixel; *1
  }
}
```

*1 Der neue Wert wird erst zugewiesen, …

*2 … nachdem er auf Gültigkeit überprüft wurde.

Hier noch eine Übersicht über die verschiedenen Variablentypen:

Variablentyp	Lebensdauer	Codebeispiel

lokale Variablen	für die Dauer des Code-Blocks	`// hier ist i noch gar nicht da`
		`for(int i=0; i<10; i++) {`
		`  // hier ist i sichtbar`
		`}`
		`// hier ist i nicht mehr sichtbar`

Objektvariablen	vom Erstellen des Objekts bis zu dem Zeitpunkt, an dem das Objekt nicht mehr referenziert wird	`public class FotoApparat {`
		`  private String marke;`
		`  …`
		`}`

Klassenvariablen	vom Laden der Klasse bis zu dem Zeitpunkt, an dem die Klasse nicht mehr geladen ist	`public class Konfiguration  {`
		`  public static int GROESSE = 10;`
		`  …`
		`}`

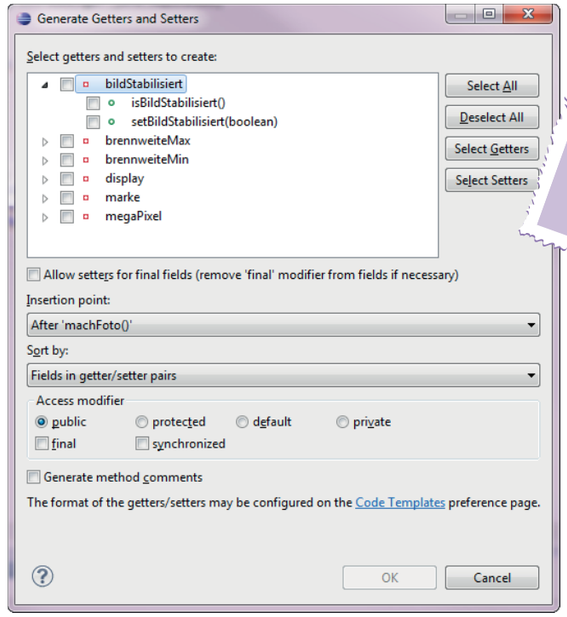
[Belohnung/Lösung]
So, das war ein langer Tag im Büro. Aber glaub mir, die Überstunden zahlen sich aus. Denn besser, du lernst **jetzt**, die Daten richtig in Objekten zu kapseln, als später, wenn dich das prozedurale Monster gepackt hat.

Der Fotoapparat unter der Lupe

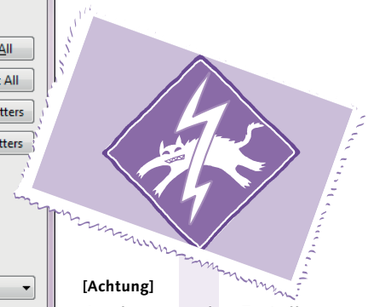
[Einfache Aufgabe]
Nimm die Klasse **`FotoApparat`**, und erstelle für alle Eigenschaften Getter und Setter.

Das schaffe ich, marke, so … uff, diese Getter- und Setter-Methoden zu schreiben, ist aber ganz schön anstrengend. Gibt's da nicht was in Eclipse?

Natürlich gibt's da was:
Source • Generate Getters and Setters.

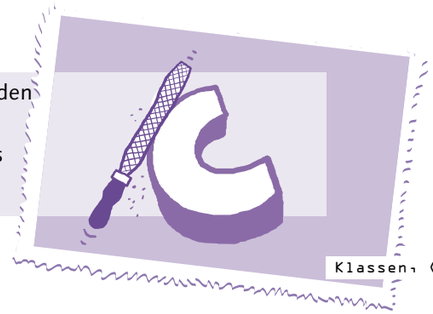


Mit Eclipse kannst du dir genau die Getter und Setter generieren lassen, die du brauchst.



[Achtung]
Auch wenn das Erstellen von Gettern und Settern damit sehr einfach ist, solltest du natürlich trotzdem immer überlegen, welche Objektvariablen du nach außen sichtbar machen möchtest.

[Code bearbeiten]
Passe jetzt die beiden generierten Methoden **`setBrennweiteMin()`** und **`setBrennweiteMax()`** so an, dass **`brennweiteMax`** nicht kleiner als **`brennweiteMin`** sein darf.



Hier die **Lösung**:

```
public void setBrennweiteMin(int brennweiteMin) {
  if(this.brennweiteMax >= brennweiteMin) { *1
    this.brennweiteMin = brennweiteMin;
  } else {
    System.err.println("Die minimale Brennweite muss kleiner gleich der maximalen
      Brennweite sein."); *2
  }
}
public void setBrennweiteMax(int brennweiteMax) {
  if(this.brennweiteMin <= brennweiteMax) { *1
    this.brennweiteMax = brennweiteMax;
  } else {
    System.err.println("Die maximale Brennweite muss größer gleich der minimalen
      Brennweite sein."); *2
  }
}
```

*1 Die minimale bzw. die maximale Brennweite darf nur gesetzt werden, wenn sie kleiner gleich bzw. größer gleich der gesetzten maximalen bzw. minimalen Brennweite ist, …

*2 … ansonsten gibt's eine Fehlermeldung. Normalerweise musst du hier mehr tun, als nur den Fehler auf die Konsole zu schreiben, zum Beispiel den **Fehler an den Aufrufer der Methode** schicken. Wie du das machst, zeige ich dir in Kapitel 9.

Setter mit mehreren Parametern

So, das ist schon mal besser als gar keine Validierung. Man kann das aber **noch besser** machen.
Das Problem ist nämlich Folgendes: Wenn du ein Objekt vom Typ **`FotoApparat`** erstellst, haben beide Objektvariablen **`brennweiteMin`** und **`brennweiteMax`** den Wert 0. Stell dir jetzt vor, du machst Folgendes:

```
fotoApparat.setBrennweiteMin(18);
fotoApparat.setBrennweiteMax(100);
```

Minimale Brennweite 18, maximale Brennweite 100, was ist das Problem?

Na ja, spiel das doch mal durch:

`setBrennweiteMin(18)` schlägt fehl, weil 18 nicht kleiner als der Defaultwert von **`brennweiteMax(0)`** ist. **`setBrennweiteMax(100)`** dagegen ist kein Problem. Danach hat deine Kamera also einen Brennweitenbereich von 0 bis 100 – nicht das, was man erwarten würde.

Erst wenn du die Setter-Aufrufe vertauschst, funktioniert's:

Wenn du erst **`setBrennweiteMax(100)`** aufrufst und dann **`setBrennweiteMin(18)`**, würden beide Aufrufe das machen, was man erwartet: Der Brennweitenbereich wäre 18–100.


```
fotoApparat.setBrennweiteMax(100);
fotoApparat.setBrennweiteMin(18);
```

Derjenige, der den **`fotoApparat`** bedient, muss also wissen, in welcher Reihenfolge die Methoden deiner Klasse aufgerufen werden müssen. **Dieses Wissen solltest du nicht voraussetzen**.
Besser geht es so: Du stellst nur einen Setter zur Verfügung, der beide Brennweiten entgegennimmt.

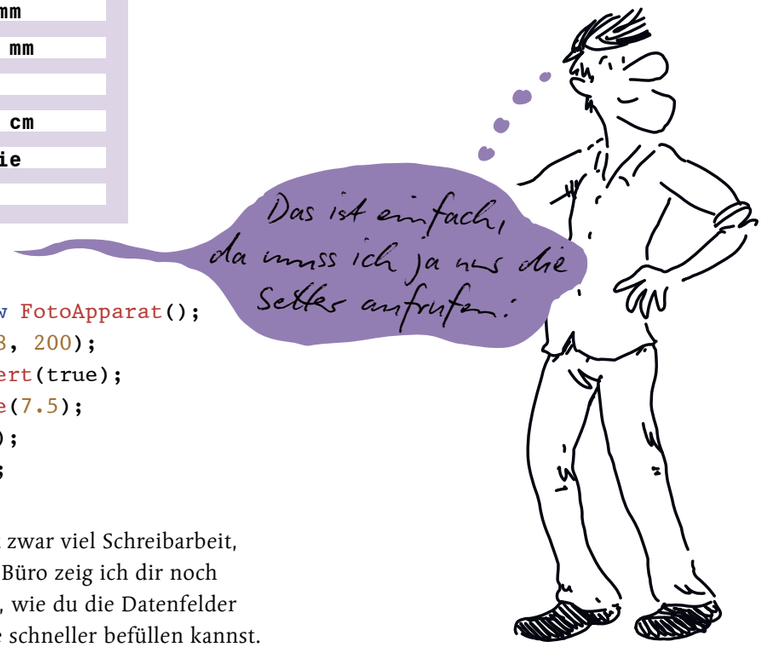
```
public void setBrennweiten(int brennweiteMin, int brennweiteMax) {
  if(brennweiteMin <= brennweiteMax) {
    this.brennweiteMin = brennweiteMin;
    this.brennweiteMax = brennweiteMax;
  } else {
    // hier die Fehlerausgabe
}
```

[Achtung]
Natürlich solltest du dann deine normalen Setter für die Brennweiten wieder löschen oder als **`private`** markieren.

Abschließend noch eine kleine einfache Übung:

[Einfache Aufgabe]
Erstelle eine Instanz von **`FotoApparat`** mit den folgenden Daten:

minimale Brennweite:	**18 mm**
maximale Brennweite:	**200 mm**
bildstabilisiert:	**ja**
Displaygröße:	**7.5 cm**
Marke:	**SoNie**
Megapixel:	**18**




```
FotoApparat fotoApparat = new FotoApparat();
fotoApparat.setBrennweiten(18, 200);
fotoApparat.setBildStabilisiert(true);
fotoApparat.setDisplayGroesse(7.5);
fotoApparat.setMarke("SoNie");
fotoApparat.setMegaPixel(18);
```

Perfekt! Ist zwar viel Schreibarbeit, aber gleich im Büro zeig ich dir noch ein paar Tricks, wie du die Datenfelder deiner Objekte schneller befüllen kannst.

Klassen vs. Objekte

[Einfache Aufgabe]
Erweitere die Kameras um die Angabe des Herstellungslandes. Welchen Typ von Variable brauchst du dafür? Lokale Variable? Klassenvariable? Objektvariable?

Eine lokale Variable bringt mir ja schon mal gar nichts, so ein Quatsch. Und eine Klassenvariable bringt auch nichts, weil ja das Herstellungsland von Kamera zu Kamera unterschiedlich sein kann. Ich nehme eine Objektvariable! Plus Setter und Getter, so wie hier:

```
public class FotoApparat {
  …
  private String herstellungsLand;
  public String getHerstellungsLand() {
    return herstellungsLand;
  }
  public void setHerstellungsLand(String herstellungsLand) {
    this.herstellungsLand = herstellungsLand;
  }
}
```

Okay, prima.
Auf zur nächsten Aufgabe:

[Einfache Aufgabe]
Wie sieht es denn aus, wenn **alle deine Kameras** auf jeden Fall eine Mindestbrennweite haben müssen und eine gewisse Maximalbrennweite nicht überschreiten dürfen? Wo würdest du das definieren?

Mmh, lass mal überlegen, wenn das für alle Kameras einheitlich sein soll, dann nehm ich doch am besten eine Klassenvariable, oder?

Genau, und je nachdem, ob du die Werte später ändern willst oder nicht, nimmst du eine Variable oder eine Konstante. Konstanten definierst du, wie du weißt, ganz einfach mit dem Schlüsselwort **`final`**:


```
public class FotoApparat {
  public static final int MIN_BRENNWEITE = 10;
  public static final int MAX_BRENNWEITE = 270;
  …
}
```

[Begriffsdefinition]
Datenfelder – egal, ob Objektvariablen oder Klassenvariablen –, die als **`final`** markiert sind, können nicht mehr verändert werden. Deswegen heißen sie auch **Konstanten**. Nicht verwechseln solltest du sie mit dem Schlüsselwort **`const`**, das du aus dem Schlüsselwort-ABC aus Kapitel 2 kennst: Das hat keinerlei Funktion!

[Notieren/Üben]
Nimm Klassenvariablen für Dinge, die allgemein für alle Instanzen der Klasse gelten, und Objektvariablen für Dinge, die sich von Objektinstanz zu Objektinstanz unterscheiden.

[Zettel]
Das dickste Buch der Welt ist 4,10 m dick und hat 50 560 Seiten. Ja, und um ein Buch zu sein, müssen schon mindestens 49 Seiten drin sein. Ansonsten ist es laut Definition etwas anderes, ein Prospekt, eine Broschüre, aber kein Buch.

[Einfache Aufgabe]
Erstelle eine Klasse **Buch** mit Eigenschaften für Titel, Autor, Seitenanzahl und der Angabe, ob es sich um ein gebundenes Buch handelt. Und sorge dafür, dass das Buch mindestens 49 Seiten hat und maximal 50 560 Seiten. (Hat hier jemand Klassenvariablen gesagt?) Und was kann man mit einem Buch typischerweise machen? Genau, lesen! Dann sieh auch dafür direkt eine Methode vor.

Schreibt du noch oder liest du schon?
Hier ein Ausschnitt aus der Lösung:

```
public class Buch {
  private static final int MAX_SEITENZAHL = 50560;
  private static final int MIN_SEITENZAHL = 49;
  private String autor;
  private String titel;
  private int seitenAnzahl;
  private boolean gebunden;
  // denk dir hier noch die ganzen Getter und Setter, die du sonst noch brauchst.
  public void setSeitenAnzahl(int seitenAnzahl) {
    if(seitenAnzahl >= MIN_SEITENZAHL && seitenAnzahl <= MAX_SEITENZAHL) {
```

```
      this.seitenAnzahl = seitenAnzahl;
    } else {
      // Fehler ausgeben
    }
  }
  public void lesen() {
    System.out.println(this.getTitel() + " von " + this.getAutor() + " wollte ich
      schon lange mal lesen. Los geht's.");
  }
}
```

Und noch eine Klasse, um das Ganze laufen zu lassen:

```
public class BuchMesse {
  public static void main(String[] args) {
    Buch buch = new Buch();
    buch.setTitel("Romeo und Julia");
    buch.setAutor("William Shakespeare");
    buch.setSeitenAnzahl(400);
    buch.setGebunden(false);
    buch.lesen();
  }
}
```

Sichtbarkeit von Variablen und Methoden

Die **Sichtbarkeitsmodifikatoren** `private` und `public` hast du ja bereits kennengelernt. Mit `private` markierte Variablen oder Methoden sind nur **innerhalb der Klasse** zugänglich, in der sie definiert sind, und in keiner anderen Klasse. Mit `public` markierte Variablen und Methoden sind von **überall** zugänglich. Neben diesen beiden gibt es in Java noch einen weiteren Sichtbarkeitsmodifikator, `protected`, sowie die Möglichkeit, den Sichtbarkeitsmodifikator ganz wegzulassen. In beiden Fällen sind die Variablen/Methoden für andere Klassen **innerhalb des gleichen Packages** sichtbar, aber nicht für Klassen in anderen Packages. Ausnahme: Im Fall von `protected` sind sie zusätzlich in Unterklassen sichtbar, auch wenn die sich in anderen Packages befinden. **Unterklassen** sind sozusagen nahe Verwandte einer Klasse, die ich dir im nächsten Kapitel genauer vorstellen werde. Lässt du den Modifikator ganz weg, sind die Variablen/Methoden nur im gleichen Package sichtbar. Man spricht dann auch von **package-private**.

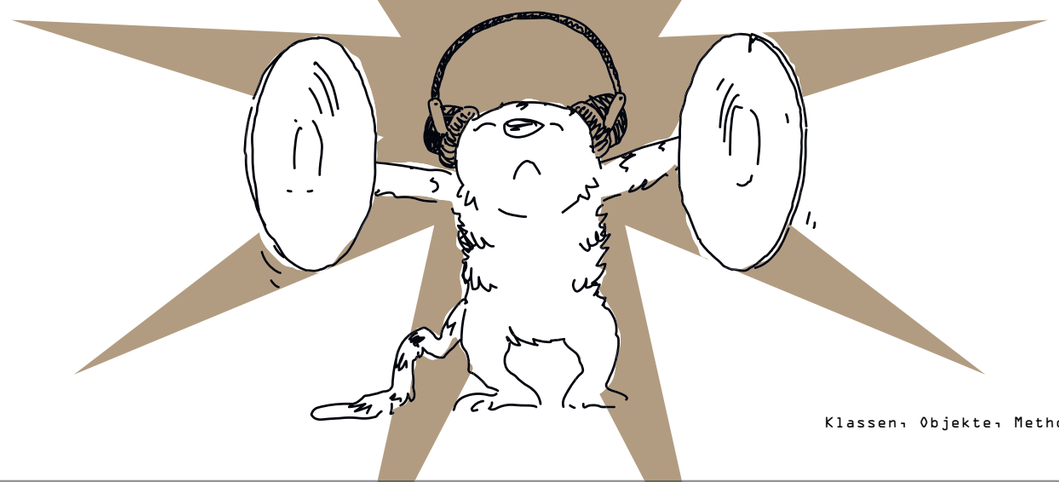
Zusammenfassend bedeutet das also:

Wenn du der Variablen oder der Methode den folgenden Wert gibst, ist sie in …	… der gleichen Klasse …	… einer anderen Klasse im gleichen Package …	… den Unterklassen im gleichen Package …	… Unterklassen in einem anderen Package …	… Nicht-Unterklassen in einem anderen Package …
`public`	… sichtbar.	… sichtbar.	… sichtbar.	… sichtbar.	… sichtbar.
`protected`	… sichtbar.	… sichtbar.	… sichtbar.	… sichtbar.	… **nicht** sichtbar.
kein Modifikator	… sichtbar.	… sichtbar.	… sichtbar.	… **nicht** sichtbar.	… **nicht** sichtbar.
`private`	… sichtbar.	… **nicht** sichtbar.	… **nicht** sichtbar.	… **nicht** sichtbar.	… **nicht** sichtbar.

[Zettel]
Beachte bitte auch die zusätzlichen Sichtbarkeitsregeln, die durch die in Java 9 eingeführten Module gelten. Nachlesen kannst du das in Kapitel 13.

[Achtung/Vorsicht]
Du kannst übrigens immer nur höchstens einen Sichtbarkeitsmodifikator verwenden.

[Notieren/Üben]
Verbirg die Variablen und Methoden deiner Klassen immer so gut wie möglich. Das bedeutet nicht, dass du alles `private` machst, dann kannst du ja kaum was Gescheites mit deinen Objekten machen. Aber du sollst eben auch nicht alles `public` machen. Überlege dir also vorher immer gut, wer wann wo welche deiner Methoden aufrufen muss, und wähle danach weise den Sichtbarkeitsmodifikator.



Konstruktoren

Gucken wir uns nochmal die Zeile an, mit der wir eben das **FotoApparat**-Objekt erstellt haben:

```
FotoApparat fotoApparat = new FotoApparat();
```

Hinter dem **new** steht ja der Aufruf des Konstruktors, das hatte ich schon gesagt. Nur, wo ist dieser Konstruktor? In unserer Klasse **FotoApparat** ist er ja nicht. Oder doch?

Okay, ich will dich nicht auf die Folter spannen. Er ist schon da, aber bisher nur implizit. Solange du nämlich **keinen eigenen Konstruktor** implementierst, legt der Compiler einen **Standardkonstruktor** für dich an. Das wäre dasselbe, wie zu schreiben:

```
public FotoApparat() {}*1
```

Sieht ein bisschen wie eine Methode aus.

Stimmt, die runden Klammern, die geschweiften Klammern, der Modifikator, aber das war's auch schon. Denn beim Konstruktor verzichtest du auf die Angabe eines Rückgabewertes wie bei einer Methodendeklaration. Denn der ist ja auch irgendwie klar: eine Objektinstanz der jeweiligen Klasse. Außerdem werden Konstruktoren im Gegensatz zu Methoden in **Upper Camel Case** geschrieben, weil sie **exakt so heißen müssen wie die Klasse**, zu der sie gehören.
Warum nun eigene Konstruktoren schreiben? Da gibt es **mindestens zwei gute Gründe**.

Fangen wir mit dem ersten an:

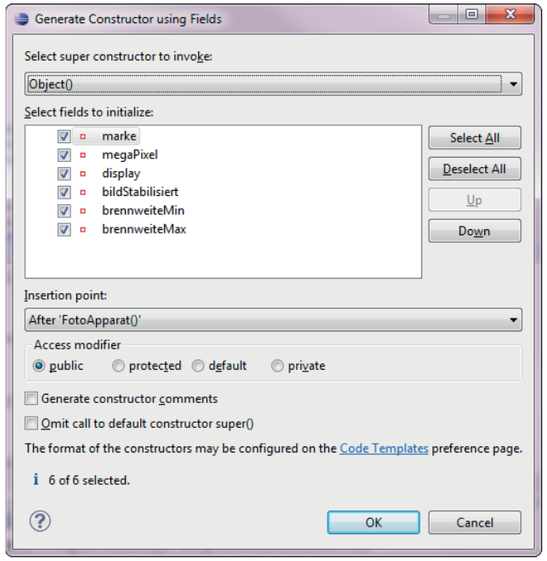
Im Konstruktor kannst du die **Objektvariablen initialisieren**. Du könntest also die Kamera von eben direkt mit Standardwerten vorbelegen.

```
public FotoApparat() {
  *1this.brennweiteMin = 18;
   this.brennweiteMax = 200;
   this.bildStabilisiert = true;
   this.displayGroesse = 7.5;
   this.marke = "SoNie";
  this.megaPixel = 18;
}
```

Damit wären standardmäßig alle Kameras gleich. Das kann in manchen Fällen ausreichen.
Wenn du allerdings die Werte bei der Erstellung deiner Objekte beeinflussen möchtest, brauchst du etwas anderes:

Der zweite gute Grund: Konstruktoren mit Parametern

Konstruktoren mit Parametern sind sinnvoll, wenn die Objektvariablen nicht mit Standardwerten, sondern aus den Parametern initialisiert werden sollen.
In Eclipse kannst du dir zu einer Klasse Konstruktoren mit Parametern ganz einfach generieren lassen: **Source • Generate Constructor using Fields**



*1 Ein Konstruktor kann genau wie Methoden auch Parameter haben.

Der Quelltext, den Eclipse generiert, sieht dann so aus:

```
public FotoApparat(*1String marke, int megaPixel, double displayGroesse,
boolean bildStabilisiert, int brennweiteMin, int brennweiteMax) {
  super();
 *2this.marke = marke;
  this.megaPixel = megaPixel;
  this.displayGroesse = displayGroesse;
  this.bildStabilisiert = bildStabilisiert;
  this.brennweiteMin = brennweiteMin;
  this.brennweiteMax = brennweiteMax;
}
```

*2 Hier solltest du das **this** nicht weglassen, ansonsten würdest du nicht die Objektvariable, sondern den Parameter ansprechen.

Was soll denn das **super()** da? Findet Eclipse es so toll, was ich da gemacht habe, oder was?

—ZWÖLF—

Programmierung mit Threads

Nicht den Faden verlieren

**Multitasking ist nur ein Gerücht?
Nicht bei uns.
Schrödinger lernt, wie er dank Java mehrere Dinge gleichzeitig tun kann. Nur muss er dabei einiges beachten, sonst gibt's ein heilloses Durcheinander.**

Prozesse und Threads

Mithilfe von Threads kannst du dein Programm dazu veranlassen, **mehrere Dinge gleichzeitig zu tun**. Statt von „gleichzeitig“ spricht man in diesem Zusammenhang auch von **„parallel“**. Bevor wir loslegen, müssen wir aber noch zwei Begriffe voneinander abgrenzen, die gerne miteinander verwechselt werden, nämlich **Prozess** und **Thread**. Wenn du zum Beispiel deine IDE startest oder deinen Browser oder irgendein anderes Programm, startet dein PC **einen neuen Prozess** für dieses Programm. Innerhalb eines solchen Prozesses kann es dann (mindestens) einen oder aber mehrere dieser Threads geben.

Threads sind also vergleichbar mit Arbeitern, die irgendetwas mehr oder weniger Produktives tun. Das Ziel bei der Verwendung **mehrerer Threads** (auch **Multithreading** oder **Nebenläufigkeit** genannt) ist es, eine höhere Performance innerhalb des Programms zu erreichen. Der Prozess wäre dann in dieser Analogie der Arbeitgeber.

Stell dir vor, deine Freundin und du, ihr erwartet Besuch, und ihr müsst vorher noch dringend eure Wohnung aufräumen. Wenn deine Freundin das ganz alleine machen müsste, bräuchte sie natürlich länger, als wenn du ihr dabei helfen würdest.

Na ja, eigentlich drücke ich mich ja eher vor der Hausarbeit.

Hausarbeit? Schrödinger, wir sind hier nicht in der Schule. Es heißt „Haushalt“. Aber ich merke schon, das scheint dir wirklich fremd zu sein. Wie auch immer, um die Analogie zu den Threads wieder hinzubekommen: Du und deine Freundin, ihr wärt jeweils Threads, die beide parallel arbeiten, so dass die Arbeit schneller erledigt wird. Wobei man hier ein bisschen einschränken muss: Im Gegensatz zu **Mehrprozessorsystemen**, innerhalb derer wirklich parallel mehrere Threads laufen können, würde das Ganze auf Computersystemen **mit nur einem Prozessor nicht wirklich parallel ablaufen**. Vielmehr würde dort immer nur ein Thread zu einem Zeitpunkt ausgeführt. Um aber dem Benutzer den Eindruck zu vermitteln, dass trotzdem alles „parallel“ geschieht, wird dort (mehr oder weniger schnell) zwischen den einzelnen Threads hin- und hergewechselt.

Bei eurer Aufräumaktion würde das also so aussehen, dass du zum Beispiel zuerst ein kleines bisschen die Fussel zusammenkehrst, dann pausierst, während deine Freundin wegen mir schon mal den Lappen in den Eimer schmeißt, dann selbst pausiert, währenddessen du wieder ein kleines bisschen Fussel zusammenkehrst und so weiter … und eben ganz, ganz schnell hintereinander, so dass es für einen Außenstehenden nicht wie eine moderne Tanzart, sondern wie eine flüssige Bewegung aussieht.

Aber auch auf Einprozessorsystemen gibt es gewisse Dinge, auf die sich die Verwendung mehrerer Threads positiv auswirken würde, immer dann nämlich, wenn auf **unterschiedliche Ressourcen** zugegriffen wird, von denen nicht alle die **volle Prozessorleistung** benötigen. Zum Beispiel könnte ein Thread einen Download aus dem Internet starten und (während der Thread dann wartet und keine Prozessorleistung verbraucht) ein anderer Thread die grafische Nutzeroberfläche aufbauen. Die einzelnen Arbeiten der beiden Threads würden also **asynchron** ablaufen.

Also ungefähr so, wie wenn ich mir einen Kaffee aufsetze und dann, während der Kaffee durchläuft, WoW spiele?

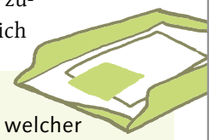
Genau. Du wärst sozusagen der Prozessor. Kaffeemaschine und Computer wären Threads, na ja, oder so ähnlich halt.

Allerdings werden Threads nicht zwangsweise immer der Reihe nach **abwechselnd** ausgeführt, sondern sie werden nach bestimmten **Kriterien ausgewählt.** Das Ding, das für diese Auswahl zuständig ist, nennt sich **Scheduler**. Dieser Scheduler sagt nämlich, wann welcher Thread arbeiten soll und wann nicht.

Klingt ein bisschen nach Bossingen … der will doch immer mir die Überstunden aufzwingen. Von Gerechtigkeit keine Spur.

Na ja, ganz so gemein ist der Java-Scheduler nicht. Damit eben nicht immer nur ein einziger Thread ausgewählt wird, sondern jeder Thread mal drankommt, gibt es verschiedene **Scheduling-Strategien**. Zu den bekannteren Strategien zählen zum Beispiel die **Round-Robin-Strategie**, bei der reihum zwischen den Threads gewechselt wird, oder die **Shortest-Job-First-Strategie**, bei der Threads, die am wenigsten Zeit für ihre Aufgabe benötigen, zuerst ausgewählt werden. Dabei muss die Bearbeitungszeit der jeweiligen Aufgabe natürlich im Vorfeld bekannt sein.

Java verwendet das sogenannte **Prioritäts-Scheduling**. Dabei wird jedem Thread eine Priorität zugeordnet, Threads mit höherer Priorität werden dann bevorzugt ausgewählt.



[Ablage]
Der **Scheduler** bestimmt, welcher Thread als nächster ausgewählt wird. Die Kriterien, nach denen er dabei vorgeht, werden als **Scheduling-Strategie** bezeichnet.

[Zettel]
Wie du selber die Priorität eines Threads bestimmen kannst, gucken wir uns gleich an.

Aber genug der Theorie. Über Multithreading kann man sicherlich ganze Vorlesungen halten. Aber du willst ja auch was **Praktisches** lernen, nicht wahr?

Der erste Thread

Um Threads zu erstellen, gibt es zwei Möglichkeiten: Entweder du erstellst eine Klasse, die das **Interface `java.lang.Runnable`** implementiert, oder du erstellst eine **Unterklasse** von **`java.lang.Thread`**. **`java.lang.Thread`** implementiert übrigens **`java.lang.Runnable`**.

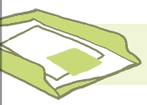
Wenn du die Lösung mit dem Interface nimmst, musst du die Methode **`run()` implementieren**, wenn du die Unterklassenvariante nimmst, solltest du die existierende **`run()`**-Methode von **`Thread` überschreiben**, damit dein Thread überhaupt was macht.

```
public class MachWas extends Thread*1 {
  @Override
  public void run()*3 {
    System.out.println("Ich mach was!");
  }
}
public class MachAuchWas implements Runnable*2 {
  @Override
  public void run()*3 {
    System.out.println("Ich mach auch was!");
  }
}
```

*1 Entweder du leitest von der Klasse **`Thread`** ab, …

*2 … oder du implementierst das Interface **`Runnable`**.

*3 In beiden Fällen kommt der Code, der in dem Thread ausgeführt werden soll, in die **`run()`**-Methode.



[Ablage]
Vorzugsweise solltest du die Variante mit dem Interface wählen. Damit bleibst du flexibler und kannst, wenn du möchtest, auch von einer anderen Klasse ableiten. Noch geläufiger ist es sogar, von dem **`Runnable`**-Interface eine **anonyme Klasse** zu erstellen.

Um einen Thread zu starten, brauchst du immer ein `Thread`-Objekt:

```
public static void main(String[] args) {
  new MachWas().start();*1
  (new Thread(new MachAuchWas())).start();*2
}
```

*1 Wenn deine Klasse von **`Thread`** ableitet, kannst du direkt auf ihr die **`start()`**-Methode aufrufen, …

*2 … ansonsten „wrappst" du deine **`Runnable`**-Implementierung mit einem **`Thread`**-Objekt und rufst auf letzterem die **`start()`**-Methode auf.

Lass das Programm ruhig ein paar Mal laufen. Fällt dir was auf?

Moment mal, mal kommt zuerst „Ich mach was!" und dann „Ich mach auch was!", mal genau andersrum.

Genau! Das ist der Scheduler in Aktion! Mal wird der eine, mal der andere Thread zuerst ausgewählt.

[Achtung]
Keine Sorge, falls die Ausgabe immer gleich ist. Das kann auch sein. Unwahrscheinlich, aber möglich, theoretisch zumindest.

[Hintergrundinfo]
Der Ober-Thread ist übrigens immer der Thread „main", also der Teil des Programms, der die **`main`**-Methode ausführt.

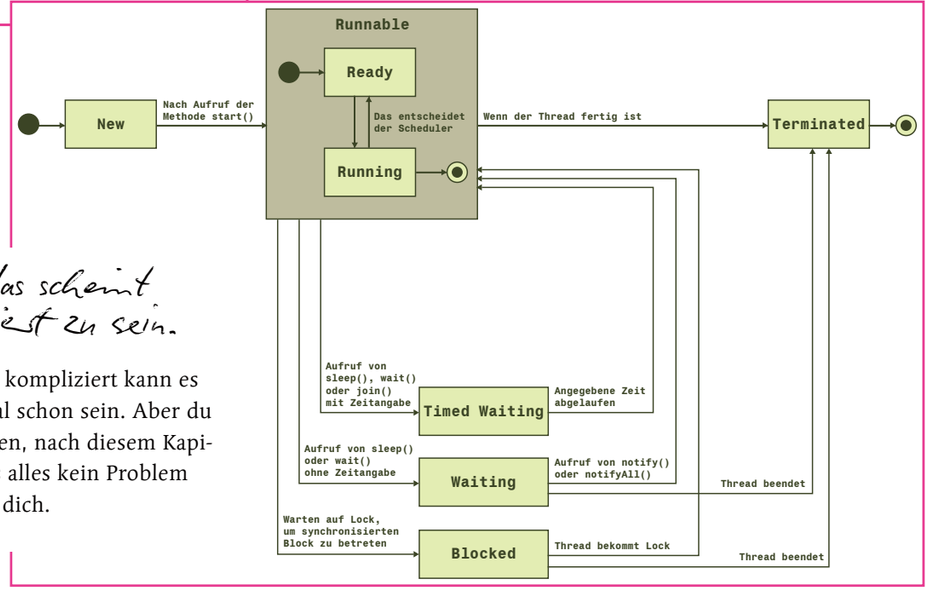
Night of the living thread

Lass mich dir zeigen, was eben im Hintergrund alles passiert ist, als du die **`Thread`**-Objekte erzeugt und die **`start()`**-Methode auf ihnen aufgerufen hast. Während seiner Lebensspanne nimmt ein Thread nämlich **verschiedene Zustände** an. Manche davon freiwillig, manche davon erzwungenermaßen. In dem folgenden Zustandsdiagramm …

Lädt …

… siehst du … gar nichts. Mmh, wahrscheinlich ist der Thread für das Laden der Bilder noch nicht fertig, warten wir mal noch ein bisschen …
ah, jetzt, da ist es …

Thread-Zustandsdiagramm



Uff, das scheint kompliziert zu sein.

Na ja, kompliziert kann es manchmal schon sein. Aber du wirst sehen, nach diesem Kapitel ist das alles kein Problem mehr für dich.

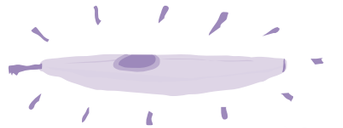
[Begriffsdefinition]
Was du hier siehst, ist ein UML-**Zustandsdiagramm**. Die Rechtecke stellen die **Zustände** dar, die Pfeile symbolisieren **Zustandsübergänge** von einem in einen anderen Zustand. Der Kreis am Anfang ist der **Startzustand**, der Kreis, in dem noch ein kleinerer Kreis ist, ist der **Endzustand**.

Was ist also gerade passiert? Mit dem Aufruf von `new MachWas().start()` wechselt der Thread zweimal seinen Zustand. Mit `new MachWas()` erreicht er den Zustand **New**, arbeitet aber noch nicht. Durch den Aufruf von `start()` gelangt der Thread in den Zustand **Ready** und ist sozusagen arbeitswillig. Arbeiten tut er aber immer noch nicht. Dazu muss er erst vom Scheduler ausgewählt werden, erst dann wechselt der Thread in den Zustand **Running**. Zum Schluss, wenn der Thread mit dem, was er machen soll, fertig ist, ändert er seinen Zustand zu **Terminated** oder auch **Dead**.

[Hintergrundinfo]
Über die Methode `yield()` kannst du einen Thread, der sich im Zustand Running befindet, dazu veranlassen, in den Zustand Ready zu wechseln, so dass der Scheduler erneut einen (anderen) Thread auswählen kann.

Damit hätten wir schon mal vier der sieben Thread-Zustände (Start- und Endzustand mal ausgenommen) abgefrühstückt. Der Rest kommt später.

Das war gerade noch ungerade



Wie so oft bei der Programmierung ist es auch bei den Threads so, dass du am besten lernst, indem du dir selber die Hände schmutzig machst.

[Schwierige Aufgabe]
Erstelle zwei Threads. Der eine soll nur die geraden Zahlen bis 100, der andere nur die ungeraden Zahlen bis 100 ausgeben.

Ich bin ja nicht so, ich helfe dir. Eine Musterlösung würde ungefähr so aussehen (auch wenn wir mit Threads arbeiten, wollen wir ja nicht die guten Tugenden eines OO-Entwicklers vergessen und fangen mit einem Interface an):

```
public interface Zahlendrucker {
  void druckeZahl(int zahl);
}
```

*1 Ein Interface, für das du zwei Implementierungen schreibst, einen `GeradeZahlenDrucker` und einen `UngeradeZahlenDrucker` ...

... und als Zwischenstufe noch eine **abstrakte Klasse**, die schon einen Teil der Arbeit übernimmt:

*1 Du implementierst das `ZahlenDrucker`-Interface und `Runnable`, damit du den Zahlendrucker auch starten kannst.

```
public abstract class AbstractZahlenDrucker implements Runnable,
  ZahlenDrucker {
  private int grenze;
  public ZahlenDrucker(int grenze) {
    this.grenze = grenze;
  }
  @Override
  public void run() {
    for(int i=0; i<=this.grenze; i++) {
      if(this.akzeptiereZahl(i)) {
        this.druckeZahl(i);
      }
    }
  }
  protected abstract boolean akzeptiereZahl(int zahl);
  @Override
  public void druckeZahl(int zahl) {
    System.out.println(zahl);
  }
}
```

*2 Die `run()`-Methode muss natürlich auch implementiert werden. Hier passiert schließlich die Äktschion! Egal, ob man ungerade oder gerade Zahlen drucken will, der Inhalt der `run()`-Methode ist in beiden Fällen komplett gleich. Deswegen bist du hier mit einer abstrakten Oberklasse am besten bedient.

*3 `grenze` gibt die Anzahl der zu druckenden Zahlen an.

*4 Ob eine Zahl gedruckt werden soll oder nicht, hältst du noch abstrakt, die Entscheidung trifft jeweils die Unterklasse.

*5 Ja, hier wird die Zahl gedruckt. Was dachtest du denn?

Das sieht doch wieder schwer nach diesem Template-Method-Entwurfsmuster aus.

[Code bearbeiten]
Bis hierhin mitgekommen? Dann kannst du die beiden Implementierungen bestimmt selbst schreiben.

Das geht, ja. Beim **GeradeZahlenDrucker** muss ich bei **akzeptiereZahl()** einfach für gerade Zahlen ein **true** zurückgeben. Das mach ich mit Modulo!

```
protected boolean akzeptiereZahl(int zahl) {
  return zahl%2==0;
}
```

Hey, endlich mal eine Verwendung dafür! Beim **UngeradeZahlenDrucker** dann umgekehrt.

Prima! Dann hätt ich jetzt noch gerne den Inhalt der **main**-Methode, die alles startet.

Bitteschön:

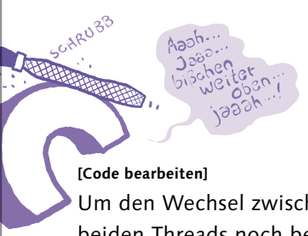
```
int grenze = 100;
(new Thread(new GeradeZahlenDrucker(grenze))).start();
(new Thread(new UngeradeZahlenDrucker(grenze))).start();
```

Aber hey, hab's gerade mal laufen lassen. Die Zahlen werden ja gar nicht der Reihe nach ausgegeben! O-Ton Eclipse: **… 89, 94, 91, 96, 93, 98 …**

Genau! Die ungeraden und die geraden Zahlen werden zusammen nicht in richtiger Reihenfolge ausgegeben, wohl aber jeweils für sich in richtiger Reihenfolge. **Innerhalb eines Threads** sind die Zahlen also in der **richtigen Reihenfolge**, bei **Kombination mehrerer Threads** ist die **Reihenfolge dagegen nicht garantiert**. Das ist so wegen des Schedulers.

Deshalb schreib dir hinter die Ohren:

[Notiz]
Wann welcher Thread arbeitet, bestimmt allein der Scheduler.



[Code bearbeiten]
Um den Wechsel zwischen den beiden Threads noch besser zu veranschaulichen, kannst du auch vor dem Aufruf von **druckeZahl()** ein **Thread.sleep()** setzen. Damit sagst du einem Thread explizit, dass er eine angegebene Zeit warten soll. 50 Millisekunden sollten reichen, um recht schön den Wechsel von ungeraden zu geraden Zahlen und zurück auf der Konsole zu sehen. **Thread.sleep()** gucken wir uns gleich im Detail an.

[Achtung]
Eventuell musst du **grenze** höher ansetzen (1 000 000 ist ganz gut). Ansonsten kann es sein, dass das Programm schneller fertig ist, als der Scheduler gucken kann und erst alle ungeraden Zahlen gedruckt werden und anschließend alle geraden Zahlen. Oder andersrum.

[Notiz]
Als kleine Hilfe, um zu sehen, welcher Thread was macht, kannst du dir jederzeit den Namen des aktuell ausgeführten Threads ausgeben lassen. Zugegeben, bei dem Zahlenbeispiel von eben gibt das wenig Sinn, falls du es aber trotzdem mal brauchst: **Thread.currentThread().getName()**.

Da krieg ich Zustände

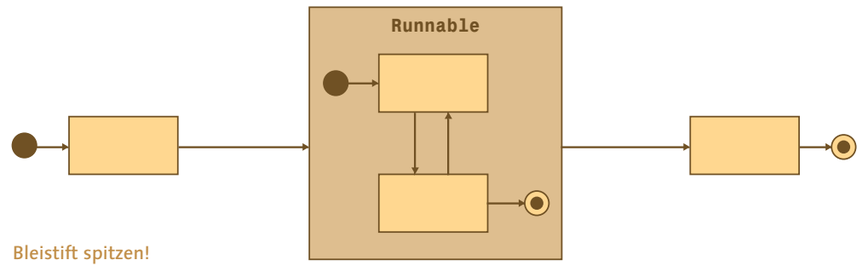
So, Schrödinger, Threads sind, zugegeben, nicht ganz einfach.
Deswegen eine kurze Zwischenprüfung über die Themen bis hierhin:

[Einfache Aufgabe]
Auf welche verschiedenen Arten kannst du einen Thread implementieren, und welche solltest du verwenden?

Ich kann entweder das **Runnable**-Interface implementieren, oder ich leite von der Klasse **Thread** ab. Am besten nehme ich aber das Interface, dann bleibt meine „Klassenhierarchie flexibler und sauberer". So zumindest hättest du das gesagt.

Richtig, du klingst manchmal echt schon wie ich …
Also, damit du einen Thread überhaupt starten kannst, brauchst du auf jeden Fall ein **Thread**-Objekt. Die Logik, die von diesem **Thread**-Objekt angestoßen wird, solltest du allerdings in eine Klasse auslagern, die das **Runnable**-Interface implementiert, oft reicht dazu eine lokale oder anonyme Klasse.

[Einfache Aufgabe]
Wir haben bisher vier Thread-Zustände besprochen. Wie hängen diese Zustände zusammen? Wie gelangt ein Thread von welchem Zustand in welchen anderen Zustand?



Bleistift spitzen!

So, alles gewusst? Die vier Zustände sind **New**, **Ready**, **Running** und **Terminated**. In den **New**-Zustand gelangt der Thread, nachdem der Konstruktor aufgerufen wurde. Von **New** nach **Ready** gelangt ein Thread durch den Aufruf der **start()**-Methode. Den Wechsel zwischen **Ready** und **Running** und zurück bestimmt der Scheduler. Und nach Vollenden der **run()**-Methode ist ein Thread **Terminated**.

[Notieren/Üben]
Über die Methode **getState()** kannst du dir den aktuellen Zustand eines Threads ausgeben lassen. Als Rückgabe bekommst du einen Enum-Wert, der den jeweiligen Zustand symbolisiert.

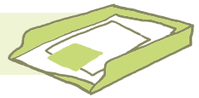
Threads schlafen legen

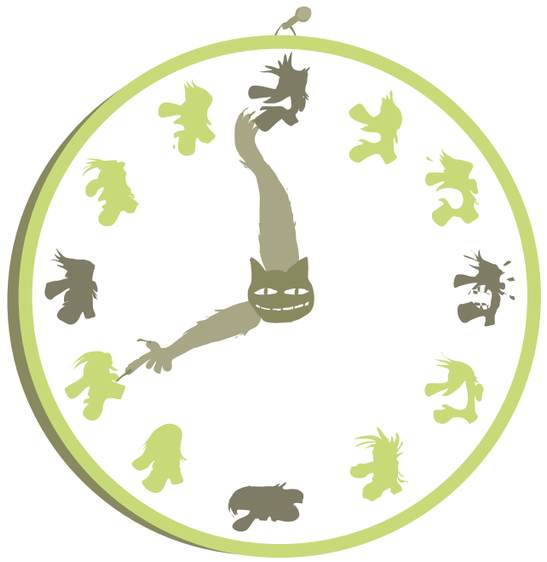
Wenn ein Thread mal überarbeitet ist oder auch aus purer Unlust an der Arbeit, kann er über die Methode **`sleep()`** eine bestimmte Anzahl an Millisekunden schlafen gelegt werden, das hatte ich ja eben schon kurz erwähnt. Mit dem Aufruf von **`sleep()`** gelangt der Thread in den Zustand **Timed Waiting**. Aus diesem Zustand kann er dann nur wieder in den **Ready**-Zustand wechseln, und zwar entweder, wenn die angegebene Zeit abgelaufen ist, oder, wenn der Thread beim Schlafen unterbrochen wurde. Das wirst du vielleicht auch von zu Hause kennen, wenn du dir abends den Wecker stellst …

… ja, stimmt, ich stell ihn immer auf 8:00 Uhr, aber meistens weckt mich die Katze schon vorher, weil sie meine Füße für zwei Wollballen hält …

[Ablage]

Schlafen kann ein Thread nur, wenn er sich vorher im **Running**-Zustand befunden hat. Wenn ein schlafender Thread aufwacht, gelangt er in den **Ready**-Zustand.





Helden, aufgepasst!

Multithreading ist ein komplexes Thema, und ich habe leider nur ein paar Seiten Zeit, dir das Ganze zu erklären. Damit du den Stoff besser verdauen kannst, lass mich dir eine Geschichte erzählen, und zwar die von zwei Helden, dem Zwerg und dem Nachtelfen, die zusammen gegen eine Gruppe von Orks in den Kampf ziehen wollen. Sollte ja kein großes Problem sein, denken sich die beiden, schließlich sie sind ja **zu zweit**.

Doch wenn zwei oder mehr Helden was Produktives zusammen machen wollen, will das Vorgehen gut geplant sein; genauso, wenn mehr als ein Thread im Spiel ist.

Um mit der Geschichte nicht den Bezug zu Java zu verlieren, darf der entsprechende Quelltext natürlich nicht fehlen. Die Basisklasse für die beiden Helden ist die abstrakte Klasse Held:

```
public abstract class Held { *1
  private String name;
  public Held(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public abstract void aufInDenKampf(Held held); *2
}
```

*1 Die Basisklasse für unsere Helden ist erstmal noch ganz simpel mit einem Datenfeld, einem Getter und …

*2 … einer abstrakten Methode **`aufInDenKampf()`**, deren Parameter der jeweils andere Held ist, mit dem in den Kampf gezogen werden soll.

Die konkrete Unterklasse für den Nachtelfen sieht dann so aus:

Die abstrakte Basisklasse

```
public class Nachtelf extends Held {
  public Nachtelf(String name) {
    super(name);
  }

  @Override
  public void aufInDenKampf(Held held) { *1
    System.out.printf("%s: Auf geht's! In den Kampf!%n", this.getName());*2
  }
}
```

*1 Die Implementierung der **`aufInDenKampf()`**-Methode. Der Elf hat's eilig, er will direkt in den Kampf.

*2 Ach so, ja, das hier kann ich dir hier schon mal erklären. Das „Formatierungskapitel" haben wir zwar noch vor uns, aber so viel schon mal vorweg: Mit der **`printf()`**-Methode kannst du Strings formatieren, hier zum Beispiel wird das **`%s`** in dem String durch den Wert ersetzt, der von **`this.getName()`** zurückgegeben wird, und für das **`%n`** wird eine neue Zeile begonnen. Praktisch oder?

Und die Unterklasse für den Zwerg sieht so aus:

```
public class Zwerg extends Held {
  public Zwerg(String name) {
    super(name);
  }
  @Override
  public void aufInDenKampf(Held held) { *1
    System.out.printf("%s: \"Auuuuufstäääääähnnnn!!!!\"%n", held.getName());
    try {
      Thread.sleep(5000); *2
    } catch (InterruptedException exception) {
    }
    System.out.printf("%s (etwas verspätet): Gääääähn, uff, guten Morgen, jaja ↵
      ich komme schon.%n", this.getName()); *3
  }
}
```

*1 Auch der Zwerg hat seine eigene Vorstellung davon, wie `aufInDenKampf()` aussehen muss, …

*2 … allerdings hat er es nicht ganz so eilig wie der Nachtelf. Mit `Thread.sleep()` wird hier nachgebildet, dass er erstmal 5 Sekunden wartet, bevor er …

*3 … dann endlich auch soweit ist und mit in den Kampf zieht.

[Einfache Aufgabe]

Wie kannst du die beiden Helden in separaten Threads in den Kampf schicken?

Bestimmt nicht, indem ich **Zwerg** und **Nachtelf** von **Thread** ableiten lasse. Das geht ja nicht, die erben ja schon von **Held**. **Held** von **Thread** ableiten zu lassen, wäre genauso doof. Bleibt mir nur, das Interface **Runnable** zu implementieren … am besten in einer anonymen Klasse. *Das war's!*

Genau, in etwa so wie hier in der Musterlösung:

```
public static void main(String[] args) {
  final Zwerg zwerg = new Zwerg("Zwerg"*3);
  final Nachtelf nachtelf = new Nachtelf("Nachtelf"*3);
  new Thread(new Runnable() { *1
    @Override
    public void run() {
      zwerg.aufInDenKampf(nachtelf); *2
    }
  }).start();
  new Thread(new Runnable() { *1
    @Override
    public void run() {
      nachtelf.aufInDenKampf(zwerg); *2
    }
  }).start();
}
```

*1 Zwei **Thread**-Objekte, damit du die Threads starten kannst. Jedes davon bekommt eine anonyme Implementierung von **Runnable**, …

*2 … in der die beiden Helden in den Kampf geschickt werden.

*3 Hier sind sie, die beiden namenlosen Helden.

*4 Und nicht vergessen: Damit innerhalb der anonymen Klasse überhaupt auf die beiden lokalen Heldenvariablen zugegriffen werden kann, müssen diese **final** sein.

Die Ausgabe wird (sehr, sehr wahrscheinlich immer) die folgende sein:

```
Nachtelf: "Auuuuufstäääääähnnnn!!!!"
Nachtelf: "Auf geht's! In den Kampf!"
Zwerg (etwas verspätet): "Gääääähn, uff, guten Morgen, jaja ich komme schon."
```

So weit, so klassisch. Allerdings ist der Nachtelf in dem obigen Code ungeduldig und zieht, **ohne auf den Zwerg zu warten**, alleine in den Kampf. Keine gute Idee, denn als er auf eine Gruppe von vier Orks trifft, von denen er eins auf die Mütze bekommt, ist er über seine voreilige Entscheidung nicht mehr so glücklich. **Nächstes Mal, beschließt er, warte ich auf den Zwerg, egal, wie lange das dauert.**

Auf andere warten

Damit unser Elf also nicht alleine in den Kampf zieht, müssen wir ihm irgendwie beibringen, auf den Zwerg zu warten. In Java übersetzt, bedeutet das, dass der Thread, in dem **`nachtelf.aufInDenKampf()`** aufgerufen wird, auf den Thread warten muss, in dem **`zwerg.aufInDenKampf()`** aufgerufen wird. Das funktioniert, indem du im Elfen-Thread auf dem Zwergen-Thread die Methode **`join()`** aufrufst. Damit wartet der Elfen-Thread mit seiner Arbeit so lange, bis der Zwergen-Thread mit dem, was er tut, fertig ist.

So wartet der Elf auf den Zwerg:

[Hintergrundinfo]
`join()` ohne Parameter sorgt dafür, dass der Thread in den Zustand **Waiting** wechselt. Die Methode gibt es aber auch noch in der Variante, in der du über einen Parameter sagen kannst, wie lange gewartet werden soll. Dann wechselt der Thread in den Zustand **Timed Waiting**.

```
public static void main(String[] args) {
  final Zwerg zwerg = new Zwerg("Zwerg");
  final Nachtelf nachtelf = new Nachtelf("Nachtelf");
  final Thread zwergenThread = new Thread(new Runnable() { *1
    @Override
    public void run() {
      zwerg.aufInDenKampf(nachtelf);
    }
  });
  final Thread elfenThread = new Thread(new Runnable() { *1
    @Override
    public void run() {
      try {
        zwergenThread.join(); *2
      } catch (InterruptedException exception) { *3
        // Exception-Behandlung hier
      }
      nachtelf.aufInDenKampf(zwerg); *4
    }
  });
  elfenThread.start(); *5
  zwergenThread.start(); *5
}
```

*1 Die Threads werden genauso erzeugt wie eben. Allerdings werden die erzeugten Objekte in den zwei Variablen **`zwergenThread`** und **`elfenThread`** gespeichert. Zumindest die Variable **`zwergenThread`** brauchst du, damit du gleich unten im Elfen-Thread darauf zugreifen kannst.

*2 Mit **`join()`** sagst du, dass **`elfenThread`** auf den **`zwergenThread`** warten sollt.

*3 **`join()`** wirft übrigens eine **`InterruptedException`**.

*4 Das hier wird erst ausgeführt, wenn der **`zwergenThread`** fertig ist.

*5 Ja, und nicht vergessen, die beiden Threads auch zu starten.

Die Ausgabe ist jetzt auf jeden Fall diese hier:

```
Nachtelf: "Auuuuufstäääääähnnnn!!!!"
Zwerg: (etwas verspätet): "Gääääähn, uff, guten Morgen, jaja, ich komme schon."
Nachtelf: "Auf geht's! In den Kampf!"
```

Ende gut, alles gut? Na ja, fast. Den Kampf haben die beiden gemeistert, die vier Orks sind jetzt mindestens ebenso viele Köpfe kürzer. Allerdings ist das Abenteuer noch nicht zu Ende. Wie die Geschichte weitergeht, erfährst du gleich. Aber erst muss ich dir noch was anderes erklären.

Synchronisierung

Ich hatte dir eben ja schon gesagt, dass **Threads standardmäßig asynchron arbeiten.** Die Reihenfolge, in der die einzelnen Threads durch den Scheduler ausgewählt werden, und vor allem der Zeitpunkt, wann der Scheduler einen Thread auswählt, sind also nicht garantiert. Dies kann in manchen Fällen zu Problemen führen. Ruf dir als Beispiel nochmal die Klasse **`MusikAbspielGeraet`** aus Kapitel 7 in Erinnerung.

Was? Wie? Wo? Welche Seite?

Das kenn ich gut, deswegen hier der relevante Quelltext, damit du nicht blättern und suchen musst:

```
public final void hoeren(Tontraeger tontraeger) {
  if(this.unterstuetztTontraeger(tontraeger)) {
    this.tontraeger = tontraeger;*1
    this.einlegen(this.tontraeger);*2
    this.abspielen(this.tontraeger);*2
  } else {
    …
  }
}
public Tontraeger auswerfen() {
  Tontraeger tontraeger = this.tontraeger;
  this.tontraeger = null;*3
  return tontraeger;
}
```

*1 Hier wird die Objektvariable **`tontraeger`** auf den Wert der übergebenen Objektreferenz gesetzt, …

*2 … anschließend werden die Methoden **`einlegen()`** und **`abspielen()`** mit der Objektvariablen aufgerufen, …

*3 … und beim Auswerfen wird die Objektvariable wieder auf **`null`** gesetzt.

Das alles funktioniert prima, **wenn nur ein Thread im Spiel ist**, denn dann kann zuerst **`hoeren()`** und anschließend **`auswerfen()`** aufgerufen werden. Oder umgekehrt erst **`auswerfen()`** und dann **`hoeren()`**, wobei das sicherlich wenig Sinn machen würde. Beides würde aber prinzipiell ohne Fehler funktionieren. Wenn aber zwei Threads mit der gleichen **`MusikAbspielGeraet`**-Objektinstanz arbeiten, und in dem einen Thread **`hoeren()`** aufgerufen wird und in dem anderen **`auswerfen()`**, ...

```
final MusikAbspielGeraet plattenSpieler*2 = new SchallplattenSpieler();
(new Thread(new Runnable() {*1
  @Override
  public void run() {
    plattenSpieler*2.hoeren(new Schallplatte("The Doors"))*3;
  }
}, "Hörer")).start();
(new Thread(new Runnable() {*1
  @Override
  public void run() {
    plattenSpieler*2.auswerfen()*4;
  }
}, "Auswerfer")).start();
```

*1 Zwei Threads ...

*2 ...arbeiten mit der gleichen **`MusikAbspielGeraet`**-Objektinstanz.

*3 In dem einen Thread wird **`hoeren()`** aufgerufen ...

*4 ... und in dem anderen **`auswerfen()`**.

... dann kann es passieren, dass **`hoeren()`** in Thread „Hörer" nur bis **`this.tontraeger = tontraeger;`** ausgeführt wird und dann vom Scheduler zum Thread „Auswerfer" gewechselt wird, der den Tonträger direkt wieder auswirft. Wird dann wieder zurück zum Thread „Hörer" gewechselt, arbeitet das **`MusikAbspielGeraet`**-Objekt mit einer **`null`**-Referenz!

Boah, wie bitte? Und ... was ... ähm ... kann ich dagegen machen?

Um das zu verhindern, musst du sicherstellen, dass alles, was innerhalb von **`hoeren()`** steht, **der Reihe nach ohne Unterbrechung** von einem Thread ausgeführt wird. Und das erreichst du, indem du die Methode als **`synchronized`** markierst. Besser sogar, du markierst alle Methoden, in denen auf die **`tontraeger`**-Objektvariable zugegriffen wird, als **`synchronized`**, so wie hier:

*1 Wenn jetzt eine der beiden Methoden aufgerufen wird, kann keine andere Methode auf der **gleichen Objektreferenz** aufgerufen werden, also auch nicht die jeweils andere Methode.

```
public abstract class MusikAbspielGeraet {
  ...
  public synchronized*1 final void hoeren(Tontraeger tontraeger) {
    ...
  }
  public synchronized*1 Tontraeger auswerfen() {
    ...
  }
}
```

`Was bedeutet das, und wieso funktioniert das?` Ganz einfach: Immer wenn du **`synchronized`** verwendest, wird ein sogenanntes **Monitor-Lock** erstellt, ein Schloss sozusagen, welches dafür sorgt, dass nur jeweils ein Thread auf das entsprechende Codestück zugreifen kann.
Steht das **`synchronized`** an einer **Objektmethode**, ist die zugehörige **Objektinstanz** das Lock. Steht das **`synchronized`** an einer **Klassenmethode**, ist die zugehörige **Klasse** das Lock. Objektinstanz bzw. Klasse sind dann jeweils „verschlossen" und für weitere Aufrufe aus anderen Threads gesperrt.

```
public static synchronized*1 methode() { … }
public synchronized*2 methode() { … }
```

*1 An einer Klassenmethode ist implizit die Klasse das Lock, ...

*2 ... an einer Objektmethode ist das unterliegende Objekt das Lock.

Noch flexibler bist du mit **synchronisierten Blöcken**. Dort kannst du **irgendein Objekt** oder **irgendeine Klasse** als Lock verwenden. Und du kannst sogar nur **Teile einer Methode synchronisieren.**

```
synchronized(lock*1) { … }
synchronized(MusikAbspielGeraet.class*2) { … }
```

*1 **`lock`** kann irgendeine Objektreferenz sein.

*2 Hier ist die Klasse **`MusikAbspielGeraet`** das Lock.

[Ablage]
In einen mit **`synchronized(X.class)`** gekennzeichneten Block kann nur ein einziger Thread hinein, in einen mit **`synchronized(this)`** gekennzeichneten Block kann pro Objektinstanz nur ein einziger Thread hinein.

[Hintergrundinfo]
Wenn ein Thread auf **`synchronized`** trifft, wechselt er in den Zustand **Blocked**. Dort verharrt er so lange, bis der Thread, der gerade das Lock gesperrt hat, dieses entsperrt und er selbst das Lock sperrt.

Das klingt gefährlich. Ich frage mich, was passiert, wenn es es nicht bekommt ...

Erst die geraden Zahlen, bitte!

Generell geht es bei Threads ja um Arbeitsteilung. Die beiden **`Zahlendrucker`**-Implementierungen von vorhin machen das ja prinzipiell richtig: Der eine druckt die geraden Zahlen, der andere die ungeraden Zahlen. Allerdings endet das, wie du selber bemerkt hast, in einem ziemlichen Durcheinander, weil die beiden **asynchron laufen**. Zeit, mal ein bisschen Ordnung reinzubringen und die **beiden Threads zu synchronisieren**.

[Schwierige Aufgabe]
Ändere die Klasse **`AbstractZahlenDrucker`** so, dass **zuerst alle ungeraden** oder **zuerst alle geraden Zahlen** ausgegeben werden. Nicht verzweifeln, wenn du nicht weiterkommst. Das ist nicht umsonst eine schwierige Aufgabe.

Mmh, komisch, ich denke, ich habe die Stelle gefunden, an der ich das **synchronized** setzen muss: an der **run()**-Methode. Aber irgendwie funktioniert das so nicht:

```
public synchronized void run() {…}
…
}
}
```

Das **synchronized** an die **run()**-Methode zu setzen, reicht nicht aus, weil es sich um eine **Objektmethode** und nicht um eine **Klassenmethode** handelt. Das bedeutet, dein Code wäre nur bezüglich der Klasse synchron, nicht aber bezüglich der Objekte. Also würde das nur funktionieren, wenn du innerhalb der beiden Threads **nur eine** Objektreferenz hättest, dann könntest du diese Objektreferenz als Lock verwenden.

So wie hier:

```
AbstractZahlenDrucker zahlendrucker = new GeradeZahlenDrucker(grenze);
(new Thread(zahlendrucker)).start();
(new Thread(zahlendrucker)).start();
```

In dem Beispiel von vorhin hast du aber **zwei Objektinstanzen**: einmal eine Instanz von **`GeradeZahlenDrucker`** und einmal eine Instanz von **`UngeradeZahlenDrucker`**.

Das funktioniert doch, ich müsste irgendwas als Lock nehmen, das beide Instanzen haben.

Genau. Und das ist naheliegenderweise die Klasse **`AbstractZahlenDrucker`**.

So würdest du diese Klasse als Lock verwenden:

```
synchronized (AbstractZahlenDrucker.class*1) {
  for(int i=0; i<=this.grenze; i++) {
    …
  }
}
```

*1 So kann nur ein Thread in den synchronisierten Block hinein.

Freu dich nicht zu früh, denn falsch verwendet, führen solche Blöcke zu dem wohl bekanntesten Problem, seit es Multithreading gibt, den …

`synchronized` verwende.

Primal, dann kann ja nichts mehr schiefgehen, wenn ich

…Deadlocks!

…Deadlocks!

Deadlocks sind eine ganz dumme, unheimliche Sache, die ich dir am besten am Beispiel unserer beiden furchtlosen Helden erkläre.

Stimmt, die Geschichte war ja noch gar nicht zu Ende!

Ja, also, wo waren wir stehen geblieben? Ach ja, beim Kampf gegen die Orks, genau. Die beiden Helden sind ziemlich geschafft von diesem Kampf, aber noch nicht so, dass sie auf einen Besuch im lokalen Wirtshaus verzichten wollen. „Erst noch kurz duschen, dann treffen wir uns, in Ordnung?“, sagt der Nachtelf. „Okay, okay, stimmt, duschen wäre mal wieder an der Zeit, bis gleich dann.“, antwortet der Zwerg und beide verschwinden zu ihren Häusern. Die beiden duschen und beschließen anschließend jeweils, zu dem Haus des anderen zu gehen. Dort angekommen, warten sie dann auf den jeweils anderen und warten und warten …, weil sie vergessen haben, einen Treffpunkt auszumachen.

Übersetzt nennt man so ein Problem, in dem zwei Threads **aufeinander warten,** einen **Deadlock**. Dann funktioniert nichts mehr, beide Threads können nicht weiterarbeiten.

[Begriffsdefinition]
Ein **Deadlock** bezeichnet einen Zustand, in dem ein oder mehrere Threads auf ein Ereignis des/der jeweils anderen Threads warten.

In Code übersetzt sähe das Problem so aus:

```
public class Held {
  …
  public synchronized *1 void besuchen(Held held) {
    try {
      Thread.sleep(50); *2
    } catch (InterruptedException exception) {
    }
    System.out.printf("%s: \"Wir treffen uns bei ihm, dann gehe ↩
      ich schon mal los.\"%n", this.getName());
    held.besuchEmpfangen(this); *3
  }
  public synchronized *1  void besuchEmpfangen(Held held) {
    System.out.printf("%s: \"Wir haben uns bei mir getroffen.↩
      \"%n", this.getName());
  }
}
```

*1 Auf einer Objektinstanz von **`Held`** kann von verschiedenen Threads aus immer nur eine der beiden Methoden, aber nicht beide gleichzeitig, aufgerufen werden.

*2 Die Helden warten ein bisschen.

*3 Und das hier ist die kritische Stelle, die den Deadlock auslöst, aber was da passiert, gucken wir uns gleich mal genauer an.

An dem Quelltext für die **main**-Methode ändert sich zum vorigen Beispiel nicht viel, wieder ein Thread pro Held, und jeder Held ruft die Methode auf, um den jeweils anderen zu besuchen.

```
public static void main(String[] args) {
  final Zwerg zwerg = new Zwerg("Zwerg");
  final Nachtelf nachtelf = new Nachtelf("Nachtelf");
  final Thread zwergenThread = new Thread(new Runnable() {
    @Overridea
    public void run() {
      zwerg.besuchen(nachtelf); *1
    }
  }, "ZwergenThread");
  Thread elfenThread = new Thread(new Runnable() {
    @Override
    public void run() {
      nachtelf.besuchen(zwerg); *2
    }
  }, "ElfenThread");
  zwergenThread.start();
  elfenThread.start();
}
```

Die Reihenfolge der Aufrufe ist also die hier: Je nachdem, ob der **zwergenThread** oder der **elfenThread** zuerst vom Thread-Scheduler ausgewählt wird, wird entweder zuerst **zwerg.besuchen(nachtelf)** aufgerufen oder **nachtelf.besuchen(zwerg)**. Da beide Methoden auf Objektinstanz-Ebene **synchronized** sind, müssen beide Methoden beendet werden, bevor auf **nachtelf** und **zwerg** etwas anderes aufgerufen werden kann. Da aber die **besuchen()**-Methode die **besuchEmpfangen()**-Methode des jeweils anderen aufruft, kommt es zum Deadlock, und das Programm bleibt zum Beispiel mit dieser Ausgabe hängen:

```
Zwerg: "Wir treffen uns bei ihm, dann gehe ich schon mal los."
Nachtelf: "Wir treffen uns bei ihm, dann gehe ich schon mal los."
```

[Achtung]

Lässt du übrigens das **Thread.sleep(50)** weg, muss es nicht zwingend zum Deadlock kommen, dann könnte die Ausgabe auch diese hier sein:

```
Zwerg: "Wir treffen uns bei ihm, dann gehe ich schon mal los."
Nachtelf: "Wir haben uns bei mir getroffen."
Nachtelf: "Wir treffen uns bei ihm, dann gehe ich schon mal los."
Zwerg: "Wir haben uns bei mir getroffen."
```

Das würde also bedeuten, dass der Zwerg schnell zum Nachtelfen flitzt und diesen besucht und dann wieder schnell nach Hause flitzt und den Nachtelfen empfängt. Indem wir die Threads schlafen lassen (wenn auch nur kurz und zu Demonstrationszwecken), erhöhen wir hier also die Wahrscheinlichkeit für den Deadlock.

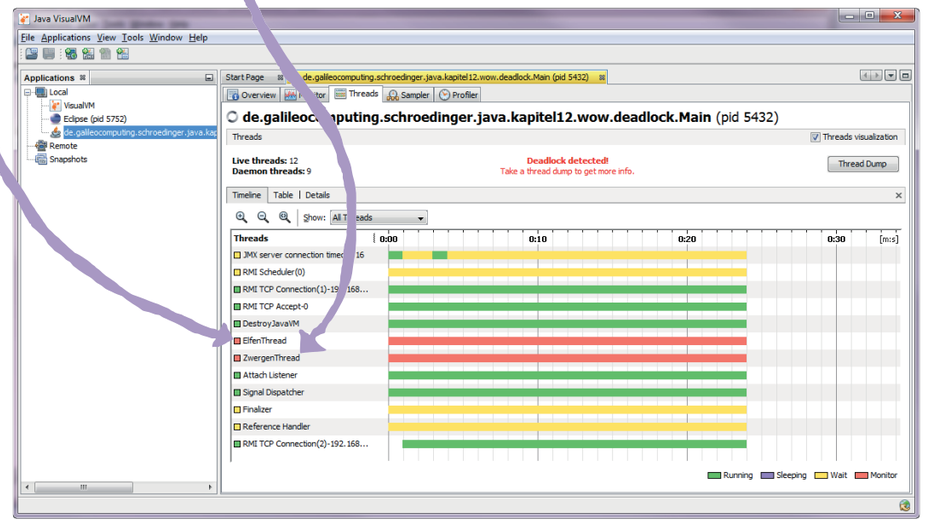
Deadlocks finden und umgehen

Um Deadlocks aufzuspüren, kann dir zum Glück wieder das Tool VisualVM ganz gut helfen.

[Einfache Aufgabe]

Starte das Programm von eben, öffne VisualVM, und wähle innerhalb des Tools wieder den entsprechenden Prozess für dein Programm aus. Anschließend wechselst du in das Tab mit der Überschrift „Threads".

Dort werden alle Threads aufgelistet, die in dem ausgewählten Prozess laufen, inklusive ihres aktuellen Zustands. Wie du im folgenden Screenshot sehen kannst, befinden sich **ElfenThread** und **ZwergenThread** beide im Zustand **Monitor**, was so viel bedeutet wie, dass sie blockiert sind. VisualVM erkennt das und meldet korrekterweise einen Deadlock.

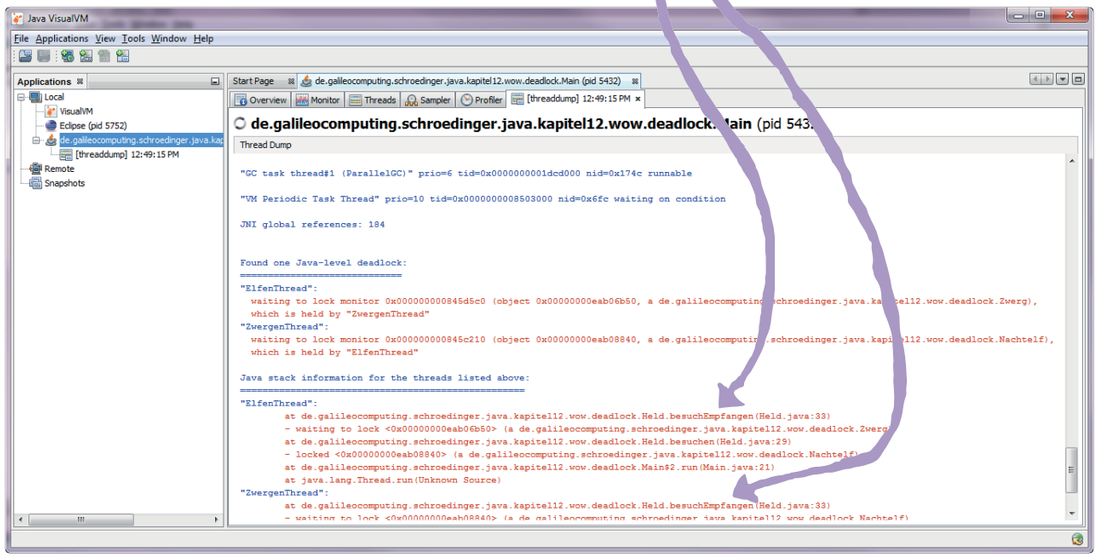


Zwei Threads im Deadlock …

Mit Klick auf die Schaltfläche „Thread Dump" erhältst du noch detailliertere Informationen über den Deadlock, hier mal ein gekürzter Auszug daraus:

```
"ElfenThread":
  waiting to lock monitor ...
  which is held by "ZwergenThread"
"ZwergenThread":
  waiting to lock monitor ...
  which is held by "ElfenThread"
```

Übersetzt bedeutet das in etwa, dass der Thread mit dem Namen „ElfenThread“ auf den mit dem Namen „ZwergenThread“ wartet und umgekehrt. Zusätzlich gibt es noch für jeden der beiden Threads einen Ausschnitt aus dem Stacktrace, damit du der Fehlerquelle noch besser auf die Spur kommen kannst.



… und der dazugehörige Thread Dump

Oje, Deadlocks machen mir irgendwie Angst. Aber gut zu wissen, dass es nützliche Helferlein gibt.

Ja, Deadlocks zu finden, ist die eine Sache, sie gar nicht erst entstehen zu lassen, nochmal eine andere. Das Problem eben war, dass beide Threads auf ein Lock-Objekt zugreifen wollten, das vom jeweils anderen Thread geblockt war. Oft macht es daher Sinn, **nur ein Lock-Objekt für mehrere Threads** zu verwenden.

Der Schlüssel zum Erfolg

Du könntest die **`synchronized`**-Blöcke von eben so erweitern, dass sie statt **`this`** ein Objekt als Lock bekommen, dass sich die beiden Helden teilen. Dafür gibt es im Paket **`java.util.concurrent.locks`** schon einige Helferklassen, die das Arbeiten mit solchen Locks um einiges einfacher und sicherer gestalten. Das Basis-Interface dafür ist das Interface **`java.util.concurrent.locks.Lock`**.

Du könntest beliebige Objekte als Lock verwenden. **Aber Lock hat gewisse Vorteile, und das siehst du hier:**

```
public class Held {
  private String name;
  private Lock lock*3 = new ReentrantLock(); *4

  …
  public boolean treffenBei(Held held) { *1
    Boolean ichSageWirTreffenUnsBeiIhm = false; *5
    Boolean erSagtWirTreffenUnsBeiIhm = false; *5
    Boolean treffenBeiIhm = false;
    try {
      ichSageWirTreffenUnsBeiIhm = lock.tryLock(); *6
      erSagtWirTreffenUnsBeiIhm = held.getLock().tryLock(); *6
      treffenBeiIhm = ichSageWirTreffenUnsBeiIhm && erSagtWirTreffenUnsBeiIhm;
    } finally {
      if(!treffenBeiIhm) { *7
        System.out.println(this.getName() + ": ↵
          \"Nächstes Mal treffen wir uns dann eben bei ihm.\"");
        if(ichSageWirTreffenUnsBeiIhm) {
          lock.unlock(); *7
        }
        if(erSagtWirTreffenUnsBeiIhm) {
          held.getLock().unlock(); *7
        }
      }
    }
    return treffenBeiIhm; *6
  }
  protected Lock getLock() { *3
    return lock;
  }

  public void *1besuchen(Held held) {
```

*1 Die **`besuchen()`**-Methode von **`Held`** ist jetzt nicht mehr **`synchronized`**. Stattdessen gibt es eine neue Methode **`treffenBei()`**, die feststellt, bei welchem Helden man sich trifft, …

*2 … und bevor ein Held den anderen besucht, stellt er über diese Methode erstmal sicher, ob der Besuch überhaupt so abgesprochen war.

*3 Und diese Absprache wird über zwei Instanzen von **`Lock`** implementiert (eine in jeder Heldeninstanz). Entweder der Zwerg bekommt die Locks oder der Nachtelf, aber nicht beide gleichzeitig.

*4 **`ReentrantLock`** ist im Übrigen nur eine der Implementierungen von **`Lock`**. Das im Detail zu erklären, würde jetzt aber wirklich zu weit gehen.

*5 Interessant ist jetzt der Inhalt der **`treffenBei()`**-Methode. Denn die darf ja immer nur für einen der beiden Helden ein **`true`** zurückgeben und muss für den anderen ein **`false`** zurückgeben. Zwei **`Boolean`**s, die festhalten, ob beide mit dem Treffpunkt einverstanden sind.

*6 Hier wird versucht, beide Locks zu bekommen bzw. sich über den Treffpunkt einig zu werden.

*7 Wenn wir nicht beide Locks bekommen haben, prüfen wir, ob wir eines bekommen haben, und geben das dann frei.

```
    if(this.treffenBei(held)) { *2
      System.out.printf("%s: \"Wir treffen uns bei ihm, ↩
        dann gehe ich schon mal los.\"%n", this.getName());
      held.besuchEmpfangen(this);
    }
  }
  …
}
```

Boah, das muss ich mir aber nochmal in Ruhe angucken. Worin liegt denn nun der Vorteil, wenn ich Instanzen von **Lock** statt meines eigenen Lock-Objekte verwende?

Na ja, verwendest du „normale“ Objekte als Lock, dann wechselt der Thread, der das Lock bekommen möchte, automatisch in den Zustand **Blocked**, aus dem er nur dann wieder rauskommt, wenn er selber das Lock bekommt. Wenn er es nicht bekommt, dann … Deadlock … nicht wahr? Mit der Methode **`tryLock()`** von **Lock** kannst du **erst prüfen**, ob der Thread das Lock bekommen würde und gegebenenfalls, sollte er es nicht bekommen, mit was anderem weitermachen und später nochmal prüfen. Kurz: Auf diese Weise verhinderst du Deadlocks.