Objektorientierte Programmierung

Kapitel 2: Objektorientierte Programmierung am Beispiel

Stefan Brass
Martin-Luther-Universität Halle-Wittenberg
Wintersemester 2012/13

http://www.informatik.uni-halle.de/~brass/oop12/

Inhalt

1. Klassen und Attribute
   - Modellierung der realen Welt, Anwendung
   - Klassen, Objekte, Attribute
   - Variablen, Zuweisungen
2. Datentypen
   - Datentypen von Java (insbesondere primitive Typen)
3. Konstruktoren, Zugriffsschutz, Methoden
   - Definition eines Konstruktors
   - Zugriffsschutz, Methoden
   - Fortsetzung des Beispiels, Beziehung zwischen Klassen
   - Verwendung eines Arrays, Berechnung mit Schleife

Modell-Bildung

- Programme dienen normalerweise dazu, Aufgaben und Aktivitäten in der realen Welt zu unterstützen.

  Manchmal sind es auch gedachte Welten, wie etwa bei Spielen.

- Dazu enthalten die Programme Abbilder von Objekten der realen Welt.

- Selbstverständlich läßt man dabei alle Details weg, die für die gegebene Aufgabe nicht relevant sind (diesen Vorgang nennt man auch “Abstraktion”).

  Wenn man z.B. die Hausaufgaben-Punkte für diese Vorlesung verwalten will, kann man Studenten auf die Eigenschaften “Name”, “Vorname”, “EMail-Adresse” und “Matrikel-Nummer” reduzieren. Dinge wie die Augenfarbe oder die Hobbies sind für diese Anwendung nicht wichtig.

Beispiel-Anwendung (1)

- Ich plane gelegentlich ein Großfeuerwerk.

  Etwa zur letzten Langen Nacht der Wissenschaften. Ich habe als Hobby eine Ausbildung als Großfeuerwerker gemacht und habe Befähigungsschein und Erlaubnis nach dem Sprengstoffgesetz.

- Ich möchte eine Software entwicklen, die mich dabei unterstützt.

  Es ist wichtig, für sich persönlich eine motivierende Anwendung zu haben.

- Bei einem Feuerwerk werden Feuerwerksartikel abgebrannt (z.B. Fontänen, Feuerwerksbatterien, Feuerwerksbomben).

  Raketen sind bei Großfeuerwerken selten, weil man wegen der unsicheren Flugbahn und der herabfallenden Stäbe einen großen Sicherheitsabstand braucht (200m). Feuerwerksbomben werden nach dem Prinzip einer Kanone aus Abschussrohren senkrecht in den Himmel geschossen, und zerplatzen dort in Leuchtsterne und andere Effekte. Der Effekt ist ähnlich zu Raketen.

Beispiel-Anwendung (2)

- Feuerwerksartikel haben viele interessante Eigenschaften (Attribute), aber für die erste Ausbaustufe des Programms sollen folgende Daten reichen:
  - Bezeichnung
  - Preis
  - Brenndauer
- Z.B. gibt es den Artikel
  - "Schweizer Supervulkan Magic Light",
  - der 5.50 € kostet (plus Mehrwertsteuer),
  - und 60 Sekunden brennt.

    Erst Stroboskop-Effekt (weißer Bengalblinker), dann goldene Fontäne mit weißen Blinksternen darin, die sich bis zur maximalen Sprühhöhe von 6m langsam aufbaut (Blinksterne erst nach 30s).

Klassen und Objekte

- Eine Klasse ist eine Zusammenfassung von ähnlichen Objekten, die also insbesondere die gleichen Attribute haben. Beispiel: Alle Feuerwerksartikel für meine Planung.

  Eine Klasse ist also ein Begriff, mit denen man über eine Menge von gleichartigen Dingen der realen Welt reden kann (auch Form der Abstraktion).

- Man kann Klassen auch als Blaupausen/Schablonen zur Konstruktion von Objekten verstehen.
- Objekte haben Identität, Zustand und Verhalten:
  - Objekte sind von anderen Objekten unterscheidbar.
  - Der Zustand wird durch Datenwerte bestimmt, die im Objekt gespeichert werden (Werte der Attribute).
  - Das Verhalten wird durch Programmcode in Methoden realisiert, die von außen aufgerufen werden können.

Klassendeklarationen (1)

- Im Beispiel gibt es also eine Klasse `Artikel` (kurz für "Feuerwerksartikel") mit den drei genannten Attributen:

```
class Artikel {
    String bezeichnung; // Zeichenkette
    int preis; // Preis in Cent ohne MwSt
    int dauer; // Brenndauer in Sekunden
    ...
}
```

Die Attribute werden durch Variablen in den Objekten realisiert.

- Wenn man ein Objekt der Klasse `Artikel` erzeugt, wird ausreichend Hauptspeicher für das Objekt reserviert, so dass man in diesen Bereich eine Zeichenkette und zwei Zahlwerte (`int`: "integer", ganze Zahl) speichern kann.

Klassendeklarationen (2)

- Eine Klassendeklaration besteht also (etwas vereinfacht) aus
  - dem Schlüsselwort "`class`",
  - dem Namen der neuen Klasse (ein Bezeichner, s.u.),
  - einer "`{`" (geschweifte Klammer auf),
  - Deklarationen der Bestandteile der Klasse, insbesondere Variablen und Methoden (s.u.),
  - einer "`}`" (geschweifte Klammer zu).

    In C++ muss man übrigens hinter der schließenden Klammer der Klasse ein Semikolon setzen. In Java ist das nicht verlangt und nicht üblich. Um ehemaligen C++-Programmierern das Leben zu erleichtern, ist ein Semikolon aber möglich (sogar beliebig viele, auch vor der Klasse, für den Compiler bedeutungslose Alternative zur Klassendeklaration).

Bezeichner (1)

- Klassen, Variablen, Methoden, u.s.w. benötigen Namen (engl. “identifier”: Bezeichner).
- Solche Namen können aus Buchstaben und Ziffern bestehen, wobei das erste Zeichen ein Buchstabe sein muß.

  Die Zeichen “_” und “$” zählen als Buchstabe, wobei “$” aber nur in von Programmen erzeugtem Programmcode verwendet werden soll.

  Umlaute sind grundsätzlich möglich, führen aber gerade bei Klassennamen möglicherweise zu Problemen, weil die Klassennamen ja auch als Dateinamen verwendet werden, und nicht jedes Betriebssystem Umlaute gleich codiert.
- Klassennamen beginnen üblicherweise mit einem Großbuchstaben. Bei mehreren Worten wählt man die “Kamelhöcker-Schreibweise”, z.B. `LagerPosition`.

  Das ist nur Konvention und Empfehlung, dem Compiler ist es egal.

  Leerzeichen in Namen wären aber nicht möglich.

Bezeichner (2)

- Bezeichner müssen in einem gewissen Kontext eindeutig sein, z.B. kann man nicht zwei Klassen mit gleichem Namen deklarieren.

  Wenn man später ein Objekt der Klasse anlegen will, muß der Compiler ja wissen, von welcher Klasse, d.h. was die zugehörige Klassendeklaration ist. (Komplexere Fälle, z.B. verschiedene Pakete, werden später behandelt.)

- Die Schlüsselwörter (oder "reservierten Wörter") können auch nicht als Namen verwendet werden. Man kann z.B. keine Klasse deklarieren, die "`class`" heißt.

  Die Schlüsselworte sind: `abstract`, `assert`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extends`, `final`, `finally`, `float`, `for`, `if`, `goto`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`, `package`, `private`, `protected`, `public`, `return`, `short`, `static`, `strictfp`, `super`, `swirtch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `try`, `void`, `volatile`, `while`.

Variablen-Deklarationen (1)

- Innerhalb der Klasse kann man Variablen deklarieren, die die Eigenschaftswerte aufnehmen.

  Variablen in Klassen werden auch als Attribute, Felder (engl. "field") oder "member variables" bezeichnet (im Unterschied zu Variablen in Methoden).

- Allgemein ist eine Variable ein Speicherbereich, der Datenwerte aufnehmen kann.

- Jede Variable kann nur Werte eines bestimmten Datentyps aufnehmen (z.B. `int`, `String`).

- Ein Datentyp ist eine Menge von Werten der gleichen Art, z.B. die Menge der ganzen Zahlen, oder die Menge der Zeichenketten.

  Zu einem Datentyp gehört nicht nur die Wertemenge, sondern auch die Operationen, die man mit Werten dieser Art ausführen kann.

Variablen-Deklarationen (2)

- Eine Variablen-Deklaration besteht im einfachsten Fall aus:
  - dem Namen des Datentyps (z.B. `int`, `String`, s.u.),
  - dem Namen der Variable (z.B. `preis`),
  - einem Semikolon "`;`".
- Es ist üblich, dass Namen von Variablen in Klassen mit einem Kleinbuchstaben beginnen, und bei mehreren Worten auch die Kamelhöckerschreibweise gewählt wird.

  Das ist nur Konvention, und stammt natürlich aus dem englischsprachigen Bereich, wo Kleinschreibung von Nomen kein Problem ist. Falls Sie es nicht mögen, könnten Sie natürlich gegen diese Konvention verstoßen (s.o.).
  Auf die Dauer empfiehlt es sich aber wohl, Programmtexte ganz in Englisch zu schreiben. Aufgrund der Schlüsselworte bekommen Sie sonst ein Deutsch-Englisches-Mischmasch, und die internationale Zusammenarbeit ist Ihnen auch verwehrt.

Variablen-Deklarationen (3)

- Natürlich müssen Variablen-Namen innerhalb einer Klasse eindeutig sein: Man kann z.B. nicht zwei Variablen "`preis`" in der Klasse "`Artikel`" haben.
- Verschiedene Klassen können aber Variablen mit gleichem Namen haben.

  Aufgrund des Kontexts kann der Compiler die Variablen auseinander halten.
- Es kann natürlich auch mehrere Objekte der Klasse `Artikel` geben, dafür ist sie ja gerade gemacht.
- Jedes Objekt der Klasse `Artikel` hat seinen eigenen Satz von den drei Variablen `bezeichnung`, `preis`, `dauer`.

  Wenn man sich im Programm auf eine dieser Variablen beziehen will, ist entweder aus dem Kontext klar, welches "aktuelle Objekt" gemeint ist, oder man muss ein Objekt angeben.

Variablen und Schubladen

- Man kann eine Variable mit einer Schublade einer Kommode vergleichen, in die man einen Datenwert hineintun kann.
- Dabei gibt es aber auch Unterschiede:
  - Eine Variable kann immer nur einen Datenwert enthalten. Wenn man einen neuen hineintut, verschwindet der alte, er wird "überschrieben".
  - Eine Variable kann niemals leer sein, es ist immer ein Wert darin.

    Die Bits im Hauptspeicher sind aus technischen Gründen immer 0 oder 1. Es gibt keinen dritten Wert "leer".
  - Wenn man den Wert abfragt, nimmt man ihn nicht heraus: Er bleibt solange in der Variable, bis ein neuer Wert explizit hineingespeichert wird.

Zuweisungen (1)

- Man speichert einen Wert mittels einer "Zuweisung" in eine Variable. Eine Zuweisung ist eine Anweisung der Art:

```
dauer = 60;
```

- Eine Zuweisung besteht also (etwas vereinfacht) aus:
  - Name der Variablen,

    So einfach geht es nur in Methoden der Klasse selbst, dort ist bereits beim Aufruf ein Objekt angegeben, so dass klar ist, von welchem Objekt die Variable stammt.
  - einem Gleichheitszeichen "=",
  - einem Ausdruck, der einen Wert liefert, also z.B. einer Konstanten wie `10` oder `"abc"`,

    Man kann hier auch rechnen, z.B. `1+1` oder `preis * 1.19`.
  - und einem Semikolon ";".

Zuweisungen (2)

- Der Typ des Wertes auf der rechten Seite muß zum Datentyp der Variable auf der linken Seite passen.

  Z.B. kann man nicht `"abc"` (einen String) eine eine Variable speichern, die nach der Deklaration den Typ `int` (ganze Zahl) hat.

- Es stellt sich natürlich die Frage, welchen Wert eine Variable vor der Ausführung der ersten Zuweisung hat.
  - Java initialisiert Variablen einer Klasse automatisch auf Standardwerte — je nach Typ die Zahl 0, der leere String, oder die Null-Referenz.

    Die erste Zuweisung heißt Initialisierung.
  - Bei Variablen in Methoden überwacht Java, dass der Programmierer sie vor der Verwendung initialisiert.

    C++ tut das nicht, aber auch dort enthalten Variablen immer einen Wert — zufälligen Müll, solange man nicht einen Wert eingetragen hat.

Deklaration mit Initialisierung

- Man kann Deklaration und erste Zuweisung (Initialisierung) auch in einer Anweisung kombinieren, z.B.:

```
int i = 1;
```

  Dies führt also eine Variable `i` vom Typ `int` (ganze Zahl) ein, und speichert gleich den Wert `1` in den für `i` reservierten Platz im Hauptspeicher.

- Normalerweise ist das übersichtlicher als Deklaration und erste Zuweisung zu trennen:
  - Die Variable hat gleich einen (sinnvollen) Wert.

    Man kann also die für Variablen in Methoden notwendige Initialisierung nicht vergessen.
  - Die Bedeutung der Variablen wird noch klarer, wenn man gleich einen Wert für sie kennt.

    Es geht aber nicht immer so (z.B. wenn Wert definiert über Fallunterscheidung).

Objekt-Erzeugung (1)

- Variablen können nicht nur einfache Datenwerte wie ganze Zahlen enthalten, sondern auch Objekte.

  Genauer Referenzen oder Verweise auf Objekte (ihre Hauptspeicher-Adresse). Daher können mehrere Variablen Referenzen auf das gleiche Objekt enthalten. Bei der Zuweisung wird nicht das ganze Objekt kopiert, sondern nur ein Verweis auf das Objekt. Der Unterschied wird später noch klarer, wenn die im Objekt gespeicherten Werte geändert werden.

- Man erzeugt ein Objekt (etwas vereinfacht) mit
  - dem Schlüsselwort "`new`",
  - gefolgt von dem Klassennamen, z.B. `Artikel`,
  - und einer Argumentliste, die im Moment noch leer ist, also "`()`".

Objekt-Erzeugung (2)

- Damit man das neu erzeugte Objekt verwenden kann, muß man es sich in eine Variable speichern.

  Sonst ist es nicht mehr zugreifbar und wird von Java automatisch gelöscht ("Garbage Collection": automatische Müll-Einsammlung).

- Man kann also eine Variable für (Referenzen auf) Objekte der Klasse `Artikel` deklarieren:

```
Artikel vulkan;
```

- Dieser Variablen weist man das neu erzeugte Objekt zu:

```
vulkan = new Artikel();
```

- Alternative (empfohlen): Deklaration und Initialisierung in einer Anweisung:

```
Artikel vulkan = new Artikel();
```

Zugriff auf Attribute (1)

- Zum Zugriff auf die Attribute eines Objektes schreibt man:
  - Objekt (z.B. Variable, die Referenz auf Objekt enthält),
  - "." (ein Punkt),
  - Attribut (d.h. Variable im Objekt).

    Wir werden später sehen, dass man den Zugriff auf Attribute von außerhalb der Klasse beschränken kann.
- Z.B. können die Attribute des neu erzeugten Objektes mit folgenden Anweisungen gesetzt werden:

```
vulkan.bezeichnung = "Vulkan Magic Light";
vulkan.preis = 550; // 5.50 Euro
vulkan.dauer = 60;  // 60s
```

Zugriff auf Attribute (2)

- "Variable.Attribut" ist wieder eine Variable.
- Attribute können auch einen Klassen-Typ haben, dann können ganze Ketten von Attribut-Zugriffen entstehen.
- Wenn Variablen links vom Zuweisungs-Operator "=" stehen, findet ein schreibender Zugriff auf die Variable statt (es wird ein Wert hineingespeichert).
- In anderem Kontext findet ein lesender Zugriff statt, d.h. der Wert wird abgefragt.
- Man kann einen Attribut-Wert also z.B. auf folgende Art ausgeben:

```
System.out.println(vulkan.dauer);
```

Erste Version des Programms (1)

- Die erste Version des Programms soll
  - die Klasse `Artikel` deklarieren, und
  - im Hauptprogramm (Methode `main` in Klasse `FW`) ein Objekt `vulkan` anlegen, Attributewerte setzen und drucken.
- Zwei Möglichkeiten zur Strukturierung des Programms:
  - Beide Klassen (`Artikel`, `FW`) zusammen in eine `.java`-Datei.

    Beachten Sie, dass der Compiler daraus aber zwei `.class`-Dateien erzeugt (eine für jede Klasse).
  - Jede Klasse in eine eigene `.java`-Datei (übliche Lösung).

    Man kann beide Dateien zusammen beim Aufruf des Compilers angeben. Wenn man zuerst "`javac FW.java`" aufruft, compiliert er `Artikel.java` automatisch mit, weil er die `.class`-Datei braucht, um z.B. das Objekt zu erzeugen oder auf die Attribute zuzugreifen.

Erste Version des Programms (2)

- Wenn man sich für die zwei getrennten Dateien entscheidet, schreibt man in die Datei "`Artikel.java`" folgenden Programmtext:

```
(1)  // Klasse fuer Feuerwerks-Artikel
(2)  // Erste Version, noch ohne Methoden
(3)  // und Zugriffs-Schutz.
(4)
(5)  class Artikel {
(6)      String bezeichnung; //Artikel-Name
(7)      int preis; // Preis in Cent ohne MwSt
(8)      int dauer; // Dauer in Sekunden
(9)  }
```

- Inhalt der Datei "`FW.java`" auf nächster Folie.

Erste Version des Programms (3)

```
(1)  class FW {
(2)      static public void main(String[] args) {
(3)          Artikel vulkan = new Artikel();
(4)          vulkan.bezeichnung =
(5)                          "Vulkan Magic Light";
(6)          vulkan.preis = 550;
(7)          vulkan.dauer = 60;
(8)          System.out.print("Bezeichnung: ");
(9)          System.out.println(vulkan.bezeichnung);
(10)         System.out.print("Preis:       ");
(11)         System.out.println(vulkan.preis);
(12)         System.out.print("Dauer:       ");
(13)         System.out.println(vulkan.dauer);
(14)     }
(15) }
```

Erste Version des Programms (4)

- Für das Compilieren gibt es folgende Möglichkeiten:
  - `javac Artikel.java FW.java`
  - `javac FW.java` (übersetzt `Artikel.java` automatisch)
  - `javac Artikel.java` (erzeugt nur "`Artikel.class`"), `javac FW.java` (übersetzt nur "`FW.java`")
  - Eine IDE wie Eclipse, Netbeans, BlueJ benutzen.

    Das wäre bei komplizierteren Abhängigkeiten zu empfehlen.
- Dann führt man das Programm mit "`java FW`" aus. Die Ausgabe ist:

```
Bezeichnung: Vulkan Magic Light
Preis:       550
Dauer:       60
```

Inhalt

1. Klassen und Attribute
   - Modellierung der realen Welt, Anwendung
   - Klassen, Objekte, Attribute
   - Variablen, Zuweisungen
2. Datentypen
   - Datentypen von Java (insbesondere primitive Typen)
3. Konstruktoren, Zugriffsschutz, Methoden
   - Definition eines Konstruktors
   - Zugriffsschutz, Methoden
   - Fortsetzung des Beispiels, Beziehung zwischen Klassen
   - Verwendung eines Arrays, Berechnung mit Schleife

Datentypen: Allgemeines (1)

- Ein Datentyp (kurz "Typ") bestimmt:
  - eine Name für den Datentyp, z.B. `int`,
  - eine Menge von möglichen Werten des Datentyps, z.B. die Menge der ganzen Zahlen:
    $$\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}.$$
    Tatsächlich ist der Wertebereich für `int` beschränkt, siehe unten.
  - ggf. eine Möglichkeit, diese Werte im Programm mit Konstanten ("Datentyp-Literalen") aufzuschreiben, z.B. als Folge von Dezimalziffern, wie etwa `123`,
  - Operationen auf den Werten, z.B. die vier Grundrechenarten `+`, `-`, `*`, `/`.
    Die Division zweier `int`-Werte liefert in Java wieder einen `int`-Wert, und zwar wird in Richtung auf 0 gerundet, z.B. $14/3 = 4$.
    Der Operator % liefert den Divisionsrest, z.B. $14 \,\%\, 3 = 2$.

Datentypen: Allgemeines (2)

- Der Datentyp einer Variablen bestimmt auch, wie viel Speicherplatz der Compiler für die Variable reservieren muss.

  Einge ganze Zahl benötigt z.B. mehr Speicherplatz als ein einzelnes Zeichen.

- Der Datentyp einer Variablen legt fest, wie die Bits in dem Speicherbereich der Variablen zu interpretieren sind.

- Die Angabe von Datentypen hilft, Fehler zu vermeiden bzw. schon bei der Compilierung zu erkennen.

  Der Compiler stellt sicher, dass nur passende Operationen auf die Daten angewendet werden können, d.h. dass die Bits an der gleichen Stelle im Hauptspeicher nicht bei verschiedenen Befehlen völlig unterschiedlich interpretiert werden können.

- Außerdem sind Datentypen eine wichtige Dokumentation, die das Verständnis des Programms erleichtert.

Datentypen: Primitive Typen vs. Referenztypen (1)

- Die Datentypen in Java gliedern sich in
  - Primitive Typen, z.B. `int`.
  - Referenz-Typen, z.B. Klassen-Typen.

    Außerdem Interface-Typen und Array-Typen, das wird später erklärt.
- `String` ist ein spezieller Klassen-Typ (Referenz-Typ): Es gibt eine vordefinierte Klasse für Zeichenketten.

  Dies erklärt auch, warum `String` groß geschrieben wird, und `int` klein.

  Die Klasse `String` ist insofern speziell, als Zeichenketten-Konstanten wie `"abc"` vom Compiler in Objekte dieser Klasse abgebildet werden.

  Das kann man so nicht für selbstdefinierte Klassen haben.
- Während die Zuweisung bei primitiven Typen den ganzen Wert kopiert, speichert sie bei Referenz-Typen nur eine Referenz (d.h. Hauptspeicher-Adresse) für das Objekt (s.u.).

Datentypen: Primitive Typen vs. Referenztypen (2)

- Dass tatsächlich nur eine Referenz kopiert wird, kann man mit folgendem Programmstück ausprobieren:

```
(1)  class FW {
(2)      static public void main(String[] args){
(3)          Artikel vulkan = new Artikel();
(4)          vulkan.bezeichnung =
(5)                          "Vulkan Magic Light";
(6)          vulkan.preis = 550;
(7)          vulkan.dauer = 60;
(8)          Artikel vulkan2 = vulkan;
(9)          vulkan.preis = 600;
(10)         System.out.println(vulkan2.preis);
(11)     }
(12) }
```

Datentypen: Primitive Typen vs. Referenztypen (3)

- Das obige Programm gibt `600` aus (also den neuen Preis).
- Die Variablen `vulkan` und `vulkan2` verweisen auf dasselbe Objekt, deswegen wirkt sich die Änderung von `vulkan` auch auf `vulkan2` aus.

  Java bietet natürlich Möglichkeiten, ganze Objekte zu kopieren, z.B. mit der Methode `clone()`. Das will man jedoch nur relativ selten. In C++ muß man sich explizit entscheiden, ob eine Variable das Objekt selbst enthalten soll, oder eine Referenz darauf (unterschiedliche Datentypen).

- Objekte haben also einen internen Zustand, der geändert werden kann. Dagegen bleibt die Zahl 3 immer die Zahl 3.
- Weil Objekte groß sein können, ist es auch aus Gründen der Effizienz günstig, nur die relativ kleine Referenz (32 bit oder 64 bit) zu kopieren.

Begrenzung des Zahlbereiches (1)

- Im Gegensatz zur mathematischen Vorstellung von "ganze Zahl" ist in Java (wie in den meisten Programmiersprachen) der Wertebereich begrenzt:
  - der kleinste Wert vom Typ `int` ist $-2\,147\,483\,648 = -2^{31}$,
  - der größte Wert ist entsprechend $2\,147\,483\,647 = 2^{31} - 1$, d.h. etwas über 2 Milliarden.
  - Der Grund für die Begrenzung ist, dass zur Darstellung eines `int`-Wertes im Hauptspeicher 32 Bit (4 Byte) benutzt werden, damit kann man insgesamt 2^{32} verschiedene Werte darstellen.

    Es int einen positiven Wert weniger als negative Werte, weil auch die 0 noch mit eingerechnet werden muss. Damit sind es dann insgesamt 2^{32} verschiedene Werte.

Begrenzung des Zahlbereiches (2)

- Falls das Ergebnis einer Rechenoperation außerhalb des darstellbaren Zahlbereiches liegt,
  - gibt es in Java keinen Fehler,
  - aber das Ergebnis stimmt natürlich nicht mit dem mathematisch erwarteten Ergebnis überein.
- Wenn man z.B. $1\,000\,000$ quadriert, so erhält man $-727\,379\,968$.

  Es werden einfach die untersten 32 Bit des richtigen Ergebnisses genommen, dabei gibt es hier auch einen Überlauf in das Vorzeichen-Bit hinein.
- Gute Programmierer versuchen, in solchen Fällen wenigstens eine Fehlermeldung auszugeben.

  Die Java-Bibliothek hat auch eine Klasse `BigInteger` zur Repräsentation von ganzen Zahlen beliebiger Länge.

Primitive Typen (1)

- Java hat insgesamt 8 verschiedene primitive Typen.
- Davon sind vier Datentypen zur Repräsentation ganzer Zahlen unterschiedlicher Größe:
  - `byte` (8 bit): Von -128 bis $+127$.
  - `short` (16 bit): Von $-32\,768$ bis $+32\,767$.
  - `int` (32 bit): Von $-2\,147\,483\,648$ bis $+2\,147\,483\,647$.
  - `long` (64 bit): Von $-9\,223\,372\,036\,854\,775\,808$ bis $+9\,223\,372\,036\,854\,775\,807$ ($>$ 9 Trillionen: $9 * 10^{18}$)

    Da $2^{10} = 1024$ etwas mehr als 10^3 ist, ist 2^{63} entsprechend mehr als $2^3 * 10^{6*3}$, d.h. $8 * 10^{18}$. In C++ ist der Datentyp `long int`, den man auch `long` abkürzen kann, üblicherweise nur 32 bit groß.
    Die genauen Zahlbereiche sind in C++ implementierungsabhängig, bei `int` ist nur garantiert, dass es mindestens 16 bit groß ist.

Primitive Typen (2)

- Hinweis zur Verwendung:
  - `int` ist für ganze Zahlen der normale Typ.
  - Mit den Typen `byte`, `short` kann man Speicherplatz sparen, sie sind aber für Anfänger nicht zu empfehlen.

    Berechnungen (wie Addition +) werden mit 32 Bit durchgeführt, auch wenn die Eingaben einen kleineren Typ haben. Bei der Zuweisung an eine Variable desselben Typs muß dann eine explizite Typumwandlung geschrieben werden, also z.B. "`b = (byte) (b+1);`" wenn `b` als Variable vom Typ `byte` deklariert ist. Das ist eine unnötige Komplikation. Man würde also auch zum Speichern von Prozentzahlen (die sicher in ein `byte` hineinpassen würden) eher den Datentyp `int` benutzen. Da keiner von beiden Typen genau die Zahlen 0 bis 100 erlaubt, wäre eine zusätzliche Fehlerprüfung nützlich (später).
  - `long` wird nur selten benötigt, wenn wirklich sehr große Zahlen auftreten könnten.

Primitive Typen (3)

- Außerdem gehört auch der Typ `char` zu den "integral types".
  - Er dient zur Repräsentation einzelner Zeichen z.B. `'a'`.

    "`char`" stammt von engl. "character" (Zeichen). Zur Aussprache: Man kann "`char`" wie den Anfang von "character" aussprechen ("kär"), oder wie eine Mischung von "child" und "car" ("tschar"). Es gibt ein englisches Wort "char" (Aussprache "tschar") mit der Bedeutung "verkohlen", "Rotforelle", "als Putzfrau arbeiten".
  - aber man kann ihn auch für Zahlen von 0 bis 65 535 verwenden (16 bit, "unsigned").

    Kapitel 1 enthält eine ASCII-Tabelle, die die Codierung von Zeichen durch Zahlen illustriert. Java benutzt Unicode, um auch deutsche Umlaute, asiatische Schriftzeichen u.s.w. darstellen zu können. Genauer benutzt Java UTF-16. Der Unicode Standard ist inzwischen auf mehr als 16 Bit gewachsen, einige seltene Zeichen müssen als zwei UTF-16 Einheiten repräsentiert werden, und passen nicht in ein `char`.

Primitive Typen (4)

- Weitere Zahldatentypen sind die Gleitkomma-Zahlen:
  - `float` (32 bit), von engl. “floating point numbers”
  - `double` (64 bit), von engl. “double precision”
- Gleitkomma-Zahlen sind Zahlen mit Nachkomma-Stellen und werden intern mit Vorzeichen, Mantisse und Exponent dargestellt, also z.B. $1.04\text{E}5 = 1.04 * 10^5 = 10400$.

  Tatsächlich werden Exponenten zur Basis 2 verwendet (s.u.).

  Statt “Gleitkomma” kann man auch “Fließkomma” sagen.
- Beispiele für Konstanten des Datentyps `double`: `3.14`, `1E10`, `1.2E-5`.

  Man beachte, dass Werte mit “`.`” notiert werden, und nicht mit Komma.

  Das Zeichen “`E`” für den Exponenten kann auch klein geschrieben werden: “`e`”.

  Konstanten des Typs `float` werden durch ein angehängtes “`f`” / “`F`” markiert.

Primitive Typen (5)

- Hinweise zur Verwendung von Gleitkomma-Zahlen:
  - Das Gegenteil von Gleitkommazahlen wären Festkomma-Zahlen, also Zahlen mit einer festen Anzahl von Stellen vor und nach dem Komma.

    Festkomma-Zahlen gibt es in Datenbanken, aber nicht in Sprachen wie Java oder C++, dort gibt es nur den Spezialfall "ganze Zahl". Durch entsprechende Interpretation kann man aber Festkomma-Zahlen simulieren, z.B. Centbetrag abspeichern statt Euro-Betrag.
  - Vorteil von Gleitkomma-Zahlen: Es können sehr große und sehr kleine Zahlen dargestellt werden.
  - Nachteil: Es finden automatische, nicht immer überschaubare Rundungen statt (s.u.).
  - `double` ist der normale Gleitkomma-Typ, nicht `float`.

Primitive Typen (6)

- Der letzte primitive Datentyp ist der Typ `boolean` zur Repräsentation von Wahrheitswerten.

  Benannt nach George Boole, 1815-1864. In C++ heißt der Typ "`bool`".

- Er hat nur zwei mögliche Werte:
  - `true`: wahr
  - `false`: falsch
- Man erhält einen booleschen Wert z.B. als Ergebnis eines Vergleichs:
  - $1 < 2$ ist wahr,
  - $3 > 4$ ist falsch.

Primitive Typen (7)

Übersicht (die 8 primitiven Typen von Java):

- Ganzzahlige Typen ("integral types" $\subseteq$ "numeric types"):
  - `byte`
  - `short`
  - `int`: Normalfall für ganze Zahlen
  - `long`
  - `char`: Für einzelne Zeichen (Buchstaben, Ziffern etc.)
- Gleitkomma-Zahlen ("floating point types" $\subseteq$ "num. types"):
  - `float`
  - `double`: Normalfall für Gleitkomma-Zahlen
- `boolean`: Für Wahrheitswerte (`true`, `false`).

Exkurs: Datentypen `float`, `double` (1)

Rundungen:

- Der Typ `float` hat (etwas mehr als) 6 signifikante Dezimalstellen.
- Man kann damit z.B. folgende Zahl darstellen:

$$0.0000123456 = 1.23456 * 10^{-6}$$

- Folgende Zahl kann man dagegen nicht darstellen:

$$1.0000123456$$

  Dies würde 11 signifikante Stellen benötigen.
- Da man nur 6 signifikante Stellen hat, würde diese Zahl gerundet zu: 1.00001

Exkurs: Datentypen `float`, `double` (2)

Rundungen, Forts.:

- Bei Gleitkommazahlen kann es zu Rundungsfehlern kommen, deren Effekt bei komplizierten Berechnungen undurchschaubar ist.
- Übliche mathematische Gesetze gelten nicht, z.B.:

$$(A + B) + C = A + (B + C)$$

(Assoziativgesetz). Verletzt für:

  - $A = +1000000$
  - $B = -1000000$
  - $C = 0.0001$

Exkurs: Datentypen `float`, `double` (3)

Rundungen, Forts.:

- Wenn sich Rundungsfehler (wie im Beispiel) fortpflanzen, ist es möglich, daß das Ergebnis absolut nichts mehr bedeutet.

  Also ein mehr oder weniger zufälliger Wert ist, der weit entfernt vom mathematisch korrekten Ergebnis ist.

- Geldbeträge würde man z.B. mit `int` in Cent repräsentieren, und nicht mit `float`.
- Es gibt Bibliotheken, die mit Intervallen rechnen, so daß man am Ende wenigstens merkt, wenn das Ergebnis sehr ungenau geworden ist.

Exkurs: Datentypen `float`, `double` (4)

Interne Repräsentation des Datentyps `float`:

- Nach dem IEEE Standard 754 werden für 32-Bit Gleitkommazahlen 24 Bit für die Mantisse verwendet, 8 Bit für den Exponenten zur Basis 2 (-126 bis $+127$), und 1 Bit für das Vorzeichen.

  Das sind eigentlich 33 Bit. Da das erste Bit der Mantisse aber immer 1 ist, wird es nicht explizit abgespeichert, man braucht also in Wirklichkeit nur 23 Bit für die Mantisse. Dies betrifft aber nur die sogenannten "normalisierten Zahlen". Der Standard behandelt auch nicht normalisierte Zahlen, $+\infty$, $-\infty$ und einen Fehlerwert ("not a number"). Dies erklärt, warum einige theoretisch mögliche Werte für den Exponenten nicht vorkommen. Der Wert des Exponenten e wird übrigens immer als $e + 127$ in den Bits 23 bis 30 abgespeichert (Bit 31 ist das Vorzeichen, Bit 0 bis 22 die Mantisse). Der Wert 0 und der Wert 255 im Exponentenfeld werden für die Spezialfälle (wie ∞) verwendet.

Exkurs: Datentypen `float`, `double` (5)

Datentyp `float`:

- (Etwas mehr als) sechs signifikante Dezimalstellen.
- Kleinster positiver Wert: $1.17549 * 10^{-38}$
- Größter Wert: $3.40282 * 10^{38}$

Datentyp `double`:

- 8 Byte (53 Bit Mantisse, 11 Bit Exponent).

  Der Exponent (zur Basis 2) läuft im Bereich -1022 bis $+1023$.
- 15 signifikate Dezimalstellen.
- Kleinster positiver Wert: $2.22507385850720 * 10^{-308}$
- Größter Wert: $1.79769313486232 * 10^{308}$

Inhalt

1. Klassen und Attribute
   - Modellierung der realen Welt, Anwendung
   - Klassen, Objekte, Attribute
   - Variablen, Zuweisungen
2. Datentypen
   - Datentypen von Java (insbesondere primitive Typen)
3. Konstruktoren, Zugriffsschutz, Methoden
   - Definition eines Konstruktors
   - Zugriffsschutz, Methoden
   - Fortsetzung des Beispiels, Beziehung zwischen Klassen
   - Verwendung eines Arrays, Berechnung mit Schleife

Konstruktoren (1)

- In der ersten Programmversion wird das Objekt zunächst mit den Standardwerten für die Attribute (0, "") erzeugt, und die Werte dann gesetzt.
- Die Standardwerte sind für die Anwendung nicht sinnvoll: Es wäre besser, wenn die Attributewerte gleich bei der Erzeugung des Objektes auf korrekte Werte gesetzt werden.
- Es ist üblich, dafür ein Programmstück in der Klasse zu schreiben, das man Konstruktor nennt.

  Es kann auch mehrere alternative Konstruktoren geben (wird später erläutert).
- Java garantiert, dass der Konstruktor einer Klasse aufgerufen wird, immer wenn ein neues Objekt der Klasse erzeugt wird.

  Deswegen heißt er Konstruktor: Er konstruiert Objekte. (Genauer formt er Objekte aus dem Speicher, der ihm von Java zur Verfügung gestellt wird.)

Konstruktoren (2)

- Hat man einen Konstruktor definiert, der die Attribute auf sinnvolle Werte setzt, so kann man das später bei der Objekterzeugung nie mehr vergessen.
- Konstruktoren können Parameter (Eingabewerte, "Argumente") haben.
- Dies wird z.B. verwendet, um dem `Artikel`-Konstrukor die Werte für Bezeichnung, Preis, Dauer zu übergeben.
- Die Objekterzeugung sieht dann so aus:

```
Artikel vulkan =
    new Artikel("Vulkan Magic Light", 550, 60);
```

Konstruktoren (3)

- Beispiel für Klasse mit Konstruktor:

```
(1)  class Artikel {
(2)      String bezeichnung; //Artikel-Name
(3)      int preis; // Preis in Cent ohne MwSt
(4)      int dauer; // Dauer in Sekunden
(5)
(6)      Artikel(String b, int p, int d) {
(7)          bezeichnung = b;
(8)          preis = p;
(9)          dauer = d;
(10)     }
(11) }
```

Parameter (im Beispiel `b`, `p`, `d`) sind Variablen, die beim Aufruf einer Methode oder eines Konstruktors mit den dort angegebenen Werten initialisiert werden.

Konstruktoren (4)

- Ein Konstruktor besteht also aus:
  - dem Namen der Klasse,
  - in runden Klammern `(...)` den Parametern des Konstruktors, für die man beim Aufruf Eingabewerte angeben muss,

    Die Parameter werden wie Variablen deklariert, nur wird die Deklaration nicht jeweils mit "`;`" abgeschlossen, sondern die einzelnen Deklarationen werden durch "`,`" getrennt. Hat ein Konstruktor keine Parameter, schreibt man einfach "`()`".
  - und in geschweiften Klammern `{...}` dem Programmcode ("Rumpf") des Konstruktors.
- Wenn man einen Konstruktor mit Parametern deklariert hat, funktioniert `new Artikel()` nicht mehr.

  Es sei denn, man deklariert einen zweiten Konstruktor ohne Parameter.

Konstruktoren (5)

- Die Deklaration eines Konstruktors sieht ähnlich aus wie die Deklaration einer Methode (wie `main`), aber er
  - heißt immer wie die Klasse, und
  - hat keinen Rückgabe-Typ.

    Er kann auch nicht `static` sein.
- Im Konstruktor kann man sowohl auf die Parameter zugreifen, als auch auf die Attribute der Klasse.

  Die Parameter sind Variablen, die mit den beim Aufruf übergebenen Werten automatisch initialisiert sind. Wenn der Konstruktor fertig abgearbeitet ist, werden sie wieder gelöscht (der Speicherplatz wird freigegeben).
  Die Attribute sind Variablen, die in dem Objekt gespeichert werden, und so lange existieren wie das Objekt.

Konstruktoren (6)

- Der Beispielkonstruktor kopiert die Eingabewerte von den Parametern in die Attribute des Objektes.
- Natürlich kann der Konstruktor mehr machen, als nur Eingabewerte 1:1 in Attribute zu kopieren:
  - Der Konstruktor könnte die Eingabewerte prüfen, z.B. negative Preise/Brenndauern zurückweisen.

    Allgemein gelten häufig "Integritätsbedingungen" ("Invarianten") für eine Klasse. Der Konstruktor muss natürlich überwachen, dass sie bei der Objekt-Erzeugung nicht schon verletzt sind (s.u.).
  - Oft gibt es auch Attribute, die mit festen Werten initialisiert werden, oder mit aus den Parametern berechneten Werten.
  - Manchmal muss das neue Objekt in Datenstrukturen zum späteren Zugriff eingetragen werden.

Konstruktoren (7)

- Die bei der Objekt-Erzeugung angegebenen Werte werden entsprechend ihrer Reihenfolge in der Liste den Parametern zugewiesen.
- Weil im Konstrukter `p` als zweiter Parameter angegeben wurde, wird der zweite Wert (`550`) diesem Parameter zugewiesen.

  Wenn man die Werte für Preis und Dauer vertauscht, bekommt man keine Fehlermeldung vom Compiler: Beides sind `int`-Werte und können `int`-Parametern zugewiesen werden. Im Konstruktor könnte man testen, dass Brenndauern nicht unrealistisch lang sind, über 5min sind sehr selten (hier hat man nur die Wahl zwischen zwei Übeln, dies wäre ja doch eine Einschränkung). Wenn man die Zeichenkette an falscher Position übergibt, liefert der Compiler dagegen einen Typfehler. Eine mögliche Lösung wäre es, eine extra Klasse für Preise zu machen, und dann ein `Preis`-Objekt zu übergeben. Das kann der Compiler natürlich von einem `int` unterscheiden.

Verschattung (1)

- Man kann Attribute und Parameter nennen, wie man will.
- Wenn Parameter des Konstruktors direkt den Attributen entsprechen, sind die naheliegenden Namen gleich.

  Wenn man die Parameter nicht künstlich umbenennt, und z.B. nur einbuchstabige Bezeichner verwendet (wie oben geschehen). Das ist aber schlechter Stil, zumindest, wenn der Konstruktor größer ist, so dass man die Bedeutung nicht mehr sofort überblickt. Variablennamen sollten verständlich sein. Natürlich sind sinnvolle Namen bei den Attributen wichtiger als bei den Parametern, weil sie in einem größeren Bereich des Programms verwendet werden.

- Der Kopf des Konstruktors wäre dann also:

```
Artikel(String bezeichnung, int preis, int dauer)
```

Verschattung (2)

- Wenn Attribute und Parameter gleich heißen, weiß der Compiler nicht, auf was man sich beziehen will.

  Es sind ja unterschiedliche Speicherstellen.

- Die Lösung ist, dass "gewinnt", was näher deklariert ist, in diesem Fall also der Parameter. Das Attribut ist "verschattet" (nicht direkt zugreifbar).

- Eine Anweisung wie

  ```
  preis = preis;
  ```

  ist ja auch offensichtlich sinnlos: Der Wert wird von einer Variablen in die gleiche Variable kopiert.

  Wenn man Glück hat, bekommt man eine Warnung, ansonsten tut der Compiler genau dies. Rechner denken sich nichts, sondern tun stupide, was man ihnen sagt.

Verschattung (3)

- Wenn Sie aber Attribute und Parameter so nennen wollen, gibt es eine Lösung. Schreiben Sie:

```
this.preis = preis;
```

- Das Schlüsselwort "`this`" bedeutet immer das aktuelle Objekt, für das der Konstruktor also gerade aufgerufen wurde.

  Es kann ähnlich wie eine Variable vom Typ der jeweiligen Klasse verwendet werden.

- Deswegen ist die Situation hier klar:
  - Bei `this.preis` kann `preis` nur das Attribut sein.
  - Rechts vom Gleichheitszeichen könnte es Attribut oder Parameter sein, aber da der Parameter näher deklariert ist, gewinnt er.

Verschattung (4)

- Damit geschieht also das Gewünschte:

  ```
  this.preis = preis;
  ```

  kopiert den Wert vom Parameter in das Attribut.

- Manche Java-Programmierer schreiben beim Zugriff auf Attribute im Konstruktor und in Methoden der Klasse immer "`this.`", um deutlich zu machen, dass es sich nicht um einen Parameter / eine Variable der Methode handelt.

  Gewöhnen Sie sich einen Programmierstil an und bleiben Sie dabei (dann brauchen Sie über solche Details nicht mehr nachzudenken).
  In einem Team sollte man natürlich einen gemeinsamen Stil verwenden.

Defensives Programmieren (1)

- Ein guter Konstruktor sollte testen, ob die Eingabewerte sinnvoll sind, also z.B. Preis und Brenndauer nicht negativ.
- Man macht bei der Programmierung manchmal Fehler. Zusätzliche Tests im Programm können:
  - die Fehlersuche erleichtern und
  - die Korrektheit des Endergebnisses verbessern.

    Der Aufwand für den zusätzlichen Test macht sich also bezahlt.
- Besonders wichtig ist die Überprüfung aller Benutzereingaben, und von Werten, die aus einer Datei gelesen wurden.
- Das Programm sollte eine sinnvolle Fehlermeldung ausgeben, und z.B. nicht ein falsches Ergebnis oder Daten zerstören.

  Auch Hacker machen sich ungeprüfte Eingaben oft zunutze.

Defensives Programmieren (2)

```
(1)  class Artikel {
(2)     String bezeichnung; //Artikel-Name
(3)     int preis; // Preis in Cent ohne MwSt
(4)     int dauer; // Dauer in Sekunden
(5)
(6)     Artikel(String b, int p, int d) {
(7)        if(p < 0) {
(8)           System.out.println("Fehler: " +
(9)              "Artikel.preis ungültig:" + p);
(10)          System.exit(1);
(11)       }
(12)       ... // Entsprechend für Dauer d
(13)       bezeichnung = b;
(14)       preis = p;
(15)       dauer = d;
(16)    }
(17) }
```

Defensives Programmieren (3)

- Das obige Programm gibt eine Fehlermeldung aus und beendet anschließend die Programmausführung durch:

```
System.exit(1);
```

  Die 1 ist ein Statuswert, den ein anderes Programm, das dieses Programm gestartet hat, abfragen könnte. Der Wert 0 bedeutet "in Ordnung", kleine positive Werte bedeuten Fehler, negative Werte sollte man nicht verwenden.

- Dies ist eine einfache, aber nicht unbedingt die beste Art der Fehlerbehandlung:
  - Es ist nicht klar, ob der Benutzer die Fehlermeldung zu sehen bekommt.

    Er könnte die Ausgabe in eine Datei umgeleitet haben.

    Bei GUI-Programmen sind Konsolenausgaben auch problematisch.
  - Eventuell gehen beim plötzlichen Programm-Abbruch wichtige Daten verloren.

Defensives Programmieren (4)

- Wir werden später noch andere Arten der Fehlerbehandlung kennenlernen, insbesondere mit Exceptions.
- Es ist war wichtig, sich gleich anzugewöhnen,
  - über Korrektheits-Bedingungen für Daten nachzudenken,

    In der Programmierung werden Bedingungen, die das Programm sicherstellt, meist "Invarianten" genannt. Bei Datenbanken sind es "Integritätsbedingungen".
  - eher zu viele Tests in das Programm einzubauen als zu wenig.

    Falls Sie Effizienz-Fanatiker sind, und die winzige Verzögerung für den zusätzlichen Test nicht mögen, gibt es auch Assertions ("Zusicherungen"), die bei der Programmentwicklung getestet werden, aber nicht in der eigentlichen Anwendung (siehe später). Dies geht natürlich nicht für fehlerhafte Benutzereingaben.

Zugriffsschutz (1)

- In der aktuellen Programmversion ist aber nicht garantiert, dass Preis und Brenndauer immer nicht-negativ sind.
- Das Problem ist, dass man im Moment noch von außerhalb der Klasse auf die Attribute der Klasse zugreifen kann.
- Z.B. könnte das Hauptprogramm `main` folgende Zeile enthalten:

```
vulkan.dauer = -10;
```

- Diese nachträgliche Änderung des Attributwertes würde den Test im Konstruktor umgehen.

  Während man bei so kleinen Programmen wie unserem Beispiel den Fehler natürlich leicht finden kann, kann das bei großen Programmen natürlich recht aufwändig sein.

Zugriffsschutz (2)

- Bei der Deklaration von Komponenten einer Klasse kann man auswählen, wer darauf zugreifen darf:
  - Mit dem Schlüsselwort "`private`" stellt man sicher, dass nur Programmcode aus der Klasse selbst zugreifen kann.
  - Wenn man dagegen nichts angibt, können alle Klassen des "Paketes" (größere Strukturierungseinheit von Programmen) darauf zugreifen.

    Pakete werden später behandelt. Unsere kleinen Programme liegen immer ganz in einem Paket (dem unbenannten Paket). Pakete sind nur für sehr große Programme interessant, oder wenn man Bibliotheken zur Verfügung stellen will, die auch in anderen Projekten genutzt werden können. Viele Programmierer halten es für unglücklich, dass Java sehr großzügige Zugriffe zulässt, wenn man nichts angibt.
    Bei C++ ist die Voreinstellung für Komponenten von Klassen "`private`".
    Es gibt noch `protected` und `public`: Siehe späteres Kapitel.

Zugriffsschutz (3)

- Es ist üblich, Attribute immer `private` zu deklarieren, und Zugriffe nur über Methoden der Klasse zu erlauben.

  Eine mögliche Ausnahme sind Attribute, die als "`final`" deklariert sind (s.u.).

  Diese können nach der Initialisierung (Konstruktor) nicht mehr geändert werden.

  Dann werden über diesen Mechanismus Änderungen von außen verhindert.

- Der Programmcode dieser Methoden kann dann Integritätsbedingungen (Invarianten) überwachen.
- Die Attributdeklaration würde dann so aussehen:

```
(1)  class Artikel {
(2)      private String bezeichnung;
(3)      private int preis;
(4)      private int dauer;
```

In C++ schreibt man "`private:`", das gilt für alle folgenden Komponenten.

In Java muss man den "Access Modifier" jeweils einzeln angeben.

Zugriffsschutz (4)

- Oft möchte/muss man aber einen lesenden Zugriff auf die Attribute der Klasse von außen erlauben.

  Wenn man z.B. im Hauptprogramm ein Objekt der Klasse `Artikel` hat, soll es möglich sein, dessen Bezeichnung, Preis und Brenndauer abzufragen. Das wäre nach der `private`-Deklaration nicht direkt möglich: Jeder Zugriff von außen ist verboten.

- Die Lösung ist, dass man Methoden deklariert, die den jeweiligen Attributwert liefern.

  Die Methode liefert den Wert, aber nicht die Variable. Daher hat der Aufrufer so keine Möglichkeit, die Variable zu ändern. Es wird nur ein Lesezugriff unterstützt, aber kein Schreibzugriff (d.h. nicht das Speichern eines neuen Wertes).

- Diese Methoden dürfen dann natürlich nicht als "`private`" deklariert sein, sonst kann man sie von außen nicht aufrufen.

Zugriffsschutz (5)

- Falls man Änderungen eines Attributes erlauben will, kann man dafür auch entsprechende Methoden deklarieren. Diese nehmen die Änderung nur vor, wenn der neue Wert die jeweiligen Korrektheits-Bedingungen erfüllt.

  Der Mehrwert ist also, dass man vor der eigentlichen Zuweisung, die dann in der Methode ausgeführt wird, prüfen kann, ob z.B. der neue Preis auch nicht negativ ist.

- Es hängt von der Anwendung ab, auf welche Attribute überhaupt ändernd zugegriffen werden muss.

  Manche Attribute werden im Konstruktor initialisiert und behalten dann den gleichen Wert für den Rest des Programmlaufs. In diesem Fall bietet man zwar eine Lesefunktion an (zum Abfragen des Wertes), aber keine zum Schreiben (Ändern) des Wertes.

Methoden (1)

- Methoden sind benannte Stücke von Programmcode, die man durch Angabe ihres Namens (und ggf. Werten für Parameter) "aufrufen", d.h. ausführen kann.
- Methoden werden wie Attribute innerhalb von Klassen deklariert.

  Es sollen jetzt Methoden der Klasse `Artikel` definieren. Methoden beschreiben Operationen/Aktionen, die man mit Objekten der Klasse ausführen kann.
- Das "Hello, World!" Beispiel enthielt die Methode `main`.

  Diese Methode war insofern speziell, als sie durch das Schlüsselwort "`static`" ohne Objekt aufgerufen werden kann, und dafür natürlich auch keinen Zugriff auf eventuelle Attribute (Variablen eines Objektes) hat. Normalerweise muß beim Aufruf ein Objekt angegeben werden (sofern nicht implizit das aktuelle Objekt `this` verwendet wird), und die Methode hat dann Zugriff auf die Variablen in diesem Objekt.

Methoden (2)

- Beispiel für Methoden zum Lesezugriff auf Attribute:

```
(5)  String getBezeichnung() {
(6)      return bezeichnung;
(7)  }
(8)
(9)  int getPreis() { return preis; }
(10)
(11) int getDauer() { return dauer; }
```

- Die Anweisung "`return` X`;`" beendet die Ausführung der Methode und liefert X als Ergebniswert des Aufrufs.

  Beim speziellen Typ `void` ist keine `return`-Anweisung erforderlich, weil es dort keinen Ergebniswert gibt. Man kann aber `return;` verwenden, um die Ausführung vorzeitig zu beenden. Ansonsten endet sie nach Ausführung der letzten Anweisung in `{...}`.

Methoden (3)

- Eine Methodendefinition besteht aus
  - dem Kopf (mit Ergebnistyp, Methodenname, Parameterliste),

    Optional können ganz zu Anfang noch "Modifizierer" (engl. "Modifier") angegeben werden, dazu zählen z.B. "`private`", "`public`", "`static`". Die Reihenfolge der Modifizierer ist egal, aber sie müssen vor dem Typ kommen. Natürlich geht "`private`" und "`public`" nicht gleichzeitig.
  - dem Rumpf (ein Block, Folge von Anweisungen).
- Die erste Methode im Beispiel
  - heißt "`getBezeichnung`",
  - liefert eine Zeichenkette (Wert von Typ "`String`"),
  - hat keine Parameter (Eingabewerte), markiert durch `()`,
  - hat einen Rumpf, der nur aus der `return`-Anweisung besteht.

Methoden (4)

- Beispiel für eine Methode zur Änderung des Preises (Schreibzugriff auf Attribut "`preis`"):

```
(13)  void setPreis(int p) {
(14)      if(p < 0) {
(15)          System.out.println("Fehler: " +
(16)              "Artikel.preis ungültig:" + p);
(17)          System.exit(1);
(18)      }
(19)      preis = p;
(20)  }
```

  Dieser Programmcode steht noch innerhalb der Deklaration der Klasse `Artikel`.

- Diese Methode hat einen Parameter `p` vom Typ `int`.
- Diese Methode liefert keinen Ergebniswert, erkennbar am Schlüsselwort "`void`" anstelle des Ergebnis-Typs.

Methoden-Aufruf (1)

- Im Hauptprogramm kann man die Bezeichnung eines Artikels jetzt so ausgeben:

```
System.out.println(vulkan.getBezeichnung());
```

- Der Methoden-Aufruf besteht also aus
  - der Angabe eines Objektes, für das die Methode ausgeführt werden soll,

    Das kann entfallen, wenn die Methode für das aktuelle Objekt ausgeführt werden soll, wenn also eine Methode der Klasse eine andere aufruft. Bei statischen Methoden (als "`static`" markiert) benötigt man auch kein Objekt, sondern nur den Namen der Klasse.
  - einem Punkt "`.`"
  - dem Name der Methode,
  - in runden Klammern `(...)` ggf. Werte für die Parameter.

Methoden-Aufruf (2)

- Da die Methode `getBezeichnung()` eine Zeichenkette als Ergebniswert liefert, kann ihr Aufruf wie eine Zeichenkette benutzt werden.
- Das Beispiel enthält noch einen weiteren Methoden-Aufruf:

  `System.out.println(...)`

  ruft die Methode `println(String)` für das Objekt `System.out` auf.

  Die Variable "`out`" ist in der Bibliotheks-Klasse "`System`" als "`public static`" deklariert, deswegen kann man auf sie durch Angabe des Klassen-Namens zugreifen (sie ist außerdem "`final`", d.h. nicht änderbar).
- Änderung des Preises (wirkt wie Zuweisung):

```
vulkan.setPreis(600);
```

Beispiele für Fehler (1)

- Selbstverständlich kann man im Hauptprogramm, d.h. außerhalb der Klasse `Artikel`, nicht mehr direkt auf das Attribut zugreifen:

```
System.out.println(vulkan.preis);
```

liefert:

```
FW.java:6: preis has private access in Artikel
    System.out.println(vulkan.preis);
                                  ^
1 error
```

Korrektur: Das Attribut ist zwar "`private`", aber die Methode "`getPreis()`" ist nicht "`private`". Als Methode der Klasse "`Artikel`" kann sie auf das Attribut zugreifen und den Wert liefern. Man ersetzt also einfach "`vulkan.preis`" durch "`vulkan.getPreis()`".

Beispiele für Fehler (2)

- Was die Methode liefert, der Wert der Variablen "`preis`", nicht die Variable selber.
- Auf der linken Seite einer Zuweisung wird aber eine Variable benötigt, daher wird dies vom Compiler zurückgewiesen:

```
vulkan.getPreis() = 600;
```

- Die Fehlermeldung ist wieder sehr klar:

```
FW.java:7: unexpected type
required: variable
found   : value
    vulkan.getPreis() = 600;
                      ^
1 error
```

Beispiele für Fehler (3)

- Falls man sich beim Zugriff auf eine Methode verschreibt, z.B.

```
vulkan.setPrs(600);
```

  bekommt man diese Fehlermeldung:

```
FW.java:8: cannot find symbol
symbol  : method setPrs(int)
location: class Artikel
    vulkan.setPrs(600);
           ^
```

- Entsprechend: Tippfehler in Variable "`vulkan`" in "`main`":

```
FW.java:8: cannot find symbol
symbol  : variable vulkn
location: class FW
    vulkn.setPreis(600);
    ^
```

Beispiele für Fehler (4)

- Wenn man den falschen Datentyp für einen Parameter angibt, z.B. `setPreis(int)` mit einer Zeichenkette aufruft, erhält man diese Fehlermeldung:

```
FW.java:9: setPreis(int) in Artikel
        cannot be applied to (java.lang.String)
    vulkan.setPreis("abc");
           ^
```

- Wenn man den Parameter-Wert vergisst, entsprechend:

```
FW.java:10: setPreis(int) in Artikel
            cannot be applied to ()
    vulkan.setPreis();
           ^
```

Auch zu viele Parameter-Werte liefern eine Meldung dieses Typs. Es wird immer die Liste der Datentypen der tatsächlich angegebenen Werte angezeigt.

Übliche Methoden-Namen (1)

- Die Zugriffsfunktionen für das Attribut "`attr`" heißen normalerweise:
  - "`getAttr`": Lesefunktion, liefert aktuellen Attributwert,
  - "`setAttr`": Schreibfunktion, ändert den Attributwert.

    Diese Funktion hat einen Parameter, nämlich den neuen Wert. Dieser wird (nach eventuellen Tests) der Attribut-Variablen zugewiesen. Möglicherweise müssen auch noch abgeleitete Daten oder Zugriffsstrukturen aktualisiert werden (d.h. dem veränderten Attributwert angepasst). Nicht alle Attribute müssen nach der Konstruktion des Objektes noch verändert werden. Falls das nicht nötig ist, deklariert man keine Funktion "`setAttr`".
- Im Beispiel liefert das einen deutsch-englischen Mischmasch. Empfohlene Lösung: Ganz auf Englisch wechseln.

Übliche Methoden-Namen (2)

- Da Methoden Aktionen beschreiben, sind Verben oder Verphrasen besonders typisch für Methodennamen.

  Das ist aber keine Vorschrift, z.B. ist `length()` eine Methode der Klasse `String`.

- Methoden, die eine Bedingung `B` testen, also einen booleschen Wert (Wahrheitswert) liefern, heißen oft `isB`.

  Z.B. `isEmpty()` für `String`-Objekte.

- Methoden, die ein Objekt in einen anderen Datentyp `D` konvertieren, heißen üblicherweise `toD`.

  Z.B. ist `toString()` eine nützliche Methode für alle Objekte. Wenn wir `toString()` für die Klasse `Artikel` definieren, erhalten wir eine sinnvolle Ausgabe, wenn wir ein `Artikel`-Objekt mit `println` drucken.

Übliche Methoden-Namen (3)

- Es ist auch möglich, die Lesefunktion für das Attribut `attr` genauso zu nennen, wie das Attribut: `attr()`.
- Im Beispiel könnte man dann statt `vulkan.getPreis()` einfach `vulkan.preis()` schreiben.
- Der gleiche Name wird hier für Variable (Attribute) und Methode verwendet: Java kann beides durch den Kontext unterscheiden, nach einer Methode kommt immer eine "`(`".

  In C++ wäre das nicht möglich, dort kann man nicht den gleichen Namen für zwei verschiedene Komponenten der Klasse verwenden. In C++ können Funktionen auch als Werte verwendet werden, daher folgt nicht immer die "(".
- Im Folgenden soll diese Namensgebung verwendet werden (weniger Sprachwirrwarr, kein Unterschied von Attributzugriff zu berechnetem Wert, aber Nicht-Standard).

Weitere Methoden (1)

- Natürlich kann man noch beliebige weitere Methoden einführen, die für die Anwendung wichtig sind.
- Z.B. ist der Preis mit 19% MwSt (Brutto-Preis) ein abgeleitetes Attribut:

```
(21)  int bruttoPreis() {
(22)      return preis + (preis * 19 + 99) / 100;
(23)  }
```

19% erhält man durch Multiplikation mit 19 und anschließende Division durch 100. Bei der Division von positiven ganzen Zahlen rundet Java immer ab. Um in diesem Fall aufzurunden, wurde vor der Division noch 99 addiert. Natürlich könnte man auch mit 1.19 multiplizieren, aber dann wäre das Ergebnis vom Typ "`double`". Dies erfordert eine explizite Typumwandlung nach `int`, außerdem muss man die Funktion `Math.ceil` zum Aufrunden verwenden: `return (int) Math.ceil(preis * 1.19);`

Weitere Methoden (2)

- Man sollte eine Methode `toString()` definieren, die eine Objektbeschreibung als Zeichenkette liefert:

```
(24)  @Override public String toString() {
(25)      return bezeichnung +
(26)          "(Preis: " + preis + " Cent, " +
(27)          "Dauer: " + dauer + "s)";
(28)  }
```

- Wenn man ein Objekt druckt, wird diese Methode verwendet:

```
System.out.println(vulkan);
```

Wenn man die Methode `toString()` nicht definiert, gibt es schon eine vordefinierte ("ererbte") Methode, die würde z.B. "`Artikel@7919298d`" drucken, was nicht besonders hilfreich ist. Weil es diese Methode schon gibt, sollte man `@Override` angeben, und weil sie `public` ist, muss auch die hier neudefinierte Version `public` sein.

Prinzip der objektorientierten Programmierung (1)

- Es ist ein wesentliches Kennzeichen der objektorientierten Programmierung, dass man Daten zusammen mit Methoden (Programmcode) zum Zugriff auf die Daten "kapselt", d.h.
  - beides ist zusammen definiert, und

    Dass man Daten immer im Zusammenhang mit Funktionen zu ihrer Verarbeitung sehen muss, war nicht neu: Dies ist z.B. ein Kernpunkt der Theorie abstrakter Datentypen.
  - auf die Daten nur über die in der Klasse definierten Methoden zugreifen kann.

    Dazu müssen die Daten, also die Attribute, natürlich "`private`" deklariert sein.

Prinzip der objektorientierten Programmierung (2)

- Indem man den direkten Zugriff auf die Attribute verbietet, kann man
  - das Interface (Schnittstelle, die von außen aufrufbaren Methoden) konstant halten, aber
  - die Implementierung (wie die Methoden intern programmiert sind) ändern.

    Wenn die Berechnung des Bruttopreises nicht so einfach wäre, wäre es eine Alternative, eine zusätzliche Variable in der Klasse für den Bruttopreis zu deklarieren, und ihn nur einmal (im Konstruktor), ggf. auch in `setPreis`, zu berechnen. Die Methode `bruttoPreis()` würde dann nur den Wert der Variablen liefern und die Berechnung nicht jedesmal neu durchführen. Von außen wären diese beiden Implementierungen ununterscheidbar. Die Trennung von Interface und Implementierung ist aber auch keine neue Idee der objektorientierten Programmierung.

Fortsetzung des Beispiels (1)

- Der Abbrennplan für ein Feuerwerk legt fest, welche Artikel zu welchem Zeitpunkt gezündet werden sollen (ggf. auch mehrere gleichzeitig).

  Damit wird insbesondere auch die Reihenfolge der Artikel festgelegt.

- Der Abbrennplan ist eine Folge von Punkten.
- Zu jedem Punkt (Zündung) gibt es folgende Daten:
  - Laufende Nummer im Plan.
  - Zeitpunkt der Zündung.
  - Artikel
  - Menge (falls mehrere Exemplare des Artikels).

Fortsetzung des Beispiels (2)

- Die laufende Nummer im Plan ist oft die Kanalnummer für eine elektrische Zündanlage.

  Großfeuerwerke werden heute fast nur noch elektrisch gezündet.

- Der Zeitpunkt der Zündung wird hier in Sekunden seit dem Start des Feuerwerks gespeichert.

  In der Realität würde man für bestimmte Effekte schon mindestens eine Auflösung von Zehntelsekunden benötigen.

- Nicht selten werden mehrere Exemplare eines Artikels gleichzeitig gezündet.

  Z.B. nicht einen Vulkan, sondern ≥ 3 in einer Linie parallel zum Publikum. Zur Vereinfachung behandeln wir nicht den Fall gleichzeitiger Zündung verschiedener Artikel.

Fortsetzung des Beispiels (3)

- Der Abbrennplan für ein Mini-Feuerwerk könnte also z.B. so aussehen:

Nr.	Zeit	Artikel	Stück
1	0:00	Vulkan Magic Light	3
2	0:10	Feuertopf Silberweide	2
3	0:12	Feuertopf Silberweide	2
⋮	⋮	⋮	⋮

Real wären etwa 40–80 Punkte bei einem mittleren Feuerwerk ohne Musik. Bei großen musiksynchronen Feuerwerken könnten es auch Hunderte von Punkten sein. Eine weitere Information, die hier zur Vereinfachung weggelassen wurde, ist die Position, an der ein Artikel abgebrannt wird: Z.B. die Vulkane in der Mitte als Fächer, die Feuertöpfe links und rechts.

Fortsetzung des Beispiels (4)

- Nennen wir die Klasse für einen Punkt im Abbrennplan "`Zuendung`".

  Natürlich sollte man möglichst aussagekräftige Namen wählen, die auch für andere Personen verständlich sind (vielleicht arbeiten später weitere Programmierer in diesem Projekt mit). Übrigens ist "`Artikel`" mehrdeutig: Gemeint ist eigentlich "Artikeltyp", von dem es mehrere Exemplare geben kann.

- Neu ist nun, dass eine Beziehung zwischen Objekten der Klasse `Zuendung` und Objekten der Klasse `Artikel` dargestellt werden muss.

  Es ist eine ganz wesentliche Information, welcher Artikel(-typ) zu einem bestimmten Zeitpunkt gezündet werden soll.

- Das ist kein Problem, da Attribute auch (Referenzen auf) Objekte enthalten können: Klassen sind auch Datentypen.

Klasse Zuendung (1)

- Klasse Zuendung, Teil 1 (Attribute und Konstruktor):

```
(1)  class Zuendung {
(2)      private int nr;
(3)      private int zeit; // Sek. seit Start
(4)      private Artikel artikel;
(5)      private int stueck;
(6)
(7)      Zuendung(int nr, int zeit,
(8)               Artikel artikel, int stueck) {
(9)          this.nr = nr;
(10)         this.zeit = zeit;
(11)         this.artikel = artikel;
(12)         this.stueck = stueck;
(13)     }
(14)
```

Klasse Zuendung (2)

- Klasse Zuendung, Teil 2 (Zugriffsfunktionen):

```
(15)     // Nr der Zeile im Abbrennplan (Kanal):
(16)     int nr() { return nr; }
(17)
(18)     // Zeitpunkt der Zuendung in Sek.:
(19)     int zeit() { return zeit; }
(20)
(21)     // Artikel, der gezuendet wird:
(22)     Artikel artikel() { return artikel; }
(23)
(24)     // Stueckzahl (gleichzeitig gezuendet):
(25)     int stueck() { return stueck; }
(26)
```

Klasse Zuendung (3)

- Klasse Zuendung, Teil 3 (Ausgabe, Anfang):

```
(27)     @Override public String toString() {
(28)        // Spalte Zeit in Minuten+Sekunden:
(29)        int min = zeit / 60;
(30)        int sek = zeit % 60; //Divisionsrest
(31)
(32)        // Beginne die Zeile mit Minuten:
(33)        String text = min + ":";
(34)
(35)        // Fuege die Sekunden hinzu
(36)        // (zweistellig):
(37)        if(sek < 10) {
(38)           text = text + "0" + sek;
(39)        else
(40)           text = text + sek;
```

Klasse Zuendung (4)

- Klasse Zuendung, Teil 4 (Ausgabe, Fortsetzung):

```
(41)
(42)            //Anschliessend den Artikel:
(43)            text = text + " " +
(44)                   artikel.bezeichnung();
(45)
(46)            // Dann Stueckzahl in (...),
(47)            // aber nur, falls >1:
(48)            if(stueck > 1)
(49)               text = text + " (" + stueck +
(50)                  " Stueck)";
(51)
(52)            // Liefere Abbrennplan-Zeile:
(53)            return text;
(54)        }
(55)    }
```

Klasse für Abbrennpläne

- Die wesentliche Funktionalität von `Plan` ist:
  - Eine Methode `add`, mit der man ein `Zuendung`-Objekt zum Plan hinzufügen kann (als jeweils letztes Element der Liste).
  - Eine Methode `gesamtpreis`, die die Summe aller Einzelpreise berechnet, und dazu die Liste der `Zuendung`-Objekte durchläuft.
- Natürlich wären in dieser Klasse weitere Analyse- und Ausgabe-/Visualisierungs-Funktionen denkbar, dafür fehlt uns jetzt aber die Zeit.
- Die Liste der Punkte wird als "Array" repräsentiert.

  Die Kommandozeilen-Argumente waren auch ein Array. Mathematisch entspricht ein Array der Größe n einem n-dimensionalen Vektor.

Exkurs: Arrays

- Ein Array ist eine Datenstruktur, die aus n Variablen für einen Basis-Datentyp besteht.

  Die einzelnen Variablen werden über einen Index angesprochen, der von 0 bis $n-1$ läuft.

- Wir benötigen hier ein Array von `Zuendung`-Variablen. Den Datentyp des Arrays schreibt man `Zuendung[]`.

- Man erzeugt ein Array-Objekt mit

  `new Zuendung[`n`]`

  Hier muß man also die Größe n festlegen.

- Wenn das Array `a` heißt, greift man auf eine Variable im Array mit `a[i]` zu. Dabei ist `i` der Index ($0 \leq$ `i` $\leq n-1$).

- `a.length` liefert die Größe n des Arrays.

Klasse Plan (1)

- Klasse Plan, Teil 1:

```
(1)  class Plan {
(2)      private String ueberschrift;
(3)      private int anzPunkte;
(4)      private Zuendung[] plan;
(5)
(6)      Plan(String ueb, int maxPunkte) {
(7)          ueberschrift = ueb;
(8)          anzPunkte = 0;
(9)          plan = new Zuendung[maxPunkte];
(10)     }
(11)
(12)     String ueberschrift() {
(13)         return ueberschrift;
(14)     }
(15)
(16)     int anzPunkte() { return anzPunkte; }
```

Klasse Plan (2)

- Klasse Plan, Teil 2 (Hinzufügen zur Liste):

```
(17)     // Anhaengen eines Punktes (Zuendung)
(18)     // an die Liste:
(19)     void add(Zuendung z) {
(20)        if(anzPunkte < plan.length) {
(21)           plan[anzPunkte] = z;
(22)           anzPunkte = anzPunkte + 1;
(23)        }
(24)        else {
(25)           System.out.println(
(26)              "Zu viele Kanäle!");
(27)           System.exit(2);
(28)        }
(29)     }
```

Klasse Plan (3)

- Klasse Plan, Teil 3 (Berechnung Gesamtkosten):

```
(27)      // Summe der Kosten aller Artikel:
(28)      int gesamtkosten() {
(29)          int kosten = 0;
(30)          int i = 0;
(31)          while(i < anzPunkte) {
(32)              kosten = kosten +
(33)                  plan[i].artikel().preis() *
(34)                  plan[i].stueck();
(35)              i = i + 1;
(36)          }
(37)          return kosten;
(38)      }
(39)
(40)  } // Ende der Klasse Plan
```

Muster in der Programmierung

- Bei der Programmierung sollten sich bestimmte Muster mit der Zeit einprägen, so dass man nicht jedes Mal neu überlegen muss, wie man eine Schleife zum Durchlaufen eines Arrays schreibt.

  Ein versierter Programmierer würde allerdings eine `for`-Schleife verwenden, und `i++` zum Erhöhen von `i`. Aber das sind nur syntaktische Abkürzungen, es passiert genau das Gleiche (später mehr zu Syntax-Alternativen).

- Ein anderes typisches Muster in diesem Beispiel ist die "Akkumulatorvariable" `kosten`.
  - Sie wird mit 0 initialisiert und dann werden die Kosten jedes einzelnen Punktes (im Abbrennplan) immer auf den aktuellen Wert aufaddiert.
  - So enthält sie am Ende die Summe aller Kosten.

Mehr Berechnungen auf Abbrennplänen

Aufgaben:

- Berechnen Sie die Gesamtdauer des Feuerwerks als das Maximum von Zuendzeitpunkt + Brenndauer des Artikels (über allen Punkten).
- Schreiben Sie eine Methode `print` der Klasse `Plan`, die den gesamten Plan ausdruckt.

  Für die Methode `toString()` wäre ein langer, mehrzeiliger Text eher ungewöhnlich. Diese Methode sollte die Ueberschrift des Plans und die Anzahl der Punkte ausgeben, eventuell auch die Maximalanzahl (Array-Länge).
- Überlegen Sie sich, wie die Preise in üblicher Weise als Euro und Cent ausgegeben werden können.
- Schreiben Sie eine Methode zur Überprüfung, ob der Plan Lücken hat (wo nichts brennt).

Hauptprogramm (1)

```
(1)  // Programm zum Testen der Klassen Artikel,
(2)  // Zuendung, Plan.
(3)  class FW {
(4)
(5)     // Hauptprogramm:
(6)     static public void main(String[] args) {
(7)
(8)        // Begruessung ausgeben:
(9)        System.out.println(
(10)          "Feuerwerk-Planungssoftware 0.1");
(11)
(12)       // Artikel-Objekte erzeugen:
(13)       Artikel vulkan = new Artikel(
(14)          "Vulkan Magic Light", 550, 60);
(15)       Artikel ft = new Artikel(
(16)          "Feuertopf Silberweide, 700, 3);
```

Hauptprogramm (2)

```
(17)
(18)          // Abrennplan mit max. 10 Punkten:
(19)          Plan mini = new Plan("Mini-FW", 10);
(20)
(21)          // Plan fuellen (drei Zuendungen):
(22)          mini.add(new Zuendung(1, 0, vulkan, 3));
(23)          mini.add(new Zuendung(2, 10, ft, 2));
(24)          mini.add(new Zuendung(3, 12, ft, 2));
(25)
(26)          // Gesamtkosten berechnen und ausgeben:
(27)          System.out.println("Gesamtkosten: " +
(28)              mini.gesamtkosten());
(29)      }
(30)  }
```

Es würde natürlich auch reichen, den Plan gleich nur mit Platz für 3 Punkte (Zündungen) zu erzeugen. Die erzeugten `Zuendung`-Objekte werden nicht in einer Variablen zwischengespeichert, sondern direkt an die Methode `add` übergeben.