Objektorientierte Programmierung

Kapitel 2: Erste Schritte in der Programmierung

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

1. Rahmenprogramm
   - Wiederholung: Ausführung von Java-Programmen
2. Java als Taschenrechner
   - Ausgabebefehl, Methoden-Aufruf
   - Konstanten und Rechenoperationen
3. Variablen
   - Einführung, Deklaration, Wertausdrücke mit Variablen
   - Zuweisungen, Berechnungszustand, Anweisungsfolgen
   - Eingabe von der Tastatur
4. Bedingungen
   - if-Anweisung
5. Schleifen
   - while-Anweisung
6. Fehlersuche
   - Compiler-Warnungen, Testausgaben, Debugger

Wiederholung (1)

- Am “Hello, World!” Beispiel haben wir gesehen, wie ein Programm ausprobiert werden kann:
  - Man benutzt einen Editor, um das Programm (den Quelltext in der Sprache Java) in eine Datei einzugeben, z.B. die Datei “`Hello.java`”.

    Man kann Notepad benutzen, aber das ist wenig komfortabel. `Notepad++` wird häufig empfohlen, bzw. `gedit` unter Linux. Andere mächtige Editoren sind z.B. `gvim` und `Emacs`. `Word` eignet sich nicht für Programmcode, da es nicht nur den reinen Text, sondern auch zusätzliche Angaben zur Formatierung in der Datei speichert.
  - Dann benutzt man einen Compiler, um diesen Quelltext in Befehle für die Ziel-Maschine zu übersetzten, in unseren Fall die “Java Virtual Machine” (JVM).

```
javac Hello.java
```

Wiederholung (2)

- Ausführung des "Hello, World" Beispiels, Forts.:
  - Der Compiler erzeugt eine Datei "`Hello.class`", die die JVM-Befehle (Java Bytecode) enthält.

    Sie ist binär (keine Textdatei). Wenn Sie sie in den Editor laden, sehen Sie nur ein großes Durcheinander von Zeichen, weil die Bits, die eigentlich JVM-Befehle sind, als Zeichencodes interpretiert werden. Wenn Sie die JVM-Befehle sehen wollen: "`javap -c Hello.class`".
  - Weil das Java-Programm nicht direkt in Maschinencode für die eigentliche CPU unseres Rechners übersetzt wurde, benutzen wir jetzt das Programm "`java`", das die JVM in Software simuliert:

    ```
    java Hello
    ```
  - Bei der Ausführung wird der Text "Hello, World!" auf den Bildschirm (Console, Terminal) ausgegeben.

Wiederholung (3)

- Anstelle von Einzelwerkzeugen (Editor, Compiler, JVM) kann man auch eine integrierte Entwicklungsumgebung (IDE) verwenden, z.B.:
  - Eclipse [http://www.eclipse.org/]
  - NetBeans [http://netbeans.org/]
  - BlueJ [http://www.bluej.org/] (für Schulen).
- Eine IDE macht mehr automatisch und enthält Hilfen.

  Während ein Editor die Syntax der Programmiersprache nur beschränkt "kennt", läuft bei einer IDE normalerweise ständig ein Compiler (zumindest dessen Syntaxanalyse) im Hintergrund. Fehler werden direkt im Text markiert.
- Großer Funktionsumfang: Orientierung anfangs schwierig.
- In der Prüfung muss YAPEX verwendet werden, keine volle IDE (zeigt bessere Beherrschung der Sprache).

Wiederholung (4)

- Die Werkzeuge zur Ausführung von Java-Programmen werden Ihnen auch in der Übung gezeigt.

  Wenn Sie damit Probleme haben sollten, fragen Sie nach bzw. holen Sie sich Hilfe. Sie brauchen die Möglichkeit, Java-Programme ausprobieren zu können!

- Unter [http://ideone.com/] gibt es das auch im Web, das ist aber nur für einfache Programme zu Anfang geeignet.

- Mit dem Programm PuTTY [http://www.putty.org/] können Sie sich auf `anubis.informatik.uni-halle.de` einloggen, und dort `javac` und `java` nutzen.

  Editoren ohne graphische Schnittstelle, die Sie in diesem Terminal-Fenster verwenden können, sind z.B. z.B. `nano` [http://wiki.ubuntuusers.de/Nano], `ne` [http://ne.dsi.unimi.it/], `vi`, `emacs`. Dateien können Sie mit dem Programm `pscp` (kommt mit `PuTTY`) zwischen `anubis` und Ihrem PC übertragen.

Wichtige Einsicht: Syntaktische Korrektheit

- Programmiersprachen sind so gemacht, dass man die Programme (mit etwas Erfahrung) in etwa wie einen englischen Text lesen kann.
- Der Compiler ist aber eine (Software-)Maschine, und hat keinerlei “Verstand”.
- Er kann sich deswegen nicht denken, was man gemeint hat.
- Man muss Programme also aus den Textbausteinen so ähnlich konstruieren wie Maschinen aus Zahnrädern.

  Oder wie elektrische Schaltungen aus Elektronik-Bauteilen.
- Wenn etwas nicht genau passt, funktioniert es nicht.

  Ein Programm sieht aus wie ein Text, ist das aber nicht, oder jedenfalls nicht nur. Der eigentliche Zweck ist die Verarbeitung durch den Compiler und anschließende Ausführung auf dem Rechner.

Erstes Beispiel: Hello, World!

- Inhalt der Datei "Hello.java":

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- Java ist eine formatfreie Sprache: Zwischen den Worten und Symbolen kann man beliebig Leerzeichen, Tabulatoren und Zeilenumbrüche einfügen.

  Man könnte das ganze Programm also auch in eine Zeile schreiben. Die Zeilenstruktur und Einrückungen helfen nur, es dem menschlichen Leser leichter zu machen — dem Compiler ist es egal.

- Die Groß/Klein-Schreibung ist dagegen wichtig.

Klassendeklarationen

- Ein Java-Programm besteht aus Klassendeklarationen.

  Normalerweise sind Klassen Blaupausen zur Erzeugung von Objekten. Z.B. könnte es eine Klasse “`Kunde`” geben, und jedes einzelne Objekt repräsentiert einen Kunden, enthält also `Name`, `Vorname`, `KdNr`, u.s.w.

  Im Beispiel dient die Klasse nur als “Modul”, also als Einheit zur Strukturierung von Programmcode. Es werden keine Objekte der Klasse erzeugt.

- 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).

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 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 (innerhalb eines "Paketes": siehe späteres Kapitel).

  Wenn man später ein Objekt der Klasse anlegen will, muss der Compiler ja wissen, von welcher Klasse, d.h. was die zugehörige Klassendeklaration ist.

- Die Schlüsselwörter (oder "reservierten Wörter") von Java 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`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `try`, `void`, `volatile`, `while`.

Methoden

- Prozeduren/Funktionen/Methoden (grob gesprochen alles dasselbe) enthalten die eigentlich auszuführenden Befehle.
- Eine Methodendeklaration besteht aus:
  - "Modifizierern", im Beispiel `public` und `static`,
  - einem Ergebnis- oder Rückgabetyp, im Beispiel `void`,
  - dem Namen der Methode, in Beispiel `main`,
  - einer Parameterliste, eingeschlossen in `(` und `)`,

    Dies sind die Argument- oder Eingabewerte der Funktion. Bei der speziellen Methode "`main`" (Hauptprogramm, Start der Ausführung) ist vorgesehen, dass Kommandozeilen-Argumente übergeben werden. Sie werden hier nicht verwendet, müssen aber deklariert werden.
  - die eigentlich auszuführenden Befehle, eingeschlossen in `{` und `}` (Methoden-Rumpf).

Rahmenprogramm (1)

- Man muss das jetzt noch nicht alles verstehen.

  Klassen und Methoden werden später noch sehr ausführlich behandelt.

- Man kann das Rahmenprogramm auch erstmal als gegeben annehmen.

- Für uns ist zunächst nur der Rumpf der Methode "main" interessant (Hauptprogramm):

```
class Hello {
    public static void main(String[] args) {
        [                    ]
    }
}
```

Rahmenprogramm (2)

- Außerdem wird man natürlich einen eigenen Klassennamen wählen:

```
class [    ] {
    public static void main(String[] args) {
        [                              ]
    }
}
```

Auch der Parameter-Name "`args`" (Kommandozeilenargumente) ist beliebig.

- Wenn man die Klasse `XYZ` nennt, sollte sie in der Quelldatei `XYZ.java` deklariert werden.

Für Klassen, die nicht als "`public`" deklariert sind, ist das nicht unbedingt nötig. Das Ergebnis des Compiler-Aufrufs steht aber auf jeden Fall in `XYZ.class`. Es ist übersichtlicher, wenn beides zusammen passt.

Inhalt

1. Rahmenprogramm
   - Wiederholung: Ausführung von Java-Programmen
2. Java als Taschenrechner
   - Ausgabebefehl, Methoden-Aufruf
   - Konstanten und Rechenoperationen
3. Variablen
   - Einführung, Deklaration, Wertausdrücke mit Variablen
   - Zuweisungen, Berechnungszustand, Anweisungsfolgen
   - Eingabe von der Tastatur
4. Bedingungen
   - if-Anweisung
5. Schleifen
   - while-Anweisung
6. Fehlersuche
   - Compiler-Warnungen, Testausgaben, Debugger

Ausgabebefehle

- Der Rumpf einer Methode (z.B. von `main`) enthält eine Folge von Anweisungen (Befehlen, engl. "Statements").
- Ein Beispiel ist die Anweisung zur Ausgabe eines Textes:

```
System.out.println("Hello, world!");
```

- Das "`ln`" steht für "line". Es gibt auch eine Variante, die nicht einen Zeilenumbruch nach dem Text ausgibt:

```
System.out.print("Hello, world!");
```

- Das ist eventuell interessant, wenn man mehrere Textstücke hintereinander ausgeben will.

  Alternativ kann man Textstücke und andere auszugebene Daten auch mit dem Operator "+" zusammenfügen, das wird unten noch gezeigt.

Methodenaufruf (1)

- Ein Methodenaufruf (wie der Druckbefehl) besteht aus
  - Dem Namen einer Klasse, oder einem Ausdruck, der zu einem Objekt auswertbar ist.

    Es gibt zwei verschiedene Arten von Methoden: Die "statischen" Methoden brauchen eine Klasse, die "normalen" Methoden ein Objekt. Dies wird alles später noch ausführlich behandelt. Im Beispiel ist "`System.out`" ein Ausdruck, der ein Objekt liefert.
  - einem Punkt "`.`",
  - dem Namen der Methode (im Beispiel "`println`"),
  - einer öffnenden runden Klammer "`(`",
  - einer Liste von Argumenten (Eingabe der Methode),

    Im Beispiel gibt es nur ein Argument, nämlich den zu druckenden Text.
  - einer schließenden runden Klammer "`)`".

Methodenaufruf (2)

- Das Semikolon “;” am Ende gehört nicht mehr zum Methodenaufruf, sondern markiert das Ende der Anweisung.

  In diesem Fall besteht die Anweisung aus einem einzelnen Methodenaufruf.

- Auch mathematische Funktionen sind in Java Methoden.

- Zum Beispiel kann man so die $\sqrt{2}$ berechnen:

```
System.out.println(Math.sqrt(2.0));
```

- Die Zahl 2.0 ist die Eingabe für die Funktion `sqrt` (“square root”: Quadratwurzel).

  Diese Funktion ist in der Klasse `Math` definiert.

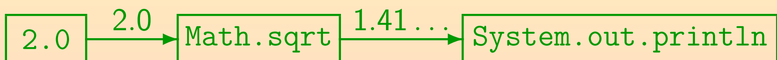
  [http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html]

- Der berechnete Wert wird dann weiter an die Methode `println` gegeben, um ihn auszudrucken.

Methodenaufruf (3)

- Die Struktur des Ausdrucks ist also so ähnlich wie ein geschachtelter Funktionsaufruf $f(g(x))$ in der Mathematik, nur dass die Funktionsnamen etwas länger sind.

  [Man könnte das Objekt für den Ausgabestrom als zusätzliches Argument der Druck-Funktion sehen, aber das wird später noch besprochen.]



- Die Methode `println` unterscheidet sich von einer mathematischen Funktion in zwei Punkten:
  - Sie liefert keinen Ergebniswert, man kann sie also nicht weiter in einen anderen Methodenaufruf schachteln.
  - Sie bewirkt eine Änderung des Berechnungszustandes, nämlich eine Ausgabe.

Konstanten/Literale: Zeichenketten (1)

- Mit Konstanten (auch Literale genannt) kann man Datenwerte aufschreiben.

- Das "Hello, world!"-Programm enthält z.B. einen Text (Folge von Zeichen, "Zeichenkette", engl. "String"):

```
"Hello, world!"
```

  Das englische Wort "String" bedeutet u.a. Schnur, vielleicht soll man sich die Zeichen wie an einer Perlenkette aufgefädelt vorstellen. In der Informatik übersetzt man "String" mit "Zeichenkette". Man kann solche Fachworte aber auch unübersetzt lassen. In Java sind Zeichenketten Objekte der Klasse `String`.

- Die Anführungszeichen am Anfang und Ende dienen nur der Markierung/Begrenzung der Zeichenkette, sie werden nicht mit ausgegeben.

  Das erste Zeichen der Zeichenkette ist "`H`", das letzte "`!`".

Konstanten/Literale: Zeichenketten (2)

- Zeichenkettenkonstanten bestehen aus
  - Einem Anführungszeichen " (Anfangs-Markierung)
  - fast beliebigen Zeichen und einigen Escapesequenzen (s.u.)
  - einem Anführungszeichen " (Ende-Markierung).
- "Fast beliebige Zeichen" bedeutet alle Zeichen außer
  - Anführungszeichen "

    Dann würde der Compiler ja denken, die Zeichenkettenkonstante ist zu Ende.
  - Rückwärtsschrägstrich \

    Dieses Zeichen wird zur Einleitung von Escapesequenzen gebraucht, s.u.
  - Zeilenumbrüchen ("Carriage Return", "Linefeed").

Konstanten/Literale: Zeichenketten (3)

- In Zeichenkettenkonstanten können verschiedene Escape-Sequenzen verwendet werden, z.B.
  - `\"` für ein Anführungszeichen,

    Durch die Markierung mit dem vorangestellten \ hält der Compiler dieses Anführungszeichen nicht für das Ende der Zeichenkette, sondern für ein Anführungszeichen.
  - `\\` für den Rückwärtsschrägstrich,

    Da der Rückwärtsschrägstrich jetzt eine Spezialbedeutung hat, braucht man auch eine Möglichkeit, ihn einzugeben.
  - `\n` für einen Zeilenumbruch.

    Eine String-Konstante muss auf derselben Zeile geschlossen werden, in der sie geöffnet wurde (falls man das schließende " vergisst, bekommt man so die Fehlermeldung am Ende der Zeile, statt am Ende der Datei). Bei Bedarf kann man mit “`\n`” einen Zeilenumbruch codieren.

Konstanten/Literale: Zeichenketten (4)

- Um Escape-Sequenzen zu verstehen, muss man Folgendes unterscheiden:
  - die Darstellung des Wertes im Java-Quellcode (Konstante)
  - den repräsentierten Wert (bei Ausführung des Programms).
- Z.B. ist `"\"\\"` eine Zeichenkette aus zwei Zeichen: einem Anführungszeichen und einem Rückwärtsschrägstrich.
  - `"\"\\"`: Erstes Zeichen: `"`.
  - `"\"\\"`: Zweites Zeichen: `\`.

  Solche unübersichtlichen Zeichenketten sind natürlich selten, meist steht eine einzelne Escape-Sequenz zwischen normalen Zeichen, so dass man sie leicht auseinander halten kann. Merken Sie sich nur, dass ein Rückwärtsschrägstich \ in Zeichenketten eine besondere Bedeutung hat, und " im Innern (ohne \) nicht erlaubt ist.

Konstanten/Literale: Zahlen (1)

- Im Beispiel mit der Berechnung der $\sqrt{2}$ stand eine Konstante für eine reelle Zahl:

```
2.0
```

- Solche Zahlkonstanten bestehen (vereinfacht) aus:
  - eine Folge von Ziffern vor dem Punkt,
  - einem Punkt "`.`" (Dezimalpunkt),
  - einer Folge von Ziffern nach dem Punkt.

    Es reicht, wenn vor oder nach dem "`.`" eine Ziffer steht (also `.3` oder `3.` wären ok). Außerdem gibt es noch die wissenschaftliche Schreibweise mit Zehnerpotenz und eine Hexadezimalschreibweise. Später ausführlich.
  - Selbstverständlich ist auch ein Vorzeichen möglich, aber das zählt formal schon als Rechenoperation (s.u.).

Konstanten/Literale: Zahlen (2)

- Selbstverständlich gibt es auch ganze Zahlen, sie werden einfach als Folge von Ziffern geschrieben, z.B.:

  `100`

  Wieder kann man effektiv ein Vorzeichen voranstellen. Weil das als eigene Operation zählt, wäre zwischen Vorzeichen und Zahl ein Leerzeichen möglich (aber nicht nötig), während man innerhalb der eigentlichen Zahlkonstante natürlich keine Leerzeichen verwenden kann (dann wären es zwei Zahlkonstanten hintereinander, das würde einen Syntaxfehler geben).

- Man muss führende Nullen vermeiden, z.B. ist `010` nicht die Zahl `10`, sondern `8`!

  Aus historischen Gründen (Kompatibilität zu C) schaltet Java auf Oktalschreibweise (zur Basis 8, nicht zur Basis 10) um, wenn die ganze Zahl mit `0` beginnt. Für eine einzelne `0` ist es egal, die bedeutet immer Null, egal, was die Basis ist.

Datentypen (1)

- Ein Datentyp ist eine Menge von Werten gleicher Art (zusammen mit den zugehörigen Operationen, s.u.).
- Java unterscheidet deutlich zwischen
  - reellen Zahlen wie `2.0`, diese haben den Datentyp "`double`",

    "`double`" kommt von "doppelte Genauigkeit": Intern kann man nur eine gewisse Anzahl Stellen darstellen, nicht beliebige reelle Zahlen. Es gibt auch einen Typ "`float`" für Zahlen mit einfacher Genauigkeit, aber den verwendet man kaum noch.
  - ganzen Zahlen wie `2`, diese haben den Datentyp "`int`",

    Von Englisch "integer": ganze Zahl.
  - Zeichenketten wie `"2"`, diese haben den Datentyp "`String`".

Datentypen (2)

- Die interne Repräsentation dieser Werte im Speicher des Rechners ist ganz unterschiedlich.

  Ein `int`-Wert braucht 32 Bit. Ein `double`-Wert braucht 64 Bit, und ist intern mit Mantisse und Exponent dargestellt (wie in "wissenschaftlicher Notation"). Ein `String`-Wert ist die Hauptspeicher-Adresse eines Objektes, dieses enthält in der OpenJDK-Implementierung ein Zeichen-Array (Speicherbereich für Zeichen) und drei ganze Zahlen, und ist damit (je nach Rechner) mindestens 128 Bit groß (plus den Speicherbereich für das Array).

- Datentypen helfen, Fehler zu vermeiden. Beispiel:

  ```
  System.out.println(Math.sqrt("abc"));
  ```

  Der Compiler gibt hier eine Fehlermeldung aus:

  "`method sqrt in class Math cannot be applied to given types; required: double, found: String`"

Datentypen (3)

- Eine Umwandlung von `int` nach `double` würde der Compiler bei Bedarf automatisch einfügen, z.B. würde

```
System.out.println(Math.sqrt(2));
```

funktionieren.

  Tatsächlich wird 2.0 an die `sqrt`-Funktion übergeben (“widening conversion”).

- Von manchen Funktionen gibt es mehrere Versionen für unterschiedliche Datentypen, z.B. gibt es von der `println`-Funktion Varianten für die Typen `int`, `double` und `String` (und noch weitere Typen).

  Die übergebenen Bits müssen ja richtig interpretiert werden. Es handelt sich hier um unterschiedliche Methoden, die zwar den gleichen Namen haben, aber aufgrund der Datentypen des übergebenen Wertes auseinander gehalten werden können (“überladene Methoden”).

Rechenoperationen (1)

- Selbstverständlich kennt Java die vier Grundrechenarten. Zum Beispiel gibt der Befehl

```
System.out.println(1 + 1);
```

  den Wert 2 aus.

  Die Leerzeichen links und rechts vom “+” sind optional (kann man weglassen).

- Java kennt die Regel “Punktrechnung vor Strichrechnung”:

```
System.out.println(1 + 2 * 3);
```

  gibt den Wert 7 aus, und nicht etwa 9.

- Wenn man möchte, kann man Klammern setzen:

```
System.out.println(1 + (2 * 3));
```

  Dies wäre die implizite Klammerung, hier wären die Klammern überflüssig.

  Will man aber die Addition vor der Multiplikation, wären die Klammern wichtig.

Rechenoperationen (2)

- Bei der Division ist zu beachten, dass bei `int`-Werten links und rechts die ganzzahlige Division (Division mit Rest) verwendet wird.
- Zum Beispiel druckt

  ```
  System.out.println(14 / 5);
  ```

  den Wert 2.

  Es wird immer in Richtung auf 0 gerundet, d.h. abgerundet (bei Ergebnis > 0).
- Den zugehörigen Rest erhält man mit dem Prozent-Operator:

  ```
  System.out.println(14 % 5);
  ```

  gibt 4 aus.

  Die Division ergibt 2 Rest 4. Die Bestimmung des Rests wird in der Mathematik auch mit dem Modulo-Operator `mod` geschrieben.

Rechenoperationen (3)

- Wenn die Operanden vom Divisions-Operator / `double`-Werte sind, findet die normale Division statt, und das Ergebnis ist ein `double`-Wert.
- Zum Beispiel liefert

```
System.out.println(14.0 / 5.0);
```

  den Wert `2.8`.

- Wenn nur einer der Operanden ein `double`-Wert ist, und der andere ein `int`-Wert, so wird der `int`-Wert automatisch in einen `double`-Wert umgewandelt, bevor die Division ausgeführt wird.

  Zum Beispiel liefert “`System.out.println(14.0 / 5);`” auch `2.8` (weil eigentlich `14.0 / 5.0` gerechnet wird).

Rechenoperationen (4)

- Der Operator `/` ist also auch überladen (wie `println`). Es gibt zwei verschiedene Versionen:
  - Die eine hat zwei Operanden vom Typ `int`, und liefert ein `int` (ganzzahlige Division).
  - Die andere hat zwei Operanden vom Typ `double`, und liefert ein `double`.

    Es gibt es noch Varianten für andere Typen (`float` und `long`): später.
- Tatsächlich gibt das für alle arithmetischen Operationen, aber bei der Division ist es am offensichtlichsten.
- Weitere Kombinationen von Eingabetypen werden behandelt, indem vor der Operation der Operand mit dem kleineren Typ `int` in den größeren Typ `double` umgewandelt wird.

Mathematische Funktionen (1)

- Damit Sie Java wie einen Taschenrechner benutzen können, hier noch einige mathematische Funktionen:
  - `Math.sqrt(x)`: $\sqrt{x}$
  - `Math.exp(x)`: e^x
  - `Math.pow(x,y)`: x^y
  - `Math.log(x)`: $\ln(x)$ (Logarithmus zur Basis e)
  - `Math.log10(x)`: $\log(x)$ (Logarithmus zur Basis 10)
  - `Math.sin(x)`: $\sin(x)$ (Argument in Bogenmaß/rad)
  - `Math.cos(x)`: $\cos(x)$ (Argument in Bogenmaß/rad)
- Die vollständige Liste finden Sie in der Dokumentation zur Java Platform SE 7 API (Klasse `Math` im Paket `java.lang`).

  [http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html]

Mathematische Funktionen (2)

- Die Klasse `Math` enthält außerdem noch zwei Konstanten:
  - `Math.PI`: $\pi = 3.14159\ldots$
  - `Math.E`: $e = 2.71828\ldots$ (Eulersche Zahl)
- Wenn man den Sinus von 30° berechnen will, kann man das z.B. so tun (180° entspricht π Radiant):

```
System.out.println(Math.sin(30*Math.PI/180));
```

  Das Ergebnis ist leider nicht ganz exakt: `0.49999999999999994`

- Es gibt aber auch eine eigene Funktion für die Umrechnung:

```
System.out.println(Math.sin(Math.toRadians(30)));
```

- Außerdem enthält die Klasse `Math` noch eine Funktion zur Berechnung von Zufallswerten:
  - `Math.random()`: Zufallswert im Intervall $[0, 1)$

Konkatenation von Zeichenketten (1)

- Der Operator + ist nicht nur für int- und double-Werte definiert, sondern man kann ihn auch auf Zeichenketten (Werte vom Typ String) anwenden.
- In diesem Fall fügt er die Zeichenketten zusammen (konkateniert sie):

```
System.out.println("abc" + "def");
```

  druckt: abcdef.
- Auch hier macht der Compiler bei Bedarf eine automatische Typanpassung: Wendet man + auf eine Zeichenkette und eine Zahl an, so wird die Zahl in eine Zeichenkette umgewandelt, und dann wird konkateniert.

Konkatenation von Zeichenketten (2)

- Wenn man z.B. nicht nur das “nackte” Ergebnis drucken will, kann man das so machen:

```
System.out.println("Die Wurzel aus 2 ist: " +
                   Math.sqrt(2));
```

- Aufgabe: Warum sind hier die Klammern wichtig?

```
System.out.println("1+1 = " + (1+1));
```

Hinweis: Wenn man keine Klammern setzt, wird implizit von links geklammert.

- Es gibt noch viele weitere Funktionen für Zeichenketten, die später in einem eigenen Kapitel behandelt werden.

[http://docs.oracle.com/javase/7/docs/api/java/lang/String.html]

Anweisungsfolge (1)

- Im Methodenrumpf kann man eine Folge von Anweisungen schreiben, z.B. drei Ausgabe-Anweisungen hintereinander:

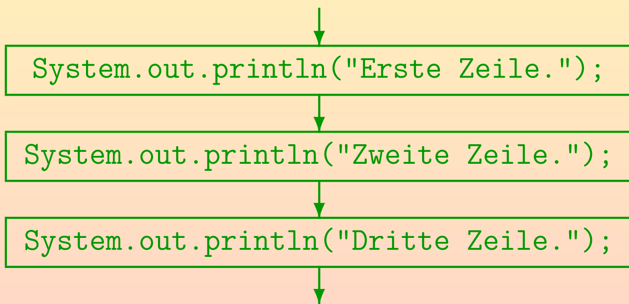
```
class Hello2 {
    public static void main(String[] args) {
        System.out.println("Erste Zeile.");
        System.out.println("Zweite Zeile.");
        System.out.println("Dritte Zeile.");
    }
}
```

- Die Ausgabe des Programms ist dann:

```
Erste Zeile.
Zweite Zeile.
Dritte Zeile.
```

Anweisungsfolge (2)

- Die Anweisungen werden in der Reihenfolge ausgeführt, in der man sie aufschreibt:



- Jede Anweisung modifiziert den Berechnungszustand, der nach der vorangegangenen Anweisung erreicht ist.

Inhalt

1. Rahmenprogramm
   - Wiederholung: Ausführung von Java-Programmen
2. Java als Taschenrechner
   - Ausgabebefehl, Methoden-Aufruf
   - Konstanten und Rechenoperationen
3. Variablen
   - Einführung, Deklaration, Wertausdrücke mit Variablen
   - Zuweisungen, Berechnungszustand, Anweisungsfolgen
   - Eingabe von der Tastatur
4. Bedingungen
   - if-Anweisung
5. Schleifen
   - while-Anweisung
6. Fehlersuche
   - Compiler-Warnungen, Testausgaben, Debugger

Variablen (1)

- Variablen ermöglichen die Speicherung von Werten während der Ausführung eines Programms (benannte Speicherstellen).
- Mit einer Variablen kann man also zwei Dinge tun:
  - einen Wert (z.B. eine Zahl) in die Variable speichern (hineintun, schreiben, zuweisen),

    Ein eventuell vorher in der Variablen gespeicherter Wert wird dabei "überschrieben" (verschwindet). In der Variablen steht immer nur ein Wert gleichzeitig. Im zeitlichen Ablauf kann sie aber verschiedene Werte nacheinander enthalten, deswegen der Name "Variable".

  - den Wert abfragen (lesen), der in der Variablen steht (um ihn in der weiteren Berechnung zu verwenden).

    So ähnlich wie man einen Text in eine Datei schreibt, und ihn dann beliebig häufig lesen kann. Im Gegensatz zu Dateien werden Variablen aber spätestens am Programmende gelöscht (genauer später).

Variablen (2)

- 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.

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

Variablen (3)

- Sie kennen Variablen aus der Mathematik.
- Auch dort sind sie ein Platzhalter in einer Formel, für den man später unterschiedliche Werte einsetzen kann, z.B.

$$f(x) = x^2 + 2 * x + 1$$

- Java löst allerdings keine Gleichungen für Sie, um unbekannte Werte zu ermitteln.

  Selbstverständlich können Sie selbst einen Gleichungslöser programmieren.

- Außerdem kann man in eine Java Variable nacheinander verschiedene Werte speichern. In der Mathematik kann der Wert zwar unbekannt oder beliebig sein, ist aber fest.

  Der zeitliche Ablauf spielt in der Mathematik keine Rolle, in Java schon.

  Dafür ist der Wert niemals wirklich unbekannt: Man hat ja vorher einen Wert in die Variable hineingespeichert.

Variablen: Motivation

- Variablen ermöglichen es,
  - Programmcode aufzuschreiben, der Berechnungen mit erst zur Laufzeit (bei Ausführung des Programms) bekannten Werten arbeitet,

    Wenn Sie das Programm schreiben, kennen Sie die vom Benutzer eingegebenen Werte ja noch nicht (auch der Compiler kennt sie nicht).
  - Programmcode aufzuschreiben, der mehrfach mit unterschiedlichen Werten ausgeführt werden kann,
  - einen Wert einmal zu berechnen und mehrfach zu verwenden,
  - kompliziertere Formeln in überschaubare Teile zu zerlegen,
  - einen Wert im Programm nur einmal aufzuschreiben, so dass er bei Bedarf leicht geändert werden kann.

Variablen: Eigenschaften

- Der Compiler speichert über Variablen folgende Daten:
  - ihren Namen (einen Bezeichner, engl. "Identifier"),

    Ein Bezeichner ist eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt, z.B. `betrag1` (keine Schlüsselworte wie "`class`").
  - ihren Datentyp, z.B. `int` (ganze Zahl, engl. "integer"),

    Jede Variable kann nur Werte eines Datentyps speichern.
    Je nach Datentyp wird unterschiedlich viel Hauptspeicher reserviert, z.B. 32 Bit (4 Byte) für ein `int` und 64 Bit (8 Byte) für ein `double`.
    Außerdem hängt die Interpretation des in der Variable gespeicherten Bitmusters vom Typ der Variablen ab.
  - eine Hauptspeicher-Adresse.

    Der Compiler reserviert für die Variable automatisch Platz im Hauptspeicher (RAM) ihres Rechners (für die Dauer der Ausführung des Programms, bzw. genauer des Methoden-Aufrufs).

Variablen: Deklaration, Initialisierung (1)

- Man teilt dem Compiler Namen und Datentyp der Variablen, die man verwenden will, in einer "Deklarationen" mit.
- Zum Beispiel wird hier eine Variable mit Namen "`n`" und Datentyp "`int`" deklariert:

```
int n;
```

- Es ist auch möglich, eine Variable gleich bei der Deklaration zu initialisieren, also einen ersten Wert in die Speicherstelle einzutragen:

```
int n = 1;
```

Wenn man dies nicht macht, stellt der Compiler sicher, dass man den Wert der Variablen nicht abfragen kann, bevor man nicht mit einer Zuweisung (s.u.) einen Wert eingetragen hat. (Dies gilt für die hier besprochenen "lokalen Variablen", andere werden ggf. automatisch initialisiert: später mehr.)

Variablen: Deklaration, Initialisierung (2)

- Eine Deklaration besteht also (etwas vereinfacht) aus:
  - Einem Datentyp, z.B. `int`, `double`, `String`,
  - einem Bezeichner (Variablennamen),
  - optional einem Gleichheitszeichen “`=`” und einem Wertausdruck (z.B. einer Konstanten passenden Typs),

    Wertausdrücke werden später noch ausführlich behandelt. Allgemein kann hier eine Formel stehen, die auch andere Variablen verwendet.
  - einem Semikolon “`;`”.
- Wenn man einen sinnvollen Wert für die Variable kennt, sollte man die Variante mit Initialisierung wählen.

  Das macht das Programm verständlicher und vermeidet Probleme, wenn der Compiler den Zugriff nicht zuläßt, weil er die Initialisierung garantieren muss.

Variablen-Namen (1)

- Es ist üblich, dass Namen von Variablen 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. 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.

- Meist haben Variablen in Methoden ("lokale Variablen") aber eher kurze Namen (nur ein Wort).

  Nicht selten auch nur ein Zeichen, wie `i`, `n`, `s`, `x`. Die Verständlichkeit des Programms darf darunter aber nicht leiden.

Variablen-Namen (2)

- Innerhalb einer Methode (wie z.B. `main`) kann man nicht zwei Variablen mit gleichem Namen deklarieren.
- Der Compiler muss ja wissen, auf welche Variable man sich bezieht, wenn man den Namen verwendet.
- Auch der Name "`args`" des Parameters geht nicht.

  Dies ist auch eine Variable, die ihren Wert automatisch beim Aufruf der Methode erhält.
- Selbstverständlich sind auch die reservierten Worte (Schlüsselworte) wie `class` ausgeschlossen.

  Liste siehe Folie 11.

Wertausdrücke (1)

- Wertausdrücke (oder kurz "Ausdrücke", engl. "expressions") beschreiben die Berechnung von Werten.
- Wertausdrücke entsprechen mathematischen Formeln, ein Beispiel für einen Wertausdruck ist:

```
2 * x + 1
```

- Z.B. wird ein Wertausdruck im Innern des Druckbefehls verwendet, um den auszugebenen Wert zu berechnen:

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

- Auch der Wert, der in eine Variable zu speichern ist, kann mit einem Wertausdruck berechnet werden:

```
double y =              ;
```

Wertausdrücke (2)

- Für Wertausdrücke hat man (etwas vereinfacht) folgende Möglichkeiten (Fortsetzung siehe nächste Folie):
  - Eine Konstante eines Datentyps (Literal), z.B.

    ```
    1.0
    ```

    In Java-Programmen wird die amerikanische Schreibweise mit Dezimalpunkt verwendet. In der Benutzer-Eingabe teils mit Komma.
  - Eine Variable (schon vorher deklariert und initialisiert), z.B.

    ```
    x
    ```

  - Ein Operator wie `+`, `-`, `*`, `/` mit einem Wertausdruck links und rechts, z.B.

    ```
    x + 1.0
    ```

    Weil links und rechts selbst ein Wertausdruck erlaubt ist, kann dort eine Konstante oder eine Variable stehen, oder auch komplexere Ausdrücke, die selbst Operatoren enthalten (später ausführlich).

Wertausdrücke (3)

- Möglichkeiten für Wertausdrücke (Forts.):
  - Man kann einen Wertausdruck in (runde) Klammern einschließen, und erhält wieder einen Wertausdruck:

    ```
    (x + 1.0)
    ```

    Weil das Ergebnis wieder ein Wertausdruck ist, könnte man es auch nochmals in Klammern einschliessen: `((x + 1.0))`. Dem Compiler ist es egal, für Menschen sieht es komisch aus.
  - Ein Aufruf einer mathematischen Funktion/Methode, z.B.

    ```
    Math.sqrt(x + 1.0)
    ```

    Das Argument des Funktionsaufrufs ist natürlich wieder ein Wertausdruck. Es gibt auch Funktionen mit mehreren Eingabe-Parametern, z.B. `Math.pow(x, y)` für x^y, dann gibt man entsprechend viele Wertausdrücke, durch Komma getrennt an.

Wertausdrücke (4)

- In der Mathematik schreibt man z.B.

$$y = x^2 + 2x - 5$$

- In den meisten Programmiersprachen
  - muss man das Multiplikationszeichen explizit setzen, also `2 * x` statt $2x$ schreiben,
  - gibt es keinen Operator für die Potenzierung, man muss also `x * x` oder `Math.pow(x, 2.0)` statt x^2 schreiben.

    `x * x` wird schneller berechnet und ist möglicherweise genauer als die allgemeine Funktion für Potenzen, die ja auch nicht ganzzahlige Potenzen behandeln muss.

- Daher schreibt man in Java:

```
y = x*x + 2*x - 5;
```

Zuweisungen (1)

- Man kann den Wert einer Variablen während der Programmausführung ändern.
- Dadurch kann man eine Anweisung, die im Programm nur ein Mal aufgeschrieben ist, mehrfach für unterschiedliche Werte der Variablen ausführen.

  Ohne diese Möglichkeit könnte man keine interessanteren Programme schreiben.
- Man speichert einen Wert in eine Variable mit einer “Zuweisung” (engl. “assignment”), bestehend aus
  - dem Namen der Variablen,
  - einem Gleichheitszeichen “=”,
  - einem Wertausdruck, der den neuen Wert berechnet,
  - einem Semikolon “;” als Ende-Markierung.

Zuweisungen (2)

- Ein Beispiel für eine Zuweisung ist

```
y = x + 2;
```

  - Dies nimmt den aktuellen Wert der Variablen `x`,
  - addiert 2,
  - und speichert das Ergebnis in die Variable `y`.

- Zuweisungen funktionieren also von rechts nach links: $\leftarrow$.

- Wenn `x` vor dieser Anweisung z.B. den Wert 5 hatte, hat `y` hinterher den Wert 7.

  Man sagt auch: "`y` wird auf 7 gesetzt." Falls die Variable `y` vorher noch keinen definierten Wert hatte (also ohne Initialisierung deklariert wurde), ist diese Zuweisung gleichzeitig eine Initialisierung.

- Am Wert von `x` ändert sich nichts, darin steht noch immer 5.

Zuweisungen (3)

- Tatsächlich funktioniert auch folgendes:

```
i = i + 1;  // Erhöht Wert von i um 1.
```

- Hier wird der aktuelle Wert von `i` genommen, 1 aufaddiert, und das Ergebnis in `i` zurück gespeichert (es überschreibt daher den bisherigen Wert von `i`).
- Spätestens hier bekommen Mathematiker Bauchschmerzen: `i` kann niemals gleich `i+1` sein.
- Eigentlich muss man diese Zuweisung so verstehen:

$$i_{neu} = i_{alt} + 1;$$

Die Zuweisung ist nicht der Vergleichsoperator, der wird `==` geschrieben (s.u.). Z.B. in Pascal wird die Zuweisung `:=` geschrieben, das war den C-Erfindern aber zu lang für eine so häufige Operation. Java folgt der C-Tradition.

Beispiel

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

        i = 1; // Zuweisung (Initialisierung)
        i = i + 1;
        i = i * i;
        System.out.print("i ist jetzt ");
        System.out.println(i);
    }
}
```

Beispiel: Ausführung (1)

- Es soll jetzt noch einmal Schritt für Schritt erklärt werden, wie das Beispiel-Programm ausgeführt wird.
- Es ist eine wichtige Fähigkeit eines Programmiers, die Ausführung eines Programms in Gedanken oder auf dem Papier zu simulieren.
- Man schlüpft dabei in die Rolle der CPU.
- Als Programmierer arbeitet man natürlich nicht auf der Ebene der Maschinenbefehle, sondern merkt sich stattdessen immer, welches die nächste auszuführende Anweisung des Java-Programms ist.
- Auf den Folien, die die Programm-Ausführung visualisieren, ist die nächste auszuführende Anweisung mit "$\Rightarrow$" markiert.

Beispiel: Ausführung (2)

- Solange man keine Kontrollstrukturen verwendet, wird einfach eine Anweisung nach der anderen abgearbeitet (in der Reihenfolge, in der sie aufgeschrieben sind).

  So wie der "Instruction Pointer" / "Program Counter" in der CPU einfach hochgezählt wird, wenn es sich nicht gerade um einen Sprungbefehl handelt. Genauer führen auch Methodenaufrufe (z.B. mit System.out.println) zu einem vorübergehenden Transfer der Kontrolle, aber je nach Abstraktionsebene kann man den Methodenaufruf auch als elementaren Schritt betrachten. Gerade Methoden aus der Java-Bibliothek (API) wird man einfach als Erweiterung der Sprache um neue atomare Befehle verstehen.

- Für reine Deklarationen wird kein Code erzeugt.
  - Sie bewirken zur Compilezeit eine Speicherreservierung,
  - werden aber zur Laufzeit im wesentlichen übersprungen.

    Außer bei Deklarationen mit Initialisierung.

Beispiel: Ausführung (3)

- Außerdem hat der Rechner noch Hauptspeicher, in den Daten geschrieben, und aus dem Daten ausgelesen werden.
- Auf Java-Ebene entspricht dem eine Tabelle mit den aktuellen Werten der Variablen.

  Natürlich steht vieles mehr im Hauptspeicher, z.B. das Programm, aber das wird bei der Simulation als fest gegeben angenommen.

- Beachte: "$\Rightarrow$" markiert die nächste auszuführende Anweisung, die gezeigten Variableninhalte entsprechen dem Zustand vor der Ausführung.
- Schließlich muss man ggf. noch über den Zustand der Ein-/Ausgabe Buch führen.

  Was wurde bisher eingelesen oder ausgedruckt? Z.B. markiert man das nächste einzulesende Zeichen der Eingabe mit $\uparrow$.

Beispiel: Ausführung (4)

i:	(unbekannt)
Ausgabe:	(leer)

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

    ⇒   i = 1; // Zuweisung (Initialisierung)
        i = i + 1;
        i = i * i;
        System.out.print("i ist jetzt ");
        System.out.println(i);
    }
}
```

Beispiel: Ausführung (5)

i:	1
Ausgabe:	(leer)

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

        i = 1; // Zuweisung (Initialisierung)
    ⇒   i = i + 1;
        i = i * i;
        System.out.print("i ist jetzt ");
        System.out.println(i);
    }
}
```

Beispiel: Ausführung (6)

i:	2
Ausgabe:	(leer)

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

        i = 1; // Zuweisung (Initialisierung)
        i = i + 1;
     ⇒  i = i * i;
        System.out.print("i ist jetzt ");
        System.out.println(i);
    }
}
```

Beispiel: Ausführung (7)

i:	4
Ausgabe:	(leer)

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

        i = 1; // Zuweisung (Initialisierung)
        i = i + 1;
        i = i * i;
    ⇒   System.out.print("i ist jetzt ");
        System.out.println(i);
    }
}
```

Beispiel: Ausführung (8)

i:	4
Ausgabe:	i ist jetzt

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

        i = 1; // Zuweisung (Initialisierung)
        i = i + 1;
        i = i * i;
        System.out.print("i ist jetzt ");
    ⇒  System.out.println(i);
    }
}
```

Beispiel: Ausführung (9)

i:	4
Ausgabe:	i ist jetzt 4

```
class Zuweisungen {
    public static void main(String[] args) {
        int i; // Deklaration der Variablen i

        i = 1; // Zuweisung (Initialisierung)
        i = i + 1;
        i = i * i;
        System.out.print("i ist jetzt ");
        System.out.println(i);
 ⇒  }
}
```

Aufgabe

Was gibt dieses Programm aus?

```
class Aufgabe {
    public static void main(String[] args) {
        int i;
        int n;

        i = 27;
        n = i - 20;
        i = 5;
        System.out.println(i * n);
    }
}
```

Beispiel: Eingabe (1)

- Programme werden erst richtig interessant, wenn man sie mehrfach mit verschiedenen Eingabewerten ausführen kann.
- Berechnung des Quadrates einer Zahl, (int-Wert), die von der Tastatur eingelesen wird:

```
import java.util.Scanner;
class Quadrat {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Bitte n eingeben: ");
        int n = input.nextInt();
        int q = n * n;
        System.out.println("n zum Quadrat: " + q);
    }
}
```

Beispiel: Eingabe (2)

- Es gibt in Java mehrere Möglichkeiten, wie man die Eingabe von der Konsole (Tastatur) durchführen kann.

  Alle sehen zunächst relativ kompliziert aus, aber man kann sich daran gewöhnen. Wenn man mehr über Klassen und Objekte weiss, wird es klarer.

- Eine Möglichkeit ist die Verwendung der `Scanner`-Klasse.

  Sie gehört seit Version 1.5 zum Java-Funktionsumfang.

- Weil sie nicht zum Java Kern gehört, braucht man eine `import`-Anweisung, um dem Compiler mitzuteilen, dass man diese Klasse verwenden will.

- In der Klasse `System` gibt es eine Variable `in`, die den Standard-Eingabestrom (normalerweise Tastatur) darstellt.

- Der Eingabestrom würde nur Folgen von Bytes liefern.

Beispiel: Eingabe (3)

- Es wird nun ein neues Objekt `input` der Klasse `Scanner` angelegt, das seine Eingabe aus dem Standard-Eingabestrom bezieht.

  `input` ist auch eine Variable, die aber keine ganze oder reelle Zahl enthält, sondern ein Objekt der Klasse `Scanner` (genauer eine Referenz auf so ein Objekt — das wird später noch sehr ausführlich diskutiert).

- Dieses Objekt kümmert sich darum, die Bytes zu Worten zusammenzufassen, und in den gewünschten Datentyp umzuwandeln.

- Die Klasse `Scanner` hat z.B. eine Methode `nextInt()`, die die nächste ganze Zahl aus der Eingabe liefert.

  Es gibt auch eine Methode `nextDouble()`, dann muss man aber bei den normalen Einstellungen für Deutschland ein Komma "`,`" statt dem Dezimalpunkt verwenden, also z.B. `100,00`.

Beispiel: Eingabe (4)

- Wenn man z.B. “`abc`” statt einer Zahl eingibt, bekommt man eine “`java.util.InputMismatchException`”, also einen Fehler.

- Das Programm wird abgebrochen, und die Fehlermeldung ist nur für Programmierer geeignet.

  Nicht für den Nutzer, der Quadratzahlen berechnen möchte.

- Wir besprechen später, wie man solche Fehler abfängt, und darauf reagieren kann, z.B.
  - eine verständliche Fehlermeldung ausgibt,
  - dem Benutzer eine neue Chance gibt, einen korrekten Zahlwert einzugeben.

Inhalt

1. Rahmenprogramm
   - Wiederholung: Ausführung von Java-Programmen
2. Java als Taschenrechner
   - Ausgabebefehl, Methoden-Aufruf
   - Konstanten und Rechenoperationen
3. Variablen
   - Einführung, Deklaration, Wertausdrücke mit Variablen
   - Zuweisungen, Berechnungszustand, Anweisungsfolgen
   - Eingabe von der Tastatur
4. Bedingungen
   - if-Anweisung
5. Schleifen
   - while-Anweisung
6. Fehlersuche
   - Compiler-Warnungen, Testausgaben, Debugger

Bedingungen (1)

- Bisher werden die Anweisungen in einem Programm einfach von oben nach unten der Reihe nach ausgeführt.
- So kann man natürlich noch keine sehr anspruchsvollen Programme schreiben.
- Es ist auch möglich, Anweisungen nur auszuführen, wenn eine Bedingung erfüllt ist.
- Im folgenden Beispiel-Programm (Folie 76)
  - wird der Benutzer gefragt, was $1 + 1$ ist,
  - eine Antwort eingelesen (ganze Zahl),
  - Falls die Antwort 2 ist, wird “`Ja, richtig!`” ausgegeben,
  - sonst “`Leider falsch`”.

Bedingungen (2)

- Ablauf des Programms als “Flussdiagramm”:

Bedingungen (3)

```
import java.util.Scanner;

class Einfach {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("Was ist 1+1? ");
        int antwort = input.nextInt();

        if(antwort == 2)
            System.out.println("Ja, richtig!");
        else
            System.out.println("Leider falsch.");
    }
}
```

Bedingungen (4)

- Bedingte Anweisungen sehen in Java so aus:

```
if ( Bedingung )
        Anweisung 1
else
        Anweisung 2
```

- Die erste Anweisung wird ausgeführt, wenn die Bedingung erfüllt ist.

  "if" bedeutet "wenn", "falls".

- Die zweite Anweisung wird ausgeführt, wenn die Bedingung nicht erfüllt ist.

  "else" bedeutet "sonst".

Bedingungen (5)

- Eine Bedingung ist ein Wertausdruck, der den speziellen Datentyp `boolean` (Wahrheitswerte, boolesche Werte) liefert.

  Benannt nach George Boole, 1815-1864.

- Dieser Datentyp hat nur zwei mögliche Werte:
  - `true`: wahr.
  - `false`: falsch.

- Die Vergleichsoperatoren `==` ($=$), `!=` ($\neq$), `<` ($<$), `<=` ($\leq$), `>=` ($\geq$), `>` ($>$) liefern boolesche Werte.

  In Klammern ist jeweils die in der Mathematik übliche Schreibweise angegeben.

- Man beachte, dass der Test auf Gleichheit "`==`" geschrieben wird, weil "`=`" schon für die Zuweisung verbraucht war.

Bedingungen (6)

- Den `else`-Teil kann man auch weglassen, wenn man nur im positiven Fall (Bedingung ist erfüllt) eine Anweisung ausführen möchte.
- Beispiel:

```
// Berechnung des Absolutwertes von x:
if(x < 0)
        x = -x;
```

- Falls mehr als eine Anweisung von `if` oder `else` abhängen, muss man sie in `{...}` einschließen.

  Beispiel siehe nächste Folie. Selbstverständlich kann man `{...}` auch verwenden, wenn nur eine Anweisung von `if` oder `else` abhängt. Manchmal wird das Anfängern empfohlen, um den potentiellen Fehler zu vermeiden, wenn man eine zweite Anweisung hinzufügt, und die geschweiften Klammern vergisst. Der Compiler beachtet die Einrückung ja nicht.

Bedingungen (7)

```
import java.util.Scanner;

class Einfach2 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("Was ist 1+1? ");
        int antwort = input.nextInt();

        if(antwort == 2)
            System.out.println("Ja, richtig!");
        else {
            System.out.println("Leider falsch.");
            System.out.println("Richtig waere: 2.");
        }
    }
}
```

Bedingungen (8)

- Formal sind
  - `if(` Bedingung `)` Anweisung
  - `if(` Bedingung `)` Anweisung 1 `else` Anweisung 2
  - `{` Anweisung 1 Anweisung 2 `...}`

  selbst wieder Anweisungen.

  Ebenso wie eine Zuweisung eine Anweisung ist, oder auch der Druckbefehl (Methodenaufruf). In einer Anweisungsfolge in `{...}` kann man auch Deklarationen als Anweisungen verwenden.

- Man kann also z.B. `else if`-Ketten wie im nächsten Beispiel bilden.

  Hier ist die von `else` abhängige Anweisung wieder eine `if...else`-Anweisung. Bedingte Anweisungen im `if`-Teil dagegen besser in `{...}` einschließen.

Bedingungen (9)

```
import java.util.Scanner;

class Vorzeichen {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("Ganze Zahl eingeben: ");
        int n = input.nextInt();

        if(n > 0)
            System.out.println("Positiv!");
        else if(n == 0)
            System.out.println("Null!");
        else
            System.out.println("Negativ!");
    }
}
```

Inhalt

1. Rahmenprogramm
   - Wiederholung: Ausführung von Java-Programmen
2. Java als Taschenrechner
   - Ausgabebefehl, Methoden-Aufruf
   - Konstanten und Rechenoperationen
3. Variablen
   - Einführung, Deklaration, Wertausdrücke mit Variablen
   - Zuweisungen, Berechnungszustand, Anweisungsfolgen
   - Eingabe von der Tastatur
4. Bedingungen
   - if-Anweisung
5. Schleifen
   - while-Anweisung
6. Fehlersuche
   - Compiler-Warnungen, Testausgaben, Debugger

Schleifen (1)

- Bisher wird jede Anweisung im Programm maximal ein Mal ausgeführt.
- In den meisten Programmen müssen aber bestimmte Anweisungen wiederholt ausgeführt werden.
- Hierfür gibt es verschiedene Konstrukte in Java. Das grundlegendste ist die `while`-Schleife:

```
while( Bedingung )
        Anweisung
```

- Hier wird zunächst die Bedingung getestet.
- Falls sie erfüllt ist, wird die Anweisung ausgeführt.

Schleifen (2)

- Dann wird wieder die Bedingung getestet.
- Ist sie noch immer erfüllt, wird die Anweisung erneut ausgeführt.
- Und so weiter (bis die Bedingung hoffentlich irgendwann nicht mehr erfüllt ist).
- Die Anweisung muss also (u.a.) den Wert einer Variable ändern, die in der Bedingung verwendet wird.

  Und zwar in eine Richtung, die schließlich dazu führt, dass die Bedingung nicht mehr erfüllt ist.

Schleifen (3)

- Falls die Schleifenbedingung immer erfüllt bleibt, erhält man eine Endlosschleife.

  Die CPU arbeitet hart, aber es geschieht nichts mehr (oder eine endlose Ausgabe rauscht vorbei, man wird immer wieder zu einer Eingabe aufgefordert ohne das Programm verlassen zu können, etc.).

- Man sagt dann auch: "Das Programm terminiert nicht."

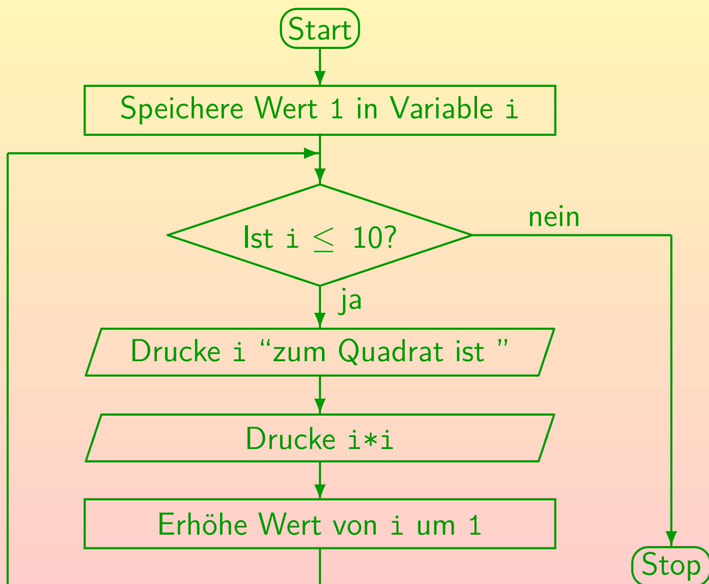
  Es hält nicht von selber an.

- Man kann Programme normalerweise mit `Crtl+C` abbrechen.

  Unter Windows kann man sich mit `Crtl+Alt+Delete` die Prozesse anzeigen lassen und das Programm abbrechen.

  Unter UNIX/Linux kann man mit `ps` oder `ps -ef` sich die Prozesse anzeigen lassen, und dann mit `kill ⟨Prozessnummer⟩` abbrechen, notfalls mit `kill -9 ⟨Prozessnummer⟩`.

Schleifen (4)



Schleifen (5)

```
class Quadratzahlen {
    public static void main(String[] args) {
        int i = 1;
        while(i <= 10) {
            System.out.print(i);
            System.out.print(" zum Quadrat ist ");
            System.out.println(i * i);
            i = i + 1;
        }
    }
}
```

Ausgabe:

```
1 zum Quadrat ist 1
2 zum Quadrat ist 4
3 zum Quadrat ist 9
...
10 zum Quadrat ist 100
```

Schleifen (6)

- Das Muster im obigen Programm ist sehr typisch:
  - Es gibt eine Laufvariable, im Beispiel `i`.
  - Diese wird zuerst initialisiert:

    ```
    i = 1;
    ```
  - In der Bedingung wird getestet, ob die Laufvariable schon eine gewisse Grenze erreicht hat:

    ```
    while(i <= 10)
    ```
  - Am Ende der Schleife wird die Laufvariable auf den nächsten Wert weitergeschaltet:

    ```
    i = i + 1;
    ```
- Die Variable `i` durchläuft also die Werte 1, 2, 3, . . ., 10.

  Weil dieses Muster so typisch ist, gibt es dafür in Java noch die `for`-Schleife als Abkürzung, die später behandelt wird (nicht unbedingt nötig).

Schleifen (7)

- Man hört öfters, dass Studenten von “If-Schleifen” reden. Das ist falsch!
- Die “If-Anweisung” (das “If-Statement”) ist keine Schleife, sondern gehört zu den bedingten Anweisungen.

  Eine Schleife ist etwas, wo die gleiche Anweisung wiederholt ausgeführt wird, eben in einer Art Kreis.

- Es gibt in Java die `while`-Schleife, die `for`-Schleife, und die `do`-Schleife. Mehr nicht.

  Von der `for`-Schleife gibt es noch eine Variante, die manchmal als `foreach`-Schleife bezeichnet wird. Sie verwendet aber das Schlüsselwort `for`. Wir besprechen später die verschiedenen Schleifentypen, aber wirklich fundamental ist nur die hier vorgestellte `while`-Schleife.

Aufgabe (1)

- Schreiben Sie ein Programm, das testet, ob eine eingegebene positive ganze Zahl n eine Primzahl ist.

  D.h. nur durch 1 und sich selbst teilbar. Primzahlen sind z.B. 2, 3, 5, 7, 11, 13.

- Den Teilbarkeitstest können Sie mit dem Modulo-Operator `%` ausführen: `a % b` liefert den Rest, der übrig bleibt, wenn man `a` durch `b` teilt.

  Z.B. ist `7 % 3 == 1`.

- Falls die Zahl keine Primzahl ist, geben Sie alle Teiler aus (von 2 bis $n - 1$).

- Falls die Zahl eine Primzahl ist, geben Sie den Text "`Primzahl!`" aus.

- Algorithmus siehe nächste Folie.

Aufgabe (2)

- Ein Algorithmus ist eine Beschreibung eines Verfahrens (Berechnungsvorschrift) für einen Menschen.

  Man kann daher natürliche Sprache verwenden, und das Verfahren auf einer etwas höheren Abstraktionsebene als Java-Befehle darstellen. Ein Algorithmus wird dann in einer Programmiersprache wie Java codiert, um ihn ausführen zu können.

- Beispiel (Primzahltest):
  - Lies die Zahl `n` ein.
  - Setze eine boolesche Variable `prim` auf `true`.
  - Lasse `i` in einer Schleife von `2` bis `n-1` laufen.
    Ist `n` durch `i` teilbar, so drucke `i` und setze `prim` auf `false`.
  - Falls nach der Schleife `prim` noch `true` ist,
    so gib “`Primzahl!`” aus.

Inhalt

1. Rahmenprogramm
   - Wiederholung: Ausführung von Java-Programmen
2. Java als Taschenrechner
   - Ausgabebefehl, Methoden-Aufruf
   - Konstanten und Rechenoperationen
3. Variablen
   - Einführung, Deklaration, Wertausdrücke mit Variablen
   - Zuweisungen, Berechnungszustand, Anweisungsfolgen
   - Eingabe von der Tastatur
4. Bedingungen
   - if-Anweisung
5. Schleifen
   - while-Anweisung
6. **Fehlersuche**
   - **Compiler-Warnungen, Testausgaben, Debugger**

Fehler in Programmen

- Programme funktionieren oft nicht sofort, wenn man sie eingetippt hat.
- Zunächst gibt der Compiler Fehlermeldungen aus, wenn der Programmtext kein gültiges Java ist.
- Nachdem man diese Fehler korrigiert hat, und das Programm durch den Compiler läuft, erhält man ein ausführbares Programm.
- Wenn man das Programm dann ausprobiert, tut es öfters nicht sofort das, was die Aufgabe verlangt.
- Dann muss man das Programm "debuggen".

    Ein "Bug" ist ein Programmierfehler.

Fehlervermeidung (1)

- Man spart viel Zeit, wenn man bei der Programmierung etwas länger nachdenkt, und dafür hinterher keinen Fehler suchen muss.
- Simulieren Sie das Programm im Kopf. Bei Schleifen sind besonders der erste und der letzte Durchlauf interessant.
- Wählen Sie sinnvolle, aussagekräftige Variablennamen.

  Nicht einfach `a`, `b`, `c`. Das Skript ist nicht immer vorbildlich, weil es auf den Folien wenig Platz gibt. Außerdem kann man bei kurzen Beispielen alles auch mit nicht aussagekräftigen Namen überblicken. Werden Sie sich aber in jedem Fall darüber klar, was genau in der Variablen stehen soll.
- Wählen Sie eine übersichtliche Formatierung Ihres Programmes (Einrückungen, Zeilenumbrüche, Leerzeilen).
- Nutzen Sie Kommentare.

Fehlervermeidung (2)

- Ein tückischer Fehler sind fehlende oder falsch gesetzte geschweifte Klammern und dazu inkonsistente Einrückungen.
- Beispiel:

```
if(x > 0)
    System.out.println("x = " + x);
    System.out.println("x ist positiv.");
```

- Die zweite Ausgabe gehört nicht zu der `if`-Struktur und wird daher immer ausgeführt.

  Wenn man keine `{...}` setzt, ist nur eine Anweisung vom `if` abhängig.

- Entwicklungsumgebungen (und spezielle "pretty printer") können Programmtext automatisch formatieren (u.a. einrücken). Dann sieht man es.

Fehlersuche (1)

- Man muss versuchen, zu verstehen, warum sich das Programm so verhält, wie man beobachtet, und nicht so, wie man beabsichtigt hat.

  Wie immer ist eine genaue Aufklärung des Fehlers Voraussetzung dafür, dass man ihn wirklich beseitigen kann. Wenn man nur an den Symptomen herumdoktert, erhält man vielleicht das richtige Verhalten für eine spezielle Eingabe, aber reißt häufig neue Löcher für andere Eingaben auf. Das kann beliebig lange dauern und sehr frustrierend sein. Investieren Sie lieber etwas Zeit, um den Fehler wirklich zu verstehen, als mit Programmänderungen herumzuprobieren (Testausgaben sind dagegen nützlich, siehe nächste Folie).

- Oft kann man die Ausführung der kritischen Stelle im Kopf simulieren, und das Verhalten erklären.

Fehlersuche (2)

- Eine übliche Methode ist, zusätzliche Ausgaben in das Programm einzubauen,
  - um zu sehen, ob bestimmte Zeilen im Programm überhaupt erreicht werden, oder auch,
  - um die Werte von Variablen zu kontrollieren.
- So kann man aktiv etwas unternehmen, um den Fehler einzukreisen.

  Wichtig ist natürlich, dass Sie in Ihrem Kopf ein klares Konzept haben, wie das Programm funktionieren soll. Sonst können Sie die Abweichung davon ja gar nicht an einer Stelle festmachen.
- Anschließend werden die Ausgaben wieder gelöscht.

Fehlersuche (3)

- Es macht allerdings Mühe, Ausgaben in das Programm einzubauen.

  Man muss es jeweils neu compilieren, und mit einer Runde von Änderungen für Testausgaben ist es selten getan. Außerdem kann sich das Programm in ungünstigen Fällen durch die eingefügten Ausgabeanweisungen anders verhalten, als ohne sie (der Unterschied könnte größer sein, als nur die zusätzliche Ausgabe).

- Professioneller ist die Nutzung eines Debuggers. Das ist ein Programm, das die Fehlersuche in anderen Programmen unterstützt, z.B. indem man
  - das Programm in Einzelschritten ausführen kann,
  - die Inhalte von Variablen inspizieren kann.