Objektorientierte Programmierung

Kapitel 3: Erste Programme mit Kontrollstrukturen

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2013/14

http://www.informatik.uni-halle.de/~brass/oop13/

Inhalt

Wiederholung: Variablen (1)

- Variablen ermöglichen die Speicherung von Werten während der Ausführung eines Programms.
- Die wesentlichen Dinge, die man mit einer Variablen tun kann, sind also:
  - einen Wert 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).

    Die Variable behält bei dieser Operation ihren Wert, der Wert wird also nicht "herausgenommen".

Wiederholung: Variablen (2)

- 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.

Wiederholung: Variablen (3)

- 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).

Datentypen (1)

- Datentypen dienen zur Klassifizierung von Werten.
- Z.B. ist die Division nur mit Zahlen möglich, nicht mit Zeichenketten.
- Auch bei Zahlen gibt es einen Unterschied zwischen
  - der Division zweier ganzer Zahlen, in diesem Fall ist das Ergebnis ganzzahlig (Division mit Rest),
  - der Division zweier reeller Zahlen, in diesem Fall ist das Ergebnis eine reelle Zahl.

    Im Rechner kann nur eine beschränkte Genauigkeit dargestellt werden.
- Indem man dem Compiler den Datentyp von Variablen mitteilt, kann er bestimmte Fehler schon erkennen, bzw. die richtige Variante einer Operation wählen.

Datentypen (2)

- Beispiele für Datentypen sind:
  - `int`: ganze Zahl
  - `double`: reelle Zahl beschränkter Genauigkeit

    Es ist schon die doppelte Genauigkeit gegenüber dem früher üblichen Typ "`float`" (von "Gleitkomma" oder "Fließkomma").
  - `boolean`: Wahrheitswerte: `true` (wahr) oder `false` (falsch)
  - `String`: Zeichenkette

    `String` ist eine vordefinierte Klasse. Es ist Konvention, dass Klassen groß geschrieben werden. Dagegen sind `int`, `double` und `boolean` primitive Typen, die fest in die Sprache Java eingebaut sind.
  - `java.util.Scanner`: Objekt zur Umwandlung von eingelesenen Zeichen in Worte oder Zahlen.

    Dies ist eine Bibliotheksklasse, die mit Java mitgeliefert wird.

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.

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, 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 oder einer Methode, wie im letzten Kapitel kurz beschrieben, 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)

- Jeder Wertausdruck hat einen Ergebnis-Datentyp.
- Dies gilt natürlich auch für Teilausdrücke in einem größeren Wertausdruck.
- Wenn ein Wertausdruck als Argument einer Funktion oder als Operand eines Operators verwendet wird, muss der Datentyp passen. Z.B. geht nicht:

```
Math.sqrt("abc")
```

- Die Funktion `Math.sqrt` erwartet einen Eingabewert vom Typ `double` (reelle Zahl).
- Eine ganze Zahl (`int`) würde der Compiler bei Bedarf automatisch nach `double` umwandeln, aber für eine Zeichenkette geht das nicht. Man bekommt einen Typfehler.

Wertausdrücke (5)

- 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;
```

Inhalt

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`.

- 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.
- Wie erwähnt, hat die CPU einen “Instruction Pointer” / “Program Counter”, der die Adresse des aktuell abzuarbeitenden Maschinenbefehls enthält.

Beispiel: Ausführung (2)

- 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.

  Meist entsprechen die Anweisungen den Zeilen. Gelegentlich erstreckt sich eine Anweisung über mehrere Zeilen, manchmal muß man auch innerhalb einer Anweisung mehrere Schritte betrachten.

- Auf den Folien, die die Ausführung des Programms visualisieren, ist die nächste auszuführende Anweisung mit "$\Rightarrow$" gekennzeichnet.

Beispiel: Ausführung (3)

- 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), die man nicht selbst geschrieben hat, wird man einfach als Erweiterung der Sprache um neue atomare Befehle verstehen.

Beispiel: Ausführung (4)

- 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.
- 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 ↑.

Beispiel: Ausführung (5)

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

    Für Experten: Beim Prozeduraufruf gibt es einen Vorspann, der Register so setzt, dass für die lokalen Variablen Platz reserviert wird. Insofern gibt es doch etwas Programmcode (Maschinenbefehle), der für die Deklaration ausgeführt wird. Wenn man mehrere Variablen deklariert, wird aber in einem Schritt ein entsprechend größerer Speicherbereich reserviert. Man kann diesen Programmcode also nicht einer bestimmten Deklaration zuordnen. Anders ist die Deklaration mit Initialisierung oder Konstruktoraufruf: Dann gibt es natürlich Programmcode für die spezielle Deklaration.

Beispiel: Ausführung (6)

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 (7)

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 (8)

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 (9)

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 (10)

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 (11)

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);
    }
}
```

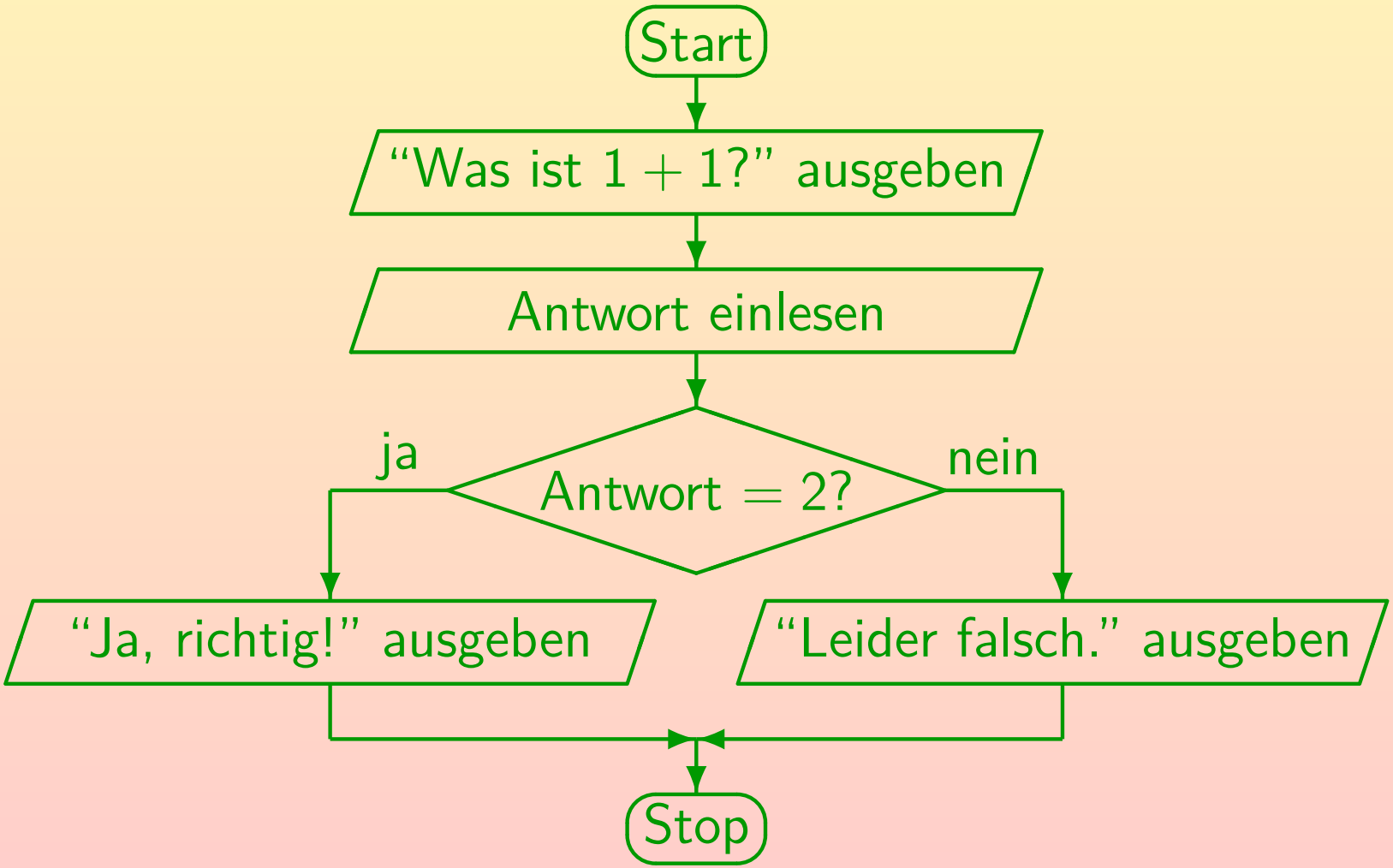
Inhalt

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 35)
  - 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, das 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

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 immernoch 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."

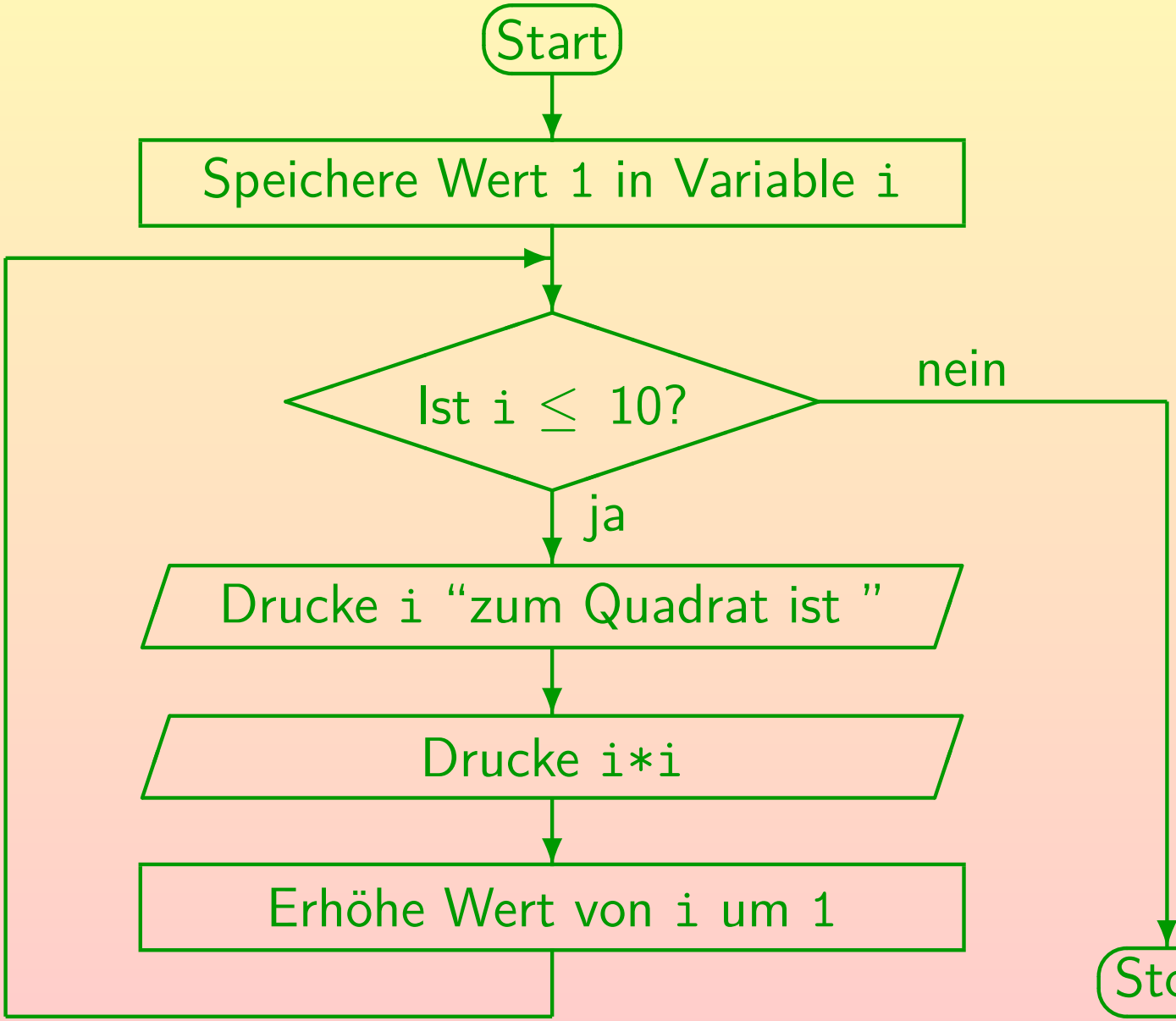
  Es hält nicht von selber an.

- 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)


Start
Speichere Wert 1 in Variable i
Ist i ≤ 10?
nein
ja
Drucke i "zum Quadrat ist "
Drucke i*i
Erhöhe Wert von i um 1
Stop

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, das 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

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)

- Man sollte probieren, die Warnungen des Compilers anzuschalten mit

```
javac -Xlint:all Hello.java
```

Warnungen werden gedruckt für Programmcode, der formal legales Java ist, aber vermutlich nicht, was Sie gemeint haben. Warnungen sind allerdings in Sprachen wie C++ wichtiger, weil die Sprache dort mehr "Unsinn" erlaubt. Java ist von Natur aus restriktiver. Deswegen erhält man auch seltener Warnungen. Wenn man aber eine Warnung bekommt, sollte man versuchen, sie zu verstehen. Falls das Programm tatsächlich korrekt ist, sollte man es so umschreiben, dass die Warnung nicht mehr auftritt. Wenn man unproblematische Warnungen einfach stehen läßt, bekommt man am Ende in einem großen Programm so viele Warnungen, dass man die eine wichtige darin übersieht.

Fehlervermeidung (3)

- 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 geschweiften Klammern `{...}` setzt, ist nur eine Anweisung vom `if` abhängig.

- Die in Kapitel 2 beschriebene automatische Formatierung (Einrückung) des Programms könnte hier helfen.

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.

Benutzung des jdb (1)

- Der "Java Debugger" `jdb` ist Teil des JDK.
- Vor Benutzung des Debuggers empfiehlt sich, beim Compilieren die Option `-g` anzugeben:

```
javac -g Hello.java
```

  Dadurch wird zusätzliche Information in der `class`-Datei abgelegt, die für den Debugger nützlich ist (z.B. die Namen der lokalen Variablen).

- Man ruft den Debugger `jdb` auf mit

```
jdb Hello
```

- Als nächstes setzt man einen "Breakpoint" auf den Anfang der Methode `main`, so dass die Programmausführung dort angehalten wird:

```
> stop in Hello.main
```

  Alternativ: mit "`stop at`" Breakpoint auf bestimmte Zeile setzen.

Benutzung des jdb (2)

- Anschließend startet man die Ausführung des Programms:

```
> run
```

- Die Ausführung hält dann beim Breakpoint an, also beim Beginn der Methode `main`.

```
Breakpoint hit: "thread=main", Hello.main(), line=4 bci=0
4       System.out.println("Hello, world");
```

- Man kann nun mit folgendem Befehl jeweils eine Anweisung ausführen:

```
> next
```

Alternativ würde man mit "`step`" in Methodenaufrufe hinein steigen, während "`next`" sie überspringt. Mit "`step out`" kann man aus dem aktuellen Methodenaufruf heraus kommen. Mit "`cont`" läuft das Programm bis zum nächsten Breakpoint (oder bis zum Ende).

Benutzung des jdb (3)

- Folgender Befehl zeigt die Werte der lokalen Variablen an:

```
> locals
```

- Folgender Befehl zeigt den Programmcode in der Umgebung der aktuellen Position:

```
> list
```

- Folgender Befehl zeigt den Wert eines Ausdrucks:

```
> print i
```

- Folgender Befehl zeigt die Interna eines Objektes:

```
> dump input
```

- Folgender Befehl zeigt Methoden-Aufrufhierachie:

```
> where
```

Benutzung des jdb (4)

- Da der jdb die Eingaben von der Tastatur selbst verarbeitet, funktioniert das Einlesen mit dem Scanner nicht.
  Man kann aber die Eingabe in das Programm schreiben:

```
Scanner input = new Scanner("123");
```

- Man beendet den jdb mit dem Befehl

```
> quit
```

- Man bekommt eine Liste der Befehle mit

```
> help
```

- Der jdb ist ein umfangreiches Programm mit vielen Befehlen (hier nur kleiner Ausschnitt).

  Anleitung:

  [http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html]

Benutzung des Eclipse-Debuggers (1)

- Selbstverständlich enthält auch Eclipse (und jede andere moderne IDE) einen Debugger.
- Man muß auch hier einen Breakpoint setzen, um die Ausführung des Programms zu unterbrechen, so dass man danach z.B. jede Anweisung einzeln ausführen kann.

  Dazu vorne vor der Zeilen in der man den Breakpoint setzen will, mit der rechten Maustaste klicken und im Pop-Up-Menu "Toggle Breakpoint" wählen (nochmal "Toggle Breakpoint" entfernt ihn wieder). Man kann auch `Ctrl+Shift+B` drücken, wenn der Cursor sich irgendwo in der Zeile befindet.
- Anschließend wählt man `Run→Debug` aus.

  Eclipse fragt dann, ob mit mit einem "Perspective Switch" einverstanden ist: Zum Debuggen ist eine andere Fensteraufteilung empfohlen. Man sollte "`Yes`" auswählen. Die aktiven Perspektiven werden oben rechts im Eclipse Fenster angezeigt, durch Drücken auf "`Java`" kann man wieder zurück.

Benutzung des Eclipse-Debuggers (2)

- Mit `Run→Step Over` (oder `F6`) kann man jeweils die nächste Anweisung ausführen.

  Sie ist in der Quellcode-Ansicht grün hinterlegt. Mit `Run→Step Into` (oder `F5`) würde man in aufgerufene Methoden hinein wechseln.

- Die Variablen mit ihren Werten werden oben rechts angezeigt.

  Dort gibt es auch eine Liste der Breakpoints.

- Mit `Run→Resume` (`F8`) kann man die Ausführung des Programms fortsetzen, entweder bis zum Ende, oder bis zum Erreichen des nächsten Breakpoints.

Benutzung des Eclipse-Debuggers (3)