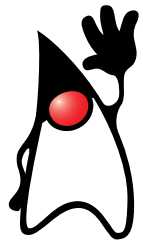
Übersicht

Übersicht

Java – Was ist das?

- Objektorientierte Programmiersprache
  - Entwickelt von Sun Microsystems (mittlerweile Oracle)
  - „Internet-Programmiersprache“
- Ziel: Plattformunabhängigkeit
  - z.B. PC (z.B. Linux, Mac OS/X, Windows), Mobiltelefone, ...
  - z.B. Intel x86, ARM, MIPS, ...
  - Programme können über Netzwerk verteilt und praktisch überall einfach ausgeführt werden
- Breite Unterstützung für Entwickler
  - Entwicklungsumgebungen
  - Bibliotheken („vorgefertigte“ Programmteile) z.B. für graphische Ausgabe, Datenbankzugriffe,...



Warum Java?

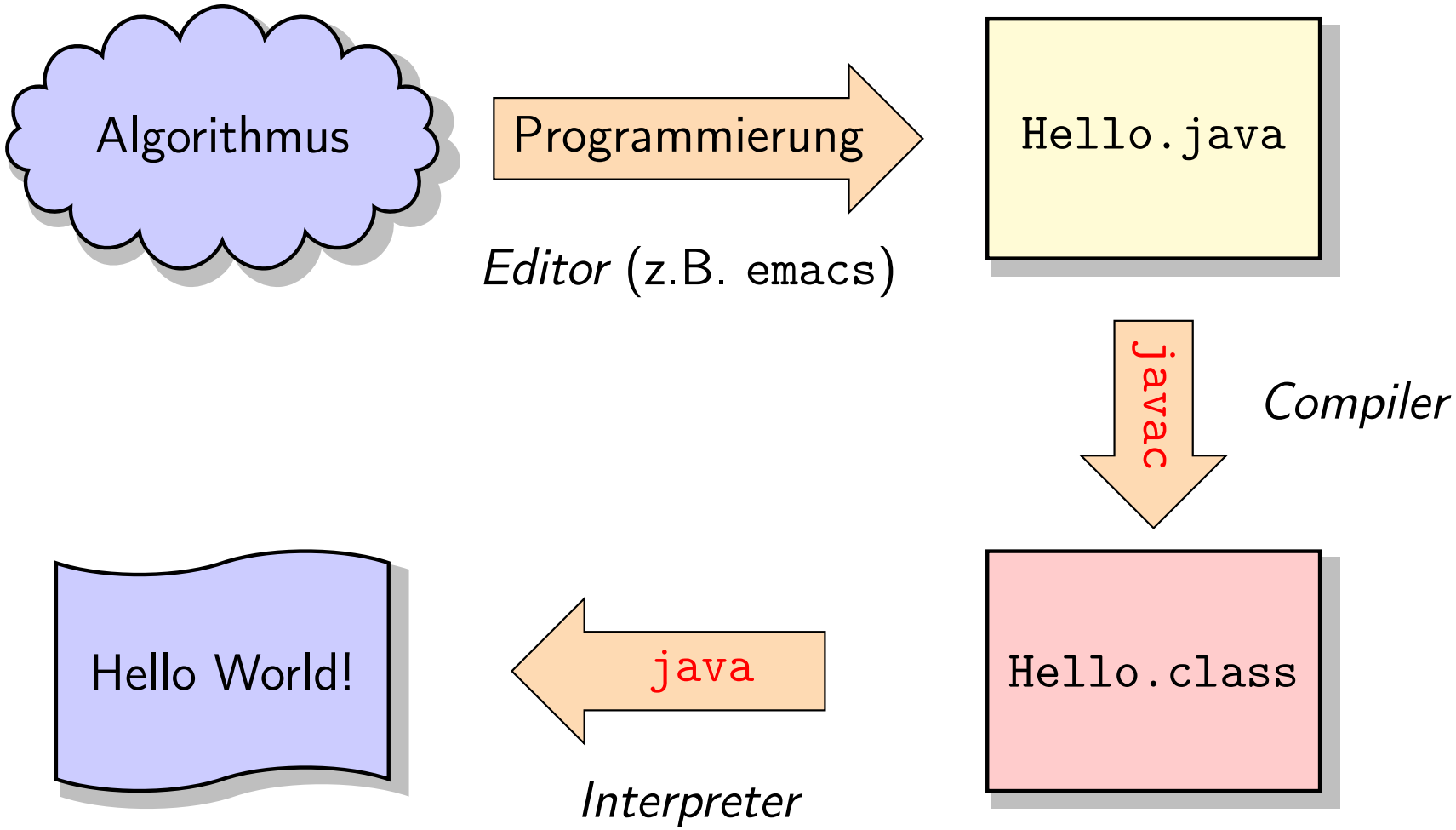
- Java ist . . .
  - leichter zu lernen als C/C++ (und viele andere Sprachen)
  - syntaktisch ähnlich zu C/C++ (für CV im nächsten Semester!)
  - *Wahrscheinlich die Sprache, die Euch am meisten nützt.*
- Java läuft (fast) überall.
- Es gibt viel Literatur zu Java.
- Es gibt zahlreiche Werkzeuge und Beispiele.

- Java ist trotz allem relativ komplex
  - z.B. im Vergleich zu Lisp, Haskell,. . .
  - *Wir fangen langsam an!*
  - *Wir werden nicht alles nutzen/behandlen!*

Eigenschaften von Java

- Objektorientierung
  - Klassen als Abstraktionskonzept
  - Weit verbreitetes, sehr populäres Programmmierparadigma
- Starke, statische Typisierung
  - Muss Typen von Daten (Variablen) im Code spezifizieren
  - Frühzeitiges Erkennen von Fehlern beim Übersetzen
- Automatisierte Speicherverwaltung
  - Bequem, viele Fehlerquellen entfallen
  - Überprüfung von Zugriffen zur Laufzeit
- Interpretiert und portabel
  - Plattformunabhängiger Zwischencode (Bytecode)
  - Ausführung durch virtuelle Java-Maschine
  - Abstraktion von Hardware und m Betriebssystem:
    Programme laufen ohne Änderung unter Unix, Windows,. . .
  - “Kleine Programme“

Essentielle Java Werkzeuge



Java Compiler und Interpreter

- Der Java *Compiler* `javac` übersetzt Quellcode in *Bytecode*.
  - Quellcode = Beschreibung des Algorithmus in Java
  - Quellcode (*source code*) in `*.java` Datei(en)
  - Compiler überprüft dabei Syntax (und anderes z.B. Typen)
- Der Bytecode ist ein *plattformunabhängiger* Zwischencode.
  - Im Gegensatz zu plattformabhängigen Maschinencode
  - Bytecode = Anweisungen an einen Prozessor
  - Bytecode wird in `*.class` Datei(en) gespeichert.
- Der Java *Interpreter* `java` interpretiert den Bytecode.
  - Führt Anweisungen aus: *virtuelle Maschine* (Java VM)
  - Überprüft dabei Speicherzugriffe
  - Beschränkt ggf. Ressourcen/Zugriffe (unter Sicherheitsaspekten)

Beispiel: Essentielle Java Werkzeuge

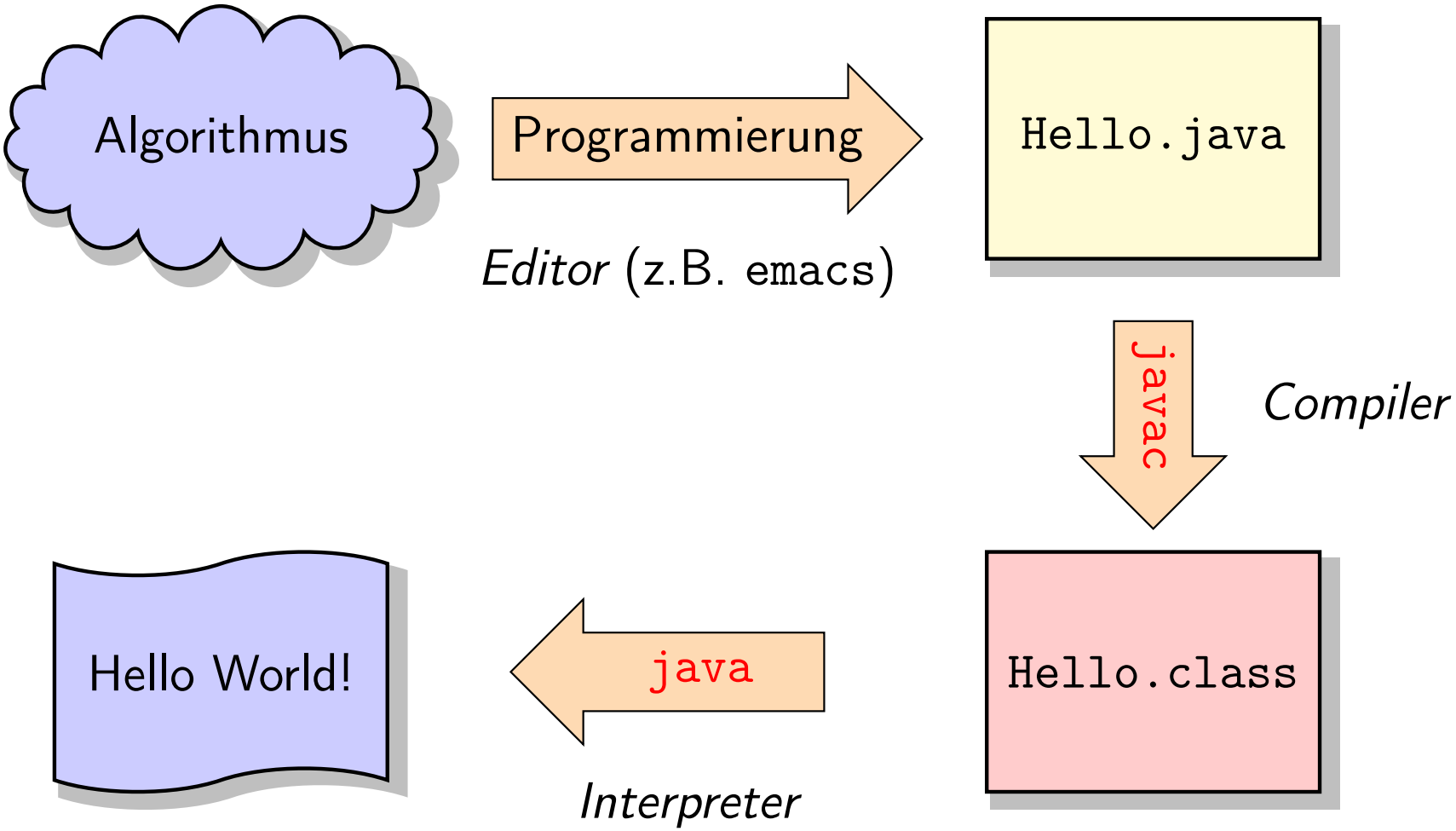
```
bash> emacs Hello.java          # Programmierung

bash> ls
Hello.java                      # => Hello.java

bash> javac Hello.java          # Compilieren
bash> ls
Hello.class Hello.java          # => Hello.class

bash> java Hello                # Ausfuehren
Hello World!
```

Essentielle Java Werkzeuge



Beispiel: `Hello.java`

- Datei `Hello.java`:

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

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

   }
}
```

- *Wir bleiben am Anfang bei diesem Grundgerüst:*
  - Klassenname `Hello` = *Dateiname* `Hello.java`
    Groß-/Kleinschreibung beachten! – Gilt *grundsätzlich* für Java!
  - Beschreibung des Algorithmus innerhalb von `main`:
    `System.out.println("Hello world!");`
- Beispiele auf Folien zeigen i.d.R. nur (Teile des) Algorithmus

Typisches Java Programm

- *Wird werden Programme sehr einfach halten!*
  z.B. ohne graphische Benutzeroberfläche

1. Einlesen von Eingabedaten
2. Verarbeitung durch Algorithmus
3. Ausgabe des Ergebnisses

- Eingabe
  - Argumente auf *Kommandozeile* (Text)
  - Von *Standardeingabe* oder Datei (meist Text)
  - Auf Folien (sonst nur in Ausnahmefällen): „Fest im Code“
- Ausgabe
  - *Standardausgabe* (Text z.B. auf Terminal)
  - Datei(en) (meist Text)
- *Mehr dazu in Übungen/Tutorien und auf der Vorlesungsseite*

Java Shell

- Alternativ interaktive *shell* („Kommandozeile“): JShell
- Ähnlich wie für viele andere dynamische Sprachen wie Python
- *Eignet sich gut zum schnell mal Dinge Ausprobieren!*
- Meist Ausführen von kleinen „Code-Schnipseln“
- Muss Compiler und Interpreter nicht explizit ausführen
- Benötige kein „Gerüst“ wie die Datei `Hello.java`

- *Ich verwende die Java Shell gerne für kurze interaktive Demos.*

Kurze Zusammenfassung

- Compiler `javac` übersetzt Java-Quelltext in Bytecode
- Interpreter `java` führt Bytecode aus
- Grundstruktur eines einfachen Java-Programms (`Hello.java`)

- Wir müssen hier etwas vorgreifen
  - sowohl für die Struktur eines einfachen Java-Programms
  - als auch für Ein-/Ausgabe
- *Learning by doing!* – Übungen und Tutorien nutzen!
  - *Nachfragen*, wenn etwas unklar ist!
  - Sofort nachfragen!

- Als nächstes *algorithmische Grundkonzepte* in Java
  - Aufbau der Sprache (Syntax, Semantik)
  - Algorithmische Bausteine
  - *Wir beschränken uns jeweils vorerst auf das nötigste!*

Übersicht

Einführung

- Auch Java-Programme gehorchen einer bestimmten *Syntax*.
  - Was sind „Wörter“? Wie werden „Sätze“ gebildet?
  - Wie wird ein Algorithmus formal spezifiziert?
- Dazu kommt *Semantik*
  - Was *bedeutet* ein Satz?
  - Was soll der Prozessor tun?
- Wir sehen uns das für die Programmiersprache Java an.
  - Wir vereinfachen die Dinge am Anfang! (Grober Überblick!)
  - *Alle Details* stehen z.B. in der Java Language Specification

„Wörter“ in Java: Beispiel

```java
/** A simple Java Program,
    which doesn't make much sense.
 */
public class HelloWorld {
  public static void main(String[] args) {
    int i=1+2;
    if (i==3 || false)
      System.out.println("Hello, world");
  }
}
```

„Wörter“ in Java

- Reservierte Schlüsselwörter
  - z.B. `int boolean if while ...`
- Literale repräsentieren konstante *Werte*
  - z.B. 0 42 true false null 'x' "Hello world"
- Bezeichner (*identifiers*) sind eindeutige *Namen*
  - z.B. für Variablen oder Funktionen
- Begrenzer (*separators*) „strukturieren“ Ausdrücke
  - z.B. `( ) { } ; ,`
- Operatoren definieren z.B. arithmetische Ausdrücken
  - z.B. `+ - * / < <= ...`

„Wörter“ in Java: Beispiel

```
public class HelloWorld {
  public static void main(String[] args) {
    int i=1+2;
    if (i==3 || false)
      System.out.println("Hello, world");
  }
}
```

- Schlüsselwörter: `public`, `class`, `static`, `void`, `int`, `if`
- Literale: `1`, `2`, `3`, `false`, `"Hello, world"`
- Bezeichner: `String`, `args`, `System`, `out`, `println`, `i`
- Begrenzer: `{}` `()` `[]` `.` `;`
- Operatoren: `=` `+` `==` `||`

Übersicht

Typen

- Ein Typ bezeichnet eine Sorte von Werten, z.B.
  - ganze Zahlen $n \in \mathbb{Z}$
  - rationale Zahlen $q \in \mathbb{Q}$
  - Wahrheitswerte $b \in \{\text{wahr}, \text{falsch}\}$
  - Zeichenketten $s \in \Sigma^*$ (aus Zeichen im Alphabet Σ)
- Wir beschränken uns vorerst auf zwei Typen in Java:
  - `int` repräsentiert eine *ganze Zahl* $-2^{31} \leqslant n < 2^{31}$
  - `boolean` repräsentiert einen *Wahrheitswert* $b \in \{\text{true}, \text{false}\}$
- Typen stehen i.d.R. nicht für sich alleine; zusätzlich definiert man *Operationen* auf Typen z.B. OOP (*später*).

Variablen

- Variablen modellieren Daten
  - *Bezeichner* („Namen“) für Werte (wie z.B. in $f(x) = x^2$)
  - Werte sind *veränderlich* („variabel“, im Gegensatz zu $f(x) = x^2$) und werden durch *Anweisungen* verändert
- In Java müssen Variablen vor Verwendung definiert werden:
  - Name *und Typ* der Variablen
  - Dabei wird Speicherplatz für den Wert „reserviert“.
  - Die Variable ~~kann~~ sollte *initialisiert* werden.
- Java Syntax

```
Type identifier;          // undefined value!
Type identifier = value;
Type id1,id2 [,...];      // same as list
```

- Listenform besser meiden, weil unübersichtlich!
- Später: weitere Eigenschaften, z.B. Sichtbarkeit, Lebensdauer

Beispiele für Definition von Variablen in Java

```
int      x;                 // Value is undefined!
int      y=1;
int      a,b=2,c=x,d;       // list of variables

boolean  p=true, q;         // Value of q is undefined!

int      x=2;               //  ERROR! Already defined!
```

- Tatsächlich initialisiert Java jede Variable automatisch.
- Das kann – muss aber nicht – mit einem sinvollen Wert sein.
- C/C++ tut das grundsätzlich nicht – bei ähnlicher Syntax.
- *Besser initialisieren!* Wir schreiben vorsichtshalber *undefiniert*!

Ausdrücke

- Operatoren verknüpfen Werte (Variablen und Konstanten) zu Ausdrücken, z.B. ganze Zahlen nach „üblichen“ Rechenregeln.
- Ausdrücke werden i.d.R. gebildet aus
  - Bezeichnern (*Variablen*), Literalen (*Konstanten*)
  - Operatoren, z.B. `+ - * /`
  - Begrenzern, z.B. `( )`
  - und aus (Teil-)*Ausdrücken*
- Ein Ausdruck liefert einen Wert von einem bestimmten Typ.
- Wir unterscheiden
  - Arithmetische Ausdrücke
  - Logische Ausdrücke
  - Zuweisungen

Arithmetische Ausdrücke

- z.B. „Grundrechenarten“ für ganzen Zahlen (`int`)
  - `+` : `int` $\times$ `int` $\rightarrow$ `int` (Summe)
  - `-` : `int` $\times$ `int` $\rightarrow$ `int` (Differenz)
  - `*` : `int` $\times$ `int` $\rightarrow$ `int` (Multiplikation)
  - `/` : `int` $\times$ `int` $\rightarrow$ `int` (*ganzzahlige* Division ohne Rest)
  - `%` : `int` $\times$ `int` $\rightarrow$ `int` (ganzzahliger *Rest* aus Division)
- Vorrangregeln *(precedence)* beachten
  - „Punkt vor Strich“
  - Auswertung *von links nach rechts*
  - Klammern
- Beispiele

```
x+y     x+2*y      (x+2)*y     -x     -(x+y)
x/y     (x/y)*y    x-x%y
```

Logische Ausdrücke

- Logische Ausdrücke liefern Wahrheitswerte (`boolean`), z.B.
  - `==` : `int` $\times$ `int` $\to$ `boolean` (Vergleich $x = y$)
  - `!=` : `int` $\times$ `int` $\to$ `boolean` (Vergleich $x \neq y$)
  - `>` : `int` $\times$ `int` $\to$ `boolean` (Vergleich $x > y$)
  - `>=` : `int` $\times$ `int` $\to$ `boolean` (Vergleich $x \geqslant y$)
  - `&&` : `boolean` $\times$ `boolean` $\to$ `boolean` („p *und* q“)
  - `||` : `boolean` $\times$ `boolean` $\to$ `boolean` („p *oder* q“)
  - `!` : `boolean` $\to$ `boolean` („nicht p“)
- Operatoren haben *niedrige Priorität* (in Tabelle nachschlagen!)
- *Im Zweifelsfall helfen Klammern!*
- Beispiele

```
x!=y          x+3>=y                  y<=x+3
!(x==y)       (x<=y  &&  y<=z)        (x==y  ||  x<z)
```

Zuweisungen

- Zuweisungsoperatoren ändern den *Wert* einer Variablen
- `x = y;` (Zuweisung $x \leftarrow y$)
  - `x` ist eine *Variable*
  - `y` kann ein arithmetischer *Ausdruck* vom Typ `int` sein
  - Nicht verwechseln mit Vergleichsoperator ==
- Beispiele

```
x=y+1;      x=x*2;      x=x+x;      x=(y+1)*(z-1);
```

- Beachte
  - Zuweisungen sind *keine Gleichungen*!
    Eine Variable kann links *und* im Ausdruck rechts stehen!
  - *Zuerst* wird der Ausdruck auf der rechten Seite ausgewertet.
    Dabei werden für Variablen im Ausdruck deren aktuelle Werte „eingesetzt“.
  - *Danach* wird dieser Wert der Variablen auf der linken Seite zugewiesen, d.h. deren Wert ändert sich i.d.R.

Zuweisungen: Varianten

- Abkürzende Varianten
  - `x += y;` entspricht `x = x+y;` $\quad (x \leftarrow x+y)$
  - analog `-= *= /= %=`
  - *Werden in der Praxis viel verwendet.*
- Beispiele

```
x+=1;      x*=2;      x+=x;      x=(y+1)*(z-1);
```

- Zuweisungsausdruck selbst hat einen *Wert*
  - ermöglicht z.B. `(x=y)<z`
  - Oft mißverständlich – Fehlerquelle! Deshalb besser meiden!

Beispiel: `int` und ganzzahlige Division

```
int seconds = 123456; // print as d, h, m, s

int minutes = seconds / 60;
int hours = minutes / 60;
int days = hours / 24;

seconds %= 60;
minutes %= 60;
hours %= 24;

System.out.println(days + "d " +
                   hours + "h " +
                   minutes + "m " +
                   seconds + "s");
```

Beispiel: `int`

```
int d = 1;   // days
int h = 10; // hours
int m = 17; // minutes
int s = 36; // seconds

// compute number of seconds
// (example for using Horner's scheme)

int seconds =
    ((d * 24 + h) * 60 + m) * 60 + s;
```

Beispiele: `double`

```java
double x = 10.0;  // distance [m]
double t =  3.0;  // time [s]

double v = x / t; // speed [m/s]
```

```java
double price = 499.0;           // w/ 19% MwST
double net = price / 1.19;      // netto

double price_21 = net * 1.16; // WiSe 20/21
```

```java
double r = 1.0;             // radius
double x = 0.4, y = 0.5; // point

boolean in_circle = (x*x + y*y <= r*r);
```

Kurze Zusammenfassung

- Typen
  - `int` (ganze Zahlen)
  - `boolean` (Wahrheitswerte)
- Variablen und konstante Werte (Literale)
- Ausdrücke (mit Hilfe von Operatoren)
  - Arithmetische Ausdrücke („Berechnungen“)
  - Logische Ausdrücke („Bedingungen“)
- Spezialfall: Zuweisungen
  - Ändern den Wert von Variablen
  - Zentrale Operation: (Zwischen-)Ergebnisse speichern
- Als nächstes: *Bausteine für Algorithmen*
  - Wir können bereits einfache Berechnungen durchführen, ähnlich einem Taschenrechner.
  - Was fehlt, um *beliebige* Algorithmen zu beschreiben? „Bedienung des Taschenrechners“?

Übersicht

Bausteine für Algorithmen

- Sequenz: Hintereinanderausführung von
  - anderen Bausteinen
  - Anweisungen (Ausdrücke, i.d.R. Zuweisungen)
  - Spezialfall: *einzelner* Baustein/Anweisung
- Fallunterscheidung: *bedingte* Anweisung
  - „Wenn Bedingung P erfüllt ist, dann führe Sequenz S aus.“
  - Logischer Ausdruck (*wahr* oder *falsch*) als Bedingung P
- Schleife: bedingte *Wiederholung*
  - „Wiederhole Sequenz S solange *Bedingung* P erfüllt ist.“

Sequenzen in Java

- Eine Sequenz ist eine Hintereinanderausführung von Anweisungen (*statements*).
- Anweisungen können beliebige Ausdrücke sein; sinnvoll sind hier Zuweisungen, denn sie *ändern* einen Zustand.
- In Java wird *jede* Anweisung mit Semikolon ; abgeschlossen.
- Anweisungen können so – durch Semikolon getrennt – zu einer *Sequenz* aneinandergereiht werden.
- Beispiel

```
int x=10, y; // definition of variables

y=x+1;
x*=y;          // sequence of 2 statements
```

- Der Lesbarkeit halber *eine Anweisung pro Zeile*!

Blöcke

- Sequenzen werden in Blöcken organisiert.
  - Blöcke werden durch geschweifte Klammern { } begrenzt.
  - Ein Block wird grob wie *eine* einzelne Anweisung behandelt.
  - Blöcke werden *nicht* durch Semikolon abgeschlossen!
  - Blöcke können beliebig ineinander *verschachtelt* werden.
- Beispiel

```
{  x=y+1;  z=2*y;
   {  x=z;  x*=3;
   }
}
```

- Das Beispiel ist nicht besonders sinnvoll. – Wir benötigen Blöcke zur *Strukturierung*, z.B. in Fallunterscheidungen. Gleich...

Blöcke und Variablen

- Blöcke definieren den Geltungsbereich (*scope*) von Variablen.
- Eine Variable ist nur sichtbar
  - in dem Block, in dem sie definiert wurde
  - in allen dort – direkt oder indirekt – verschachtelten Blöcken
- d.h. mit Beenden dieses Blocks wird die Variable *ungültig*.
- Beispiel

```
{ int x;     // visible: x
  { int y;  /* visible: x,y */ }
  { int z;  /* visible: x,z */ }
            // visible: x
}           // visible: -
```

- Im folgenden steht eine Sequenz S für
  - eine einzelne Anweisung, z.B. `x=y+3;` *oder*
  - einen Block, z.B. `{ x=y; x+=3; }`

Fallunterscheidungen in Java

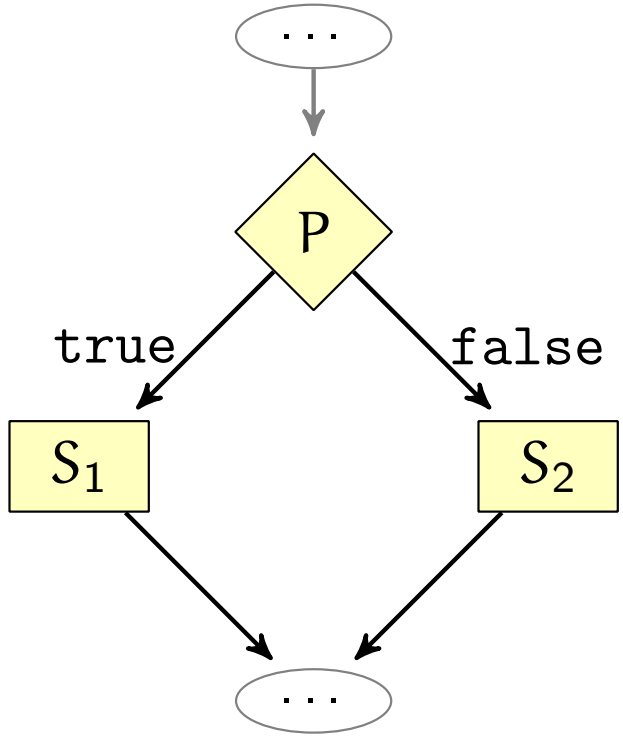
- Schlüsselwörter `if` (*wenn*), `else` (*sonst*)

Syntax

```
if ( P ) { // condition
  S
}
```

oder

```
if ( P ) {
  S1
} else {
  S2
}
```



- Alternativen
  - Operator `? :` (Fallunterscheidung als arithmetischer Ausdruck)
  - `switch`-Ausdruck *(betrachten wir nicht weiter)*

Beispiel: Berechne Absolutbetrag $y \leftarrow |x|$

$$|x| = \begin{cases} x & \text{falls } x \geqslant 0 \\ -x & \text{sonst} \end{cases}$$

```
if (x>=0)
  y=x;
else
  y=-x;
```

Äquivalent statt `else` explizit zwei Bedingungen: $\neg(x \geqslant 0) \Leftrightarrow x < 0$

```
if (x>=0)
  y=x;
if (x<0)
  y=-x;
```

Beispiel: Signum-Funktion (Vorzeichen) $z \leftarrow \text{sgn}(x)$

Wir betrachten hier *drei* Fälle:

$$\text{sgn}(x) = \begin{cases} -1 & \text{falls } x < 0 \\ 0 & \text{falls } x = 0 \\ +1 & \text{falls } x > 0 \end{cases}$$

```
if (x==0)
  z=0;                  // special case
else if (x<0)
  z=-1;
else                    // x>0
  z=+1;
```

- {...} Blöcke nur für „echte“ Sequenzen (>1 Anweisung) nötig
- Lesbarkeit: else if bei mehreren *Alternativen* nicht „schachteln“

Beispiel: Vertausche Werte

- Vertausche Werte von x und y falls $x < y$,so dass so dass $(x, y) \leftarrow (\max\{x, y\}, \min\{x, y\})$.

```
if (x<y) {
   int z=x;    // must store value temporarily
   x=y;
   y=z;
}
```

- *Wir werden das Vertauschen oft benötigen!*
- z.B. beim Sortieren

Schleifen in Java: `while`

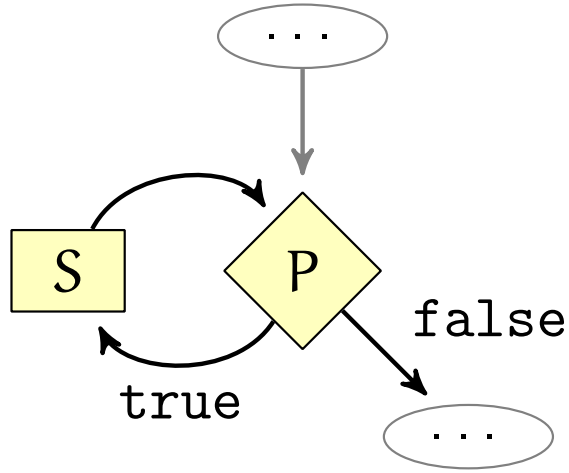
- Schlüsselwort `while` (*solange*)

Syntax

```
while ( P ) {

   S

}
```



- Mit `while` lassen sich *alle* Schleifen ausdrücken.
- Varianten
  - `do ... while`
  - `for (...;...;...)`
- Zusätzliche Möglichkeit zum ...
  - *Abbruch* der Schleife mit `break`
  - *Fortsetzen* der nächsten Iteration mit `continue`

Beispiel: Fakultät mit `while` Schleife

- Berechne

$$x = n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

- In Java:

```
int n=...;  // input n >= 0
int i=1;    // counter
int x=1;    // Mind initialisation!

while (i<=n) {
   x=x*i;
   i=i+1;
}
            // result  x = n!
```

- Später: *Funktionen* `x=factorial(n)`

Beispiel: Quersumme mit `while` Schleife

- Berechne Quersumme x einer Dezimalzahl $n \geqslant 0$
- In Java:

```
int n=...; // input n (will be overwritten)
int x=0;

while (n!=0) {
  x=x+n%10; // add last digit
  n=n/10;   // "shift" right by 1 digit
}
          // result: digit sum in x
          //   and n=0
```

Felder (*arrays*)

- Felder (*arrays*) halten eine Anzahl von gleichförmigen Daten (d.h. gleicher Typ).
- Die einzelnen Elemente werden mit Hilfe eines *Index* bezeichnet.
- Wir können uns das vorstellen wie eine Tabelle oder einen Vektor z.B. $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}^n$
  - Hier besteht $\mathbf{a}$ aus n ganzen Zahlen.
  - Wir sprechen hier von n als *Länge* und
  - Elementen a_i mit *Index* $i \in \{0, \ldots, n-1\}$.
  - Die Elemente sind selbst *Variablen*.
- *Indices in der Informatik beginnen grundsätzlich bei* 0*!*
  - Im Gegensatz zur Mathematik (Vektoren, Matrizen)
  - Abhängig von Programmiersprache!
    Ausnahmen z.B. Fortran, Julia oder Matlab / GNU Octave.

Felder (*arrays*) in Java: Erzeugung

- Feld**typ**: aus `int` wird `int[]`
  Die Länge ist *nicht* Teil des Typs!
- Definiere Feld**variable** mit

```
int[] a;   // define an array of int
```

  - *Länge und Werte* sind noch undefiniert!
  - Zugriff führt zu Fehler: `java.lang.NullPointerException`
- Reserviere **Speicher** mit `new` Neu!

```
int n=10;
a = new int[n];   // reserve storage
```

  - z.B. Erzeuge Feld von `int` der Länge $n = 10$
  - Erst jetzt können Werte im Feld gespeichert werden.
  - *Wir gehen wieder von undefinierten Werten aus!*
- Dasselbe kürzer

```
int[] a = new int[10];
```

Felder (*arrays*) in Java: Initialisierung & Zugriff

- Wenn Länge und Werte bekannt sind, kann eine **Liste** in `{}` angegeben werden (statt `new`)

```
int[] a = {1,2,3};
```

ist äquivalent zu

```
int[] a = new int[3];
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

- **Zugriff** auf Werte mit Index

```
int i = 0;          // index (an integer!)
int value = a[i]; // read
a[i] = value + 1; // write
```

Felder (*arrays*) in Java: Zugriff & Länge

- **Länge** des Felds mit `.length`

```
int[] a = {2,3,5,7};
int n = a.length;    // query length: 4
```

  Undefiniert für nicht initialisiertes Feld (kein `new`, keine Liste): Fehler `java.lang.NullPointerException`

- Zulässige Indices für `a[i]` sind `i = 0,...,a.length-1`
- Java überprüft Zugriff, unzulässiger Index führt zu Fehler: `java.lang.ArrayIndexOutOfBoundsException`
- Auf Feldern sind sonst *keine* Operatoren definiert außer
  - Gleichheit == bzw. != und Zuweisung =
  - *Vorsicht!* – Deren Verhalten ist anders als für z.B. `int`.
  - Insbesondere sind == und != i.d.R. nicht sinnvoll.

Beispiel

```
int[] a=new int[3];
int   i=0;

while (i<a.length) { // 0..2
  a[i]=i+1;
  i=i+1;
}
                     // now: a = {1,2,3}

a[3]=4;              // ERROR!
```

- Die Schleife *initialisiert* Werte der Elemente.
- Schleifen und Felder werden oft zusammen verwendet.
- „Schleifenende“ (Abbruchbedingung) beachten!
  „Meist < statt <= . . . “

Feldvariablen werden referenziert

- Eine Referenz ist ein *Verweis* auf ein Datum.
- Folge: Zuweisung ändert Verweis (*nicht* Datum)
- Beispiel:

```
int    x=1, y;       // integers
int[] a={1,2}, b;    // arrays

y=x;         // Mind type int!
y=y+1;       // ⇒ x = 1 ∧ y = 2

b=a;         // Array variables are references!
b[0]=99;     // ⇒ a = (99,2) ∧ b = a
```

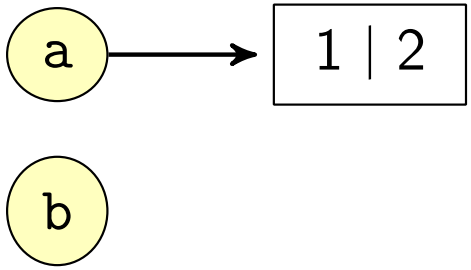
- Variablen vom Typ `int` speichern *Werte*!
- Feldvariablen `int[]` speichern Referenzen auf Daten!

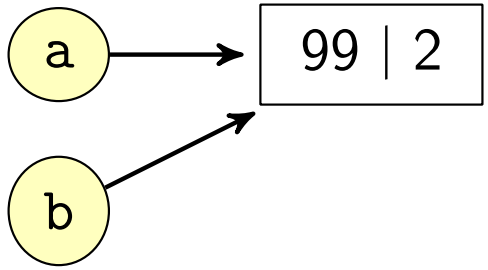
Feldvariablen werden referenziert

- Eine Referenz ist ein *Verweis* auf ein Datum.
- Folge: Zuweisung ändert Verweis (*nicht* Datum)
- Beispiel

```
int[] a={1,2}, b;
```

```
b=a;
b[0]=99;
```





- Jetzt: Scheinbar *Besonderheit* im Unterschied z.B. zu `int`
- Tatsächlich sind Referenzen die **Regel** in Java!

Beispiel: Summe

- Berechne Summe der Werte im Feld `a`
- Die Länge von `a` darf beliebig sein.
- Wenn kein Speicher reserviert wurde, dann Summe 0.

```
int[] a=...;          // input
int sum=0;            // output
int i=0;

if (a!=null) {        // got storage?
  while (i<a.length) {
    sum+=a[i];
    i+=1;
  }
} // else: have already sum==0
```

- Symbol `null` für *nicht-initialisierte Referenz*

Beispiel: berechne die ersten n Primzahlen

Definition (Primzahl)

Eine *Primzahl* ist eine natürliche Zahl, die größer als eins und nur durch sich selbst und durch eins teilbar ist.

- *Konsequenz:* Wenn $x \in \mathbb{N}$ teilbar durch Primzahl $p \neq x$, dann kann x *keine* Primzahl sein.
- *Wir wollen die ersten n Primzahlen aufzählen.*
- Ein möglicher Algorithmus
  - Zähle ganze Zahlen $x = 2, 3, 4, \ldots$ auf *und dabei*
  - Teste, ob x durch eine *bereits gefundene* Primzahl teilbar ist
  - Wenn nein, dann ist x Primzahl und wird gespeichert
- Verwende *Feld* der Länge n, um Primzahlen zu speichern

Beispiel: berechne die ersten n Primzahlen (2)

```
int    n=...;                  // input
int[]  p=new int[n];           // output: primes
int    m=0;                    // number of primes in p
int    x=2, i;

while (m<n) {                  // find first n primes
  i=0;
  while (i<m && (x%p[i]!=0)) { i=i+1; }

  if (i==m) {                  // Was NOT divisible?
    p[m]=x;                    // store new prime
    m=m+1;
  }

  x=x+1;                       // test next number...
}
```

Beispiel: berechne die ersten n Primzahlen (3)

- `m` zählt PZ, die schon gefunden und in `p` gespeichert wurden.
- Äußere Schleife: *Solange noch nicht n PZ gefunden wurden...*

```
while (m<n) {                  // find first n primes

  ...

  if (/* x is prime */) {
     p[m]=x;                   // store new prime
     m=m+1;                    // got one more prime
  }

  x=x+1;                       // test next number
}
```

- Jede neue PZ wird in `p` an Stelle `m` gespeichert.
- Den Test, `x` prim ist, übernimmt die innere Schleife!

Beispiel: berechne die ersten n Primzahlen (4)

- Innere Schleife: *Ist x prim, d.h. durch keine PZ aus p teilbar?*

```
i=0;                              // enumerate p[i]
while (i<m &&                     // end of list? (i=m)
       (x%p[i]!=0)) {             // p[i] divides x?
         i=i+1;                   // test next prime
}
```

- Zählt alle gefundenen `p[i]` beginnend von `i=0` auf.
- Bricht ab, wenn entweder
  1. das Ende der PZ-Liste erreicht ist, d.h. `i==m`, *oder*
  2. `p[i]` die Zahl `x` ohne Rest teilt, d.h. `x` ist keine PZ und `i<m`.

  Wenn `i==m`, dann wird *sofort* abgebrochen, und Teilbarkeit wird nicht überprüft. – *Was, wenn das nicht so wäre?*
- Damit gilt: `x` ist prim $\Leftrightarrow$ `i==m`

Beispiel: Nochmal Auswertung von Konjuktion &&

- Im Beispiel die Schleifenbedingung

```
i<m  &&  (x%p[i]!=0)
```

- Semantischer Unterschied zwischen Operatoren
  && (Java) und $\wedge$ in (Logik, Mathematik)
  - Kommutativ: $P \wedge Q = Q \wedge P$
  - Nicht kommutativ: `(p && q)` $\not\equiv$ `(q && p)` !
- Beachte insbesondere *Effekte* durch Zuweisungen bei Auswertung einer Bedingung.
  - durch Funktionen/Methoden — später!
  - Fehlerquelle – *Nach Möglichkeit meiden!*

Kurze Zusammenfassung

- **Sequenz**: Hintereinanderausführung
  - Anweisungen getrennt durch Semikolon A_1`;` A_2`;`
  - Gliederung in *Blöcke* `{` S `}` (*Sichtbarkeit* von Variablen)
- **Fallunterscheidung**: *bedingte* Anweisung
  - `if (P) {` S_1 `} else {` S_2 `}`
- **Schleife**: *bedingte* Wiederholung
  - `while (P) {` S `}`
- *Weitere Varianten von Fallunterscheidung und Schleifen folgen*
- **Feld**: „vektorwertige“ Variable bestimmter *Länge*
  - Zugriff auf Komponenten über Index (oft in Schleifen)
  - Felder werden referenziert (`int[]` im Gegensatz zu z.B. `int`)!
- **Referenz**: Variable bezeichnet *Verweis* auf ein Datum
  - Zuweisung ändert Verweis, *nicht* Datum
  - Mehrere Verweise auf ein Datum möglich (siehe Beispiel)

Mehr Java ...

- *Wir kennen bereits die wesentlichen Bausteine, um beliebige Algorithmen zu implementieren.*
- Datentypen in Java
  - Primitive Datentypen
  - Zeichenketten (*strings*)
  - Typumwandlung
  - Einfaches Lesen von Eingabewerten
- Operatoren
  - Übersicht
  - Inkrement und Dekrement
  - `?:` Operator
- Schleifen
  - `break` und `continue`
  - `do ... while` und `for` Schleifen

Primitive Datentypen

- „Fest eingebaute Datentypen, mit denen Maschinenbefehle (der Java VM oder der CPU) arbeiten“

Arten:

- Wahrheitswerte: `boolean`
- Ganzzahlige (*integer*) Werte: `byte`, `int`, `short`, `long`
- Zeichen im Unicode-Zeichensatz: `char`
- Gleitkommazahlen (*floating point numbers*): `double`, `float`

Variablen und Primitive Datentypen

In Java speichern Variablen von **primitiven Datentypen** Werte.
Für **alle anderen** Datentypen werden Referenzen gespeichert.

Übersicht über primitive Datentypen

Datentyp	Größe	Wertebereich
`byte`	8 Bit	$-128 \ldots 127$
`short`	16 Bit	$-32.768 \ldots 32.767$
`int`	32 Bit	$-2^{31} \ldots 2^{31}-1$
`long`	64 Bit	$-2^{63} \ldots 2^{63}-1$
`char`	16 Bit	`'\u0000'` ... `'\uffff'`
`float`	32 Bit	$\pm 3.4 \times 10^{\pm 38}$
`double`	64 Bit	$\pm 1,7 \times 10^{\pm 308}$

- Alle ganzzahligen Datentypen in Java haben ein Vorzeichen.
- `char` hat kein Vorzeichen und wird als Teil von `Zeichenketten` (*strings*) verwendet.
- Zu `float` und `double` ist jeweils (ungefähr) die kleinste ($|x| > 0$) und größte ($|x| < \infty$) darstellbare Zahl x angegeben.

Reelle Zahlen: `double` und `float`

- *Approximative* Darstellung *reeller* Zahlen
- Literale `0.012345` bzw. `0.012345f`
- Oft Exponentialschreibweise $1,2345 \times 10^{-2} \equiv$ `1.2345e-2`
- Ganz grob: `double` und `float` stellen *maximal etwa* 15 bzw. 6 signifikante Dezimalstellen dar („Genauigkeit“).
- *Verwende im Zweifel `double`!*
- Rechnen mit Gleitkommazahlen kann tückisch sein!
  z.B. nicht jede Dezimalzahl kann exakt dargestellt werden
- Tatsächlich gelten nicht mal alle der üblichen Rechenregeln!
- Spezielle Symbole für ∞ und “ungültige Zahlen“

Darstellung reeller Zahlen und ihre Tücken

- Das ist ein Thema für sich!
- Hier nur Beispiele als „Warnung“

```
boolean is_equal= (1.0/3.0 == 1.0-2.0/3.0);
System.out.println(is_equal);

System.out.println(1.0/0.0);
System.out.println(0.0/0.0);

System.out.println(1/0);  // here: int division!
```

```
false
Infinity
NaN
Exception in thread "main" java.lang.ArithmeticException: /
    by zero
```

Zeichenketten (*strings*)

- Zeichenketten dienen zur Darstellung/Verarbeitung von Text.
- `char` stellt einzelnes Zeichen dar, `String` eine *Zeichenkette*.
  - *Kein* primitiver Datentyp!
  - Eigentlich *Klasse* `java.lang.String` (später!)
- `String` ist besonders, denn
  - **Literal**: `"Das ist eine Zeichenkette."`
  - + **Operator** `"Hello "+"World"` setzt zusammen
  - auch `"Wert="`+42 – **Umwandlung** in *string*
  - Unveränderbar! (*immutable*)
- Anwendungen
  - zur Ausgabe z.B. von Ergebnissen (`System.out.println`)
  - zum Einlesen von Werten
  - als Argumente (Eingabewerte) in `main(String[] args)`

Typumwandlung

- Manchmal ist es nötig, einen Typen in einen andern *umzuwandeln*.
- Solche Typumwandlungen können *implizit*, d.h. ohne eigenes Zutun, geschehen, z.B. um in einen „größeren“ Wertebereich zu wechseln wie etwa für `1+2.0`.
- Typumwandlungen können auch *explizit* erzwungen werden, das heißt auch *type cast* oder nur *cast*.
- Das ist sinnvoll und nötig, wenn
  - ein kleinerer Wertebereich verwendet werden soll,
  - Objekte und Klassenhierarchien im Spiel sind. (später)
- Nicht jede Umwandlung ist erlaubt bzw. sinnvoll!
- *Vermeide Typumwandlungen, wenn möglich!*

Typumwandlung (2)

- Implizite Umwandlung nur unter bestimmten Bedingungen
- Für primitive Datentypen gilt
  - Wertebereich des neuen Typs muss größer sein.
  - Keine Umwandlung von Gleitkomma- zu ganzen Zahlen
  - (*Niemals* Umwandlung „Zeichenkette in Zahl“!)
- *Explizite* Typumwandlung lockert die Regeln

```
x=(T) y; // new type T
```

  - Vorsicht: Wert kann außerhalb des neuen Wertebereichs liegen!
  - Java überprüft den Wertebereich *nicht*!
- Beispiel

```
short s=1;
int   i=s;    // Ok! -- implicit type cast
s=(short) i; // explicit type cast
```

Typumwandlung (3)

- Wir wollen Typumwandlung möglichst vermeiden – sowohl *implizit* als auch *explizit* (mit *type cast*) !
  - Manchmal nötig – aber häufiger Fehlerquelle ...
  - ... ggf. Algorithmus nicht durchdacht!
- Insbesondere rechnen wir in den *Vorlesungsbeispielen* meist mit *ganzen Zahlen* (`int`)
  - Wir bleiben dann beim Typ `int`!
  - Wir wechseln nicht unnötig zu `double`!
- Wir werden auch Algorithmen konstruieren, die Java schon mitbringt (z.B. Potenzieren x^n) – aber ...
  - ... auf anderen Typen (z.B. `double`)
  - ... uns geht es eben genau um den Algorithmus

Werte von Kommandozeile

- Programmaufruf

```
bash> java Greeter Habe a nice day
```

- `String[] args` $\equiv$ `{"Have", "a", "nice", "day"}` in `main()`
- Umwandeln von `String` in `int` , `double`, ...

```
public static void main(String[] args) {
  int n=args.length();  // # of arguments
  int i=Integer.parseInt(args[0]);
  double d=Double.parseDouble(args[1]);
  ...
```

Das ist *keine Typumwandlung*!

Werte einlesen von System.in

- Programm liest vom Standardeingabekanal

```
bash> echo "Give me input" | java Processor
```

oder auch „Tastatureingabe“

- Lesen von System.in mit util.Scanner

```
import java.util.Scanner;

public class Processor {
  public static void main(String[] args) {
    Scanner scanner=new Scanner(System.in);
    int i=scanner.nextInt();
    double d=scanner.nextDouble();
    String s=scanner.next();
    ...
```

Übersicht über Operatoren

- Klassifizierung nach Funktion
  - Vergleichs- oder relationale Operatoren (z.B. `!=` oder `<`)
  - Logische Operatoren (z.B. `&&`)
  - Arithmetische Operatoren (z.B. `+`)
  - *Bit-Operatoren* (z.B. `&`) (Betrachten wir nicht weiter!)
  - Zuweisungsoperatoren (z.B. `+=`)
  - *Inkrement* und *Dekrement*
  - *Fallunterscheidung* `?:`
- Klassifizierung nach Stelligkeit
  - Unäre Operatoren (z.B. `-x`)
  - Binäre Operatoren (z.B. `x+y`)
  - Ternärer Operator (z.B. `x<=y ? x : y`)
- Nach Vorrang
  - Etwas komplizierter als „Punkt vor Strich“
  - *Im Zweifel Klammern setzen!*

Operatoren nach Vorrang

Funktion	Operatoren
Postfix Inkrement/Dekrement	`x++ x--`
Vorzeichen, Negation, Bit-Komplement, Präfix Inkrement/Dekrement	`-x +x !p ~b ++x --x`
Multiplikation, Division, Rest	`* / %`
Addition, Subtraktion	`+ -`
Bit-Shift	`<< >> >>>`
Relational, Typ-Vergleich	`< <= > >= instanceof`
Gleich/Ungleich	`== !=`
bit-weise Und-Verknüpfung	`&`
bit-weise exklusive Oder-Verknüpfung	`^`
bit-weise (inklusive) Oder-Verknüpfung	`|`
Logische Und-Verknüpfung	`&&`
Logische Oder-Verknüpfung	`||`
Fallunterscheidung	`?:`
Zuweisung	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Inkrement- und Dekrementoperatoren

- Bisher ändern *nur Zuweisungen* den Wert einer Variablen.
- Dazu kommen jetzt noch
  - Inkrement – Erhöhung eines Werts um 1
  - Dekrement – Erniedrigung eines Werts um 1

```
x++;
y--;
```

äquivalent zu

```
x=x+1;
y=y-1;
```

Besser nur mit *ganzzahligen Typen* verwenden!

- Warum extra Operatoren?
  - Historisch: von `C`, Maschinenbefehle (z.B. `inc` auf 8086)
  - Oft kurz und praktisch (z.B. in Schleifen), wird gerne verwendet
  - Aber auch tückisch – potentielle Fehlerquelle!
- Zwei Formen mit *unterschiedlicher Semantik*: Präfix / Postfix

Präinkrement vs **Post**inkrement

- Gleichzeitig „Zuweisung“ *und* arithmetischer Ausdruck
  - Präinkrement `++x` erhöht *zuerst* und liefert *neuen* Wert
  - Postinkrement `x++` liefert „alten“ Wert und erhöht „*danach*“
- Beispiel

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

äquivalent zu

```
x=x+1;
y=x;
```

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

äquivalent zu

```
y=x;
x=x+1;
```

- Kompliziertere Ausdrücke sind möglich – besser meiden!
- Typische Verwendung: Index in Schleife erhöhen,
  z.B. `++i` oder `a[i++]=b[j++]`

Nochmal Fallunterscheidung: ?: Operator

- Der ternäre ?: Operator wertet eine von zwei Alternativen aus.
- Im Gegensatz zu `if-else` kann `?:` in einem Ausdruck stehen: Man kann *direkt* mit dem Ergebnis „weiterrechnen“.
- **Syntax**

$$\texttt{P ? } A_1 \texttt{ : } A_2 \texttt{ // ==} \begin{cases} A_1 & \texttt{wenn P} \\ A_2 & \texttt{sonst} \end{cases}$$

  - P ist ein logischer Ausdruck
  - A_1 und A_2 sind arithmetische Ausdrücke vom *gleichen* Typ
  - Der gesamte Ausdruck liefert einen *Wert*.
- Beispiele

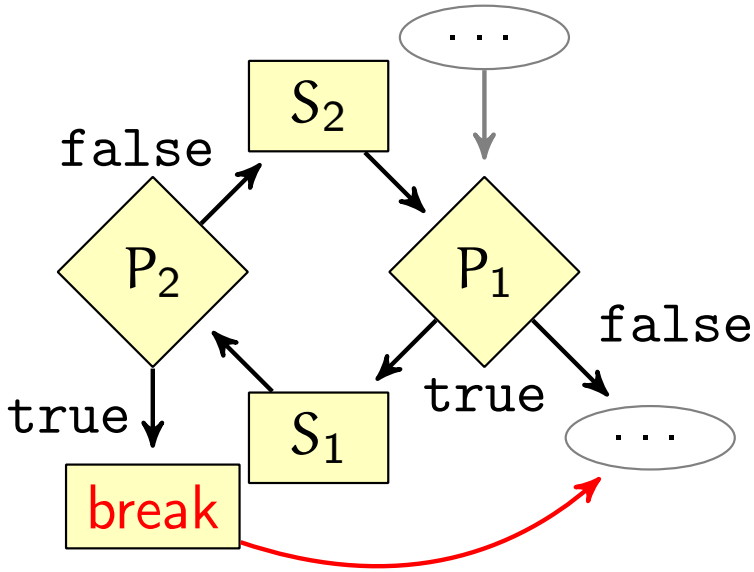
```
int diff_xy = x>y ? x-y : 0;
int abs_x   = x>=0 ? x : -x;
int max_xy  = x>=y ? x : y;
int sign_x  = x==0 ? 0 : (x>0 ? +1 : -1);
```

Schleifen in Java: `while` und `break`

- Schlüsselwort `while` (*solange*)

  Syntax

```
while ( P1 ) {
   S1
   if ( P2 ) break;
   S2
}
```



- `break` bricht die Iteration ab: „springt“ zum Ende der Schleife

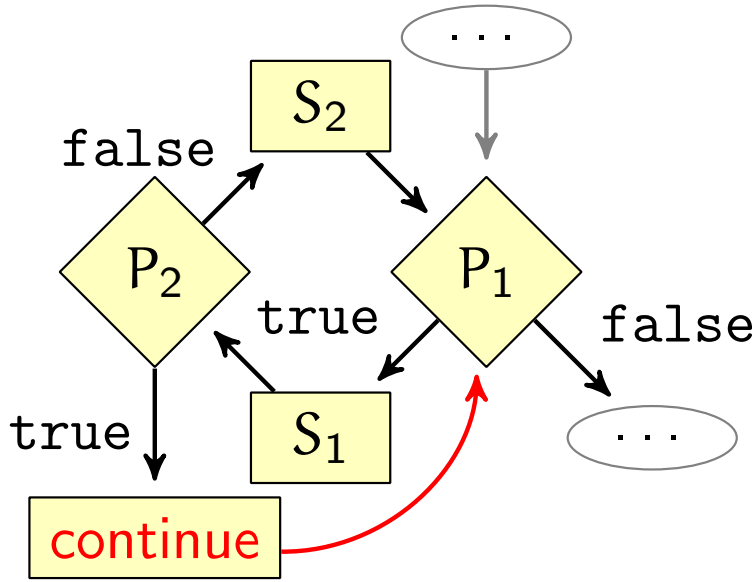
```
boolean stop=false;  // this is equivalent
while ( P1  && !stop) {
   S1
  if ( P2 ) { stop=true; } else {  S2  }
}
```

Schleifen in Java: `while` und `continue`

- Schlüsselwort `while` (*solange*)

  Syntax

```
while ( P1 ) {
   S1
   if ( P2 ) continue;
   S2
}
```



- `continue` startet die *nächste* Iteration: „Sprung“ zum Anfang

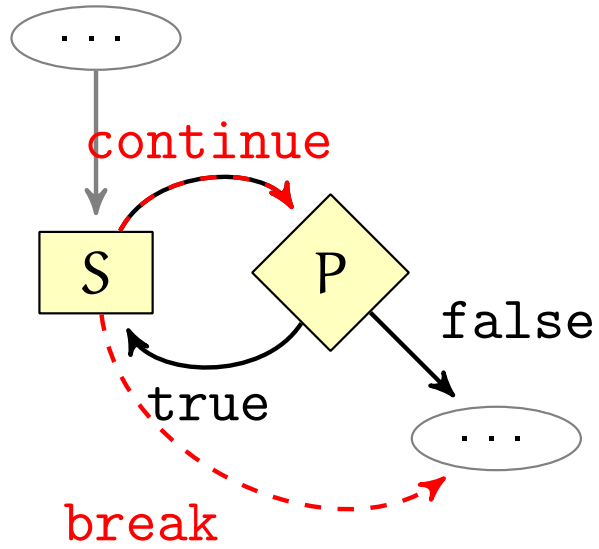
```
while ( P1 ) { // this is equivalent
   S1
   if (!( P2 )) {  S2  }
}
```

Schleifen in Java: `do ... while`

- Variante: Schleife wird *mindestens einmal* durchlaufen
- Prüfung der Bedingung *nach* jeder Iteration

Syntax

```
do {
   S
} while ( P );
```



- Äquivalent zu

```
S
while ( P ) {  S  }
```

```
while (true) {
   S
   if ( !P ) break;
}
```

Beispiel: approximiere Quadratwurzel

- Nach Heron (Anwendung des Newton-Verfahrens)
- Für $x \geqslant 0$ berechne $s \approx \sqrt{x}$, so dass der Fehler $|s^2 - x| < \epsilon$

```
double x=...;           // input (expect x>=0)
double EPS=x*1e-12;     // maximum error
double s=1.0;           // output: square root
double error;           // current error

do {
  double t=x/s;         // invariant: s*t=x
  s=(s+t)/2.0;          // we want: s≈t

  error=s*s-x;
  error=(error>=0.0 ? error : -error);

} while (error>=EPS);
```

Schleifen in Java: `for` (1)

- Schleifen sehen oft ähnlich aus z.B. für Aufzählungen
- Beispiel: berechne Fakultät von n

$$x = n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

  - Variable i wird nur innerhalb des Produkts $\prod$ verwendet.
  - i dient als Zähler und wird immer um 1 erhöht
  - Ausgehend von $i = 1$ und solange $i \leqslant n$

- Mit Hilfe einer `for`-Schleife in Java

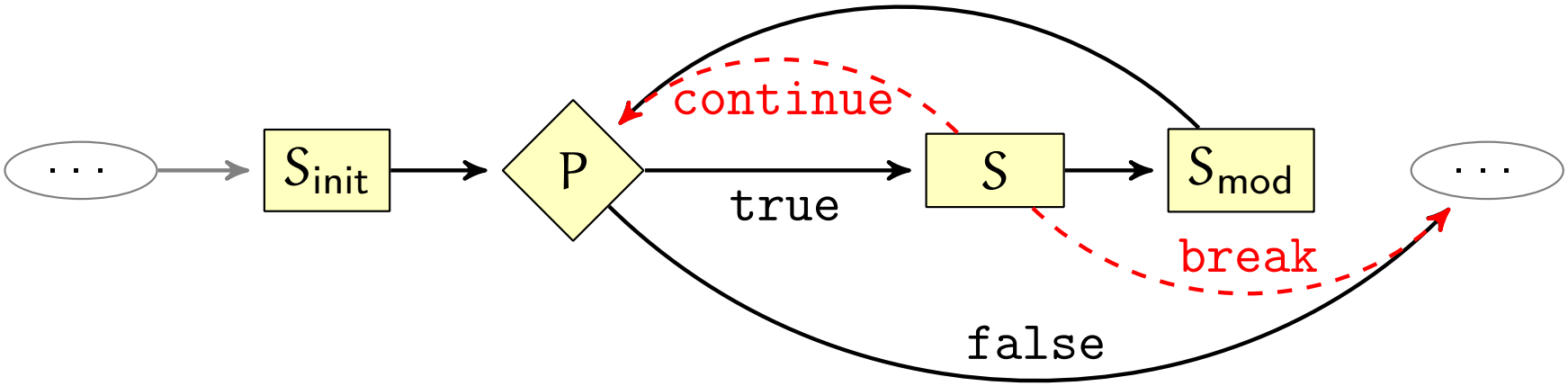
```
int n=...; // input
int x=1;   // initialization, 0! = 1

for (int i=1;i<=n;i+=1) {
  x*=i;
}
```

Schleifen in Java: `for` (2)

- Schleifen sehen oft ähnlich aus, z.B. Aufzählungen
- *Idee:* alle relevanten Informationen in einer `for` Anweisung
  - Initialisierung S_{init} $(i \leftarrow 1)$
  - Bedingung P $(i \leqslant n)$
  - Modifikationsanweisung S_{mod} („Weiterzählen“ $i \leftarrow i + 1$)
- **Syntax**

```
for ( S_init ; P ; S_mod ) {  S  }
```



Schleifen in Java: `for` (3)

- Initialisierung S_{init} *kann* eine neue, lokale Variable definieren

```
for (int i=1;i<=n;++i) { x*=i; }
```

  - Im Beispiel durch `int i=1;`
  - `i` ist *nur innerhalb* der Schleife sichtbar und gültig

  Das ist sehr praktisch und wird oft so verwendet!

- S_{init} und S_{mod} können *Sequenzen* sein
  - Folge von Anweisungen durch *Komma* getrennt
  - Beispiel ($S_{\text{mod}} = \mathtt{x} \leftarrow \mathtt{x} \cdot \mathtt{i}, \mathtt{i} \leftarrow \mathtt{i}+1$) *(Reihenfolge beachten!)*

```
for (int i=1;i<=n;x*=i,i++) {}
```

  Manchmal nützlich, aber oft unübersichtlich – wie z.B. *hier*!
  Besser meiden!

Beispiel: berechne die ersten n Primzahlen (2)

```
int    n=...;                  // input
int[]  p=new int[n];           // output: primes
int    m=0;                    // number of primes in p
int    x=2, i;

while (m<n) {                  // find first n primes
  i=0;
  while (i<m && (x%p[i]!=0)) { i=i+1; }

  if (i==m) {                  // Was NOT divisible?
    p[m]=x;                    // store new prime
    m=m+1;
  }

  x=x+1;                       // test next number...
}
```

Beispiel: berechne die ersten n Primzahlen (`for`)

```
int    n=...;                  // input
int[]  p=new int[n];           // output: primes
int    x=2, i;

for (int m=0;m<n;++x) {
  for (i=0;i<m && (x%p[i]!=0);++i) {}

  if (i==m)                    // Was x NOT divisible ?
    p[m++]=x;                  // store new prime
}
```

- Beachte: Sichtbarkeit von `i` (in `for` *und* `if`)
- Hier sinnvoll: Postfix-Inkrement `p[m++]=x;` $\equiv$ `p[m]=x; ++m;`

  Solche Kurzformen mit Inkrement oder Dekrement auf einen *Feldindex* werden in der Praxis häufig verwendet!
  Sie sind anfangs aber etwas gewöhnungsbedürftig.

Endlosschleifen

- Endlositeration: *Schleife bricht nie ab*
- Außer durch `break` Anweisung
- Umsetzung in Java, z.B.
  - als `while` Schleife

```
while (true) {  S  }
```

  - als `for` Schleife

```
for (;;) {  S  }
```

Spezialfall: *leere* Bedingung in `for` ist immer *wahr* ($\mathrm{P} \equiv$ `true`)

- Manchmal ist Endlosschleife & `break` sinnvoll.
  Wähle immer die am besten lesbare Variante!

Kurze Zusammenfassung

- Primitive Datentypen in Java
  - Keine Referenzdatentypen *(Ausnahme!)*
  - Typumwandlung
- Operatoren
  - Vorrang beachten, im Zweifel Klammern setzen
  - Inkrement und Dekrement in Präfix- und Postfix-Variante
  - Fallunterscheidung als *Ausdruck* mit `?:`
- Mehr zu Schleifen
  - Verlassen (`break`) und „Fortsetzen“ (`continue`)
  - `do-while` und `for` Schleife

- Als nächstes: *Funktionen*
  - Weitere Abstraktion des Programmablaufs
  - Danach: Objektorientierte Programmierung