2. Grundlegendes zur Programmierung mit Java

Warum eigentlich Java?

- Java ist nicht gerade als Lernsprache entwickelt worden.

- Man kann auch nicht gerade behaupten, dass Java als erste Programmiersprache für Neulinge gut geeignet ist.

Warum Java:

- Java ist heute und in weiterer Zukunft die wichtigste Programmier-sprache wegen ihrer Möglichkeiten in Internet–Programmierung (Applets), Datenbankanbindung usw.

- Java bietet eigentlich alles an Konzepten und Schwierigkeiten, was man auch in anderen gängigen Programmiersprachen findet.

→ Wer Java systematisch gelernt hat, kommt mit anderen Programmiersprachen besser klar.

Vorgeschichte von Java

- Java lehnt sich in den Äußerlichkeiten stark an die weit verbreiteten Programmiersprachen C und C++ an.

- *Hauptgrund*:

C/C++–Programmierern soll der Umstieg erleichtert werden.

→ Höhere Akzeptanz für Java in der Wirtschaft.

- *Unerwünschter Nebeneffekt*:

Java erbt von C/C++ viele ”Altlasten“.

Beispiel

◊ Zuweisung mit ”=“:

```
i = 1;
```

◊ Test auf Gleichheit mit ”==“:

```
if (i == 1)
```

Eigentlich intuitiver (und auch eher üblich in Programmiersprachen):

- Zuweisung mit ”:=“

```
i := 1;
```

- Test auf Gleichheit mit ”=“

```
if (i = 1)
```

Hintergrund:

- C stammt aus dem Jahr 1970.
- Die Länge von Programmen war 1970 noch ein ernsthaftes Speicherplatzproblem.
- Die Zuweisung kommt in Programmen im allgemeinen öfter vor als der Test auf Gleichheit.
- Entscheidung 1970: lieber bei der Zuweisung ein Zeichen einsparen!

  → Diese ”prähistorische“ Entscheidung hat sich über C++ (entwickelt in den 80ern) auch auf Java (entwickelt in den frühen 90ern) vererbt.

Geschichte von Java

- seit 1991: Entwicklung von der Firma SUN
  - Haupt-Designer: James Gosling

- Die ursprüngliche Vorgabe war, eine Programmiersprache zu entwickeln, die sich zur Programmierung von elektronischen Geräten der Konsumgüterindustrie eignen sollte
  - Programmierung von Toastern, Kaffeemaschinen, Videogeräten, usw. (Ubiquitous Computing)
  - Projekttitel: OAK (Object Application Kernel)

- Neue Zielsetzung:
  - Eine Programmiersprache entwickeln, die sich in besonderer Weise zur Programmierung auf verschiedensten Rechnertypen im Internet eignen sollte.

Allgemeine Java-Philosophie

- Hinter C und C++ steckt die Philosophie:

  Der Programmierer soll durch die Programmiersprache maximale Freiheit bekommen.

→ Alles, was nicht ausdrücklich verboten ist, ist erlaubt, egal, ob es Sinn macht oder nicht.

- Damit einhergehend wird dem Programmierer natürlich auch maximale Selbstverantwortung aufgebürdet.
- In der Philosophie lehnt sich Java **nicht** an C und C++ an:

  Beim Entwurf von Java ist man davon ausgegangen, dass diese Selbstverantwortung für die meisten Programmierer zu hoch ist.

→ Alles, was aller Erfahrung nach keinen Sinn macht, ist von vornherein verboten.

2.1. Compiler und Interpreter

Arbeitsgang:

- Der Programmierer erstellt ein oder mehrere *Source Files* (*Quelltext*).
- Diese Source Files werden dann
  - ◊ *kompiliert* durch einen *Compiler* und/oder
  - ◊ *interpretiert* durch einen *Interpreter*.

Prinzipieller Unterschied:

- *Kompilieren*: Das Source File wird in ein Maschinenprogramm übersetzt, das danach jederzeit als ein normaler Prozess des Betriebssystems aufgerufen werden kann.
- *Interpretieren*: Das Source File wird ohne vorherigen Arbeitsgang vom Interpreter Schritt für Schritt ausgeführt.

Diskussion

- *Kompilieren*: Das kompilierte Maschinenprogramm ist in der Regel *wesentlich* schneller in der Ausführung als die Abarbeitung des Source Files durch einen Interpreter.

  → Vorteilhaft für den Einsatz des Programms in der Praxis.

- *Interpretieren*: Die (bei großen Programmen mitunter recht hohe) Zeit fürs Kompilieren wird eingespart.

  → Vorteilhaft bei der Entwicklung des Programms (bei der ja laufend das Programm ein wenig verändert und dann neu kompiliert wird).

Zusammenhang mit Programmiersprachen

- Im Prinzip könnte jede Sprache sowohl kompiliert als auch interpretiert werden.
- Aber verschiedene Sprachen sind für das eine oder das andere besser geeignet.
- Die meisten gängigen Programmiersprachen sind auch von vornherein schon daraufhin entworfen worden, dass sie für eins von beiden besonders gut geeignet sind.

Beispiel fürs Interpretieren:

- Wenn eine HTML–Seite in einen WWW–Browser geladen wird, startet der Browser ein weiteres Programm, nämlich einen HTML–Interpreter.
- Dieser Interpreter liest den HTML–Text und stellt die WWW–Seite gemäß der darin enthaltenen Formatierungsbefehle dar.

Java-Konzept

- Für Java ist die Frage Kompilieren vs. Interpretieren strikt geregelt.
  - Und zwar salomonisch: sowohl als auch.
- *Genauer*:

  ◊ Ein Java Source File wird nicht in Maschinencode, sondern in den sogenannten *Java Byte Code* kompiliert.

  → Compiler–Programm ”`javac`“.

  ◊ Die Abarbeitung des Programms besteht dann darin, dass der Java Byte Code interpretiert wird.

  → Interpreter–Programm ”`java`“.

- Es gibt auf den gängigen Computersystemen auch Compiler zur Übersetzung direkt in Maschinencode.

  → Im allgemeinen bessere Laufzeit.

Java Byte Code

- Eine idealisierte und standardisierte Variation von Maschinencode.

- Basiert auf einem idealisierten Rechnermodell, das stark an das abstrakte Von–Neumann–Modell angelehnt ist:
  die sogenannte *Java Virtual Machine*.

- Durch diese Abstraktion von konkreten Hardwareplattformen und Betriebssystemen kann im Prinzip jedes kompilierte Java–Byte–Code–Programm auf jedem System laufen.

- Neben den typischen Elementen von Maschineninstruktionen enthält Java Byte Code noch weitere Informationen, die es z.B. erlauben zu prüfen, ob ein Java–Programm nur auf die Daten und Ressourcen zugreift, die ihm gestattet sind.

Laufzeitsystem

- Ein Java-Programm interpretieren heißt, die einzelnen Anweisungen im Java-Programm abzuarbeiten. Das bedeutet aber mehr, als man dem Java-Quelltext direkt ansehen kann.

- Neben der Abarbeitung der Instruktionen im Source File werden noch einige unterstützende Dienste (Beispiele siehe nächste Folie) nebenher miterledigt.

- Die Gesamtheit aller solchen Dienste, die so im Hintergrund vom Interpreter erledigt werden und dafür sorgen, dass der selbst geschriebene Java-Quelltext überhaupt erst sinnvoll und sicher ausgeführt werden kann, nennt man das *Laufzeitsystem*.

- Diesbezüglicher Unterschied zum Compiler: Der Compiler ”mixt“ die Anweisungen aus dem Source File bei dessen Übersetzung mit den vordefinierten Anweisungen des Laufzeitsystems.

Laufzeitsystem (2)

- Programmiersprachen unterscheiden sich u.a. auch darin, was das Laufzeitsystem so alles leistet.
- In Java leistet das Laufzeitsystem wesentlich mehr als in den meisten anderen Programmiersprachen.
- *Beispiele:*
  - Die Prüfung des Zugriffs auf Daten und Ressourcen
  - Umgang mit mathematischen Ausnahmesituationen wie z.B. Division durch 0. Wird für einen solchen Fall keine besondere Vorkehrung im Programm getroffen, ist es Aufgabe des Laufzeit-systems, das Programm mit einer Fehlermeldung zu beenden.

Bestandteile

- SUN stellt zur Verfügung
  - Eine Ausführungsumgebung:
    - JVM: Java Virtual Machine bzw.
    - JRE: Java Run-time Enviroment
    - notwendig zur Ausführung von in Byte-Code übersetzten Java-Programmen
    - erlaubt die Ausführung von Java-Programmen auf verschiedensten Computer-Systemen
  - Eine Entwicklungsumgebung:
    - JDK: Java Development Kit
    - notwendig zur Entwicklung von Java-Programmen
- Anderer Anbieter bieten alternative Java-Implementierungen bzw. Erweiterungen
  - z.B. Entwicklungsumgebung Eclipse (www.eclipse.org)
  - Java Virtual Machines werden typischerweise mit Web Browsern mitgeliefert

2.1½ Programmierung in Java

1. Erstellen des Programms in einem Editor
   - Z.B. `nedit meinProgramm.java &`
2. Abspeichern des Programms
   - mit "Save" im Editor
3. Übersetzen des Programms in Java Byte Code
   - `javac meinProgramm.java`
   - erzeugt den Byte Code in einem File namens `meinProgramm.class`
4. Ausführen des Programms
   - `java meinProgramm`

- Eine IDE (Integrated Development Enviroment) integriert diese Schritte in eine Benutzeroberfläche
  - BlueJ
  - Eclipse
  - oder sogar die KarelJ IDE

Java Dokumentation

Keiner kann sich alles merken (und wir können auch nicht alles in der Vorlesung behandeln), daher schlagen auch erfahrene Programmierer hin und wieder in der Dokumentation nach.

→ Sie sollten das auch tun!

- Umfangreiche Java Dokumentation von Sun:
  - `http://java.sun.com/docs/`
- Bücher
  - Empfehlungen auf der Web-Seite
  - Insbesondere on-line Buch von Guido Krüger
    `http://www.javabuch.de`
    - z.B. Grafik & Farben: Kapitel 23-25
    - z.B. Applets: Kapitel 39 (weiterführend: 40)

2.1½.1 Einige neue Programm-Konstrukte in Java

- KarelJ ist eine “abgespeckte” Variante von Java
  - Das “J.” in Karel J. Robot steht (natürlich) für Java
- Viele Dinge, die wir von KarelJ kennen, lassen sich direkt in Java weiter verwenden
  - Schleifen (`while`, aber nicht `loop`)
  - Konditionale (`if {} else {}`)
  - grundlegende Typen (`int, boolean`)
  - Klassen- und Methoden-Deklarationen
  - prinzipielle Syntax (`{}`, `;`, u.v.m.)
- Die Robot-spezifischen Deklarationen sind natürlich nicht weiter verwendbar
  - die Klassen `World`, `UrRobot` und alle Unterklassen
  - die darin implementierten Methoden
  - der Typ `direction`
  - bestimmte Programmier-Konstrukte (`task`, `loop`)

for-Schleife

- In Java gibt es keine `loop`-Schleife!
  - die `while`-Schleife können Sie aber nach wie vor verwenden.

- Statt dessen gibt es die for-Schleife:

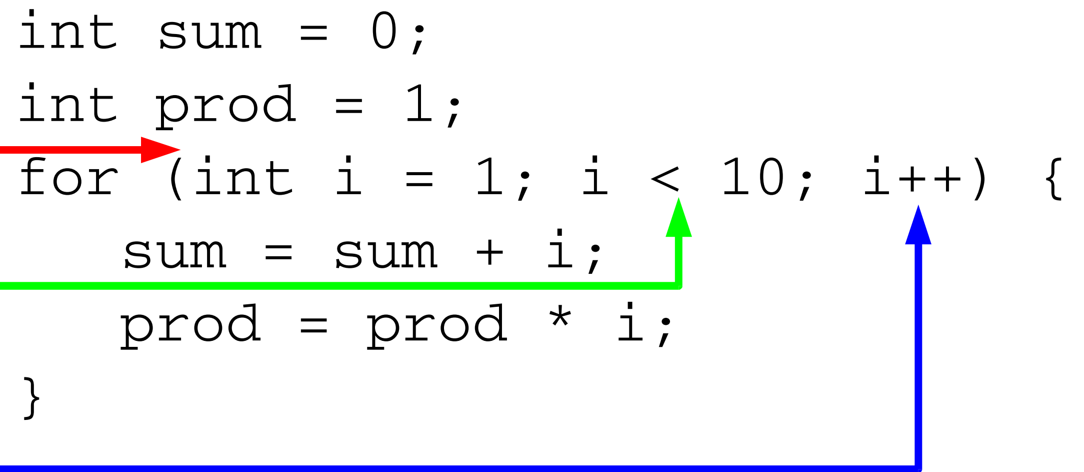
```
for (<Init>; <Test>; <Update>)
    Anweisung;
```

  - `<Init>`: Initialisierung der Schleife
    - wird genau einmal am Beginn ausgeführt
  - `<Test>`: Abbruchkriterium
    - wird vor jedem Schleifendurchlauf überprüft
    - wenn der Test true retourniert, wird die Anweisung (oder der mit { } begrenzte Block von Anweisungen) der Schleife durchgeführt.
  - `<Update>`: Veränderung der Schleifenvariablen
    - wird nach jedem Schleifendurchlauf durchgeführt

Beispiel

- `while`-Schleife
- äquivalente `for`-Schleife

```
int sum = 0;
int prod = 1;
int i = 1;
while (i < 10) {
    sum = sum + i;
    prod = prod * i;
    i++;
}
```



Anmerkung:

`i++` steht für `i = i+1`

Reelle Zahlen

- Bis jetzt hatten wir nur ganze Zahlen (int)
- Zwei Typen von reell-wertigen Zahlen:
  - `float`
    - definiert eine Gleitkommazahl
    - Beispiel: `float pi = 3.14159`
  - `double`
    - wie `float`, nur werden die Zahlen doppelt so genau abgespeichert (d.h. die Mantisse ist doppelt so lang)
    - wird häufiger als `float` verwendet
- **Beachte:**
  - Das Ergebnis einer arithmetischen Operation ist immer vom selben Typ wie die Operanden!
  - Beispiel:

```
int i = 11;
int j = 3;
System.out.println(i/j);
// 3 wird ausgegeben
```

```
double i = 11;
double j = 3;
System.out.println(i/j);
// 3.666666 wird ausgegeben!
```

Konvertierung von Zahlentypen

- Die verschiedenen Zahlentypen `int`, `float`, `double` können nicht so einfach ineinander übergeführt werden.
  - der Compiler erlaubt nur Zuweisungen zwischen Variablen gleichen Typs!
  - alle Variablen innerhalb einer Berechnung (eines arithmetischen Ausdrucks) müssen denselben Typ haben!
  - Grund: Verschiedene Zahlentypen werden ja (wie wir gesehen haben) intern verschieden abgespeichert!
- Dazu benötigt es eine explizite Konvertierung
  - Diese erfolgt indem man den gewünschten Typ in runden Klammern vor die Variable schreibt!
  - Falsch:

```
int i = 5;
double x = 10.0;
i = i + x;
```

Richtig:

```
int i = 5;
double x = 10.0;
i = i + (int) x;
```

Automatische Konvertierung

- der Compiler führt eine automatische Konvertierung durch, wenn sie ohne Beeinträchtigung der Genauigkeit möglich ist
  - Also: `int` → `float` → `double`
  - Aber nicht: `double` → `float` → `int`
- **Achtung:**
  - Die Konvertierung findet erst statt, wenn sie aufgrund der Inkompatibilität der Typen notwendig ist
    - das ist oft nicht dann, wenn man sie erwartet
  - Beispiel:

```
int x = 5;
int y = 3;
double z1 = x / y;
double z2 = (double) x / y;
```

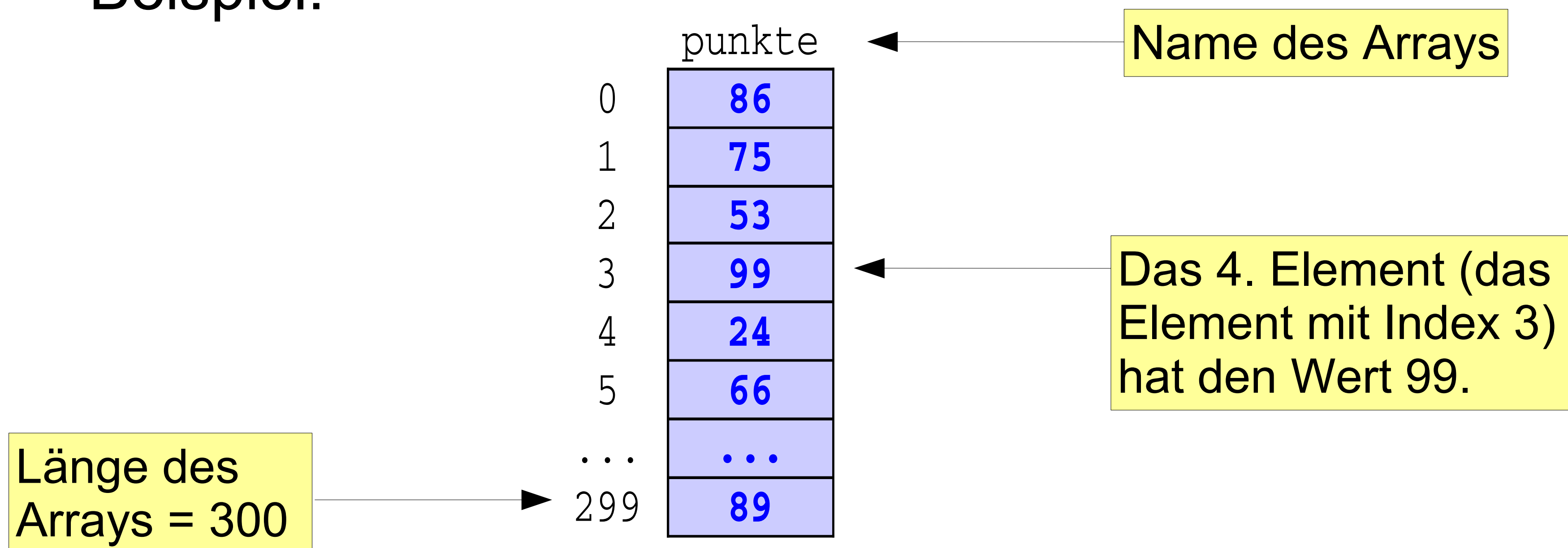
Division zweier `int`-Zahlen
→ `int` Resultat wird auf double konvertiert

`(double) x` bewirkt Konvertierung von `x`
→ `y` wird auch auf double konvertiert, erst dann wird dividiert!

Arrays

- Ein Array ist eine geordnete Liste von Variablen
  - gleichen Datentyps
  - die mit dem selben Namen angesprochen werden
  - aber jedes Element hat eine Nummer (Index)
    - ein Array mit n Elementen ist von 0 bis n-1 durchnumeriert

- Beispiel:



Arrays in Java

- Deklaration einer Array-Variable
  - *Typ* `[]` *Arrayname*
  - z.B. `int[] punkte;`
- Erzeugung eines Arrays
  - `new` *Typ* `[`*n*`]` oder `new` *Typ* `[] {` <Elements> `}`
  - z.B. `punkte = new int[3];`
  - z.B. `punkte = new int[] { 86, 75, 53 };`
- Lesen oder Schreiben von Array-Elementen
  - durch Angabe des Namens der Array-Variable
  - und der Indexnummer des Eintrags (0...n-1)
  - z.B. `punkte[2] = 53; int p = punkte[2];`
- Anzahl der Elemente eines Arrays
  - wird bei der Erzeugung unveränderbar festgelegt
  - Abfrage mit *Arrayname*`.length`
  - z.B. `punkte.length`

Beispiel

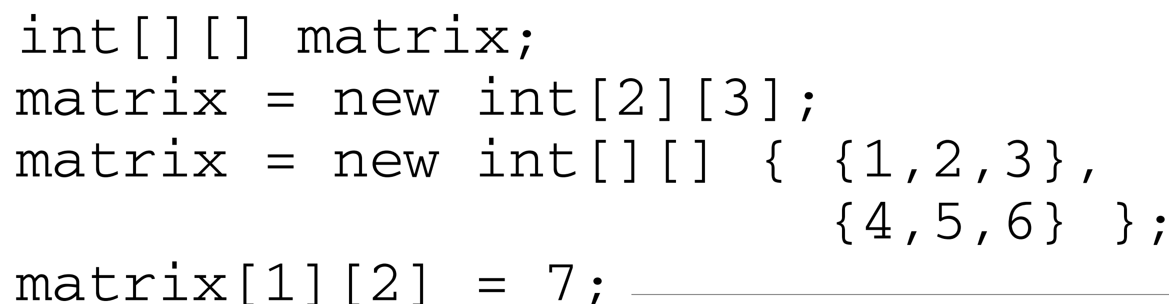
```
// Neuen Array  definieren
int[] punkte;

// Klausur wird benotet
punkte = new int[] { 86, 75, 53, 99, 24, 66,
                     // ... Hier fehlen weitere Werte
                     89 };

// Auswertung min, max, avg
int max_punkte = punkte[0];
int max_index = 0;
int sum = punkte[0];
for (int i = 1; i < punkte.length; i++) {
    if (punkte[i] > max_punkte) {
        max_punkte = punkte[i];
        max_index = i;
    }
    sum = sum + punkte[i];
}
float avg_punkte = (float) sum / (float) punkte.length;
System.out.println("Am besten war Student " + max_index
                   + " mit " + max_punkte + " Punkten.");
System.out.println("Durchschnitt:" + avg_punkte);
```

Mehrdimensionale Arrays

- Arrays können auch mehr als eine Dimension haben.
  - das heißt im Prinzip, daß jeder Eintrag in der Liste wiederum eine Liste ist
  - effektiv wird daher aus der Liste eine Matrix (in 2-dimensionalen Fall, bzw. ein (Hyper-)Würfel im allgemeinen Fall

- Beispiel:

matrix	0	1	2
0	**1**	**2**	**3**
1	**4**	**5**	**6**



public

- Java vergibt Benutzungsrechte für Klassen
  - Dadurch wird möglich, daß gewisse Klassen nur von gewissen anderen Klassen nutzbar werden

  → später im Skriptum

- einstweilen:
  - Allen Klassen- und Methoden-Definitionen wird das Keyword `public` vorangestellt
  - das erlaubt allgemeinen Zugriff auf die Klassen

import

- Komplexe Programme werden üblicherweise auf mehrere Files verteilt
  - Vorteil:
    Änderungen (und damit verbundene Neu-Übersetzungen) können relativ modular vorgenommen werden
- Java verlangt, daß für jede (public) class in einem eigenen File steht
  - Das File muß den gleichen Namen wie die Klasse haben
  - Also:

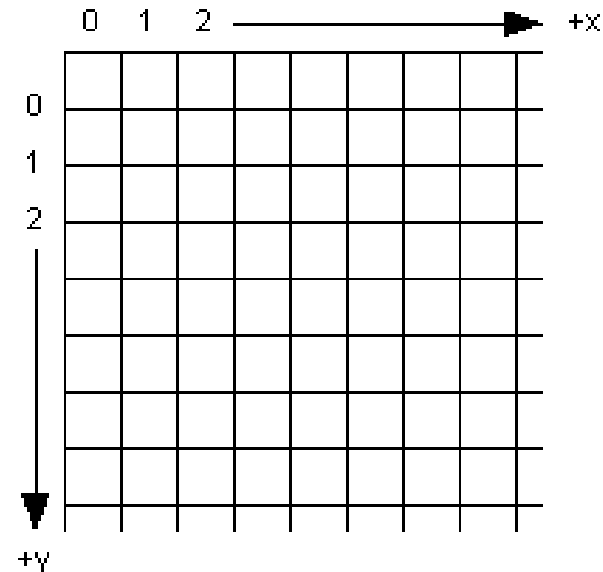
    ```
    public class MeineKlasse { }
    ```

    steht in einem File namens `MeineKlasse.java` bzw. `MeineKlasse.class` (nach der Übersetzung)
- mit dem Kommando import können Klassen aus dem momentanen Directory oder der Java-Installation importiert werden
  - Beispiel: `import MeineKlasse;`

2.1½.2 Graphik in Java

- Um die Grafikfähigkeiten von Java nutzen zu können, muß das Paket `java.awt` eingebunden werden.
  → `import java.awt.*;`

- Klasse `Graphics`
  - stellt eine universelles Ausgabegerät für Grafik und Schrift zur Verfügung
  - legt eine Koordinatensystem zugrunde, bei dem die linke obere Ecke (0,0) ist.



- die Methode `void` **`paint`**`(Graphics` **`g`**`)`
  - durch Überlagerung dieser Methode kann man auf ein `Graphics` Objekt zeichnen

Methoden der Klasse Graphics

- `void` **`drawLine`**`(int` **`x1`**`, int` **`y1`**`, int` **`x2`**`, int` **`y2`**`)`
  - Zieht eine Linie von `(x1,y1)` nach `(x2,y2)`

- `void` **`drawRect`**`(int` **`x`**`, int` **`y`**`, int` **`b`**`, int` **`h`**`)`
  - Zeichne ein Rechteck der Breite `b` und der Höhe `h`, dessen linke obere Ecke an der Position `(x,y)`
- `void` **`fillRect`**`(int` **`x`**`, int` **`y`**`, int` **`b`**`, int` **`h`**`)`
  - wie `drawRect`, nur wird das ganze Rechteck gefüllt

- `void` **`drawOval`**`(int` **`x`**`, int` **`y`**`, int` **`b`**`, int` **`h`**`)`
  - Zeichne den größten Kreis bzw. die größte Ellipse, die in ein Rechteck mit den angegebenen Spezifikationen paßt
- `void` **`fillOval`**`(int` **`x`**`, int` **`y`**`, int` **`b`**`, int` **`h`**`)`
  - wie `drawOval`, nur wird das ganze Oval gefüllt

Methoden der Klasse Graphics (2)

- `void drawPolyline(int[] x, int[] y, int n)`
  - Zeichnet ein offenes Polygon durch `n` Punkte, deren x-Koordinaten im Array `x` und deren y-Koordinaten im Array `y` abgespeichert sind.

- `void drawPolygon(int[] x, int[] y, int n)`
  - Zeichnet ein geschlossenes Polygon durch `n` Punkte, deren x-Koordinaten im Array `x` und deren y-Koordinaten im Array `y` abgespeichert sind.

- `void fillPolygon(int[] x, int[] y, int n)`
  - Füllt ein geschlossenes Polygon durch `n` Punkte, deren x-Koordinaten im Array `x` und deren y-Koordinaten im Array `y` abgespeichert sind.

Farben in Java

- Farben bilden eine eigene Klasse Color

- Color stellt einige vordefinierte Farben zur Verfügung:
  - `Color.white`
  - `Color.lightGray`
  - `Color.gray`
  - `Color.darkGray`
  - `Color.black`
  - `Color.red`
  - `Color.blue`

  `Color.green`
  `Color.yellow`
  `Color.magenta`
  `Color.cyan`
  `Color.orange`
  `Color.pink`

- Alternativ läßt sich eine neue Farbe nach dem RGB-Schema basteln
  - Konstruktor: `Color(int` **`r`**`, int` **`g`**`, int` **`b`**`)`
  - Beispiel:

```
Color meineFarbe = new Color(0,128,255);
```

Verwendung von Farben in Graphics

- `Color` **`getColor`**`()`
  - retourniert das Farbobjekt für die momentane Farbe

- `int` **`getRed`**`()`
  - retourniert den Rot-Anteil der momentanen Farbe
- `int` **`getGreen`**`()`
  - retourniert den Grün-Anteil der momentanen Farbe
- `int` **`getBlue`**`()`
  - retourniert den Blau-Anteil der momentanen Farbe

- `void` **`setColor`**`(Color` **`c`**`)`
  - setze die Farbe auf einen neuen Wert
  - Beispiele:

```
Graphics g;   // g ist ein Graphics-Objekt
g.setColor(Color.green);
g.setColor(new Color(255,255,0));
```

Strings und Textausgabe

- `String`
  - Zeichenketten werden durch sogenannte Strings repräsentiert
  - Strings können direkt mit Hilfe von Anführungszeichen angegeben werden:
    - Beispiel:

```
String str = "Das ist ein String";
```

  - oder durch "Addition" aneinandergekettet werden
    - Beispiel:

```
String str = "Das";
str = str + " ist ein String";
```

- `void` **`drawString`**`(String` **`str`**`, int` **`x`**`, int` **`y`**`)`
  - Zeichnet einen String str auf der Position (x,y)

2.1½.3 Applets

- Konventionelle Web-Seiten:
  - HTML erlaubt nur die Anzeige von Text
  - Oft ist es jedoch nützlich, Web-Seiten programmieren zu können

- Zwei Arten von Programm-Unterstützung für Web-Seiten:
  - **Applets:**
    - Programme, die in Web-Seiten eingebunden sind und auf dem Rechner des Browsers ausgeführt werden
    - z.B. Interaktive, dynamische und/oder grafische Komponenten auf Web-Seiten
  - **Servlets:**
    - Programme, die auf dem Server ausgeführt werden, und Web-Seiten dynamisch erstellen
    - z.B. Web-Seiten, die automatisch auf Datenbanken generiert werden

Einbindung von Applets

- In der Web-Seite wird (mit einem speziellen Tag) ein Link zu einem Programm in Java Byte Code angegeben
  - d.h. zu einem übersetzten Java Programm
- Der Browser lädt das Programm und übergibt es an die JVM des Clients (auf dem der Browser läuft)
  - reserviert einen Bildschirmbereich im Browser-Fenster für die Ausgabe des Applets
- Die JVM führt das Programm aus
  - Die Ausgabe des Programms wird in das Browser-Fenster eingebunden

Java für Applets

- Java eignet sich ausgezeichnet zur Programmierung von Applets
  - da der Java Byte Code auf allen gängigen Betriebssystemen lauffähig ist
  - das heißt, das jeder Benutzer die Programme ausführen kann, unabhängig von seiner Computer-Ausstattung
- Java stellt einfache Programm-Konstrukte zur Verfügung, die es erleichtern, Applets zu programmieren.
  - insbesondere das Handling von (grafischem) Output und die Kommunikation mit dem Browser werden dem Programmierer abgenommen
  - Um ein Applet in Java programmieren zu können, muß das Paket `java.applet` eingebunden werden.

    → `import java.applet.*;`
- Nicht zuletzt dadurch hat sich Java zu *der* Internet-sprache schlechthin entwickelt.

Die Applet-Klasse

- `Applet` ist eine Klasse, die es erlaubt, Applets zu programmieren
  - genauso, wie `UrRobot` eine Klasse ist, die es erlaubt, Robots zu programmieren
- programmiert werden muß üblicherweise nur eine Methode, die die Bildschirmausgabe produziert
  - alle anderen Methoden können ererbt werden
- die Klasse stellt generische Methoden zur Verfügung, die angeben, was passieren soll
  - wenn das Applet geladen wird
  - wenn die Ausführung des Applets begonnen wird
  - wenn die Ausführung des Applets beendet wird
  - wenn das Applet wieder beendet wird

Die Methoden von Applet

- `init()`
  - wird (von Browser) nach dem Laden des Applets aufgerufen (genau einmal)
- `start()`
  - wird aufgerufen, nachdem die Initialisierung abgeschlossen wurde (kann mehrmals aufgerufen werden)
- `stop()`
  - wird aufgerufen, wenn die HTML-Seite verlassen wird. Gestoppte Applets können mit `start` wiederbelebt werden.
- `destroy()`
  - wird aufgerufen, wenn das Applet beendet wird (z.B. bei Beenden des Browsers)
- `paint(Graphics g)`
  - erlaubt grafische Ausgabe auf das Grafik-Objekt g, das automatisch von `Applet` angelegt wird.
  - bei einfachen Applets muß nur diese Methode programmiert werden, die anderen können geerbt werden.

Einfaches Beispiel

- Ein Applet, das den Text “Hello World!” im Fenster des Browsers anzeigt

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet
{
   public void paint(Graphics g)
   {
      g.drawString("Hello World!",10,50);
   }
}
```

Einfaches Beispiel

- Ein Applet, das den Text “Hello World!” im Fenster des Browsers anzeigt

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet
{
   public void paint(Graphics g)
   {
      g.drawString("Hello World!",10,50);
   }
}
```

Graphik importieren

Applets importieren

Wir definieren eine Klasse HelloWorld

HelloWorld soll ein Applet sein

Das Applet implementiert die `paint` Methode, alle anderen werden geerbt.

In `g` bekommt das Applet das Zeichenbrett übergben

Eigentliches Programm: Der Text wird auf den Koordinaten (10,50) ausgegeben.

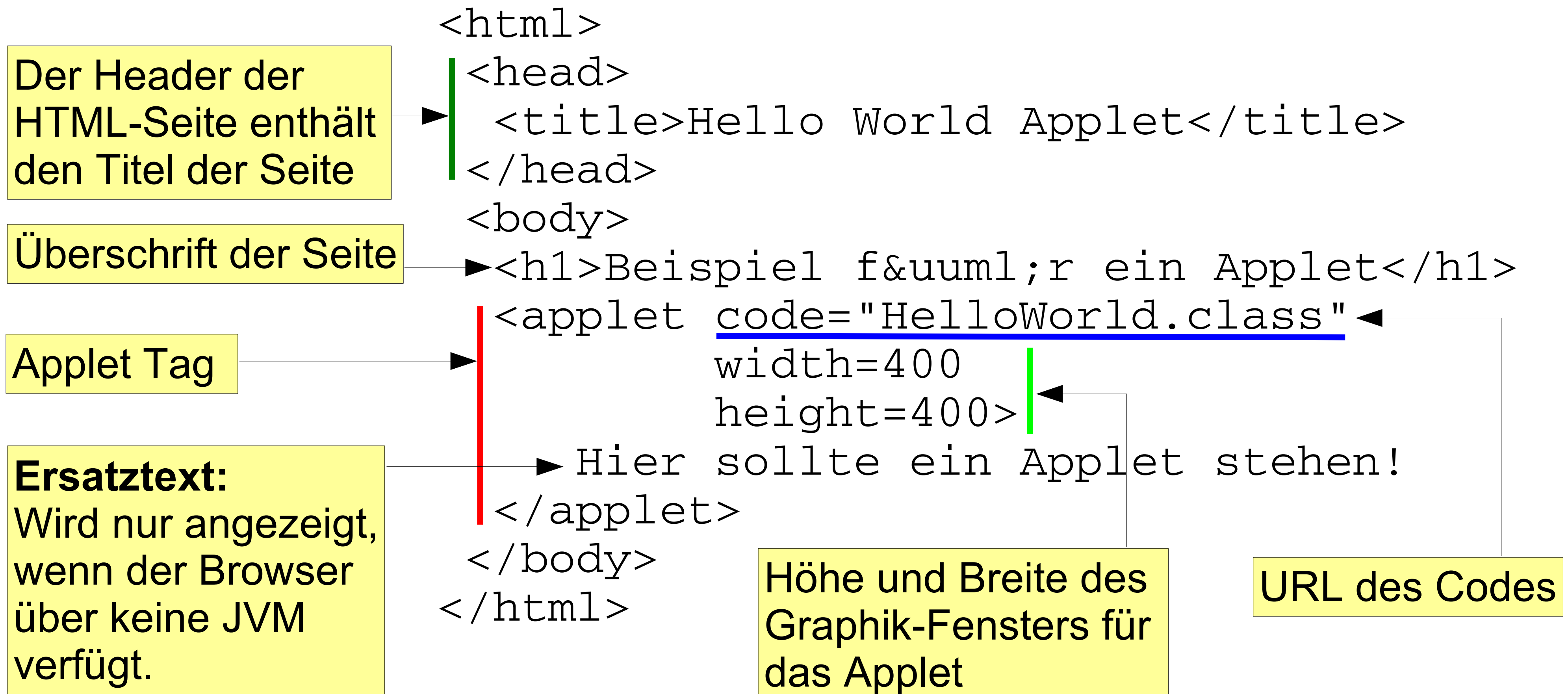
Einbindung in eine Web-Seite

- Eine Beispiel-Seite, die das Applet HelloWorld.java von der vorigen Folie einbindet.
  - **Anm.:** Der HTML-Teil kann beliebig gestaltet werden

```
<html>
 <head>
  <title>Hello World Applet</title>
 </head>
 <body>
  <h1>Beispiel f&uuml;r ein Applet</h1>
  <applet code="HelloWorld.class"
          width=400
          height=400>
     Hier sollte ein Applet stehen!
  </applet>
 </body>
</html>
```

Einbindung in eine Web-Seite

- Eine Beispiel-Seite, die das Applet HelloWorld.java von der vorigen Folie einbindet.
  - **Anm.:** Der HTML-Teil kann beliebig gestaltet werden



Ausführen bzw. Testen von Applets

- Das Applet wird nicht direkt ausgeführt, sondern über die Web-Seite, in die es eingebunden wird
  - Erinnerung: Wir haben drei Files!
    - `MeineWebSeite.html`:
      - die Web-Seite
    - `MeinApplet.java`:
      - Java-Programm, das das Applet MeinApplet implementiert
      - wird vom Browser nicht benötigt!
    - `MeinApplet.class`:
      - Java Byte Code des Programms MeinApplet.java
      - wird durch Aufruf von javac generiert
      - dieses File wird vom Browser geladen
- 2 Möglichkeiten:
  - Anzeigen der Seite in einem Web-Browser, der Java unterstützt (und in dem es nicht deaktiviert worden ist)
  - Anzeigen der Seite mit einem speziellen Programm:

```
appletviewer MeineWebSeite.html &
```

2.2 Abstraktionsebenen

- Lexikalische Ebene (Wortebene)
- Syntaktische Ebene (Satzebene)
- Semantische Ebene (Bedeutungsebene)
- Logische Ebene
- Spezifikatorische Ebene

Beachte:

- Die Grenze zwischen den Ebenen ist fließend.
- Die Grenzziehung in dieser Vorlesung ist daher bis zu einem gewissen Grad auch willkürlich.
- Was allerdings nicht gleichbedeutend ist mit ”ohne Überlegung“!

Allgemeine Vorbemerkung

In Programmiersprachen sind die lexikalischen, syntaktischen und semantischen Regeln, was korrekt ist und was nicht, sehr viel rigider und pedantischer als etwa in natürlichen Sprachen.

Gründe:

- Es ist wesentlich schwieriger und aufwendiger, einen Sprach-erkenner und Compiler oder Interpreter für eine weniger pedantische Sprache zu entwickeln.

  → Für natürliche Sprachen ist dieses Problem bis heute ungelöst!

- Die Möglichkeit mehrdeutiger Texte wäre
  - ◊ ab einem gewissen Komplexitätsgrad der Sprache wohl kaum vermeidbar,
  - ◊ bei normaler Kommunikation in natürlichen Sprachen ein häufiges Ärgernis,
  - ◊ bei Programmiersprachen eine Katastrophe.

Die Vorige Folie nach automatischer Übersetzung D-E-Chinesisch-E-D

In der Programmiersprache ist das Glossar, die Grammatik und die semantische Richtlinie, ist korrekt und ist nicht, viel mehr rigider und mehr pedantischer vergleicht spricht zum Beispiel in der Natur.

Grund:

- Er ist enorm sogar mehr ist schwierig und eine kompliziertere Entwicklung spricht mehr erkenner und der Kompilator oder der Interpret ist eine Art von weniger pedantische Sprache
  → Spricht diese Frage für die Natur, bis heutiger Tag die Lösung ist!
- Die vieldeutige Textmöglichkeit kann sein
  - Anfänge von der Sprache irgendein komplizierter Grad vermutlich fast, zum nicht in der Lage zuSEIN zu vermeiden
  - In der normalen häufigen Korrespondenzperiode wird in der Naturrede gestört
  - Programmierspracheunfall.

Übersetzung: © Babel Fish
`http://world.altavista.com/`

Grundregel

- Lexikalische und syntaktische Fehler resultieren in Fehlermeldungen beim Kompilieren (also beim Aufruf von ”`javac`“).

  → Kein Source File mit lexikalischen oder syntaktischen Fehlern kann in Java Byte Code übersetzt werden.

- Fehler der anderen Arten resultieren in Fehlverhalten des Programms zur Laufzeit (also beim Laufenlassen mit ”`java`“):

  ◊ Programmabsturz,

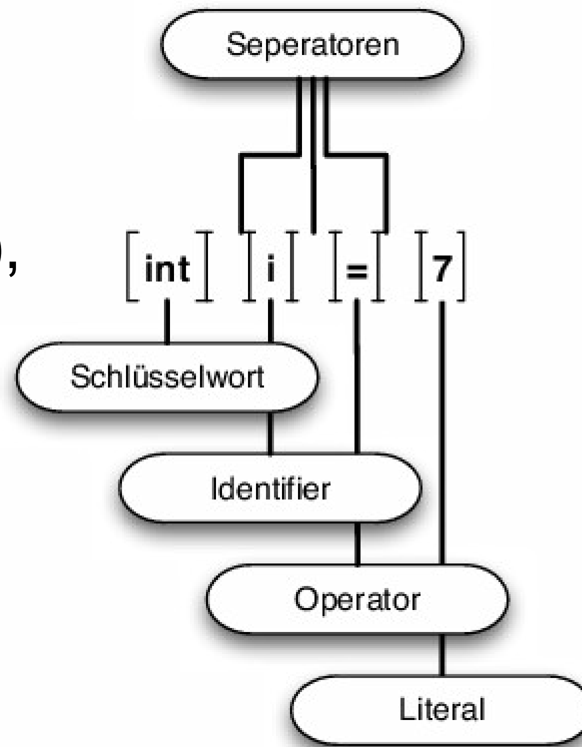
  ◊ korrekte Programmbeendigung mit falschen Endergebnissen,

  ◊ falsche oder unerwünschte Effekte des laufenden Programms auf Fileinhalte, Bildschirmausgaben etc.

2.2.1 Lexikalische Ebene

- Auf unterstem Abstraktionsniveau ist ein Programm zusammengesetzt aus
  - ◊ lexikalischen Einheiten (auch *Lexeme* oder *Token genannt*)
  - ◊ sowie trennenden Elementen (*Separatoren*).

- *Lexeme*:
  - ◊ Schlüsselwörter,
  - ◊ Bezeichner (*Identifier*),
  - ◊ Literale,
  - ◊ Operatoren,
  - ◊ Klammerungszeichen,
  - ◊ Semikolon.



- *Separatoren*:
  - ◊ Leerzeichen
  - ◊ Tabulatorzeichen,
  - ◊ Zeilenumbruch

Schlüsselwörter

- Reservierte Zeichenketten mit besonderer Bedeutung.
- *Beispiele* in Java: ”`if`“, ”`for`“, ”`class`“, ”`return`“.
- In Java bestehen alle Schlüsselwörter aus Kleinbuchstaben.

- *Auflistung* aller Java–Schlüsselwörter:
  - Die wichtigsten werden wir sukzessive kennen lernen

  → Java Dokumentation

Identifier (Bezeichner)

- *Bedeutung*: Jede Bezeichnung, die vom Programmierer des Source Files selbst ausgedacht ist.

  → Namen von Variablen, Konstanten, Klassen, Methoden, Paketen (später in der Vorlesung).

- Im Prinzip ist jede beliebige Zeichenkette erlaubt, die aus Groß- und Kleinbuchstaben, Ziffern sowie dem Unterstrich "_" (engl. *Underscore*) besteht.

- *Ausnahmen* zur Vermeidung von Zweideutigkeiten:

  ◊ Schlüsselwörter dürfen natürlich nicht als Identifier gewählt werden.

  ◊ Ein Identifier darf nicht mit einer Ziffer beginnen, um Verwechslungen mit Zahlen zu vermeiden.

Groß- und Kleinschreibung

In Java wird im Gegensatz zu einigen anderen Programmiersprachen zwischen Groß- und Kleinschreibung bei Identifiern unterschieden:

- ”`hallo`“, ”`Hallo`“ und ”`HALLO`“ sind drei verschiedene Identifier.
- ”`IF`“, ”`If`“ und ”`iF`“ sind zulässige Identifier, obwohl ”`if`“ ein Schlüsselwort ist.

Beispiele:

```
int i = 7
```

Hier ist „i” der Identifier.

```
int iF = 7; int If; int IF
```

Alle diese Identifier sind zulässig . . .

```
if (iF==If) return IF;
```

. . . aber nicht unbedingt sinnvoll!

Wahl von Identifiern

- In Java ist gemäß letzter Folie bei der Wahl von Identifiern sehr viel erlaubt.
- Aber damit ein Programm lesbar und verständlich bleibt, sollten die Identifier so gewählt sein, dass sie so etwas wie ”sprechende Kommentare“ bilden.
- Das ist vor allem wichtig,
  - ◊ wenn das Programm deutlich größer ist als unsere einfachen Beispiele in Vorlesung und Übungen
  - ◊ wenn man selbst das Programm in 2 Monaten wieder verstehen will
  - ◊ wenn andere Leute das Programm auch verstehen sollen.

→ Also praktisch immer.

Identifier als Kommentare

- Der Identifier selbst ist der einzige Kommentar zur Funktion dieses Identifiers, der bei jedem Auftreten des Identifiers garantiert dabei ist.

  → Man muss nicht erst umständlich irgendwo anders nach einem Kommentar zu diesem Identifier suchen.
- ”Richtige“ Kommentare hinter ”`//`“ oder in ”`/* ... */`“ sind erfahrungsgemäß in der Praxis eher rar und oft falsch oder veraltet.
- Das natürliche Bedürfnis des Programmierers, möglichst wenig zu tippen und daher extrem kurze Identifier zu verwenden, muss dahinter zurücktreten.

Positive Beispiele:

- Methoden ”`drawString`“, ”`drawLine`“, ”`drawOval`“, ”`fillOval`“ von ”`Graphics`“.
- ”`zeichneHausVomNikolaus`“ für eine Methode, die das Haus vom Nikolaus zeichnet.

Stilistische Regeln

- Im Gegensatz zu anderen Programmiersprachen hat sich in Java ein Regelwerk für die ”Stylistik“ von Identifiern allgemein durchgesetzt.
- Viele vorgefertigte Java–Bausteine und unzählige Java–Anwendungsprogramme vertrauen darauf, dass diese Konventionen streng eingehalten werden.

- *Grundregeln*:
  - ◊ Der Underscore ist eher verpönt.
  - ◊ Wortanfänge innerhalb eines Identifiers werden mit Großbuchstaben gekennzeichnet.
  - ◊ Namen von Klassen beginnen mit Großbuchstaben.

    → ”`UrRobot`“
  - ◊ Namen von Variablen und Methoden beginnen mit Kleinbuchstaben.

    → ”`karel`“, ”`putBeeper`“

Operatoren

- mit Operatoren können Zuweisungen und Berechnungen durchgeführt werden
- Arithmetische Operatoren:
  - `+`, `-`, `*`, `/`: Grundrechnungsarten
  - `%` berechnet den Rest einer Division (→ später)
  - `++`, `--`: Den Wert einer Variablen um 1 erhöhen/vermindern
- Vergleichsoperatoren
  - `<`, `>`, `==`, `<=`, `>=`, `!=`
- Logische Operatoren
  - `!`: nicht
  - `&`, `&&`: und
  - `|`, `||`: oder
  - `^`: exlusives oder
- Es gibt noch mehr!

  → Java Dokumentation

Klammerungszeichen

Runde, geschweifte und eckige Klammern, d.h.

- ( )
  - Zur Festlegung der Auswertungsreihenfolge von Operatoren (Änderung der → Priorität von Operatoren)
  - Zur Angabe von Parametern bei Methoden-Aufrufen oder syntaktischen Konstrukten
- { }
  - Zur Zusammenfassung von mehreren Anweisungen in eine einzelne Anweisung, einen sogenannten Block
  - Zur Angabe von Array-Elementen bei der Initialisierung von Arrays
- [ ]
  - Zur Angabe der Index-Variablen und Dimensionen in Arrays

Literale

- Literale sind explizit hingeschriebene Werte.
- *Beispiele*:
  - ◊ `69534`: ganzzahliges Literal,
  - ◊ `3.14159`: reellwertiges Literal,
  - ◊ `’a’`: Zeichenliteral (Anführungsstriche gehören dazu!)
  - ◊ `”Hello World”`: Stringliteral (inkl. Anführungsstriche!)
  - ◊ `true` und `false`: die beiden Literale zum Datentyp `boolean`.

Separatoren

- In Java dürfen (wie in eigentlich allen modernen Programmier-sprachen) beliebig viele Separatoren in beliebiger Mischung zwischen Lexeme gesetzt werden.
- Andererseits besteht nur in bestimmten Konstellationen überhaupt der Zwang, mindestens einen Separator zwischen zwei unmittelbar aufeinanderfolgende Lexeme zu platzieren.

  Beispiel: zwischen Typname und Variablenname bei der Variablen-deklaration.

```
int  i;        //  Erlaubt
int       i;   //  Erlaubt
inti;          //  Verboten
```

- In den meisten Programmiersprachen (einschl. Java) ist es logischerweise strikt verboten, Separatoren *innerhalb* von Lexemen zu plazieren.
- Die Freiheit der Platzierung zwischen Lexemen kann man in Java zur übersichtlichen, gut lesbaren Formatierung des Source Files nutzen.

Beispiel

- Zweimal derselbe Quelltext bis auf Platzierung von ”unnützen“ zusätzlichen Separatoren.
- Welche Version ist übersichtlicher und besser lesbar?

Version 1:

```
double summe = 0;
double quadratsumme = 0;
for ( int i=0; i <= n; i++ )
{
   summe += i;
   quadratsumme += i * i;
}
```

Version 2:

```
double summe=0;double quadratsumme=0;for(int
i=0;i<=n;i++){summe+=i;quadratsumme+=i*i;}
```

Abschließende Bemerkung

- In einigen älteren Programmiersprachen (bzw. älteren Versionen heutiger Programmiersprachen) müssen wesentlich strengere lexikalische Regeln befolgt werden.

- *Beispiel*: Alte Versionen von Fortran.

- *Konkretes Beispiel* daraus: Zeilenformatierung.

  ◊ Die Zeilen dürfen nicht zu lang werden (z.B. nicht mehr als 80 Zeichen).

  ◊ Einzelne Bestandteile einer Programmzeile müssen an vorgegebenen Abständen zum Zeilenbeginn platziert werden.

→ Altlast aus dem Zeitalter der Lochkarten.

2.2.2 Syntaktische Ebene

- Hier geht es nicht um die Korrektheit einzelner Lexeme,
- sondern um die korrekte Gruppierung von Lexemen in Java Source Files.
- *Erinnerung*: Syntaktische Fehler werden wie lexikalische Fehler schon beim Kompilieren abgefangen.

Beispiel: Korrekte Klammerungen

- Öffnende und schließende Klammern dürfen nur in Paaren im Source File auftreten.

- Die öffnende Klammer kommt dabei vor der schließenden.

- Die Textabschnitte innerhalb zweier Klammerpaare dürfen
  - ◊ nacheinander ohne Überlappung platziert sein:

    `( . . . ) . . . ( . . . ) . . . { . . .   }`

  - ◊ ineinander enthalten ("geschachtelt") sein:

    `( . . . [ . . . { . . . } . . . ( . . . ) . . . ] . . . )`

  - ◊ aber sich nicht anderweitig überlappen:

    `( . . . [ . . . ) . . . ]` ← Verboten!

Beispiel: Platzierung von Schlüsselwörtern

- Ein Schlüsselwort ist Teil eines Programmkonstrukts.
- Es darf nur innerhalb dieses Programmkonstrukts auftreten,
- und auch dort nur in genau festgelegter Form an genau festgelegter Stelle.

Beispiel:

Das Schlüsselwort ”`while`“ steht am Anfang des Konstrukts ”`while`–Schleife“:

`while` (*Bedingung*) *Block*

Anweisungen

- Elementare Anweisungen wie Zuweisungen mit "=" oder Inkrement/Dekrement ("++"/"--").

- Methodenaufrufe wie

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

- `if`-/`while`-/`for`–Konstrukte etc.

- Zusammengesetzte Anweisungen
  - mehrere Anweisungen können durch geschwungende Klammern in einen Block zusammengefaßt werden

```
int i = 0;
  while ( i < 10 )
    {
      if ( i < 5 )
         System.out.println (i);
      else
         System.out.println (10-i);
      i++;
    }
```

Anweisung

Anweisung

Schachtelung

- `if`-/`while`-/`for`–Konstrukte u.ä. können beliebig ineinander geschachtelt werden.
- Insbesondere können auch zwei oder mehr Konstrukte des selben Typs ineinander geschachtelt werden.
- Schachtelung wird durch Einrückung offensichtlicher!

```
int i = 0;
while ( i<10 ) {
  if ( i==5 ) {
    int j = 0;
    while ( j<10 ) {
      if ( j==i ) {
        int k = 0;
        while ( k != i-j ) {
          if ( k != j )
            System.out.println(j);
          k++;
        }
      }
    j++;
    }
  }
  i++;
}
```

2.2.3 Semantische Ebene

- Bisher haben wir zwei Ebenen betrachtet, die mit der *Form* von Texten zu tun hatten.

  → Wortebene und Satzebene.

- Der Begriff ”Semantik einer Sprache“ bezieht sich im Gegensatz dazu auf die *Bedeutung* von Texten, die lexikalisch und syntaktisch korrekt formuliert sind.

Beispiel 1: Bindungsstärke von Operatoren

- *Erinnerung*: Es gibt verschiedene Operatoren in Java.
- Die verschiedenen Operatoren haben verschiedene *Bindungsstärken*.
- Die Bindungsstärke richtet sich nach der *Priorität* des Operators.

→ Damit wird die Auswertungsreihenfolge in einem Ausdruck mit mehreren Operatoren festgelegt.

- *Beispiel*:

```
int i = 3 - 10/2*4 + 1;
```

  ◊ Wie in der Mathematik gilt in Java Punkt- vor Strichrechnung.

  ◊ Ansonsten werden zumindest die hier verwendeten vier arithmetischen Operatoren strikt von links nach rechts ausgewertet.

→ Der Wert von "i" in der obigen Quelltextzeile ist $\left(3-\left(\frac{10}{2}\cdot 4\right)\right)+1=-16$

Klammerung zur Änderung der Bindungsstärke

Beispiel:

```
int i = 3 -  10 /  2 *  4 + 1   ;
int j = 3 - (10 / (2 * (4 + 1)));
```

Erläuterungen:

- Durch die Klammerung werden Einheiten, Untereinheiten und Unteruntereinheiten des mathematischen Ausdrucks gebildet.
- Der Inhalt jedes Klammerpaars wird als Einheit berechnet und mit dem Rest weiterverarbeitet.
- Bei ineinandergeschachtelten Klammerpaaren folgt daraus Auswertung von innen nach außen.

→ Der Wert von "`j`" im obigen Quelltextfragment ist $3-\frac{10}{2\cdot(4+1)}=2$

Beispiel 2: Operatoren und Datentypen

Speziell Divisionsoperator:

```
int     i  =  17;
int     j  =  3;
int     k  =  i  /  j;     // k == 5
float   x  =  17;
float   y  =  3;
float   z  =  x  /  y;     // z == 5,66...
```

Unterschied in der Semantik:

- Ganzzahlige Division, wenn beide Operanden ganzzahlig sind.

→ Der Wert von ”`k`“ ist 5.

- Reelle Division, wenn mindestens einer der beiden Operanden reell ist.

→ Bis auf numerische Ungenauigkeiten ist der Wert von ”`z`“ gleich 5.666 ... .

Achtung Falle!

```
float z = 17 / 3;
```

- Der Datentyp für das Ergebnis der Division ist zwar reell.
- Die beiden Operanden sind aber ganzzahlig, also wird eine Ganzzahldivison durchgeführt, deren Ergebnis verlustfrei in einen float umgewandelt werden kann.

→ Gemäß ganzzahliger Division mit Rest ist der Wert von ”`z`“ in diesem Fall gleich 5.0, nicht 5.666 ... .

Modulo-Operator

- Der Operator ”`%`“ berechnet den Rest bei einer Division mit Rest:
  - ◊ `10 % 3` ➔ `1`
  - ◊ `11 % 3` ➔ `2`
  - ◊ `12 % 3` ➔ `0`
- Da diese Operation in der Mathematik ”modulo“ genannt wird, heißt dieser Operator der *Modulo–Operator*.

Beispiel

```
int wochentag = 0;    //  Sonntag
...

if (  istMitternacht ( )   )
   wochentag = ( wochentag + 1 ) % 7;
```

Erläuterungen:

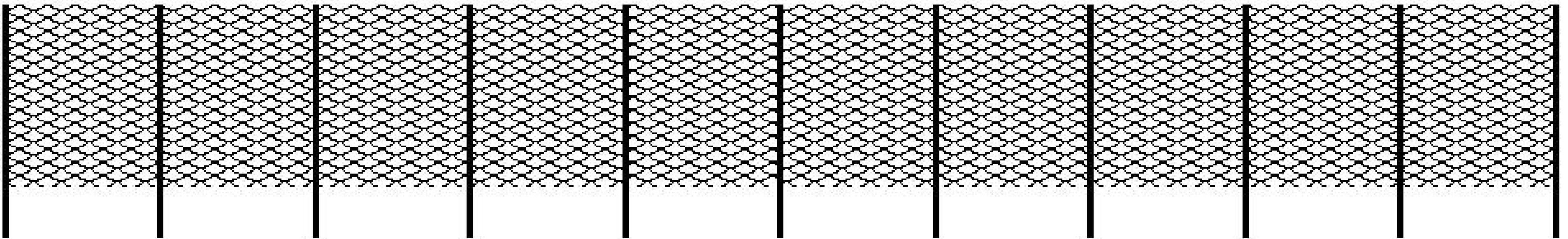
- Um den aktuellen Wochentag zu speichern, könnte man zum Beispiel eine "int"–Variable nehmen, die die Werte `0...6` annimmt.
- Dabei würde dann zum Beispiel `0` für Sonntag, `1` für Montag usw. bis `6` für Samstag stehen.
- Wenn der Wert von Tag zu Tag umgeschaltet wird, soll er von `0` bis `6` wachsen und dann wieder auf `0` gehen.
- Das ist aber genau das Verhalten "`% 7`".

2.2.4 Logische Ebene

- Wir gehen in diesem Punkt von lexikalisch/syntaktisch korrekten Quelltexten aus, die in Programme übersetzbar sind und auch keine semantischen Fehler enthalten.
- *Frage*: Was heißt es dann eigentlich noch, dass ein Programm korrekt bzw. nicht korrekt ist?
- *Salopp formulierte Antwort*: Ein Programm ist dann korrekt, wenn es
  - ◊ alles das tut, was es soll,
  - ◊ und das mit allen erwünschten Effekten,
  - ◊ und darüber hinaus keine Effekte hat, die es definitiv nicht haben soll.
- Die Formulierung des erwünschten und unerwünschten Verhaltens eines Programms ist seine *Spezifikation*.
- Nur relativ zu einer Spezifikation macht es überhaupt Sinn, von Korrektheit/Nichtkorrektheit eines Programms zu sprechen.

Beispiel für logischen Fehler

- Das sogenannte ”Fencepost Syndrom“ (auch ”off–by–one error“).
- What’s in a name: Wie viele Zaunpfähle braucht man, wenn
  - ◊ der Zaun 20 Meter lang werden soll und
  - ◊ alle 2 Meter ein Pfahl steht?
- *Antwort*:
  - ◊ Die unwillkürliche Antwort ist ”10“.
  - ◊ Die korrekte Antwort ist aber ”11“:



Programmierbeispiel dazu

```
int linkestesPixel   = 187;
int rechtestesPixel = 379;
...
int fensterWeite = rechtestesPixel
                 - linkestesPixel;
```

Erläuterungen:

- Die Weite eines Fensters ist die Anzahl der nebeneinanderliegenden Pixel von ganz links bis ganz rechts.
- Im obigen Beispiel geht das Fenster von Bildschirmpixel Nr. 187 ganz links bis Bildschirmpixel Nr. 379 ganz rechts.
- Unwillkürlich ist man geneigt, die Fensterweite als Differenz der beiden Pixelwerte zu berechnen.
- *Fencepost Syndrom*: Tatsächlich ist die Weite gleich

```
rechtestesPixel - linkestesPixel + 1
```

Beispiel für Array-Fehler

- In einem nicht ganz unwichtigen Programm ist für den (englischen) Namen des Wochentags ein Array von acht Zeichen reserviert worden.

- *Konsequenz*:
  - ◊ Das Programm lief eigentlich immer problemlos und korrekt...
  - ◊ ...außer mittwochs!

M	O	N	D	A	Y			
T	U	E	S	D	A	Y		
W	E	D	N	E	S	D	A	Y
0	1	2	3	4	5	6	7	

→ Beispiel dafür, dass ein logischer Fehler häufig nicht so ohne weiteres reproduzierbar ist, um ihn einzukreisen und aufzuspüren.

Erkennen von logischen Fehlern

- Programmierfehler auf der logischen Ebene werden nicht beim Kompilieren des Quelltextes mit Fehlermeldung abgefangen.
- Statt dessen führen solche Fehler zu *undefiniertem* (d.h. unvorhersehbarem) Verhalten des Programms bei seinem Ablauf als Prozess.
- *Zum Beispiel*:
  - ◊ Programmabsturz, d.h. vom Programm nicht beabsichtigte Termination durch das Eingreifen des Laufzeitsystems
  - ◊ keine *Termination* (= Beendigung),
  - ◊ kontrollierte Termination mit inkorrekten Ergebnissen,
  - ◊ kontrollierte Termination mit korrekten Ergebnissen (durchaus möglich!),
  - ◊ Rechnerstillstand (bei gewissen Betriebssystemen...),
  - ◊ ...

Beispiel: Keine Termination

```
int i = 0;
while (i < 10)
   System.out.println(i);
   i++;
```

Erläuterung:

- Eigentlich sollte das obige Java–Programmfragment die Zahlen `0...9` ausgeben.
- Durch eine Unachtsamkeit ist aber vergessen worden, Klammern um die beiden Anweisungen zu setzen.

  → Passiert sehr leicht, wenn man in eine Schleife mit einer einzigen Anweisung nachträglich noch weitere Anweisungen einfügt.
- Der Wert von "`i`" bleibt daher immer gleich `0`.

  → Die Bedingung "`i < 10`" bleibt immer wahr.

  → Nie abbrechende "Endlosschleife".

2.2.5 Spezifikatorische Ebene

- Auf der logischen Ebene haben wir es mit Diskrepanzen zwischen Spezifikation und Programmverhalten zu tun.
- Auf der spezifikatorischen Ebene geht es um Diskrepanzen zwischen den eigentlichen Intentionen des Programms und seiner Spezifikation.

(Nicht mehr ganz) aktuelles Beispiel:

- Die Spezifikation verlangt, dass für ein Datum der Tag, der Monat und das Jahr abzuspeichern sind.
- Aber der Programmierer hat entschieden, für jedes Jahr nur die letzten beiden Ziffern abzuspeichern.
- Bis zum 31.12.1999 stimmten Spezifikation und Programm hundertprozentig überein...

Idealbild einer Spezifikation

- Ein übersichtlich und systematisch strukturiertes Schriftstück.
- Unmissverständliche, unzweideutige, nicht interpretierbare Formulierungen.
- Verständlich und zugleich exakt.
- Deckt jeden möglichen Fall, der prinzipiell auftreten kann, auch ab.

Problem:

In der Realität wäre wohl

- die Erstellung einer solchen ”idealen“ Spezifikation viel zu aufwendig und
- die resultierende Spezifikation zu umfangreich und komplex.

→ Liest kein Mensch!

Beispiel

- Eine saloppe Spezifikation für das UNIX–Programm ”`cal`“ würde besagen:

  Das Programm ”`cal`“ berechnet den Kalender für das angegebene Jahr und den angegebenen Monat.

- Eine exakte Spezifikation müsste noch konkret enthalten:

  ◊ Welcher Kalender eigentlich (gregorianisch, julianisch, ...).

  ◊ Der Bereich von Jahren, in dem das Programm korrekte Kalender berechnen können soll.

  ◊ Die Berücksichtigung von Angleichungen wie im September 1752 (siehe Man Page zu ”`cal`“).

  ◊ Die Regeln für die Berechnung von Schaltjahren.

  ◊ Die Behandlung fehlerhafter Benutzereingaben.

→ Eine ganze Menge schriftlicher Stoff allein für eine simple Komponente zur Kalenderberechnung.