Vorkurs Informatik

Programierung

FaRaFin

22. März 2019

Warum Programmierung?

Warum Programmierung?

- Um Computer Anweisungen zu geben.
- Um abstrakte Konzepte den PC verständlich zu machen.
- Um Algorithmen zu verständigen.

Erfahrungen

Was sind eure Erfahrungen mit Programmierung?

Etwas zur Geschichte

- Ada Lovelace (1815-1852) gilt als erste Programmierer*in
- Alan Turing (1912-1954) gilt als Erfinder des Computers
- C (1972) ist eine noch heute weit verbreitete Programmiersprache
- IBM Personal Computer (1987) gilt als erster Personal Computer
- Java (1996) ist die Programmiersprache die wir lernen und ist auch weitverbreitet.

Programmierkonstrukte

Variablen

- Eine Variable ist ein reservierter Speicherbereich.
- Alle Variablen haben einen festen Datentyp.
- Variablen können einen Datenwert zugewiesen bekommen.
- Beispiele für einfache Datentypen sind:
  Ganzzahlen, Fließkommazahlen, Wahrheitswerte und Zeichen.

Primitive Datentypen

- `boolean` für Wahrheitswerte
- `int` für Ganzzahlen
- `float` für Kommazahlen
- `char` für Zeichen
- `String` für Zeichenketten/Text

Programmierkonstrukte

- Programme bestehen aus Kombinationen folgender Grundkonstrukte:
  - Sequenz
  - Bedingte Ausführung / Fallunterscheidung
  - Wiederholung
- Daraus lässt sich *jedes* Programm bilden.

Notationen

- Verschiedene Notationen (Schreibformen) sind:
  - Umgangssprachlich
  - Pseudo-Code

Sequenz

- Nacheinander Ausführen mehrerer Anweisungen

  Führe `Anweisung 1` aus,
  danach `Anweisung 2`,
  führe letzte `Anweisung N` aus.

Sequenz

Pseudo-Code

```
1 Anweisung_1
2 Anweisung_2
3 Anweisung_N
```

Anweisung

- Anweisungen sind einzelne Befehle.
- Dazu zählen zum Beispiel:
  - Ausgaben auf die Konsole
  - Einzelne Rechen Operationen.
  - Funktionsaufrufe.

Anweisungs-Block

- Ein Anweisungs-Block ist eine Folge von einer oder mehreren Anweisungen.
- Er wird in Java durch Geschwungeneklammern

```
{
  Anweisung 1;
  Anweisung 2;
}
```

  gekennzeichnet.
- Dieser Block wird auch Scope genannt.
- Eine Anweisung kann auch ein Anweisungs-Block sein.

Fallunterscheidung: Einseitig

- Ausführung abhängig von einer Bedingung

Umgangssprachlich

Wenn `Bedingung B` gilt,
dann führe `Anweisung A` aus.

Fallunterscheidung: Einseitig

Pseudo-Code

```
1 if Bedingung B then
2   Anweisung A
3 endif
```

Bedingung

- Eine Bedingung ist ein Variable oder ein Funktionsaufruf der einen Variable zurück gibt die nur zwei Zustände haben kann. (boolean, Wahrheitswert)

TRUE oder FALSE.

- Dazu zählen zum Beispiel:
  - `Variable A` ist gerade. (A % 2 == 0)
  - `Text T` hat 7 Zeichen. (T.length() == 7)
  - `Wahrheitswert W` ist TRUE/Wahr.

Fallunterscheidung: Zweiseitig

- Ausführung ebenfalls abhängig von einer Bedingung
- Wenn Bedingung falsch wird andere Anweisung ausgeführt.

Umgangssprachlich

Wenn `Bedingung` *gilt,*
führe `Anweisung A` *aus,*
ansonsten `Anweisung B` aus.

Fallunterscheidung: Zweiseitig

Pseudo-Code

```
1 if Bedingung then
2   Anweisung_A
3   else
4     Anweisung_B
5 endif
```

Fallunterscheidung: Mehrseitig

- Einer von mehreren Fällen wird anhand eines Selektors ermittelt und die entsprechende Anweisung werden ausgeführt.
- „default“ ist für alle anderen Fälle, die nicht über den Selektor abgefragt werden
- Selektor ist eine Variable

Fallunterscheidung: Mehrseitig

Umgangssprachlich

Hat der `Selektor` *S den Wert* x_0*,*
führe `Anweisung` `für` x_0 *durch.*
Hat der `Selektor` *S den Wert* x_1*,*
führe `Anweisung` `für` x_1 *durch.*
Für alle anderen Fälle führe `Anweisung` `K`*.*

Fallunterscheidung: Mehrseitig

Pseudo-Code

```
1 if variable_1:
2    1:
3       Anweisung_1
4    2:
5       Anweisung_2
6    else
7       Anweisung_n
8 endif
```

Wiederholung

- Wiederholen von Anweisungen
  - Abhängig von einer Bedingung

Umgangssprachlich

Solange `Bedingung B` *gilt, wiederhole* `Anweisung A`.

while-Schleife

- Kopf gesteuerte Schleife.
- Überprüfung der Bedingung vor Beginn der Schleife.

while-Schleife

Pseudo-Code

```
1 while Bedingung_B do
2    Anweisung_1
3    Anweisung_2
4    ...
5    Anweisung_n
6 endwhile
```

do-while-Schleife

- Fuß gesteuerte Schleife.
- Überprüfung der Bedingung nach ersten Durchlauf.
- Wird mindestens einmal ausgeführt

do-while-Schleife

Pseudo-Code

```
1 repeat
2   Anweisungen_A
3 until Bedingung_B
```

for-Schleife

- Kurzschreibweise einer while-Schleife
- Häufiger Anwendungsfall: Für jeden Wert zwischen a und b

Umgangssprachlich

Führe die `Anweisung A` N-mal durch.

for-Schleife

Pseudo-Code

```
1 for i := Anfangswert to Endwert do
2   Anweisung_1
3 endfor
```

Kombination

- Die Konstrukte lassen sich beliebig oft verschachteln

Pseudo-Code

```
1 for i := 1 to 10 do
2   if i gerade then
3     gib i ist gerade aus
4     else
5       gib i ist ungerade aus
6   endif
7 endfor
```

- Dadurch lassen sich alle möglichen Algorithmen durch Programmierung darstellen.

Was ist Java?

- Objektorientierte Programmiersprache
- Seit 1993 von Sun entwickelt
- 2010 wurde die Firma Sun von Oracle übernommen.
- Vorteile von Java:
  - Plattformunabhängiger Programmcode
  - Einfache Programmierung durch Verzicht auf komplexere Konstrukte (z.B. Zeiger)
  - Trotzdem keine eingeschränkten Nutzungsmöglichkeiten
  - Umfangreich dokumentiert

Grundgerüst - Programmaufbau

- Programmbeispiel: *HelloWorld.java*

```
1 public class HelloWorld {
2   public static void main(String[] args) {
3     //Dieses Programm gibt "Hello World!" aus
          .
4     System.out.println("Hello World!");
5   }
6 }
```

- Gibt „Hello World!“ auf der Konsole aus.

Grundgerüst - öffentliche Klasse

```
1 public class HelloWorld {
```

- Jedes ausführbare Programm besteht aus mindestens einer öffentlichen Klasse (hier: *HelloWorld*).
- Klassenname = Dateiname
  - public class HelloWorld $\Leftrightarrow$ HelloWorld.java

Grundgerüst - Main-Methode

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

- Einsprungpunkt des Java-Interpreters
- Jedes ausführbare Programm hat genau eine main-Methode
- *args* beinhaltet die Kommandozeilenparameter

Grundgerüst - Schlüsselwörter

- Schlüsselwörter:
  - Von Java reservierte Wörter, die den Programmablauf steuern
  - Schlüsselwörter dürfen niemals als Variablen- oder Methodennamen verwendet werden
  - z.B.: *if*, *else*, *public*, *private*, *static*, *void* ...

Grundgerüst - Anweisungen

- Anweisungen:

```
4 System.out.println("Hello World!");
```

  - Anweisungen in Java werden mit „;“ abgeschlossen.
  - Diese Anweisung gibt „Hello World!“ auf der Konsole aus.

Grundgerüst - Kommentare

- Kommentare:

```
3 // Dieses Programm gibt "Hello World!"
      aus
```

  - Kommentare werden vom Compiler ignoriert
  - Einzeilige Kommentare werden mit // eingeleitet
  - Mehrzeilige Kommentare:

```
1 /* Dieses Programm gibt "Hello World!"
      aus */
```

Variablen und Datentypen

- Variable = anderer Name für Speicherbereich → ein Behälter für einen bestimmten Datenwert
- Der Inhalt einer Variable kann verändert werden.
- Variablen müssen deklariert werden!
  - Syntax:

```
1 typ variablenname;
```

  - z.B.:

```
1 int variable;
```

- Verwendung ohne vorherige Deklaration nicht möglich!

```
1 // int variable;
2 variable = 2; // Kompilierfehler
```

Grundlagen der Anweisungen

- Eine Anweisung ist ein einzelner Schritt, den der Computer ausführen kann.
- In Java ist das:
  - Variablendeklaration (evtl. mit Wertzuweisung)

```
1 int variable;
2 float variable2 = 13.5f;
```

  - eine Wertzuweisung

```
3 variable = 13;
```

  - ein Methodenaufruf

```
4 System.out.println("Text ausgeben");
```

Wertzuweisung einer Variablen

Eine Wertzuweisung einer Variablen kann bestehen aus:

- festem Wert

```
5 variable = 13;
```

- Methodenaufruf

```
6 variable = Math.pow(2,5);
```

- anderer Variablen

```
7 variable = variable2;
```

- Berechnung

```
8 variable = (variable2  + Math.pow(2,5)) * 13;
```

Die Berechnung erfolgt dabei nach Punkt-Vor-Strich-Rechnung, Klammerung ist möglich.

arithmetische Operationen

Folgende arithmetische Operationen sind möglich:

Operation	**Beispiel**	**Kommentar**
Addition	$a + b$	nur mit Zahlen
Subtraktion	$a - b$	nur mit Zahlen
Multiplikation	$a * b$	nur mit Zahlen
Division	$a \ / \ b$	ist Ganzzahldivision, falls a und b ganze Zahlen sind
Modulo	$a \ \% \ b$	berechnet den Rest der Division a/b

Strings

- Objekt der Klasse String
- wird zur Speicherung von Zeichenketten verwendet
- Erzeugung eines neuen Strings:
  - Literal zuweisen

```
1 String str = "Hallo";
```

Operationen auf Strings

- Konkatenation (Aneinanderreihen von Strings) mit +

```
1 String a = "Das ist " + "ein Test";
2 String b = a + " mit Strings";
```

Wahrheitswerte

- Ein Wahrheitswerte ist etwas, was entweder wahr oder falsch ist.
- In Java ist ein Wahrheitswerte ein Wert vom Datentyp „boolean“.
- Ein boolean-Wert kann erzeugt werden durch:
  - festen Wert

```
1 boolean b1 = true;
```

  - einen Vergleich

```
2 boolean b2 = variable == 2;
```

Vergleiche

- Folgende Vergleiche für primitive Datentypen sind möglich:

Vergleich	**Beispiel**	**Kommentar**
Gleichheit	$a == b$	a und b müssen vom gleichen Typ sein
Ungleichheit	$a != b$	a und b müssen vom gleichen Typ sein
Größer	$a > b$	a und b müssen Zahlen sein
Kleiner	$a < b$	a und b müssen Zahlen sein
Größergleich	$a >= b$	a und b müssen Zahlen sein
Kleinergleich	$a <= b$	a und b müssen Zahlen sein

Logische Verknüpfungen

- Logische Verknüpfung können nur zwischen boolean-Werten stattfinden.
- Verschachtelung und Klammerung ist möglich.
- Folgende Verknüpfungen sind möglich:

Verknüpfung	**Code**	**Bedingung**
Nicht	$!a$	falls a unwahr ist
Und	$a \&\& b$	falls a und b wahr sind
Oder	$a \|\| b$	falls a oder b wahr sind

if

- Einseitige Fallunterscheidung

```
1 if (Bedingung) {
2   Anweisungen;
3 }
```

- z.B.

```
1 if (x == 3) {
2   System.out.println("x ist 3");
3 }
```

```
1 boolean bedingung = true;
2 if (bedingung) {
3   System.out.println("bedingung ist wahr");
4 }
```

if-else

- Zweiseitige Fallunterscheidung:

```
1 if(Bedingung) {
2   Anweisungen;
3 } else {
4   andereAnweisungen;
5 }
```

- z.B.

```
1 if(x == 3) {
2   System.out.println("x ist 3");
3 } else {
4   System.out.println("x ist nicht 3");
5 }
```

switch-case

- Mehrseitige Fallunterscheidung:
- Eine Variable mit konstanten Werten vergleichen.

```
1 switch(variable) {
2 case KONSTANTER_WERT_1: Anweisungen;
3   break;
4 case KONSTANTER_WERT_2: Anweisungen;
5   break;
6 ...
7 default: Anweisungen;
8   break;
9 }
```

- Ausdruck ist numerisch, nicht logisch!

switch-case

```
1  int x = 1;
2  switch(x) {
3  case 0:
4    System.out.println("x ist 0");
5    break;
6  case 1:
7    System.out.println("x ist 1");
8    break;
9  default:
10   System.out.println("x ist was anderes");
11   break;
12 }
```

Verkürzte Operatoren

- Kürtzere schreibweisen für Variablen Operatoren.

```
1 int a = 1;
2 int b = 1;
3 a += 5; // a = a + 5;
4 b -= a; // b = b - a;
5 a *= b; // a = a * b;
6 b /= 5; // b = b / 5;
7
8 a++;    // a = a + 1;
9 b--;    // b = b - 1;
```

while-Schleife

- Solang Bedingung erfühlt ist tue Anweisung.

```
1 while (Bedingung) {
2   Anweisungen;
3 }
```

```
1 int n = 0;
2 while (n <= 10) {
3   System.out.println("2 hoch" + n + "ist"
4                      + Math.pow(2,n));
5   n = n+1;
6 }
```

do-while-Schleife

- do-while-:
- when die Anweisungen mindestens ein mal ausgeführt werden soll.

```
1 do {
2   Anweisungen;
3 } while (Bedingung);
```

```
1 int n = 0;
2 do {
3   System.out.println("2 hoch "+n+" ist "
4                      + Math.pow(2,n));
5   n=n+1;
6 } while (n <= 10);
```

for-Schleife

- for-Schleifen
- When vorher die Anzahl der durchläufe bekannt ist.

```
1 for(Initialisierung; Bedingung; äVernderung) {
2   Anweisungen;
3 }
```

```
1 for(int n=0; n<=10; n=n+1) {
2   System.out.println("2 hoch "+n+" ist "
3                      + Math.pow(2,n));
4 }
```

for- als while-Schleife

- Jede for-Schleife lasst sich auch als while-Schleife ausdrücken.
- also:

```
1 for(Initialisierung; Bedingung; äVernderung)
      {
2   Anweisungen;
3 }
```

- lässt sich ausdrücken als:

```
1 Initialisierung;
2 while(Bedingung) {
3   Anweisungen;ä
4   Vernderung;
5 }
```

Achtung vor Endlosschleifen

- Achtung: Bei Schleifen muss man sicherstellen, dass die Abbruchbedingung erreicht wird!
- z.B.:

```
1 int n=0;
2 while(n <= 10) {
3   n=n-1;
4 }
```

- Da n gegen $-\infty$ läuft wird die Abbruchbedingung n $>$ 10 nie erreicht $\rightarrow$ Endlosschleife.
- Also immer Iterations-Schritt und Abbruchbedingung überprüfen.

Nutzereingabe über Scanner

- Um Nutzereingaben von Konsole machen zu können, verwenden wir ein Objekt der Klasse Scanner.
- Für die Nutzung der Klasse Scanner müssen wir die Systemquelle java.util.Scanner einbinden.
- Die Konsole können wir über System.in ansprechen.
- Die Dokumentation der Klasse Scanner findet ihr hier: `https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`

Methoden der Klasse Scanner

```
1  Scanner scn = new Scanner(System.in);
2  String string1 = scn.next();
3  //liest die Eingabe bis zum ächsten Leerzeichen
4
5  String string2 = scn.nextLine();
6  //liest die Eingabe bis zum ächsten Enter
7
8  int a = scn.nextInt(); // Integer einlesen
9
10 long b = scn.nextLong(); // Long einlesen
11
12 double c = scn.nextDouble(); //Double einlesen
13
14 scn.close(); //Freigabe des Input Streams
```

Beispiel 1: Einlesen von Strings

```
1  import java.util.Scanner;
2
3  public class ScannerDemo {
4
5    public static void main(String[] args) {
6      Scanner scn = new Scanner(System.in);
7      System.out.println("Gib einen String ein");
8      String text = scn.nextLine();
9      System.out.println("Gib noch einen String
           ein");
10     String text2 = scn.nextLine();
11
12     scn.close(); //Freigabe des Input Streams
13   }
14 }
```

Beispiel 2: Einlesen von ganzen Zahlen

```
1  public static void main(String[] args) {
2    Scanner scn = new Scanner(System.in);
3    //mehrere integer in einer Zeile eingeben.
4    System.out.println("gib 2 ganze Zahlen " +
5    "auf einmal ein");
6    int a = scn.nextInt();
7    int b = scn.nextInt();
8
9    scn.close(); //Freigabe des Input Streams
10 }
```

- Achtung: Dieser Code überprüft nicht ob der Nutzer wirklich ganze Zahlen eingibt!

Datentypen Größen

Typ	Größe (bit)	Bereich
boolean	1	{true; false}
byte	8	−128 bis 127
short	16	−32.768 bis 32.767
int	32	−2.147.483.648 bis 2.147.483.647
long	64	−9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
char	16	'\u0000' bis '\uFFFF'
float	32	$-3,40282347 \times 10^{38}$ bis $3,40282347 \times 10^{38}$
double	64	$-1,79769313486231570 \times 10^{308}$ bis $1,79769313486231570 \times 10^{308}$

Über- oder Unterlauf

- Über- oder Unterlauf
  - Variablen haben einen festen Wertebereich.
  - Wird dieser über- oder unterschritten, kann der neue Wert nicht mehr korrekt abgespeichert werden.
    $\rightarrow$ Über- bzw. Unterlauf.
  - Stattdessen wird wieder am anderen Ende des Wertebereichs fortgesetzt.
  - Java erkennt dabei keinen Fehler.
  - Beispiel

```
1 int x = 2147483647; //groesstmoeglicher
      Wert
2 x = x + 1; //Überlauf x = -2147483648
```

Casting

- wandelt Basisdatentypen ineinander um, z.B. double zu int.
- funktioniert nur mit Basisdatentypen → keine Strings, Objekte oder komplexe Datentypen
- Beispiel int zu float:

```
1 int intWert1 = 5;
2 int intWert2 = 4;
3 float dWert;
4 dWert = (float) intWert1 / (float) intWert2;
5 //Gleitkommerzahl statt Ganzzahl Division
```

- Beispiel float zu int:

```
1 float dWert = 3.893926;
2 int iWert;
3 iWert = (int) dWert;   //iWert = 3;
```

Beispiel: Erstellen von Zufallszahlen

- Das folgende Programm erstellt ganzzahlige Zufallszahlen zwischen 50 und 100.
- Math.random(); erzeugt eine Zufallszahl x, wobei 0 <= x < 1 ist.

```
1 int kleinsteZahl = 50;
2 int öanzahlMglicherZahlen = 51;
3 int zufall = (int)(Math.random() *ö
4   anzahlMglicherZahlen) + kleinsteZahl;
```

Arrays

- Ein Array besteht aus mehreren Variablen des selben Typs, z.B.

int	int	int	int	int	int	int

- Die Größe eines Feldes wird bei der Initialisierung festgelegt und ist fix. Im obigen Beispiel beträgt die Länge sieben.

Erzeugung von Arrays

- Erzeugung
  - Syntax

```
1 typ[] name = new typ[Anzahl_Elemente];
```

  - Z.B. ein Feld mit zwölf double-Werten:

```
1 double[] feld = new double[12];
```

  - Alternativ mit Initialisierungsliste:

```
1 int[] feld = {1,1,2,3,5,8,13};
```

Zugriff auf Arrays

- Zugriff:
  Die Elemente eines Arrays sind nummeriert, wobei man mit 0 zu zählen beginnt.

```
1 int[] zahlen = new int[6];
```

```
1 zahlen[0] = 13;
2 zahlen[1] = 4;
3 zahlen[5] = 42;
```

- Array-Größe:

```
1 int a = zahlen.length; // a = 6
```

Iterativer zugriff auf Arrays

- Zugriff:
  Die Elemente eines Arrays sind nummeriert, wobei man mit 0 zu zählen beginnt.

```
1 Scanner scn = new Scanner( System.in);
2 int[] zahlen = new int[6];
3 for( int i = 0; i < zahlen.lenght; i++) {
4   zahlen[i] = scn.nextInt();
5 }
6 scn.close();
```

foreach Schleife

- Folgende foreach-Schleife gibt alle Elemente des Arrays *myArray* aus:

```
1 int[] myArray = { 3, 7, 13, 44, 54 };
2 for (int element : myArray) {
3   System.out.println(element);
4 }
```

- **Dabei können keine Zuweisungen gemacht werden!**
- zum Vergleich die Ausgabe mit einer normalen for-Schleife:

```
1 int[] myArray = { 3, 7, 13, 44, 54 };
2 for(int i=0;i<myArray.length;i++){
3   System.out.println(myArray[i]);
4 }
```

break-Statment

- Durch ein `break` kann man **eine** Schleife vorzeitig verlassen, ohne den Rest der Anweisungen im Schleifenkörper auszuführen. Das Programm wird hinter der Schleife fortgesetzt.

```
1 for (int i=0; i<500; i++) {
2   if (i==20){
3     break; //äverlsst die Schleife, wenn i
          ==20
4   }
5   System.out.println(i);
6 } // Ende der for-Schleife
7 //Nach der Schleife geht es normal weiter
```

continue-Statment

- Durch `continue` wird die Ausführung des aktuellen Schleifendurchlaufs übersprungen und mit dem nächsten Schleifendurchlauf begonnen.
- Beispiel: Folgender Code gibt die Zahlen von 1 bis 100 ohne die Zahl 42 aus.

```
1 for (int i=1; i<=100; i++) {
2   if (i==42){
3     continue;
4   }
5   System.out.println(i);
6 }
```

Methoden

- Kleine Aufgaben und Algorithmen, die man oft benötigt, werden in seperaten Funktionen (in Java „Methoden“ genannt) geschrieben.
- z.B. Potenzen berechnen oder Strings vergleichen

Methoden

- Analogie zur Mathematik:
  - $y = f(x)$
    x $\rightarrow$ Parameter
    y $\rightarrow$ Rückgabewert
    f $\rightarrow$ Methode
  - $f(x) = x^2$

```
1 public static double f(double x) {
2   return x*x;
3 }
```

Methoden

- Prinzipieller Aufbau:

```
1 Sichtbarkeit [static] Rueckgabetyp Name(
      Parameter) {
2   //Quelltext
3   return üRckgabe;
4 }
```

- *Sichtbarkeit* kann entweder **private**, **public** oder **protected** sein.
- `static` kann gesetzt werden, muss es aber nicht.
  Regel: in static-Methoden (wie main) können direkt nur static-Methoden aufgerufen werden.

Methoden

- *Rückgabetyp* kann Klasse, Array, primitiver Datentyp oder `void` sein. `void` bedeutet „keine Rückgabe“.
- *Name* ist die Bezeichnung der Methode.
- Eine Funktion kann beliebig viele *Parameter* haben. Sie sind innerhalb der Funktion bereits initalisierte Variablen.
- Durch **return** wird die Methode beendet. Zugleich muss die Variable angegeben werden, dessen Wert zurückgegeben wird. Bei **void** muss nichts angegeben werden.

Konventionen für Methodennamen

- Methodennamen sollten
  - beschreiben was die Methode macht
  - mit klein geschriebenen Buchstaben beginnen
  - in CamelCase sein (erste Buchstabe klein, weitere Worte groß schreiben)
- sinnvolle Zeichen
  - a-z, A-Z, Zahlen
  - möglichst keine Sonderzeichen, Umlaute und ß verwenden
- Beispiele für gute Methodennamen:
  - toString(); getNextInputValue(); calculateMean(); isEqual();

Beispiel 1

- eine Methode die überprüft ob eine Zahl gerade ist:

```
1 public static boolean isEven(int number) {
2   if (number % 2 == 0) {
3     return true;
4   } else {
5     return false;
6   }
7 }
```

- kürzer geht auch:

```
1 public static boolean isEven(int number) {
2   return number % 2 == 0;
3 }
```

Beispiel 2

- Potenzfunktion für positive Exponenten

```
1 public static int potenzBerechnen(int base,
2                                   int exponent) {
3   int potenz=1;
4   for(int i = 1; i <= exponent; i++){
5     potenz= potenz * basis;
6   }
7   return potenz;
8 }
```

Aufrufen

- Aufruf der Methode

```
1 public class Methoden {
2   public static void main(String[] args) {
3     int ergebnis = potenzBerechnen( 2, 10);
4     System.out.println("2 hoch 10 ist gleich"
          + ergebnis);
5   }
6
7   public static int potenzBerechnen(int base,
8                                     int exponent) {...}
9 }
```

Methoden überladen - Beispiel 1

Beim überladen einer Methode wird der selbe Methodenname jedoch mehr oder weniger Parameter verwendet.

```
1  public static void main(String[] args) {
2    int x1 = 3;
3    int x2 = 1;
4    ausgabe(x1);      //springt in Zeile 8
5    ausgabe(x1,x2);   //springt in Zeile 12
6  }
7
8  private static void ausgabe(int x1) {
9    System.out.println("x1 = " + x1);
10 }
11
12 private static void ausgabe(int x1, int x2) {
13   System.out.println("x_n = " + x1 + "\t" + x2);
14 }
```

Was ist ein Scope?

- Ein Scope ist der Bereich in der eine Variable sichtbar ist.
- Er wird gekennzeichnet durch:

```
1 {
2    Variable;
3 }
```

- Für jedes Scope würd auf den Stack ein neuer Speicherbereich Reserviert.
- Jede Variable ist nur in ihren und den untergeordneten Scopes sichtbar.

- Jeder Name kann in einen Scope und untergeordneten nur ein mal benutzt werden.
- Wen das Scope geschlossen wird werden alle Variablen gelöscht. (Speichrbereich wird vom Stack entfernt)
- Ein Scope wird als eine Anweisungs-Sequenz ausgeführt

Beispiele für Scopes

- Methoden

```
1 public static void main () {
2                                \\ ^ this is a Scope
3 } \\ <-- it ends here
```

- Schleifen

```
1 while( true) { \\ <-- this is also a Scope
2
3 } \\ <-- this one ends here
```

Scope einer for-Schleife

```
1  public static void main( String[] args) {
2    for( int i = 0; i < 10; i++) {
3      \\ In diesen Scope gibt es die Variable i
4    }
5
6    \\ Hier wiederum nicht mehr
7    \\ Somit kann in einen neuen Scope
8    \\  die Variable i wieder verwendet werden.
9
10   for( int i = 0; i < 10; i++) {
11     \\ Hier gibt es eine neue Variable i
12   }
13
14 }
```

Code Conventions...

... verbessern die Lesbarkeit des Codes und erlauben damit eine schnellere und bessere Einarbeitung in den Code.

Zeilenumbrüche

- keine Zeile länger als 80 Zeichen
- Zeilenumbrüche:
  1. nach Kommata
  2. vor Operatoren
  3. höherwertige Klammerpaare bevorzugen
  4. neue Zeile auf die Höhe des vorhergehenden Ausdrucks ausrichten
  5. sollte Punkt 4 zu unleserlichem Code führen, die nachfolgende Zeile um 1 Tab einrücken

Zeilenumbrüche (Bsp. 1/3)

- Beispiel zu 1. (Kommata) und 4. (gleiche Höhe)

```
1 private static void aMethod(int aParameter,
2                               int nextParameter
3                              ) {
4
5 }
```

- Beispiel zu 2. (vor Operatoren) und 4. (gleiche Höhe)

```
1 int sumOfFollowingNumbers = number1 + number2
2                           + number3 + number4
                              ;
```

Zeilenumbrüche (Bsp. 2/3)

- Beispiel zu 3. (höherwertige Klammerpaare)

```
1 float aFloatingNumber = num1 / (num2
2                               - (num3 + num4));
```

folgendermaßen **NICHT**:

```
1 float aFloatingNumber = num1 / (num2 - (num3
2 + num4));
```

Zeilenumbrüche (Bsp. 3/3)

- Beispiel zu 5. (unleserlicher Code)

```
1 if(isThisTrue && isThisTrueToo
2                || ThisOneIsTrue) {
3                makingThingsHappen();
4 }
```

folgendermaßen **NICHT**:

```
1 if(isThisTrue && isThisTrueToo
2 || ThisOneIsTrue) {
3 makingThingsHappen();
4 }
```

Deklarationen

Eine Deklaration führt eine Variable, Methode oder Klasse im Quellcode ein. Dies ist notwendig, damit der Compiler weiß, dass eine Variable mit einem bestimmten Typ im Codeblock benutzt wird.

1. eine Deklaration pro Zeile
2. wenn möglich Variable bei ihrer Deklaration initialisieren
3. Deklarationen am Anfang eines Codeblocks
4. keine verdeckten Deklarationen
5. weitere Hinweise für die Deklaration von Methoden

Deklarationen (Bsp. 1/3)

Beispiel zu 1. (Eine Deklaration pro Zeile) und 2. (gleich initialisieren):

```
1 int aNumber = 0; //schoen kommentieren
2 String aText = "Some text..."; //auch mit Kommentar
```

folgendermaßen **NICHT**:

```
1 int aNumber, nextNumber; //kann man nicht öschn
2   //kommentieren ...
3   //ausserdem sind die Variablen nicht
        initialisiert
```

Deklarationen (Bsp. 2/3)

Beispiel zu:

- 2. (gleich initialisieren)
- 3. (Deklaration am Anfang)
- 4. (keine verdeckten Deklarationen)

```
1 void aMethode() {
2     int counter1 = 0;
3     if(somethingHappens) {
4         int counter2 = 0;
5       // ... etwas Code
6     }
7     //... und noch mehr Code
8 }
```

Hinweis: `counter1` und `counter2` könnten auch gleich benannt werden, dies würde aber Hinweis 4 verletzen.

Deklarationen (Bsp. 3/3)

Beispiel zu 5. (weitere Hinweise)

```
1  int aMethode(int firstValue, int secondValue) {
2    // zwischen Methodenname und '(' kein
         Leerzeichen
3    // '{' gleich nach der Parameterliste durch
4    //      ein Leerzeichen getrennt
5    // '}' kommt in eine extra Zeile
6  }
7
8  //eine Zeile zwischen den Methoden frei lassen
9
10 void emptyMethode() {} //leere Methode
11   //bei leeren Methoden die '{' '}' gleich
12   //hintereinander schreiben
```

Java Code Convetions

Die *Java Code Conventions* beschreiben die Verwendung und Schreibweise weiterer Ausdrücke. Im Folgenden sollen daher nur die Wichtigsten genannt werden.

Java Code Convetions

- einfache Anweisungen

```
1 aNumber += aNumber;
2 nextNumber += aNumber;
```

Wie bei Deklarationen sollte auch bei Anweisungen darauf geachtet werden, dass jeweils nur eine pro Zeile steht.

- `if`-Statement

```
1 if(somethingTrue) {
2   doThat();
3 }
```

Auch bei einzeiligen `if`-Blöcken geschweifte Klammern nutzen!

Java Code Convetions

Namenskonventionen

- Klassenbezeichner sollten Substantive sein
  (`MyObject, Date, Storage`)
- Methodenbezeichner sollten Verben sein
  (`run, runFast, getObject, isObjectBlue`)
- Variablenbezeichner beginnen klein und beschreiben den gespeicherten Wert
  (`tableWidth, position, myObject, date, storage`)
- Konstant werden groß geschrieben und Wörter durch den Unterstrich getrennt
  (`MAX_WIDTH, MAX_AMOUNT_OF_OBJECTS, STD_POSITION, BLUE`)

Grundsatz

„Guter Quelltext braucht keine Kommentare!“

(Das heißt **nicht**, dass man keine benutzen darf bzw. sollte.)

Schlechter Code

```
1   private void dolt(int a) {int[] b={4,11}; if (a==0)
2   {System.out.println("Keine Verbindung"); c.set(b);c.reload();}else}
3   if(a==1){System.out.println("Verbindung
4   wird hergestellt"); c.set(b);c.reload();}else if(a==2)
5   {System.out.println("Verbindung hergestellt");
6   int[] b={1,2,3,5,6,7,8,9,15,16,14,13};
7   c.set(b); c.beep(); c.reload();} else
8   { System.out.println ("INTERNER FEHLER! "
9   + "STATUS nicht erkennbar");c.set(b);c.reload();}
10  }
```

- Was soll der Code machen?
- Wer findet den Fehler?

Guter Code

Schon besser! Aber noch nicht wirklich clean.

```
1   private void dolt(int a) {
2   int[] b = {4,11};
3   if (a == 0) {
4   System.out.println("Keine Verbindung");
5   c.set(b);
6   c.reload();
7   }else if(a == 1) {
8   System.out.println("Verbindung wird hergestellt");
9   c.set(b);
10  c.reload();
11  }else if(a == 2) {
12  System.out.println("Verbindung hergestellt");
13  int[] b={1, 2, 3, 5, 6, 7, 8, 9, 15, 16, 14, 13};
14  c.set(b);
15  c.beep();
16  c.reload();
17  } else {
18  System.out.println ("INTERNER FEHLER! "
19  + "STATUS nicht erkennbar");
20  c.set(b);
21  c.reload();
22  }
23  }
```

Vollkommen Ausgearbeiteter Code

```
1  private final static int NOT_CONNECTED = 0;
2  private final static int TRY_TO_CONNECT = 1;
3  private final static int CONNECTED = 2;
4
5  /**
6  * Gibt den Status der Modem-Verbindung
7  * auf der Konsole und dem Modem-Display aus.
8  * (aus üPlatzgrnden fehlt die Definition der Display funktionen.)
9  * @param status - Status der Modem-Verbindung
10 */
11 private void showConnectionStatus(int status) {
12   if (status == NOT_CONNECTED) {
13     System.out.println("Keine Verbindung");
14     setDisplayOFF();
15   }else if(status == TRY_TO_CONNECT) {
16     System.out.println("Verbindung wird hergestellt");
17     setDisplayOFF();
18   }else if(status == CONNECTED) {
19     System.out.println("Verbindung hergestellt");
20     setDisplayON();
21   } else {
22     System.out.println("INTERNER FEHLER! "
23                        + "STATUS nicht erkennbar");
24     setDisplayOFF();
25   }
26 }
```

Abschließende Bemerkungen

Kommentare sollten dafür genutzt werden, die Aufgabe einer Methode oder eines Codeabschnittes zu erläutern, nicht aber dessen Umsetzung aufzuzeigen (ausgenommen sind komplizierte bzw. sehr komplexe Codeabschnitte). Dies erleichtert das *ungestörte* Einarbeiten in den Code und das Auffinden eventueller Fehler oder Verbesserung.

Definition Rekursion

Als Rekursion, von lateinisch *recurrere* = *zurücklaufen*, bezeichnet man den Aufruf oder die Definition einer Funktion oder Struktur durch sich selbst.

Quelle: http://www.uni-koeln.de/rrzk/kurse/unterlagen/java/spinfo/kap18/rekursiv.htm

Rekursion

Bei der Rekursion führt man ein Problem auf Teilprobleme gleicher Art zurück, die leichter zu lösen sind als das Ausgangsproblem. Dieses wird solange weiter geführt, bis ein Problem entstanden ist, dass trivial ist und mit so gut wie keinem Aufwand gelöst werden kann.

Begriffe

- Rekursionsabstieg: Funktion ruft sich selbst auf.
- Rekursionsabbruch: Teilproblem ist gelöst und Teilergebnis kann zurück gegeben werden.
- Rekursionsaufstieg: Teilergebnis nutzen um größeres Problem zu lösen.
- **Wird die Abbruchbedingung nicht erreicht entsteht ein Stackoverflow**

Anzahl der Reihen

Max sitzt in der letzten Reihe eines großen Höhrsaals.
Problem: *Max will wissen wie viele Reihen der Höhrsaal hat.*

- Weil er zu Faul ist auf zu stehen um aller Reihen nach einander zu zählen, fragt er seinen Vordermann wie vielle Reihen vor ihn sind.
- Dieser wiederum Fragt seinen Vordermann und so weiter.
- Bis die Person in der ersten Reihe zurück gib das vor ihr keine Reihen mehr sind.
- Der nächste Rechnet eins drauf und gibt dies weiter.

Bis es bei Max ist und er weiß wie viele Reihen der Höhrsaal hat.

Beispiel: Datei finden

- Will man eine Datei auf seinen Computer finden so muss man zunächst in seinen Home Verzeichnis danach suchen.
- Findet man die Datei dort nicht so muss man alle Ordner in Home Verzeichnis auf gleiche Weise durch suchen.
- Dies tue man so lang bis man die Datei gefunden hat oder man alle unter Ordner durch sucht hat.

Datei finden als Rekursion

- Problem Datei auf den Computer finden zurück geführt auf Problem in Ordner finden.
- Gleiches Lösungsschema für Unterordner anwenden.
- Abbruch bei wenn Datei gefunden oder alles durchsucht.

Fibonacci Reihe

$$fib(x \leq 1) = 1$$

$$fib(x) = fib(x-1) + fib(x-2)$$

Rekursionsabbruch Aus der Definition geht hervor das fib von jeder Zahl kleiner der gleich als 1, 1 ist. Somit ist das der Rekursionsabbruch.

Rekursionsschritt Die n-te Fibonacci Zahl ist die Summer der beiden vorherigen. Somit ist das unser Rekursionsschritt.

fib(x) - Ablauf

$fib(3)$

$3 > 1$

- $fib(3) = fib(2) + fib(1)$ **Rekursiver Abstieg**
  - $fib(2)$
  - $2 > 1$
  - $fib(2) = fib(1) + fib(0)$ **Rekursiver Abstieg**
    - $fib(1)$
    - $1 = 1$
    - $fib(1) = 1$ **Rekursionsabbruch**
    - $fib(0)$
    - $0 < 1$
    - $fib(0) = 1$ **Rekursionsabbruch**
  - $fib(2) = 2$ **Rekursionsabbruch**
- $fib(3) = 3$ **Rekursionsabbruch**

Addition zweier Zahlen

$plus(x, y) = x + y$ mit $x, y \in \mathbb{N}$ Aufbau

Rekursionsabbruch Wenn einer der beiden Werte 0 ist, kann der andere sofort zurückgegeben werden, da eine Zahl mit 0 addiert immer sich selbst ergibt.
Vereinfachung: Rekursionsabbruch bei $y = 0$.
- **plus(x,0) = x**

Rekursiver Abstieg Wenn $y \neq 0$ ist, dann kann man die Summe wie folgt bilden:
- **plus(x,y) = plus(x+1, y-1)**

(z.B.: $plus(3, 3) = plus(4, 2)$)

plus(x,y) - Ablauf

$plus(3, 2)$

$2 \neq 0$

- $plus(3, 2) = plus(4, 1)$ **Rekursiver Abstieg**

  $plus(4, 1)$

  $1 \neq 0$

  - $plus(4, 1) = plus(5, 0)$ **Rekursiver Abstieg**

    $plus(5, 0)$

    $0 = 0$

    - $plus(5, 0) = 5$ **Rekursionsabbruch**

plus(x,y) - Implementierung in Java

```
1 public static int plus(int x, int y) {
2    if (y = = 0) {
3       return x;
4    } else {
5       return plus(x+1, y-1);
6    }
7 }
```

Hilfsfunktionen

- In manchen Fällen ist die Anzahl der Parameter für ne Rekursion zu wenig
- In den Fall wird eine Hilfsfunktion benötigt

Das kann so aussehen:

```
1 public static int mult(int x, int y) {
2   return multHelp( 0, x, y);
3 }
4
5 private static int multHelp(int x, int toAdd,
      int times) {
6   // The Code for the actual Rekursion
7 }
```

Rekursion als Iteration

- Jede rekursive Funktion ist auch durch eine iterative, also aus Schleifen und Anweisungen aufgebaute, Funktion darstellbar.
- Dabei hat die Rekursion den Vorteil, wenn man sie mal verstanden hat, große Probleme in kleine überschaubare Teile auf zu teilen.
- Jedoch ist sie auch in gegen Satz zum iterativen Ansatz ein Stück langsamer.

Grundlagen (1/2)

- Programme entwickeln ist wie das Schreiben eines Rezeptes, die benötigten Zutaten (Variablen) müssen angegeben werden und auch in welcher Reihenfolge diese bearbeitet werden (Methodenaufrufe etc.).
- Der Programmierer schwingt also Zettel und Stift (alternativ Tasten) und reicht das Ergebnis an den Koch weiter.
- Der Koch ist jedoch Italiener und versteht nicht einmal ansatzweise, was der Programmierer will ...

Grundlagen (2/2)

- Folglich muss das Rezept in eine Form gebracht werden, die der Koch ausführen kann.
- Wir sprechen in der Programmiersprache Java, der Kochcomputer versteht nur Maschinencode (bzw. ByteCode).
- Compiler nehmen uns die Übersetzung einer Sprache in eine andere ab. Netterweise findet er auch gleich Grammatikfehler und vereinfacht umständliche Formulierungen, gemeinerweise bleibt es jedoch dem Programmierer überlassen, seine Fehler zu beheben.

Javaprogramme kompilieren (1/2)

- Javaprogramme werden nicht direkt von der Hardware ausgeführt, sondern von einer Software-Abstraktionsschicht, die unter anderem die Platformunabhängigkeit sicherstellt.
- **Java Virtual Machine (JVM) / Java Runtime Environment (JRE)**
- Das Format für ausführbare Programme der JVM ist ein Zwischending aus Maschinenprogramm und Programmiersprache und kann nur von dieser interpretiert werden (dies z.B. unter Windows, Linux, Mac OS X, ...).

- Für die Programmierung dient das **Java Development Kit (JDK)**

`http://java.sun.com/javase/downloads/`

Javaprogramme kompilieren (2/2)

- Nachdem wir ein kleines Programm im Editor geschrieben haben, wollen wir dieses laufen lassen. Dazu rufen wir auf der Konsole (im Projektverzeichnis) folgenden Befehl auf, um eine einzige Datei zu übersetzen:

  $ javac HelloWorld.java

- Ausgeführt wird das Resultat mit:

  $ java HelloWorld

- Zu beachten ist, dass **javac** zwar eine *.class*-Datei erstellt, die Endung darf beim Aufruf von **java** jedoch nicht angegeben werden.

Fehlerarten

- Syntaktische Fehler
- Laufzeitfehler
- Logische Fehler

Syntaktische Fehler

- auch grammatikalische Fehler genannt
- verhindert die erfolgreiche Compilierung des betreffenden Programms
- Beispiele:
  - falsch geschriebene Objekte und Methoden (z.B. Groß- und Kleinschreibung)
  - falsch gesetzte Klammern
  - fehlerhafte Semikolons
  - falsch importierte Packages
- Syntaktische Fehler stellen im Allg. kein Problem dar, sie sind leicht auffind- und behebbar.

Laufzeitfehler(1/2)

- treten erst beim Ausführen des Programms auf (zur Laufzeit)
- Beispiele:
  - fehlende, aber benötigte Startparameter (z.B. Eingabedatei)
  - Lesezugriff auf nicht initialisierte Variable/Objekt
  - Zugriff auf nicht vorhandene Speicherstellen
  - illegale Typumwandlung
  - Referenzierung durch Nullpointer
  - ungültige arithmetische Operation
  - nicht erreichbarer Code

Laufzeitfehler(2/2)

- Bei Laufzeitfehlern werden von der Java-Runtime Ausnahmen (Exception) geworfen, z.B. :
  - **NullPointerException**
    beim Versuch auf einen Verweis zuzugreifen, der auf kein Objekt zeigt
  - **ArrayOutOfBoundException**
    bei ungültigen Feldzugriff (Index zu groß oder zu klein)
  - **BufferOverflowException**
    wenn das Pufferlimit erreicht ist (Endlosrekursion)
  - **ArithmeticException**
    Division durch 0 oder Wurzel einer negativen Zahl, ...
- Sollte keine Ausnahme geworfen werden, so terminiert das Programm nie, es hängt in einer Endlosschleife fest.

Logische Fehler(1/5)

- Logische Fehler sind falsche Ergebnisse oder unerwartetes Programmverhalten bei korrektem Code.
- Lösungsmöglichkeiten:
  - Herausfinden, wann der Fehler bzw. für welche Testdaten der Fehler auftritt.
  - Das Programm durchlesen bzw. überfliegen und nachsehen ob ein Zahlendreher, eine falsche Formel oder ein einfacher Denkfehler vorliegt.
- Schlagen die oben genannten Varianten fehl, so geht es an das eigentliche Debuggen.

Logische Fehler(2/5)

- Ein Beispiel:
  Implementieren Sie eine Methode swap(int a, int b), die die beiden Werte a und b miteinander vertauscht.
- Erster Ansatz:

```
1   public void swap (int a, int b) {
2   int temp = a;
3   a = b;
4   b = temp;
5   }
```

Logische Fehler(3/5)

- Beim Testen der Funktion wird festgestellt, dass die Werte außerhalb der *swap*-Methode nicht vertauscht sind.
- Die *swap*-Methode vertauscht nur die Werte miteinander, speichert diese aber nicht in den übergebenen Variablen ab.
- Wenn man dementsprechend außerhalb der *swap*-Methode auf die Originalwerte zugreift, dann erhält man die unvertauschten Werte.

- **Java übergibt den Wert eines Objektes als Funktionsparameter (auch „copy by reference“).**

Logische Fehler(4/5)

- **Übergabe des Wertes**: Der Parameterliste wird eine Kopie der Werte der Argumente übergeben.
- **Übergabe durch Referenz**: Die Speicheradresse des Wertes des Argumentes wird übergeben. Wird der Wert durch die aufgerufene Routine (Funktion) geändert, bleibt die Änderung des Wertes auch nach Verlassen der Routine bestehen.

- Beide Varianten sind wichtig!

Logische Fehler(5/5)

- eine Lösung des swap-Problems:
- Die *swap*-Methode muss mit einem Datentyp realisiert werden, der die Übergabe durch Referenz umsetzt.

```
1   public void goodSwap (int[] feld) {
2   int temp = feld[0];
3   feld[0] = feld[1];
4   feld[1] = temp;
5   }
```

- Diese Lösung benutzt ein Feld (Array).

Historische Programmfehler (1/2)

- **1962** führte ein fehlender Bindestrich in einem Fortran-Programm zum Verlust der Venus-Sonde Mariner 1, welche über 80 Millionen US-Dollar gekostet hat.
- **1985-1987** gab es mehrere Unfälle mit dem medizinischen Bestrahlungsgerät Therac-25. Infolge einer Überdosis, die durch fehlerhafte Programmierung und fehlender Sicherheitsmaßnahmen verursacht wurde, mussten Organe entfernt werden, drei Patienten verstarben.

Historische Programmfehler (2/2)

- **1999** verpasste die NASA-Sonde Climate Orbiter den Landeanflug auf den Mars, weil die Programmierer das falsche Maßsystem verwendeten, Yard statt Meter. Die NASA verlor dadurch die mehrere hundert Millionen Dollar teure Sonde.

- **2002** schaltete sich Siemens Mobiltelefon im April bei Aufruf der Kalenderfunktion ab. Bekannt ist der Fehler auch als Aprilbug. Als Folge könnte angesehen werden, dass der Mobilfonhersteller von dem Nutzen eines benutzergesteuerten Softwareupdates überzeugt wurde.

Debuggen

- Bei der Fehlersuche mit einem Debugger spricht man auch von Debuggen. Der Wortbestandteil Bug, für „Programmierfehler“ wurde von der Computerpionierin Grace Hopper geprägt. Mit Bugfix wird die Behebung eines Programmfehlers bezeichnet.
- Dazu gib es verschiedene Möglichkeiten:
  - System.out.println();
  - Unterbrechungspunkte und schrittweises Durchlaufen

Debuggen durch Unterbrechungspunkte

- Ein Unterbrechungspunkt (Breakpoint) ist eine vom Nutzer bestimmte Stelle, an der das Programm die Ausführung unterbricht bzw. pausiert.
- Somit hat der Programmierer die Möglichkeiten:
  - Werte verschiedener Variablen zu beobachten
  - Programm schrittweise abzuarbeiten bzw. Schritte zu überspringen
  - Bedingungen testen

Objektorientierung - eine kurze Definition

- auf dem Konzept der Objektorientierung basierendes Programmierparadigma
- Grundidee: Softwarestruktur an Grundstruktur der Wirklichkeit ausrichten
- d.h. Klassen als „Bauplan“, Objekte als fertiges „Produkt“

Allgemeines Prinzip

- Klassen bilden den abstrakten Bauplan eines Objekts (z.B. Blaupause eines Autos)
- Objekte sind die Ausprägung/Instanz dieses Bauplans (z.B. das fertige Auto)
- in Java erstellt man eine Instanz z.B. so:
  String myString = new String();

Eigenschaften und Methoden

- Ein Objekt hat bestimmte Eigenschaften (Attribute) und Methoden.
- z.B ein Auto
  - hat eine Farbe, eine Anzahl von Rädern und eine Anzahl von Türen = Attribute
  - und kann außerdem fahren, blinken und einiges mehr (Objektmethoden)
- Dies ist das Grundkonzept eines Autos, aber noch ist es kein bestimmtes Auto!

Geheimhaltungsprinzip

- Attribute und Methoden sollen immer nur so öffentlich wie nötig sein
- die meisten Attribute sollten private sein und nur über getter und setter zugänglich sein
  - Getter: geben nur den Wert des Attributs zurück (read only/nur Lesezugriff)
  - Setter: ändern den Wert eines Attributs
- getter und setter können in Eclipse automatisch generiert werden

Das wisst ihr schon

- Jedes lauffähige Programm ist eine Klasse, wobei der Klassenname = Programmnamen
- Der Datentyp String ist eine Klasse und man kann ein neues String-Objekt z.B. mit String myString = new String(); erstellen.
- Auch die Matheklasse Math habt ihr schon kennengelernt.
- Java bietet viele schon vorgefertigte Klassen, trotzdem muss man auch eigene schreiben.

Aufbau einer Klasse(1/2)

- Prinzipieller Aufbau:

```
1 Sichtbarkeit class Name {
2   //Attribute
3   //Konstruktor
4   //Methoden
5 }
```

```
1 public class Car{
2 }
```

Aufbau einer Klasse(2/2)

- eine Klasse hat eine Sichtbarkeit, die bestimmt in welchem Bereich man ein Objekt der Klasse erstellen kann
- auch Attribute und Methoden haben eine bestimmte Sichtbarkeit (Folie Java Fortführung)

```
1 //Attribut
2 Sichtbarkeit [static] typ variablenname [= Standardwert];
3
4 //Methoden
5 Sichtbarkeit [static] Rueckgabetyp methodenname(typ parameter,....){
6     //code
7 }
```

Konstruktor(1/2)

- der Konstruktor „baut“ das Objekt d.h. reserviert und belegt Speicher für das Objekt
- übernimmt die Initialisierung von Attributen, die keinen Standardwert haben
- wird von der JVM automatisch generiert, wenn der Programmierer keinen schreibt
- 2 Arten von Konstruktoren:
  - Default-/Standardkonstruktor: keine Parameter
  - Konstruktoren mit (beliebig vielen) Parametern

Konstruktor(2/2)

- Prinzipieller Aufbau:

```
1 Sichtbarkeit Klassenname(Parameter){
2     //code
3 }
```

- Beispiel:

```
1 //Standardkonstruktor
2 public Car(){
3     //code
4 }
```

Java-Codebeispiel Car-Klasse

```java
public class Car{
    //Attribute
    public String colour;
    public int numberOfDoors;

    //Konstruktoren
    public Car(){
        this.colour = "Black";
        this.numberOfDoors = 4;
    }

    public Car(String c){
        this.colour = c;
        this.numberOfDoors = 4;
    }

    //Methode
    public void honk(){
        System.out.println("Tut tut");
    }
}
```

static und non-static

- Methoden und Attribute können als static deklariert werden
- static bedeutet, dass es diese Attribute/Methoden auch ohne Objekt aufgerufen werden können
- das heißt, sie sind Eigenschaften einer ganzen Klasse und nicht eines bestimmten Objekts
- ihr habt z.B. die Methoden der Klasse Math benutzen können ohne ein Math-Objekt zu erstellen

Beispiele static und non-static

- Anstatt ein Objekt mit Math zu erstellen, konntet ihr einfach den Klassennamen tippen, einen Punkt und die Methode, die ihr benutzen wollt.
- Das nennt man auch Klassenmethode/-variable.

```
1 Math.pow(2,6);
2 Math.abs(-3);
```

- Die Scannermethoden sind dagegen aber nicht static, hier braucht ihr ein Objekt, um die Methode aufzurufen.
- Das nennt man auch Membermethode/-variable.

```
1 Scanner scan = new Scanner(System.in);
2 int a = scan.nextInt();
```

Objekte erstellen und zuweisen

- Ein Objekt wird mit dem Schlüsselwort new und einem Konstruktoraufruf erstellt.

```
1 Car myNewCar = new Car();
2 Car myOtherCar = new Car("Red");
3
4 String aString = new String();
```

- Ein neues Objekt wird immer nur mit einem neuen Konstruktoraufruf erstellt.

```
1 //erstellt ein neues Objekt von Car
2 Car myNewCar = new Car();
3 //erstellt KEIN neues Object von Car
4 Car myOtherCar = myNewCar;
```

Kurzexkurs: Referenzen

- Nicht primitive Datentypen sind in Java generell als Referenz angelegt.
- Referenz = Verweis auf ein Objekt
- Eine Referenz ist damit ein Aliasname für ein bereits bestehendes Objekt.
- In diesem Fall sind myNewCar und myOtherCar IDENTISCH, sie bezeichnen das selbe Objekt.
- Es ist sozusagen das selbe Auto mit unterschiedlichen Namen.

Zugriff auf Methoden und Attribute

- Zugriff erfolgt mit dem Punkt-Operator
  - wenn static deklariert: Klassenname.methode(), Klassenname.attribut
  - wenn Membervariable/-methode: objekt.methode(), objekt.attribut

Vererbung - Allgemeines Prinzip

- einsparen von Code durch Erweitern einer bereits bestehenden Klasse
- Klassen können ihre Attribute und Methoden an andere Klassen vererben.
- Das kann z.B. praktisch sein, wenn die erbende Klasse eine spezielle Form der ursprünglichen Klasse ist.
- Z.B. ist ein Polizeiauto eine besondere Art Auto.

Begriffe

- Elternklasse/Superklasse/Basisklasse = Klasse, von der geerbt wird (Car)
- Kindklasse/Subklasse/erbende Klasse = Klasse, die erbt (PoliceCar)
- Im Englischen spricht man von parent und child class oder auch base und derived class.

Syntax in Java

```
1 Sichtbarkeit class Kindklasse extends
      Elternklasse{
2      //Klassendefinition
3 }
```

- Schlüsselwort extends zeigt Vererbung an.

Attribute und Methoden

- Attribute und Methoden werden von der Elternklasse übernommen = vererbt.
- Dabei können Methoden überschrieben werden, müssen sie aber nicht.
- @Override stellt sicher, dass Methoden überschrieben werden und nicht aus Versehen eine neue Methode mit gleichem Namen erstellt wird (Stichwort überladen).

```
1 public class PoliceCar extends Car{
2     @Override
3     public void honk(){
4         System.out.println("Ta Tue Ta Ta");
5     }
6 }
```

Sichtbarkeit in erbenden Klassen

- Auf public und protected Attribute und Methoden kann problemlos aus der erbenden Klasse zugegriffen werden.
- Anders sieht es bei der Sichtbarkeit private aus:
  - Kindklassen können auf private Member ihrer Elternklasse nicht direkt zugreifen.
  - Trotzdem besitzen sie diese Member auch indirekt, da sie ebenfalls Ausprägungen der Elternklasse sind.
  - Mit public/protected getter und setter lässt sich dieses Problem lösen.

Erwähnung: Mehrfachvererbung

- Java erlaubt keine direkte Mehrfachvererbung, d.h. hinter extends kann nur eine Klasse stehen.
- Um trotzdem mehrere „Klassen“ zu vererben, nutzt man Interfaces.

this Operator

- Das Schlüsselwort this stellt den Bezug zu dem Objekt her, in dem man sich gerade befindet.
- So kann z.B. eine globale Variable (Attribut) den gleichen Namen haben wie eine lokale (z.B. Parameter).

```
1 public class Example{
2     public int attribut;
3     public Example(int attribut){
4         this.attribut = attribut;
5     }
6 }
```

super Operator

- Mit super greift man auf Methoden/Konstruktor der Elternklasse zu.

```
1 public class ChildExample extends Example {
2     public int anotherAttribut;
3     public ChildExample(int attribut){
4         super(attribut);
5         anotherAttribut = 5;
6     }
7 }
```

- Der super-Konstruktor initialisiert somit alle Variablen die vererbt wurden.
- Er muss im Konstruktor der erbenden Klasse als erstes aufgerufen werden.

Laborordnung

- Essen und Trinken in Laboren verboten
- keine Kabel ziehen oder verändern
- keine illegalen Webseiten besuchen, keine illegalen Downloads
- keine Seiten mit verfassungsfeindlichen, pornografischen oder gesetzlich verbotenen Inhalt öffnen.
- keine Hackangriffe
- keine Schadsoftware programmieren
- kein Spam versenden
- beim Verlassen der Räume Labortür schließen
- Rechner nicht abschalten

Einleitung

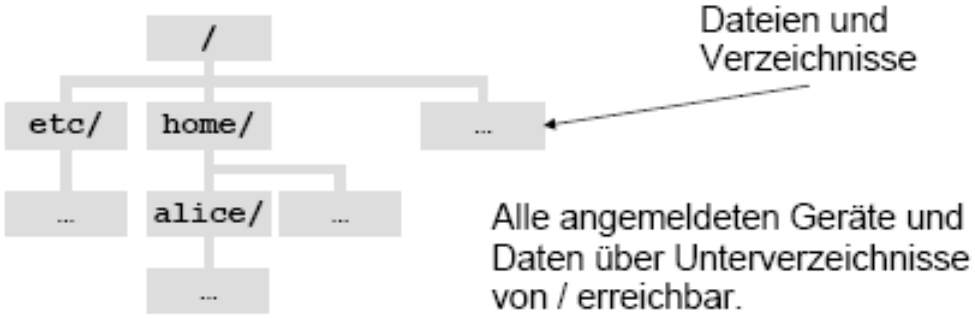
- Anmeldung mit Benutzername und Passwort
- Benutzerverzeichnisse von allen Arbeitsplätzen in den jeweiligen Räumen nutzbar
- Ähnliche Kerne bei Linux und Solaris
  - Konsolenbefehle für unsere Bedürfnisse auf beiden Systemen gleich

Frage an euch:

- Warum Linux?

Verzeichnisstruktur

- Wurzelverzeichnis /
  - Alle anderen Verzeichnisse sind Unterverzeichnisse



- "case sensitive"
  - Unter UNIX werden i. Allg. Groß- und Kleinbuchstaben unterschieden

Konsole - Was ist das?

- textbasierte Benutzerschnittstelle zu Linux und Solaris
- vollständige Systemverwaltung und -steuerung
- Arbeit in aktuellem Verzeichnis
- führt Befehle aus

Befehlssyntax

- Gewöhnliche Befehlssyntax:
  (Ausnahmen in “man”- oder “info”- Einträgen):

  $ command *[option] [parameter]*
- Inhalt des aktuellen Verzeichnisses:

  $ ls
- Liste des aktuellen Verzeichnisses:

  $ ls -l
- Liste eines Verzeichnisses inklusive versteckter Dateien:

  $ ls -la *verzeichnis*

Hilfe!

- Informationen zu einem Befehl durch Eingabe von

  $ man *command*

  [q] - Beendet die Hilfe

  bzw.

  $ info *command*
- Und wenn man den Programmnamen nicht kennt?

  $ apropos *keyword*
- Automatische Vervollständigung von Datei- und Befehlsnamen mit der Tabulator-Taste

Dateiverwaltung - Navigation

- Eigenes Benutzerverzeichnis

  /home/*Benutzername*

  (Kurzform: ~/)
- aktuelles Verzeichnis ausgeben

  $ pwd
- in Verzeichnis “dir” wechseln:

  $ cd *dir*
- Spezielle Verweise
  - . - aktuelles Verzeichnis
  - .. - übergeordnetes Verzeichnis

Dateiverwaltung - Erstellen & Löschen (1/2)

- Erstellen einer neuen Datei

  $ touch *file*
- Löschen (remove) einer Datei

  $ rm *file*
- Warnung: keine Nachfrage und keine Wiederherstellung möglich

Dateiverwaltung - Erstellen & Löschen (2/2)

- Erstellen eines neuen Verzeichnisses

  $ mkdir *dir*
- Löschen eines leeren Verzeichnisses

  $ rmdir *dir*
- Löschen eines Verzeichnisses mit Inhalt

  $ rm -r *dir*

Dateiverwaltung - kopieren, verschieben, umbenennen

- Kopieren (copy) einer Datei an eine bestimmte Stelle

  $ cp *source target*

- Verschieben (move) einer Datei oder eines Verzeichnisses an eine bestimmte Stelle

  $ mv *source target*

- Umbenennen einer Datei oder eines Verzeichnisses, wobei “target” der neue Name ist

  $ mv *source target*

Dateiverwaltung - Zugriffsrechte

- Dateien gehören einem Besitzer und einer Gruppe
- Mögliche Ausgabe von *$ ls -l*

  drwxr-xr-x alice stud 24 2007-09-10 15:09 dir
  -rw-r–r– alice stud 1035 2007-09-12 21:25 file.txt
  - **d** - Verzeichnis
  - **rwx** - Zugriffsrechte von Besitzer, Gruppe, Andere (Lesen, Schreiben, Ausführen)
  - Name des Besitzers und der Gruppe
  - Größe, Änderungsdatum und Dateiname

Dateiverwaltung - Zugriffsrechte

- Ändern des Besitzers

  $ chown *user file*
- Ändern der Zugriffsrechte

  $ chmod *changes file*
- Änderungen symbolisch: **go+w** gibt Gruppe und Anderen Schreibrechte
- Oder numerisch

  640 gibt dem Besitzer Schreib- Und Leserechte, Gruppe darf lesen, Andere nichts

Dateiverwaltung - Zugriffsrechte

- Symbolische Rechte
  - **u** Besitzer, **g** Gruppe, **o** Andere, **a** Alle
  - **+** Rechte geben, **-** Rechte entziehen
  - **r** Lesen, **w** Schreiben, **x** Ausführen
- Numerische Rechte
  - Drei Ziffern jeweils bestehend aus
    4 + 2 + 1
    Lesen + Schreiben + Ausführen

Dateiverwaltung - Textausgabe

- Inhalt aller Dateien hintereinander augeben

  $ cat *file file ...*
- Datei ausgeben, wartet nach jeder vollen Seite

  $ more *file*
- wie “more”, Zurückblättern möglich

  $ less *file*
- “text” ausgeben

  $ echo *text*

Dateiverwaltung - Suchen & Finden

- **find** sucht nach Dateien mit bestimmten Eigenschaften
  - Alle Dateien im Verzeichnis, inklusive Unterverzeichnisse

    $ find *dir*
  - Alle Java-Quelltextdateien im Verzeichnis

    $ find *dir* -name *.java
- **grep** sucht Inhalte in Dateien
  - Zeilen von Java-Dateien, die das Wort “test” enthalten, ausgeben

    $ grep test *.java

Prozessverwaltung

- **STRG-C** bricht den aktuellen Prozess ab
- mit **top** öffnet man den Prozessmanager von Linux

Terminalanmeldung

- Verbindung mit einem anderen Rechner über ein Netzwerk herstellen

  $ ssh *username@computername*

  - Passwort wird abgefragt
  - Terminal, funktioniert wie lokale Konsole
  - Unter Windows kann man mit Putty über ssh auf die Konsole zugreifen
  - Ist der Benutzername auf der aufrufenden Maschine derselbe wie auf dem Fernrechner, kann *username@* weggelassen werden.

Dateien übertragen

- Übertragung von Dateien zwischen Rechnern

  $ sftp *username@computername*

  bzw.

  $ gftp

  (grafische Variante)
- Unter Windows gibt es viele freie sftp Programme (z. B. fileZilla)

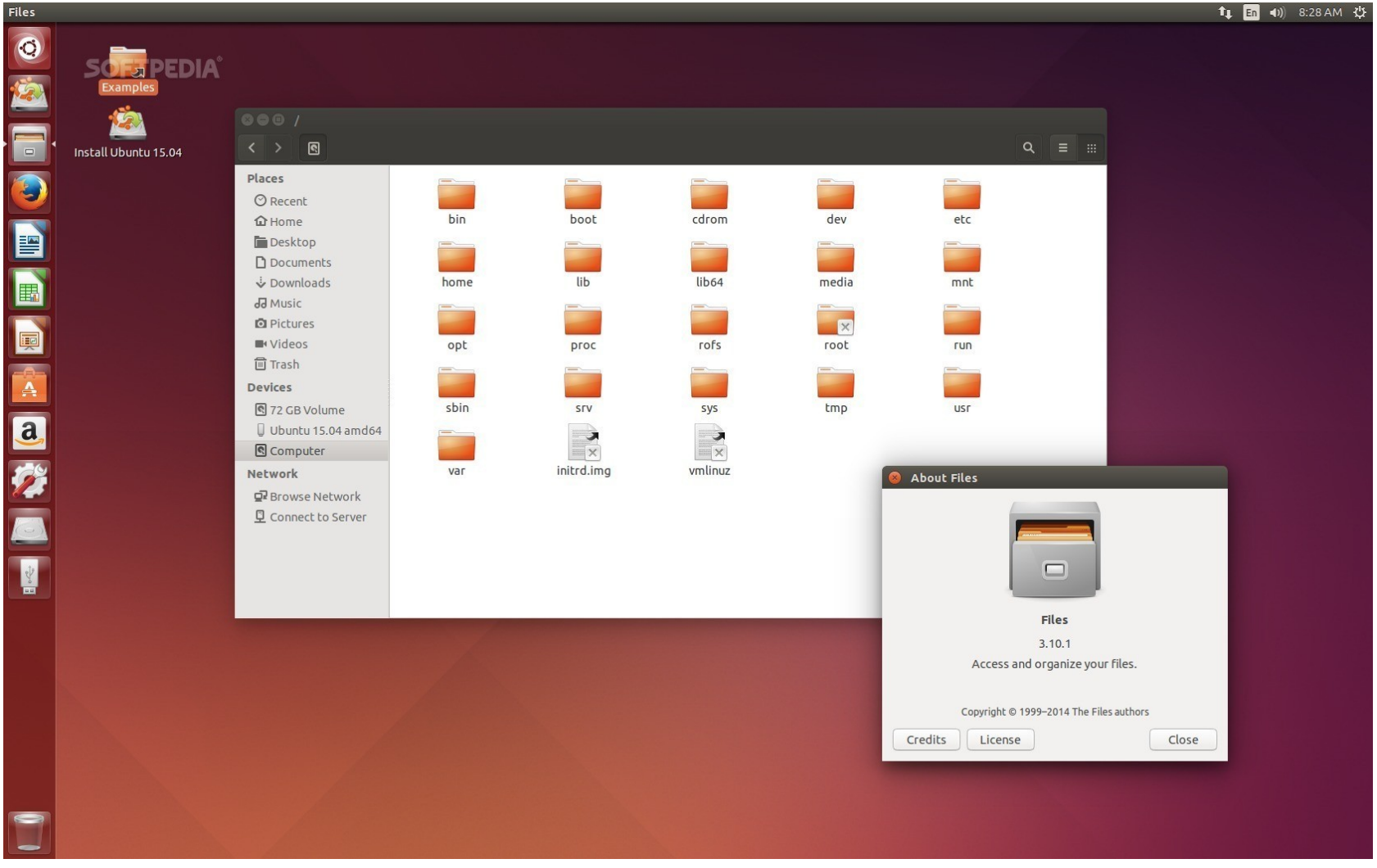
Sinnvolle Einstellungen

- Zugriffsrechte für private Dateien setzen
- Deaktivieren des Caches im Browsers
  (Verletzung der Speicherbegrenzung vorbeugen)

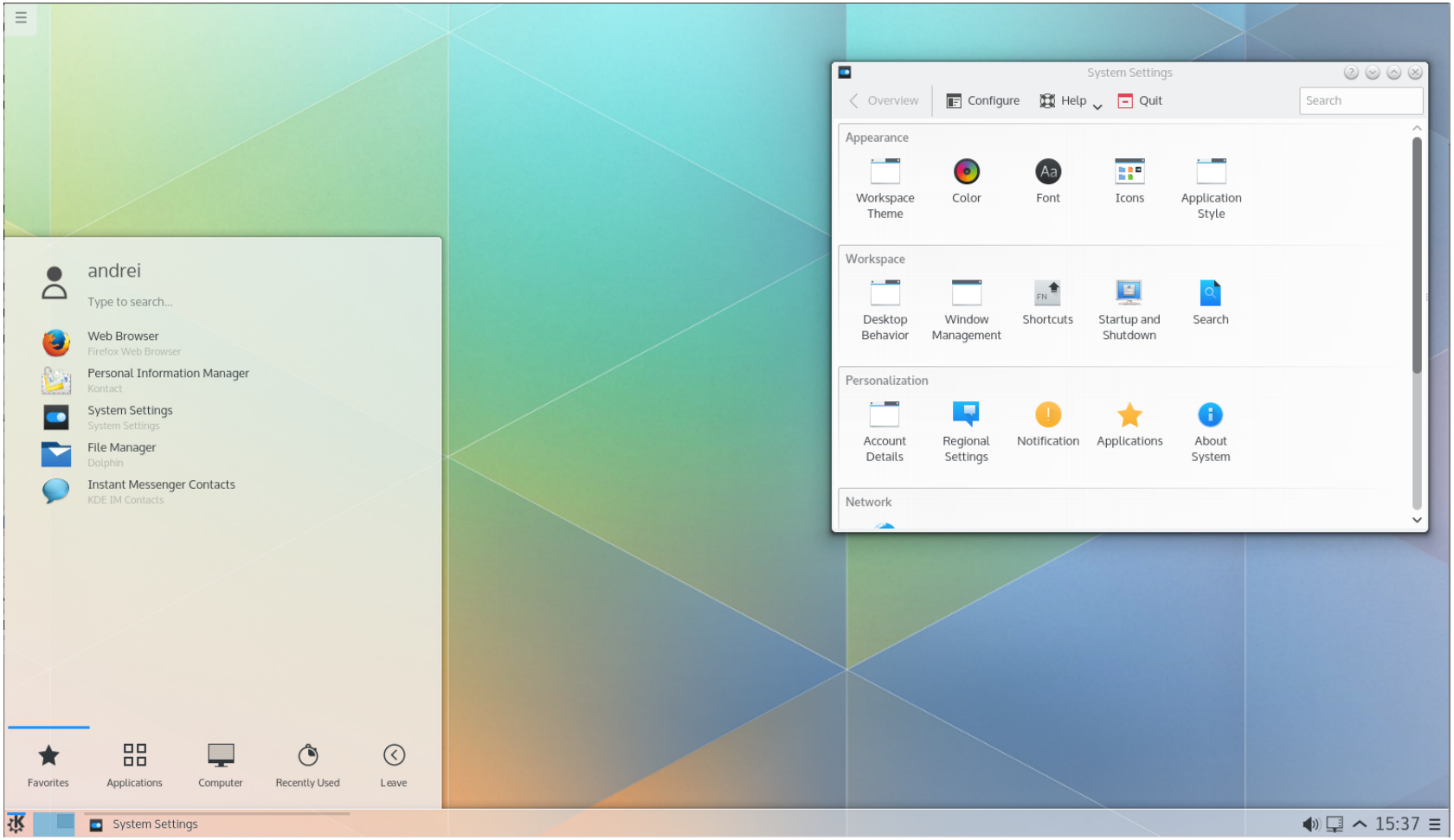
Dein Linux!

- Weit verbreitet:
  - Ubuntu
  - Debian
  - Kubuntu (Windows ähnlich)
  - Linux Mint (Windows ähnlich)
- schnell, stabil, ressourcen-schonend
- viele Sachen sind intuitiv
- im Web findet man immer Hilfe
- viele Anleitungen enthalten alle nötigen Befehle
- einfach zu updaten (OS, Treiber UND Software)

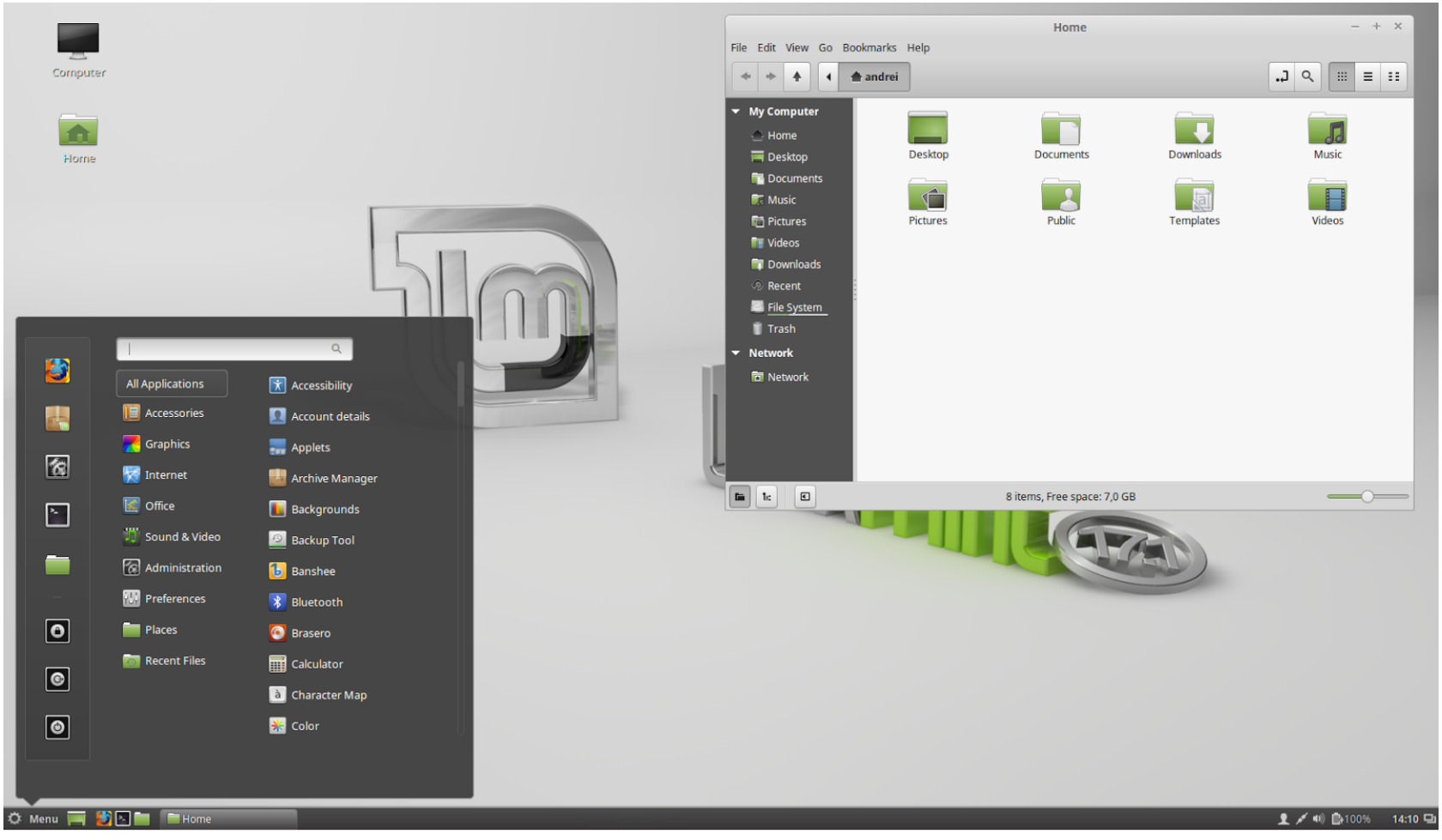
Ubuntu Unity Desktop



Kubuntu Plasma 5 Desktop



Linux Mint Cinnamon Desktop



Linux updaten

- Admin-Rechte: sudo | sudo -s

apt-get update	=	aptitude update (Paketliste aktual.)
apt-get upgrade	=	aptitude safe-upgrade (Pakete updaten)
apt-get install XYZ	=	aptitude install XYZ (Paket installieren)
apt-get autoclean	=	aptitude autoclean (Cache leeren)
apt-get (auto)remove	=	aptitude remove (Paket(e) entfernen)

- Paketquelle hinzufügen (nicht alle sind standard):
- sudo ppa-purge ppa:LP-BENUTZER/PPA-NAME
- grafische Alternative: Paketverwaltung, Softwarecenter

An diesen Folien haben mitgewirkt:

- Christian Braune
- Markus Durzinsky
- Benjamin Espe
- Kai Friedrich
- Gerhard Gossen
- Johannes Hintsch
- Matthias Kliche
- Stefanie Klum
- Markus Köppen
- Michael Kotzyba
- René Meye
- Michael Neike
- Sascha Peilicke
- Julia Preusse
- Michael Schiefer
- Hagen Schink
- Gerd Schmidt
- Martin Zettwitz
- Lydia Rohr
- Manuel Liebchen