PROCESSING

EINE ZUSAMMENFASSUNG

Created by Michael Kirsch & Beat Rossmy

INHALT

TYPEN UND OPERATOREN

DATENTYPEN

Der Computer kennt generell gewisse Arten von Daten. Diese primitiven Datentypen decken primär Zahlen und zeichen ab. Alles Darüber hinaus muss von uns festgelegt und dem Computer vermittelt werden.

DATENTYPEN

BOOLEAN

Wahrheitswert	`boolean b1 = false;` `boolean b2 = true;`
Wertebereich:	falsch oder wahr

DATENTYPEN

INTEGER

Ganzzahlige positive und negative Werte

```
int i1 = 2;
int i2 = -138;
```

Wertebereich: -2147483648 ... 2147483647

DATENTYPEN

FLOAT

Negative und positive Gleitkommazahlen

```
float f1 = 2.3902;
float f2 = -129.368;
```

Wertebereich: 10 Stellen

DATENTYPEN

CHARACTER

Einzelne Zeichen und Ziffern

```
char c1 = 'a';
char c2 = '7';
```

Merke: Intern werden Character von Zahlen repräsentiert!

```
char c2 = '7';
int i = c2; // -> i = 55
```

OPERATOREN

Operatoren erlauben es uns bestimmte Berechnungen, Vergleiche oder Veränderungen auf Daten durchzuführen.

OPERATOREN

ZUWEISUNGSOPERATOREN

Mit Hilfe von Zuweisungsoperatoren können wir Werte in Variablen Speichern oder diese verändern.

Einfache Zuweisung: **=**	`int x = 1; // -> x = 1` `String s = "Test";`
Addition und Zuweisung: **+=**	`x += 1; // -> x = 2`
Subtraktion und Zuweisung:**-=**	`x -= 1; // -> x = 1`

OPERATOREN

ARITHMETISCHE OPERATOREN

Einfache Mathematik ist möglich.

Addition: **+**	`int x = 1 + 1; // -> x = 2`
Subtraktion: **–**	`x = 10 - 1; // -> x = 9`
Multiplikation: *****	`x = 10 * 10; // -> x = 100`
Division: **/**	`x = 6 / 3; // -> x = 2`
Inkrementieren: **++** *Entspricht Addition mit 1*	`x++; // -> x = 3`
Dekrementieren: **––** *Entspricht Subtraktion mit 1*	`x--; // -> x = 2`

OPERATOREN

VERGLEICHSOPERATOREN

Das Ergebnis einer Vergleichsoperation ist immer ein boolscher Wert und kann somit als Aussagen verwendet werden.

Gleichheit: **==**	`a == b`
Ungleichheit: **!=**	`a != b`
Größer: **>**	`a > b`
Größer oder Gleich: **>=**	`a >= b`
Kleiner: **<**	`a < b`
Kleiner oder Gleich: **<=**	`a <= b`

OPERATOREN

BOOLSCHE OPERATOREN

Boolsche Operatoren bilden Aussagen, die wiederum boolsche Werte erzeugen.

Negation: **!**	`!true // -> false`

Und: **&&**	`true && true // -> true`
	`true && false // -> false`
	`false && true // -> false`
	`false && false // -> false`

Oder: **||**	`true || true // -> true`
	`true || false // -> true`
	`false || true // -> true`
	`false || false // -> false`

TYPKONVERSION

Typkonversion ist relevant um Daten z.B. in andere Werte-bereiche zu übertragen. Dabei kann es passieren, dass Informationen verloren gehen.

Downcast:

```
int i = (int)3.9; // -> i = 3
i = (int)4.1; // -> i = 4
```

Upcast:

```
float f = (float)3; // -> f = 3.0
f = (float)4; // -> i = 4.0
```

VARIABLEN

Variablen sind Speicher für Werte oder komplexere Datenstrukturen.

Deklarieren: `int xCoordinate;`
Typ Name;

Initialisieren: `xCoordinate = 1;`
Name Zuweisung;

STRUKTUREN

KONTROLLSTRUKTUREN

Durch Kontrollstrukturen wird es uns ermöglicht eine beliebige Anzahl von Befehlen unter bestimmten Bedingungen oder auf bestimmte Weise zu behandeln.

KONTROLLSTRUKTUREN

BEDINGUNGEN

Blöcke von Befehlen können abhängig von bestimmten Aussagen ausgeführt werden. Der zu **`else`** gehörige Rumpf wird nur ausgeführt wenn die Aussage der Bedingung **`false`** ist.

Kopf: *Schlagwort (Aussage)*	`if (a > b)`
Rumpf	`{ /* wenn a größer */}`
Schlagwort	`else`
Rumpf	`{ /* wenn a nicht größer */}`

KONTROLLSTRUKTUREN

SCHLEIFE

Schleifen ermöglichen es uns Blöcke von Befehlen abhängig von Bedingungen wiederholt auszuführen.

Kopf:
Schlagwort (Initialisierung; Limitierung; Zählen)

```
for (int i=0; i<10; i++)
```

Rumpf

```
{
  // hier stehen beliebig viele Befehle
}
```

FUNKTIONEN

Mehrere Befehle können als Block unter einem neuen Namen ausgeführt werden. Dabei ist es möglich eine Verallgemeinerung durch die Verwendung von Parametern zu erreichen.

Kopf:
Rückgabetyp Name (Parameter)

```
int dasDoppelteVon (int i)
```

Rumpf

```
{
  return 2*1;
}
```

FUNKTIONEN

Hat die Funktion einen Rückgabetyp, so ist der letzte Befehl im Rumpf immer **`return`** dies ermöglicht es einen Wert auszugeben. Dieser muss mit dem festgelegten Rückgabetyp übereinstimmen. Ansonsten ist der Rückgabetyp immer **`void`**.

Ohne Rückgabe	`void ... (...) {...}`
Mit Rückgabe	`int ... (...) {...; return ...;}`

KLASSEN UND OBJEKTE

KLASSEN UND OBJEKTE

Klassen ermöglichen es eigene Datenstrukturen und damit verbundene Funktionalität zu beschreiben. Instanzen dieser Struktur nennen sich Objekte.

Kopf:
Schlagwort Name

```
class Ball
```

Rumpf:
Felder, Konstruktor, Methoden

```
{
  // Felder
  int x,y,r;

  // Konstruktor
  public Ball (int x, int y) {
    this.x = x;
    this.y = y;
  }

  // Methoden
  void plot () {...}
}
```

KONSTRUKTOR

Der Konstruktor hat immer den selben Namen wie seine Klasse. Er kann von überall aus aufgerufen werden, dies beschreibt das Schlagwort **`public`**. **`this`** bezieht sich auf die Felder der durch den Konstruktor erzeugten Instanz.

Kopf:
Schlagwort KlassenName (Parameter)

```
public Ball (int x, int y)
```

Rumpf:

```
{
  this.x = x;
  this.y = y;
}
```

Konstruktoraufruf

```
Ball b = new Ball (100,100);
```

OBJEKTE

Wurde eine Instanz einer Klasse erzeugt so nennen wir diese eine Objekt. Dieses Objekt ist quasi ein Behälter für die in der Klasse beschriebenen Felder und kann sämtliche Funktionalität der Klasse ausführen. Dazu dient der **.** Operator.

Zugriff auf Felder

```
Ball b = new Ball (100,100);
println(b.x); // -> 100
b.x = 200;
println(b.x); // -> 200
```

Zugriff auf Methoden

```
b.plot();
```

STRING

Ein **`String`** ist eine Zeichenkette. Bei der Initialisierung werden die Zeichen der Kette zwischen **`" "`** aufgeführt.

Initialisieren	`String s = "Das ist ein String.";`
Zusammenfügen	`String s = "Das ist ";` `String t = s + "ein String.";`
Länge	`String s = "abc";` `println(s.length()); // -> 3`
Index des ersten Zeichens	`String s = "abc";` `println(s.indexOf('b')); // -> 1`

ARRAYS

Arrays ermöglichen es uns Daten eines Typs in einem Speicher abzulegen. Arrays sind Objekte und werden deshalb mit dem **`new`** Operator erzeugt. Die erste Stelle eines Arrays hat den Index 0.

Deklarieren: *Datentyp [] Name;*	`int[] intArray;`
Initialisieren: *ohne konkrete Werte*	`intArray = new int[7];`
Initialisieren: *mit konkreten Werten*	`intArray = new int[] {1,6,3,8,4,8,0};`
Zugriff	`println(intArray[3]); // -> 8` `intArray[3] = 100; // [3] -> 100`
Anzahl der Speicherplätze	`println(intArray.length); // -> 7`

GÜLTIGKEIT UND KONVENTIONEN

GÜLTIGKEITSBEREICHE

Wird eine Variable in einem Rumpf/Block deklariert, so ist sie in diesem und allen untergeordneten gültig. Variablen die im Kopf einer Funktion, Bedingung oder Schleife deklariert werden, sind im dazugehörigen Rumpf gültig.

Die Variable **`y`** ist nur gültig im Block ihrer Deklaration.

```
int x;

void setup () {
  // ...
  x = 0;
  int y = 0;
}

void draw () {
  ellipse(x,y,50,50); // -> Fehler!
  // y ist NUR in setup gültig!
}
```

BENENNUNG

Durch die einheitliche Benennung von Variablen und Klassen kann die Verständlichkeit des Codes gesteigert werden. Generell gilt, dass bei Namen aus zusammengesetzten Worten jedes weitere Wort mit einem Großbuchstaben an das letzte hinzugefügt wird.

Variablen *erster Buchstabe klein*	`int xCoordinate;`
Funktionen *erster Buchstabe klein*	`int doubleValue (int n) {...}`
Klassen *erster Buchstabe groß*	`class Animation {...}`

EINRÜCKUNG

Der Inhalt jedes Blocks wird auf eine neue Ebene mit dem Tabulator eingerückt. Die den Block schließende geschweifte Klammer steht also wieder auf der ausgehenden Ebene.

Eine Block

```
if () {
  // eine Ebene eingerückt
}
```

Block in Block

```
if (...) {
  // eine Ebene eingerückt
  for (...) {
    // eine weitere Ebene eingerückt
  }
}
```

QUELLEN