PROCESSING

ELTERN UND KINDER

Created by Michael Kirsch & Beat Rossmy

INHALT

RÜCKBLICK

PROCESSING BASICS

keyPressed: Variable

```
if (keyPressed) {...}
```

keyPressed: Funktion

```
void setup () {...}
void draw () {...}

void keyPressed () {
  // beliebiger code
  println(key);
}
```

keyReleased: Funktion

```
void setup () {...}
void draw () {...}

void keyReleased () {
  // beliebiger code
  println(key);
}
```

KLASSEN

Signalwort + Name + Rumpf-Anfang	`class Kreis {`
Felder	`int x,y,d;`

Konstruktor: Name + Übergabewerte	`public Kreis (intx ,int y, int d) {`
	`  ...`
	`}`

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

Klassenrumpf-Ende	`}`

OBJEKTE

Objekt deklarieren	`Kreis k1;`
Objekt initialisieren	`k1 = new Kreis(122,321,20);`
Objekt-Methoden Aufruf	`k1.plot();`
Objekt-Feld Zugriff	`println(k1.x);` `k1.x++;`

THEORIE

ES GIBT JA NICHT NUR EINE ART BALL.

- Anstatt einer Klasse **`Ball`** können wir auch mehrere bestimmen, wie z.B. Fußball, Tennisball, Golfball, ...
- Müssen wir nun für jede Art Ball eine ganz neue Klasse schreiben? Mit all den selben Feldern und Methoden von **`Ball`**?

```
Ball b;
Fussball f;
Tennisball t;
Golfball g;

class Ball {
  int x,y,d;

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

  void plot () {
    ellipse(x,y,d,d);
  }
}

...
```

WO LIEGEN GEMEINSAMKEITEN, WORIN UNTERSCHIEDE?

- Unterschiede: Größe, Aussehen, ...
- Gemeinsamkeiten: Koordinaten, Methoden, ...
- Das sind doch alles Bälle! Warum Teilen sie sich nicht einfach all die Gemeinsamkeiten?

```
Ball b;
Fussball f;
Tennisball t;
Golfball g;
```

EXTENDS

- **`extends`** hilft uns dabei, dass unterschiedliche Klassen ihr Gemeinsamkeiten teilen können.
- Diese Beziehung unterschiedlicher Klassen zueinander nennt sich Vererbung.

```
class Ball {
  ...
}

class Fussball extends Ball {
  ...
}

class Tennisball extends Ball {
  ...
}

class Golfball extends Ball {
  ...
}
```

EXTENDS

- Kindklassen erben alle Felder und Methoden der Elternklasse.
- Eine Kindklasse muss also mindestens einen eigenen Konstruktor enthalten, kann aber auch eigene Felder und Methoden definieren.

```
class Ball {
  ...
}

class Fussball extends Ball {
  Color c;
  float bar;

  public Fussball (...) {
    ...
  }

  void aufpumpen () {}
}
```

EXTENDS

- Wie greife ich nun auf vererbte Felder zu?
- **`super`** erlaubt uns zwei Dinge:
  1. Aufruf des Eltern-klassen-Konstruktors
  2. Zugriff auf Felder und Methoden der Eltern-klasse

```
class Ball {
  ...
}

class Fussball extends Ball {
  Color c;
  float bar;

  public Fussball (int x, int y) {
    super(x,y,21);
    bar = 0.5;
    c = color(255);
  }

  void aufpumpen () {
    if (super.d<23) super.d++;
  }
}
```

EXTENDS

- In der Elternkasse bereits definierte Methoden können abgeändert/ überschrieben werden.
- Dazu wird einfach die selbe Methode mit neuem Rumpf definiert. Nun wird statt der in der Eltern-klasse enthaltenen Methode die der Kind-klasse verwendet.

```
class Ball {...}

class Fussball extends Ball {
  Color c;
  float bar;

  public Fussball (int x, int y) {
    super(x,y,21);
    bar = 0.5;
    c = color(255);
  }

  void aufpumpen () {
    if (super.d<23) super.d++;
  }

  void plot () {
    fill(c);
    ellipse(x,y,d,d);
  }
}
```

EXTENDS

- Hilfreich ist das ganze, da wir uns nun auf diese Gemeinsamkeiten verlassen können!
- Wir wissen jeder **`Ball`** besitzt eine Methode **`plot()`** egal ob geerbt oder überschrieben.

```
Ball[] b;

void setup () {
  size(600,600);
  b = new Ball[]{
    new Tennisball(100,100),
    new Fussball(400,400),
    new Ball(300,300,30)
  };
}

void draw () {
  background(0);
  for (int n=0; n<b.length; n++) {
    b[n].plot();
  }
}
```

AUCH UNTERSCHIEDLICHE DINGE KÖNNEN DIE SELBE FUNKTIONALITÄT BESITZEN.

- Wäre es nicht hilfreich, wenn alle grafischen Elemente im Sketch eine Methode **`plot()`** hätten?
- So könnten wir diese einfach mit nur einem Befehl zeichnen!

```
Ball b;
Pferd p;
Haus h;

void setup () {
  ...
}

void draw () {
  background(0);
  b.plot();
  p.plot();
  h.plot();
}
```

INTERFACES

- Ein **`interface`** erlaubt es uns eine Schnittstelle zu definieren, die sich unterschiedliche Klassen teilen können.
- **`interface`** steht vor dem Namen der Schnittstelle. Diese Namen enden oft auf "-ble", da sie gewisse Fähigkeiten beschreiben (Ability -> able).

```
interface Plotable {
  void plot();
}
```

IMPLEMENTS

- Alle im **`interface`** definierten Methoden, muss die implementierende Klasse enthalten.
- Hinter dem Klassennamen leitet **`implements`** eine Liste der zu implementierenden Interfaces ein.

```
interface Plotable {
  void plot();
}

class Ball implements Plotable {
  ...

  void plot () {
    ...
  }
}

class Haus implements Plotable, ...
 {
  ...

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

IMPLEMENTS

- So können auch die unterschiedlichsten Klassen in Arrays zusammengefasst weren, solange sie sich nur das Interface **`Plotable`** teilen.
- Dies kann hilfreich sein um gemeinsame Schritte vereinheitlicht auszuführen.

```
interface Plotable {...}

class Ball implements Plotable {...
}

class Haus implements Plotable {...
}

Plotable[] p;

void setup() {
  size(600,600);
  p = new Plotable[]{new Ball(...),
 new Haus(...)};
}

void draw() {
  for (int n=0; n<p.length; n++)
    p[n].plot();
}
```

PROCESSING BASICS

Transformations

Verschieben

```
int x = 300;
int y = 200;
translate(x, y);
rect(200,200,200,200);
```

Rotieren

```
rotate(PI/2);
rect(200,200,200,200);
```

Scale

```
scale(2.0);
rect(200,200,200,200);
```

PROCESSING BASICS

Transformations

verhindere Auswirkung der Transformation auf nachfolgende Objekte

```
pushMatrix();
translate(200,200);
rect(0,0,200,200);
popMatrix();

pushMatrix();
translate(100,300);
rect(0,0,200,200);
popMatrix();
```

PROCESSING BASICS

Tabs

Tabs helfen deinen Sketch zu organisieren. Es bietet sich an für Klassen neue Tabs anzulegen.
Klicke auf den ▼ neben dem Sketch Namen und wähle "Neuer Tab". Benenne diesen so, dass nachvollziehbar ist, was der Tab enthält.

```
void setup () {...}

void draw () {...}

// ------------------
// Klasse -> neuer Tab

class Ball {...}

// ------------------
// Klasse -> neuer Tab

class Wall {...}
```

ANWENDUNG

EXTENDS

```
void setup () {
  size(800,800);
}

void draw () {

}
```

IMPLEMENTS

```
// definiere Interface Randomizeable -> zufällige Werte für Objektvariablen

// definiere Klassen die Randomizeable implementieren

void setup () {
  size(800,800);
  // initialisiere Objekte
}

void draw () {
  // RANDOMIZE!
}
```

VERKNÜPFUNG

KEYBOARD VISUALIZER

- Rechts sehen wir den bisherigen Stand unseres Projektes.
- Nun wollen wir die Funk-tionalität des Programms durch Vererbung der Klasse an ihre Kindklassen erweitern.
- Wir versuchen das Konzept "Animation" in seine Bausteine zu zerbrechen.

```
class Animation {
  int animationCounter;
  char triggerKey;

  public Animation (char c) {
    triggerKey = c;
    animationCounter = 0;
  }

  void handleInput (char c) {
    if (keyPressed && animationKey
== c) {
      animationCounter++;
    } else {
      animationCounter = 0;
    }
  }

  void plot () {
    if (animationCounter > 0) {...}
  }
}
```

INTERFACES

- Bestandteile, die nicht zwingend auf Animationen zu beschrenken sind, können in Interfaces ausgelagert werden.
- Auch andere Klassen könnten durch **`plot`** gezeichnet werden.
- Auch andere Klassen könnten auf Tasteninput reagieren.

```
interface Plotable {
  void plot();
}

interface Triggerable {
  void triggerOn(char c);
  void triggerOff(char c);
}

class Animation implements Plotable
, Triggerable {
  int animationCounter;
  char triggerKey;

  public Animation (char c) {
    triggerKey = c;
    animationCounter = 0;
  }

  void triggerOn (char c) {}
  void triggerOff (char c) {}
  void plot () {}
}
```

INTERFACES IMPLEMENTIEREN

- **`triggerOn()`** soll soäter aufgerufen werden wenn die Taste gedrückt wurde.
- **`triggerOff()`** wird aufgerufen wenn die Taste losgelassen wurde.
- Ist der **`animationCounter`** 0 so ist die Animation nicht aktiv.

```
class Animation implements Plotable
, Triggerable {
  int animationCounter;
  char triggerKey;

  public Animation (char c) {...}

  void triggerOn (char c) {
    if (triggerKey == c) {
      animationCounter = 1;
    }
  }

  void triggerOff (char c) {
    if (triggerKey == c) {
      animationCounter = 0;
    }
  }

  void plot () {}
}
```

KEYPRESSED & KEYRELEASED

- Wo werden **`triggerOn()`** und **`triggerOff()`** aufgerufen?
- Die Funktion **`keyPressed`** wird aufgerufen, wenn eine Taste gedrückt wurde. Wir behandeln den Tastendruck.
- Genauso gibt es die Funktion **`keyReleased()`**. Wir behandeln das Ende des Tastendrucks.

```
Animation a;

void setup () {...}

void draw () {
  a.plot();
}

void keyPressed () {
  a.triggerOn(key);
}

void keyReleased () {
  a.triggerOff(key)
}
```

KINDKLASSE

- Unsere Klasse erweitert die Klasse **`Animation`** und ist somit ein Kind davon.
- Ein Konstruktor muss implementiert werden, der den super-Konstruktor aufruft.
- Da **`triggerOn`** nur einmal überprüft wird, wird der **`animationCounter`** nun in **`plot`** inkrementiert.

```
class GrowingBall extends Animation
 {

  public GrowingBall (char c) {
    super(c);
  }

  void plot () {
    if (animationCounter>0) {
      animationCounter++;
      int r = animationCounter*3;
      ellipse(300, 200, r, r);
    }
  }
}
```

AUSBLICK

NÄCHSTE SITZUNG

ÜBUNG

QUELLEN