BASICS IN PROGRAMMING

INTERFACES

Created by Beat Rossmy

SCOPE

1. Variables and Data Types
    1. Variables
    2. Datatypes
    3. Boolean(-operators)
    4. Colors
2. Control Structures
    1. If and Else
    2. Loops
    3. Functions
3. Arrays
    1. Arrays
    2. Iterate
4. Classes
    1. Class
    2. Inheritance
    3. Interfaces

VARIABLES AND DATA TYPES

VARIABLES

Declaration: give values a keyword/name (variable) to make them "memorize-able".

```
int x;
```

Initilization: give these variables initial values.

```
void setup () {
    size(600,600);
    x = 100;
}
```

Usage: use variables instead of static values (e.g arguments). PC looks up the values of variables during execution.

```
void draw () {
    background(0);
    rect(x,200,200,200);
}
```

VARIABLES

`int`	`x`	`;`
Datatype: variables can be of different types.	**Name:** names can be single letters but also words. Always start with lowercase.	**End**

DATATYPES

- If we declare variables we have to specify their types.
- Different datatypes require different space in the working memory.

Integer	`int i = 10;`
Float	`float f = 3.33;`
String	`String s = "hello world!";`
Character	`char c = 'a';`
Boolean	`boolean b = false;`

BOOLEAN(-OPERATORS)

Statements generate boolean values.

greater than	`x > 100`
greater or equal	`x >= 100`
equal	`x == 100`
smaller or equal	`x <= 100`
smaller than	`x < 100`
unequal	`x != 100`

BOOLEAN(-OPERATORS)

Booleans can be **combined or manipulated** to new boolean values.

and	`true && true == true`
	`true && false == false`
	`false && true == false`
	`false && false == false`
or	`true || true == true`
	`true || false == true`
	`false || true == true`
	`false || false == false`
not	`!true == false`
	`!false == true`

COLORS

- Colors are either entered as gray values or RGB values.
- The number of arguments specifies the color type.
- Each color channel can take values from **0-255**.

Gray

```
background(0);
fill(123);
stroke(255);
```

RGB

```
background(255,0,0);
fill(0,255,0);
stroke(0,0,255);
```

Transparent colors.

```
fill(r,g,b,a);
fill(g,a);
```

CONTROL STRUCTURES

IF AND ELSE

- Based on a **condition** we can execute specific code sections.
- **`if`** the condition is **`true`** execute **`{...}`**. **`else`** execute **`{***}`**

```
void draw () {
    background(0);
    x = x+1;

    if (x>100) {...}
    else {***}

    rect(x,200,200,200);
}
```

IF AND ELSE

Keyword

```
if
```

Condition:

```
(x>0)
```

a statement that describes a certain state. A statement is either **`true`** or **`false`**.

Consequence:

```
{...}
```

if the condition is **`true`** the included commands are performed and otherwise skipped.

LOOPS

- Loops help us to solve recurring patterns.
- The three instructions in the **( )** define the execution.

```
for (int i=0; i<10; i = i+1) {
    ellipse(300,300,200-10*i,200-10*i);
}
```

LOOPS

`for`	`(int i=0;`	`i<100;`	`i=i+1)`	`{...}`
	Start: initial value of the counter.	**End:** what is the maximum value of the counter?	**Steps:** how to increment after each loop.	**Body:** commands to be performed.

FUNCTIONS

- Functions allow us to use **generalized** sets of instructions.
- A set of commands is performed when calling the function, using the **attributes** from the attribute list.
- Functions can return values of datatypes or do not return any value (`void`).

```
void printRandomCharacters () {
    for (int i=0; i<100; i = i+1) {
        print(char((int)random(255)));
    }
}

int double (int v) {
    return 2*v;
}
```

ARRAYS

ARRAYS

- In an array you can **store** multiple values of one datatype.
- You can **access** these values by referencing the array and the specific index.

```
int [] a;
void setup () {
    size(600,600);
    a = new int [3];
    a [0] = 255;
    a [1] = 100;
    a [2] = 30;
}

void draw () {
    background(a[0],a[1],a[2]);
}
```

ITERATE

- Use loops to automatically **iterate** over all elements of an array.
- Use this technique for: initialization or handling all elements..

```
int [] a;
void setup () {
    size(600,600);
    a = new int [3];
    for (int i = 0; i<3; i = 1+1) {
        a[i] = (int)random(255);
    }
}

void draw () {
    background(a[0],a[1],a[2]);
}
```

CLASSES

CLASS

- A **`class`** allows us to define data structures.
- The class (**`Ball`**) is the abstract description and the objects (**`b`**) are instances of that class.
- Each objects contains it own set of variables defined in the class as fields.

```
Ball b;

void setup () {
  size(600,600);
  b = new Ball (235,237,52);
}

void draw () {...}

class Ball {
  float x;
  float y;
  float d;

  Ball (float x, float y, float d) {
    this.x = x;
    this.y = y;
    this.d = d;
  }
}
```

CLASS

keyword **`class`** + classname

```
class Ball {
```

Fields

```
  float x;
  float y;
  float d;
```

Constructor: classname + arguments

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

Methods:

```
  void move () {
    x = x+1;
    y = y+1;
  }
```

End of class.

```
}
```

INHERITANCE

- Inheritance enables us to define functionalities and datastructures once that are shared by different classes.
- The child class (`B`) inherits all the **fields** and **methods** of its parent class (`A`).
- A class can have **multiple children** but only **one parent**.

```
class A {
  float x;
  ...
  void doSomething () {...}
}

class B extends A {
  // float x;
  ...
  // void doSomething () {...}
}

A a;
B b;

void setup () {
  a = new A();
  b = new B();
  a.doSomething();
  b.doSomething();
}
```

INHERITANCE

keyword **`extends`** + parentname

```
class B extends A {
```

Fields

```
  float x;
  float y;
  // fields of A are inherited
```

Constructor: call parent constructor

```
  public B (int x ,int y) {
    super(...); // call A constructor
    this.x = x;
    this.y = y;
  }
```

Methods:

```
  void doSomething () {...}
  // methods of A are inherited
```

End of class.

```
}
```

INTERFACES

- Interfaces define methods that **have to** be implemented by the classes implementing this interface.
- The interface specifies the **return type**, the **name** and the **arguments**, but **not the commands** that are performed.
- A class can implement **multiple** interfaces.

```
interface Doable {
  void doSomething ();
  void doSomethingElse (int x);
}

class A implements Doable {
  ...
  void doSomething () {...}
  void doSomethingElse (int x) {...}
}

class Z {
  ...
  void doSomething () {...}
  void doSomethingElse (int x) {...}
}
```

REFERENCES