Programming Methodology Zusammenfassung

Main

```
public class xy {
    publig static void main (String []args) throws $Exception${
        System.out.println("Hello World");
    }
}
```

Convertions

```
int → double : (double) a
int → float  : (float) a
int → String : Integer.toString(a)
String → Int : Integer.parseInt(a)
```

String manipulation

```
String.Split("char") → String[]
String.substring(pos1, pos2) → char pos1 - (pos2 - 1)
String.toUpperCase() → UpperCase
String.toLowerCase() → lowerCase
xyString.charAt(index); → char at pos index
xyString.compareTo(zString); // compareToIgnoreCase();
```

Scanner

```
Scanner xy = new Scanner (System.in);
String  z  =  xy.nextLine ;
int     z  =  xy.nextInt
```

Print

```
System.out. println ( );   → print String + \n      (newLine)
System.out. print ( );     → print String
System.out. printf ( );    → insert %f, %d %s .. and add
                             variables behind String with commas
```

\t ~ tab

format : z.B %.2f → 2 Nachkommastellen
%2.2f → 2 vor, & 2 Nachkommastellen

Switch case

```
switch () {
  case a :
        :
  default :
}
```

if else

```
if(boolean expression) {
} else {      }
```

shorthand

```
int z = (x>y) ? 1 : 2;
```

(✓ 1, ✗ 2)

Loop

```
for (int i=0;  i <= xy ;  i++) {
}
```

```
while (xy <= z) {
}
```

||

```
do { ... } while( );
```

File Manipulation

write

```
FileWriter ab = new FileWriter("xy.txt", true); // append to file
                                          ~ append
PrintWriter cd = new PrintWriter(ab); // will delet file if
                                      ;inserted without FileWriter
cd.println("123");
cd.close();
```

Read

```
File ab = new File("xy.txt");
Scanner input = new Scanner(ab);
while (input.hasNext()){
String str += input.nextLine();
}
input.close();
```

Binary File Manipulation

read

```
FileInputStream     streamName = new FileInputStream("File.dat");
DataInputStream     inputName  = new DataInputStream(streamName);
String   strName   = inputName.readUTF();
ClassXY  obj;
obj=(ClassXY)inputName.readObject();  //requires ObjectInputStream
```

write

```
FileOutputStream  streamName = new FileOutputStream("File.dat", true);
DataOutputStream  outName    = new DataOutputStream(streamName);
outNam.writeUTF(strName);
outNam.writeObject(obj);  //requires ObjectOutputStream
```

Append = true

Random Access File

→ treated as Binary File

```
RandomAccessFile rAF = new RandomAccessFile("File.dat", rw);
DataInputStream / DataOutputStream  stream = .....
```

r = reading
rw = read / write

```
rAF.seek(99);          // move file Pointer to Location
byte b = rAF.readByte(); //read Bytes starting at Pointe
```

Math

```
Math.pow(num, pow);   // num^pow
Math.sqrt(num);       // √num
Math.min(x,y);        // smallest value
Math.max(x,y);        // highest value
Math.abs(x);          // positive absolute value
Math.PI;              // π
```

Methods

```
evtl throw Exceptions
method  modifiers   return Type   name          arguments
public  static      void          thisIsAMethod(  ){}
```

↳ public usable outside of class

↳ private usable only inside of class

Commenting

@ param xy → describe functionality

@ return → describe what is returned

Operations

```
++var;    pre-increment operation
var++;    post-increment operation
```

Random

```
import   java.util.Random;
Random  xy  =  new  Random;
int  z  =  xy. nextInt(bound 1,  bound 2);
                  nextDouble();
                  nextFloat();
```

classes

accessible : public

constructor Method ⇒ same name as class ∀

↓

public name();

create object :

$classname$ xy = new $constructor()$;

methods:

- accessor → methods that access Data
- modifier → methods that modify Data
- toString() → returns name & hash (can be overwritten)
- equals () → to compare object fields

static:

Methods / Fields are the same for all instances

→ referenced globaly via the classname.field-/method-name

copy an object

```
classname name1 = new classname(xy);
classname name2 = new classname(name1);
```

This

.this references vars inside objects

Enum

group of constants

```
enum Level{EASY, ...}
Level xy = Level.EASY;
```

var of object ↓ var of method ↓

e.g. this.num = num;

Abstract Classes

```
abstract class name {
    public abstract void Methodname();
}
```

Serves as a "template"
body has to be defined in subclass

⇒ you can't create an object from a abstract class only of its sub classes

```
class name2 extends name {
    public void Methodname() { //body } }
```

Interfaces

Template for classes; no Method has a body

```
interface name {
int xy = 1;
public void m1();
}
```

Fields are alway static final
↳ has to be defined

an interface has to be implemented into a class:

multiple interfaces can be implemented

```
class ClassName implements name, name2 {
 // define all Methods
}
```

Arrays

var [] name = new var[size] // var = int/float/String... (can also be object)

var [] name = {element0, element1, ..., elementn}

= var name[]; => size later: name1 = new int[size]

var[] name1 name2;

array Length: array.length;

loop through elements in array: for(int val : array){
system.out.println(val);}

change length of array to smaller:

array = new var[size]; // overflow → garbage collection

refrencing:

var[] array2 = array1; //creates a reference not a copy

2D Array (or more)

var[][] name = new var[rows][columns];

ragged → rows do not have the same length

ArrayList

expands / shrinks if item is added / removed

Initialisation:

```
ArrayList<Type> name = new ArrayList<Type>(size); // default size=10
                                  ~ also objects
if object ArrayList: name.add(new object());
```

Operations:

```
name.add(index, element)   // index not required
name.size();               // returns size
name.get(index);           // returns element at index
name.remove(index);
```

Linked List

same operations like ArrayList, but different structure

↳ containers with content linked together □→□→□...

String Builder

class StringBuilder

methods: .append(" ");

.toString();

Iterator

→ traverse a collection without exposing it
↳ z.B ArrayList, HashSet

2.B
```
ArrayList <String> cars = new ArrayList<>();
cars.add("Honda");
Iterator <String> it = cars.iterator();
while (it.hasNext()) { System.out.println(it.next()); }
```
↳ Loop through collection

forEach

→ other form of Iterator

```
cars.forEach ( car -> System.out.println(car) );
```

Lambda

parameter -> expression

(parameter, parameter) -> expression

(parameter, parameter) -> { //code Block }

Method refevence

2.B `cars.forEach( System.out::println );`

Queue

↳ first in, first out

```
Queue <String> q = new LinkedList <>();
q.add("hello");    // add at the end of the queue
q.remove();        // remove and returns first element in q
q.peek();          // returns first element in q
```

Priority Queue

```
PriorityQueue <Integer> pq = new PriorityQueue<>();
```

↳ Same operation as Queue but it alway tries to sort according to given type (eg. Integer 0-∞)

Sets

storing duplicate-free elements

```
Set<String> set = new HashSet();
                  // or LinkedHashSet();   } like List / LinkedList
set.add("test");
set.remove("test");
set.contains("ex");   // returns true if "ex" is in the set
set.clear();          // clears the set
set.sizeC();          // returns the length (int) of the set
```

⇒ you can use iterators or a simple (x : set) loop

Maps

maps a unique key to a value; key can be any type

```
HashMap<String, String> HM = new HashMap<String, String>();
HM.put("key1", "value1");
HM.get("key1");          // returns the value of "key1"
HM.remove("key1");
HM.clear();
HM.size();
```

Loop through

```
for(String : HM.keySet()){}   // Loop through keys
for(String : HM.values()){}   // loop through values
```

Inheritance

Superclass

↳ Subclass extends superclass → subclass inherits fields & methods
only if public

super:

super(); → lets the user define the constructo values of the superclass
↳ has to be 1 argument in constructor!

You can @override inherited superclass methods
↳ final prohibits overides

Polymorphism

you can creak an object through classes

Testclass name 1 = new otherconstructor();
↳ is constructor of a subclass of Testclass

Generics

↳ improve code reuse & type safety

↳ type as parameter (variable type definition)

```
class name <K> {                    // K can be int/str/obj...
    K varname;
    public K name (){ return varname; }
    }
```

also in Methods

```
public K Methodname (K element) {
System.out.println (element.getClass().getName()
                    + "=" + element );
```

Threads

class eighter extends Threads or implements Runnable

```
Runnable name = () -> System.out.println("executed");
Thread name2 = new Thread(name);
name2.start();
```

→ Runnable name = // new customClass Object, where the class implements Runnable and has the Method void run(){}

if a Thread is executed it stats at the run Method

Methods

```
start();    // executes thread and calls run();
run();      // thread behavior
sleep();    // suspend execution by provided time in miliSec.
join();     // pause current thread to wait for another
interrupt(); // stop the execution of a thread
isAlive();  // check if thread was started and is still going
```

Thread Pool

→ define a pool of threads for execution

```
int numThreads = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(CnumThreads);
Future<void> fname = executor.submit(() -> {a});
```

"some Method to execute

⇒ this futures can be added to a Future List to execute them at once

```
List<Future<Void>> namelist = new ArrayList();
```

⇒ loop through to run:

```
for (Future<void> fname2 : namelist) { } // runs all
fname.get(); // returns true if a future is done

executor.shutdown(); // end executor at end of program
```

Synchronize

→ Sync tasks that are critical eg. access bank
→ to not change sth. while someone else is changing

inside method:

```
synchronized (this) { //thing to sync }
```

Locks

→ Lock something while you are changeing it

```
Lock name = new ReentrantLock();
name.lock();
try {  // access shared resource
} finally { name.unlock(); }
```

Atomicity

→ peform operations in a safe way, so that it gets executed correctly, even if variable is used in multiple threads

normal one thread:

```
int i = 0;
i++;
```

Atomic multiple threads:

```
AtomicInteger i = new AtomicInteger(0);
i.incrementAndGet();
```

Atomic CompareAndSwap:

```
i.compareAndSet(7,5);   // if i = 5 it is set to 7
                        // returns true if set, false if not.
```

DeadLocks

→ two Threads have a lock on an object and are waiting on each other, cause they bothe need the other object

Solution

implement a an order of accessing, so when obj 1 is Locked another thread waits until obj 1 is unlocked and doesn't direktly go to obj 2.

Imports

```
import java.util.*;      // ArrayList Scanner, etc...
import java.io.*;        // system input, serialisation
import java.math.*;      // Math functions
import java.util.concurrent.*   // Futures, executor, etc...
```

Errors & Exceptions

- ArrayIndexOutOfBaundsException
- FileNotFoudException

catch exceptions:

```
try {      }
catch (Exeption xy) {  System.out.println(xy.getMessage); }
finally {      }
```

Throw custom exception:

```
throw new Exception("This is a Exception");
```

code that can throw an Exception:

2.B File reading

↳ add throws to the function with the possible Errors

→ public vod method() throws FileNotFoudException { }