SUMAR – Curs Java (FasttrackIt) pe fiecare lectie

**CURS 1**

Install:

Java: https://adoptium.net/temurin/releases/

Sublime: https://www.sublimetext.com/

Git: https://git-scm.com/

Add in environment variable PATH the sublime app directory

Java history:

1)

- 1995 by Sun Microsystems team lead by James Gosling

- innovation: JVM : the byte code. What is that?

2) javac, java, jar

3) JDK, JRE, JVM

4) how to set env variables

- what is PATH env variable?

5) what is an IDE?

Bash (Command Line)

cd - change directory

cd c:

cd myDir

mkdir - make directory

mkdir newDir

mkdir -p newDir1/newDir2/newDir3 - makes all directories in the path

ls - list directory content

cat - prints the file content

cat myfile.txt

pwd - print working directory

- shows current working directory

rm - remove

rm -rf directory -> remove recursively (with children) and force(will delete if directory has files)

ls > studenti.txt prints the result of ls inside a txt file

**CURS 2 –** [Variables, Primitive Types & Operators](https://recruit-me.it/moodle/course/view.php?id=71#section-3) **(Variabile. Tipuri Primitive, Operatori)**

- defining an executable class:

public class Main {

public static void main(String[] args){

}

}

- defining a static function :

public static <return type> numeleFunctiei(<inputParams>){

// function body

}

exemplu:

public static int add(int a, int b){

return a+b;

}

return type poate lipsi:

public static void ceva(String a){

}

input params pot lipsi

public static String faraInputParams(){

return "fara input";

}

ambele pot lipsi

public static void faraNiciunul(){

}

- defining a variable:

only definition: <type> <name> eg. int a;

int a,b,c;

definition + initialization eg. int a=1;

int a, b=2, c;

data types:

- primitives: short, byte, char, int, long, float, double

- objects : String, Object, <any other custom or platform object>

- primitive have different allocated sizes

- byte : 1 byte : -128 to 127

- short : 2 bytes : -32,768 to 32,767

- int : 4 bytes : -2,147,483,648 to 2,147,483,647

- long : 8 bytes : -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

- float : 4 bytes : 3.4e−038 to 3.4e+038

- double : 8 bytes : 1.7e−308 to 1.7e+038

- boolean : 1 bit : true or false

- char : 2 bytes : ASCII characters

literals:

- representation of a fixed value

- int : 1, 100, 20000, 1\_000\_000

- float : 2F, 3.4F

- double: 23D, 5.2D

- long : 2L, 2000L

- char : 'c'

- string: "a value"

- hex : 0x1A

- binary: 0B110101

variables:

- instance variables = non-static fields

- class variables = static fields

- local variables = used in methods

- parameters = input for methods and functions

- ALWAYS named with lowercase and if they are multi word we use camelCase;

- constants are all UPPERCASE

arrays:

- definition: <type>[] <name> or <type> <name>[];

eg: int[] array;

- initialization:

- specifying length:

eg: int[] array = new int[10];

- specifying content

eg: int[] array = new int[]{15,23,5,1,2}

- element access: array[<index>]

eg: array[7]

- assigning value to element: array[7]=8;

- finding the length of an array: array.length

- the index of an array ALWAYS starts with 0. So the last index will be array.length-1

Unary operators

+ : transforms the number or variable into a positive number. It is implicit if you don't put it

- : transforms the number or variable into a negative number

++ : increments a variable : ++a or a++ is equivalent with a=a+1

- it's importatn if you put it

- before the variable, the increment will happen before the value read (prefix)

- after the variable, the increment will happen after the value read (postfix)

so, if we have

int a=1;

b=++a; // b will have value 2 and a=2

but if we have

int a=1;

b=a++; // b will have value 1 and a=1

-- : decrements a variable : --a or a-- is equivalent with a=a-1

- the same rule of prefix or postfix apply to -- as to ++

! : reverts a boolean value.

- !true is false

- if we have boolean a = true; then !a is false

- also this is used to revert conditions

Arithmetic operators:

+ : Additive operator (also used for String concatenation)

- adds two numbers

- : Substraction operator

- substracts the second number from the first one

\* : Multiplication operator

- multiplies two numbers

/ : division operator

- divides two numbers. For integer numbers the remainder will be ignored

% : remainder operator

- divides two numbers and returns the remainder for their division

Getting input from console:

We use java.util.Scanner

Scanner scanner = new Scanner(System.in);

int intValue = scanner.nextInt();

System.out.println(intValue);

String word = scanner.next();

System.out.println(word);

double doubleValue = scanner.nextDouble();

System.out.println(doubleValue);

float floatValue = scanner.nextFloat();

System.out.println(floatValue);

### CURS 3 - [Control Structures, Loops](https://recruit-me.it/moodle/course/view.php?id=71#section-5)

Conditional operators:

Equality operator : ==

- compares two variables or literals

eg. 1==1 is true

int a=3, b=4; then a==b is false

- for objects, the operator doesn't compare their content, but their address (two different objects have different addresses)

eg. String str1= new String("abc"); String str2=new String("abc"); then str1==str2 is false

- if you use string literals the identical string literals will point to the same object. So "abc"=="abc" is true

Not Equal operator : !=

- compares two variables or literals

eg. 1!=1 is false

int a=3, b=4; then a!=b is true

- for objects, the operator doesn't compare their content, but their address (two different objects have different addresses)

eg. String str1= new String("abc"); String str2=new String("abc"); then str1!=str2 is true

- if you use string literals the identical string literals will point to the same object. So "abc"!="abc" is false

Greater Than operator : >

- compares two variables or literals

eg. 1>1 is false;

int a=3, b=4, then a>b is true

- for equal variables, the operator will return false:

eg 1>1 is false

Greater Or Equal Than operator : >=

- compares two variables or literals

eg. 5>=2 is true;

int a=3, b=4, then a>=b is false

- for equal variables, the operator will return true:

eg 1>=1 is true

Smaller Than operator : <

- compares two variables or literals

eg. 5<2 is false;

int a=3, b=4, then a<b is true

- for equal variables, the operator will return false:

eg 1<1 is false

Smaller Or Equal Than operator : <=

- compares two variables or literals

eg. 5<=2 is false;

int a=3, b=4, then a<=b is true

- for equal variables, the operator will return true:

eg 1<=1 is true

Bit Manipulation operators:

- the bit manipulation operators work on the bit representation of a number. So, in order to see it more clearly, you should work in the binary representation of a number.

- to switch a number to its binary representation you ca use Integer.toBinaryString()

AND Operator: &

- applies for each bit of the factors the AND operation

AND : 1&1=1, 1&0=0, 0&1=0, 0&0=0

- so for 1001 & 0101 = 0001

OR Operator: |

- applies for each bit of the factors the OR operation

OR : 1|1=1, 1|0=0, 0|1=0, 0|0=0

- so for 1001 | 0101 = 1101

XOR Operator: ^

- applies for each bit of the factors the XOR operation

OR : 1^1=0, 1^0=1, 0^1=1, 0^0=0

- so for 1001 | 0101 = 1100

Shift Right Operator: >>

- moves the bits to right with the specified number

- the moved positions are lost

- so for 1001>>2 = 0010

Shift Left Operator: <<

- moves the bits to left with the specified number

- the new positions are filled with 0

- so for 1001<<2 = 100100

Control statements

IF: -helps us implement a decision if a condition is true or false

- if:

- syntax:

if(<condition>){

<statements>

}

- if-else

if(<condition>){

<statements>

} else{

<statements

}

- if-else-if-else

if(<condition>){

<statements>

} else if(<another-condition>){

} else{

<statements>

}

SWITCH: used when we have multiple branches based on a variable

- syntax :

switch(<variable>){

case <value> : <statement>

break;

case <value> : <statement>

break;

default: <statement>

}

Getting input from console:

We use java.util.Scanner

Scanner scanner = new Scanner(System.in);

int intValue = scanner.nextInt();

System.out.println(intValue);

String word = scanner.next();

System.out.println(word);

double doubleValue = scanner.nextDouble();

System.out.println(doubleValue);

float floatValue = scanner.nextFloat();

System.out.println(floatValue);

### CURS 4 [- String, Intellij IDEA](https://recruit-me.it/moodle/course/view.php?id=71#section-7)

Type casts

- you can transform a primitive into another one

for example, this is not accepted:

int a = 22;

byte b = a; // byte cannot hold an int value

But if you really want to transform it, you can do it explicitly:

int a = 22;

byte b = (byte)a;

But what if you transform an int to byte that exceeds the byte's capacity:

int a = 200;

byte b = (byte)a; //b will overflow: will have value -56

Packages

- directory structure

Importing classes/packages

import ro.itscool.Functions;

import ro.itschool.\*;

Strings

- Creating a string

String s= “abc”;

String s2= new String(“abc”);

Important methods on String: (for more check https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/String.html)

- length : the length of the string

- trim : deletes the empty spaces at the beginning and the end of the string

- substring(start) : returns the substring starting at position <start> to string's end

- substring(start, end) : returns the substring starting at position <start> to position <end>

- charAt(pos) : the character at position <pos>

- toCharArray : returns the char[] array containing all characters of the string

- toLowerCase : returns the same string, but with all letters lower cased

- toUpperCase : returns the same string, but with all letters upper cased

- split(delim) : splits the string using the delimitator <delim>

- indexOf(str) : returns the first starting index of <str> if it is contained in the source string.

Otherwise, it'll return -1

- lastIndexOf(str) : returns the last starting index of <str> if it is contained in the source string. Otherwise, it'll return -1

- contains(string) : returns true if the string <string> is contained into the source string

Conversions:

String to Integer : Integer.parseInt(string)

String to Double : Double.parseDouble(string)

any object to String : String.valueOf(<object>)

Formatting:

String.format(<pattern>, <vars>...)

pattern can contain:

%d for integer numbers

%s for strings

%d for decimal numbers

### CURS 5 [- Git](https://recruit-me.it/moodle/course/view.php?id=71#section-9)

Introduction to GIT:

- GIT is a distributed version control system

- it tracks files. It remembers all the history of the file. You can create branches to develop the code at different paces

- it has two parts: the repository and the workspace.

Working with git:

- cloning : the process of getting your own copy of the repository on your local machine

- git clone <git\_url> <local\_name>

- git clone git@github.com:itschool-java-oradea/maven-java-base.git

- git from scratch: if you want to start from your local machine with a git environment, create a directory with the repo name, and the run git init inside it. This will initialize the local repository.

- setting up the ssh key:

- ssh-keygen -> Enter...

- this will create in user's directory (~) .ssh/id\_rsa and .ssh/id\_rsa.pub

- copy .ssh/id\_rsa.pub to clipboard; add it to Github: Settings/SSH and GPG keys/New SSH keys. DO NOT expose id\_rsa content to anybody.

- getting the status of the repository:

- git status

- tracking new files:

- add a new file to repository

- echo test >test.txt

- add to git: git add test.txt

- staging new files:

- if you change a file that is already tracked, the modifications will be detected

- git add text.txt

- ignoring files:

- add it to .gitignore. It supports regex patterns

- commiting your changes

- all the staged changes comprise a changeset

- git commit -m "message"

- viewing history:

- git log

- pushing commited changesets

- git push origin master

- getting the changes into the local repository:

- git fetch

- updating the workspace with the latest changes:

- git update

- get the changes AND update the workspace:

- git pull [origin master]

----

Maps

A Map is an object that maps keys to values.

A Dictionary

A map cannot contain duplicate keys: Each key can map to at most one value.

If you want to map a value to multiple ones, you map it to a list

Available methods:

put(key, value)

putAll(Map)

get(key)

remove(key)

keySet() - returns all the keys

values() -returns all values

entrySet() - returns Map.Entry objects which contains (key, value) pairs.

usage:

Map<String, String> mapOfStrings = new HashMap<>();

mapOfStrings.put("Ana", "mere);

mapOfStrings.put("Bob", "pere);

mapOfStrings.put("Chris", "mure);

### CURS 6 [- Git, Introduction to Object Oriented Programming](https://recruit-me.it/moodle/course/view.php?id=71#section-11)

OOP:

What is an object?

- the center concept of the Object Oriented Programming Paradigm

- STATE & BEHAVIOR

- state & behavior in real life:

- clock: state: position of arms

behavior: moving forward, moving backward

- oven: state: temp set, time set, oven temp

behavior: set temp, time, start cooking

- door: state: closed/opened, lock status

behavior: open/close, lock/unlock

- lamp: state: on/off

behavior: turn on/off

- you shouldn't access/change the state directly. Behavior exposes the information from the state and modifies the state of an object.

- hiding the internal state and requiring that all interactions with the object is done through the behavior is called ENCAPSULATION

Object state

- object's fields

- initializing the fields:

- inline : at defintion

- in constructor

- in the static block : you can define a static{} that can contain any initialization code that is allowed to access. This will be called before the constructor.

Object behavior

- object's methods

ENCAPSULATION

- there is no direct access to the object's state. Any interaction, read or change, will happen throu the object's behavior

- think about not having encapsulation for a real lamp: You could access the wires inside the lamp and have to knot them to turn the lamp on. This is a dangerous and error prone procedure

Why do we use OOP?

- Modularity: each object is it's own world, it's written and maintained independently of other objects in the system.

- Information hiding: using encapsulation we do not expose the internal implementation detail of an object

- Code reuse: existing objects, refined by other developers can be easily integrated for your needs

- pluggability: if you find that an object is misbehaving or you want to change it, you just remove it and replace it with another one

Class = Object Blueprint

- in Java an object is defined by a Class

- definition:

class <Name>{

//state

//behavior

}

exercise: build an object that defines a lamp;

class Lamp{

//state

private boolean on;

//behavior

public void turnOn(){

on = true;

}

public void turnOff(){

on=false;

}

}

Object instances

- a Class is the blueprint of an object. Then you are able to instantiate different objects with different states

- Lamp l1 = new Lamp();

- in order to create an object instance, we use the reserved word new: new Lamp().

- this will call the class constructor. The constructor is a special method that doesn't have a return type and has the exact same name with the object.

- by default Java adds an empty constructor to the classes that have none defined

The state of an object:

- the state of an object is defined by its fields.

- the Object fields are non-static object variables:

eg:

class Lamp{

private boolean on;

}

The behavior of an object:

- the behavior of an object in Java is defined by its non-static methods

class Lamp{

..

void turnOn(){

}

}

Visibility:

- In order to achieve encapsulation, we need to restrict the access of the object's user to the elements that won't break encapsulation

- for this we have access modifiers

--------------class----package----subclass----everybody----

- public - yes ---- yes ---- yes ---- yes ----

- protected - yes ---- yes ---- yes ---- no ----

(default)- - yes ---- yes ---- no ---- no ----

- private - yes ---- no ---- no ---- no ----

-----------------------------------------------------------

- these access modifiers govern the visibility of a field or a method

RECOMENDATION:

- for fields always use private, with the exception when you have a really good motive

- use the most restrictive modifier you can afford

Class constructors:

- in order to be able to build an instance of a class, you need to provide a method that builds that object. This special method is generically called a constructor

- a constructor has no return types

- a constructor can have as many parameters you need

- a class can have multiple constructor with different parameters

- if you don't provide a constructor, a default one, with no parameters will be provided for you if possible

eg:

class Lamp{

private boolean on;

public Lamp(){

this.on = true;

}

public Lamp(boolean state){

this.on = state;

}

}

THIS

- inside an object you can get the reference to the object you are in by using the reserved word: this.

### CURS 7 [- Interfaces, Static](https://recruit-me.it/moodle/course/view.php?id=71#section-13)

Interfaces:

- describes the behavior of a class.

- does not provide implementation

- does not have fields / state;

- cannot be instantiated

- can be implemented by a class or extended by another interface

- implementing an interface means giving it implementation for all defined methods

Static:

Static defines a class-level method or field

If you don’t define a field/method as static, each instance of that object will refer to a different object

Static fields are on the class object, not on the instance. So all instances will share the same reference.

### CURS 8 [- Inheritance, Overloading, Overriding](https://recruit-me.it/moodle/course/view.php?id=71#section-15)

Overriding:

- when extending a superclass, you can re-define or add unto the existing logic. You do this by defining a method with the same name and signature as the one from the base-class.

- in order to call the one from the parent, you can use 'super' reserved word

Overloading:

- the concept of overloading is not really related to inheritance

- you can overload methods

- in order to overload methods, you just need to define a new method that has the same name as the existing one, but different arguments

eg.

public void drive()...

public void drive(int speed)...

public void drive(int speed, int direction);

Polymorphism:

- many forms

- polymorfism is the concept throu which a variable defined with a genreic type can behave differently depending on their implementation.

- ALWAYS use the most generic class you can to define an object instance;

FINAL:

- the reserved word 'final' can be used for classes, fields and methods

- when it's on the class it means that the class can't be extended

- when it's on the method, it means that the method can't be overriden

- when it's on the fields, it means that the reference of the variable can't be changed.

### CURS 9 [- Polymorphism,Abstract Classes, Collections](https://recruit-me.it/moodle/course/view.php?id=71#section-17)

Collections

Collection -

- List

- ArrayList

- LinkedList

- Set

- HashSet

- TreeSet

Map

- HashMap

usefull methods:

add, get(i)

contains

remove

Iterators

Passing parameters by reference or by value:

ALL objects in java that are passed as parameters are passed as reference

ALL primitive variables in java are passed as value

- parameters passed as reference, if their content is changed, then the variable that was called as a parameter will also change, as it points to the same reference

- parameters passed as value will not influence the calling variable, regardles of how you change their content

### CURS 10 - [- Reading & Writing Files](https://recruit-me.it/moodle/course/view.php?id=71#section-19)

Conversions:

String to Integer : Integer.parseInt(string)

String to Double : Double.parseDouble(string)

any object to String : String.valueOf(<object>)

Formatting:

String.format(<pattern>, <vars>...)

pattern can contain:

%d for integer numbers

%s for strings

%d for decimal numbers

Reading data from file:

- we use Scanner with a FileReader

Scanner scanner = new Scanner(new FileReader("test.txt"));

- when you do this you need to put throws FileNotFoundException in the caller function

public static void main(String[] args) throws FileNotFoundException {

- then read it as we do with System.in:

- next() - reads next word

- nextLine() - reads the whole line

- nextInt() - reads next int

- nextDouble() - reads next double

- hasNext() - returns true if there are words to read

- hasNextInt() - returns true if there are integers to read

also, you can use a FileInputStrean

Scanner scanner = new Scanner(new FileReader("test.txt"));

as with FileReader, you need to put throws FileNotFoundException

### [Course 11 - Maps, Enums](https://recruit-me.it/moodle/course/view.php?id=71#section-21)

Maps

A Map is an object that maps keys to values.

A Dictionary

A map cannot contain duplicate keys: Each key can map to at most one value.

If you want to map a value to multiple ones, you map it to a list

Available methods:

put(key, value) -adauga perechi cheie,valoare

putAll(Map)

get(key) – intoarce cheia

remove(key) - sterge perechea cu cheia..

keySet() - returns all the keys

values() -returns all values

entrySet() - returns Map.Entry objects which contains (key, value) pairs.

usage:

Map<String, String> mapOfStrings = new HashMap<>();

mapOfStrings.put("Ana", "mere);

mapOfStrings.put("Bob", "pere);

mapOfStrings.put("Chris", "mure);

--------------------------

Enums

- enumeration: a predefined set of fixed values

- naming: always with uppercase

- examples: CardinalPoints {NORTH,SOUTH,WEST,EAST}, days of week, months.

You can also model business related information. eg. Chocolate type enum: WHITE, MILK, DARK;

defintion:

public enum CardinalPoints{

NORTH,

SOUTH,

WEST,

EAST;

}

- actually they are an instance of that class

eg:

enum Day{

MONDAY,

TUESDAY,

...

}

- each constant is basically an instance of type Day

- that's why you can define attributes on them:

enum Day{

MONDAY(1),

TUESDAY(2);

private final int position;

Day(int position){

this.position=position;

}

int getPosition(){

return this.position;

}

}

-----

Passing parameters by reference or by value:

ALL objects in java that are passed as parameters are passed as reference

ALL primitive variables in java are passed as value

- parameters passed as reference, if their content is changed, then the variable that was called as a parameter will also change, as it points to the same reference

- parameters passed as value will not influence the calling variable, regardles of how you change their content

-----

Nested Classes

- Java allows you to create classes inside another class

- Nested classes are of 2 types: static, inner( non-static)

WHY?

- when you need a class just in one place

- thight encapsulation between the class and nested class (nested class can access private fields)

Inner classes (Non-static Nested Classes)

- in order to instantiate one, you need an instance of the enclosing object

eg.

class A{

class B{

}

}

A a = new A();

A.B b = a.new B()

- inner classes cannot have static fields

Static Nested Classes

- Non-static nested classes can be instantiated without an enclosing instance

class A{

static class B{

}

}

A.B b = new A.B();

Anonymous Classes

interface EventHandler{

String handle(Event e);

}

- nested classes with no name

EventHandler handler = new EventHandler(){

String handle(Event e){

return "anonymous class";

}

}