Java 101 - Magistère BFA

Lesson 1

Stéphane Airiau

Université Paris-Dauphine

Evaluation

- small project mainly to make you implement a small application

Slides (in English) will be posted at this webpage.

There are also lecture notes, used in my course for L3 (in French, covering more material).

`http://www.lamsade.dauphine.fr/~airiau/Teaching/L3-Java/2018.pdf`

Object Oriented Programming in `Java`

Objects and Classes

An **object** can be defined by its **states** and its **behaviours**

A car seen as an *object*

States	Behaviours
brand	accelerate
model	gear up
power	gear down
fuel level	turn wheel
oil level	opening door
tires pressure	closing door
rpm	break

A **class** can be seen as a *blue print* used for creating objects

- *states* are represented by variables
- *behaviours* are represented by *methods*.

An **object** is a class **instance**.

The state of an object can only be changed by the behaviour of the object
⇨ using the methods of the object.

Object

An object is an instance of a class.

The running example for the course will be a class of characters of comic books such as Astérix.

We will create a class `Character`. When we will create a particular character, say Astérix, we will instantiate the class `Character` to create the instance / the object Astérix.

By **convention**, the name of a class **always** start by an **upper case letter**. The instances/objects and everything else will start by a **lower case letter**.

We will write a class `MyClass` in a file `MyClass.java`. We will write the code of the class starting with the keyword **`class`**

```java
class Character {
   ...
}
```

to be saved in a file `Character.java`

Java comes with *many* classes!

`Java` comes with a large class library. The library is organised in different `package`s.

`http://docs.oracle.com/javase/8/docs/api/overview-summary.html`

For instance, the `package java.lang` contains all the basic classes of `Java`. The class to manipulate strings of characters is located in that package and is called `String`.

Instance variables

- **instance variables** :
  these variables define the characteristics of the objects.
  - initialisation is optional.

  access : `<name object>.<instance variable name>`

```
Character asterix;
...
asterix.name;
```

- **class variables** : these variables are *common to all* the instances of the class,
  - declaration with the keyword `static`
  - initialisation is compulsory

  access : `<class name>.<class variable name>`

example : the class `Float` encapsulates a floating point number `float`.

- class variables : `MAX_VALUE`, `MAX_EXPONENT`, `NaN`, etc.

Class method and Instance methods

- **Instance methods** : these methods allows to *access* or *modify* the *state* of an instance/object
- **Class methods** : these methods do *not* modify the state of an object. Usually, there are utility methods to work with object of the class.

For now, class methods may not appear so useful, just remember they exist.

Example : `Float` class

- instance method `String toString()`
  - ➩ returns a representation of the **current object** as a character string
- class method `static String toString(Float f)`
  - ➩ returns a representation of an object passed in parameter

```
Float f;
...
System.out.println(f.toString());
System.out.println(Float.toString(3.1419));
```

Encapsulation

The behaviour or the state of an object can be *known by every other object* ⇨ `public`

Any class can

- execute a public method
- access or modify a public variable

hidden to other classes ⇨ `private`
one can call a private method, access or modify a private variable only if it is inside the class

- ⇨ goal is to hide what is "under the hood"
  (one will be able to change code without affecting any other class).
- ⇨ protection

Creating an object : call a constructor

- A class is just a *blue print* to create instances.
- To create an object, we use a special method called a **constructor**.
- In the class, we need to implement a constructor
- *signature of the constructor*
  - the name of the method is the name of the class
  - there is not return type nor `void`

The **default** constructor is the constructor with **no** argument :

```
public class <class name> {
   // declare variables
   // (class or instance)

   ...
   // default constructor
   public <class name>() {
     // body
   }
}
```

Example

In Java, we can use the same name for a method but with different orders and/or numbers of parameters. We can have many constructors to make it easier to build an object.

```
public class Character {
   public String name;

   // default constructor
   public Character(){
     name = "unknown";
   }

   public Character(String my_name){
     name = my_name;
   }
}
```

In the example, we have two constructors.

Declaring and Creating an Object

Declaration

- In `Java`, we can declare and object telling `Java`the nature of an object : every time `Java`will see the variable name, `Java`will know its declared nature.
- In `Java`, any variable **must** be declared !

➩ `<class name> <object name>`

Creation

- we use the keyword **`new`** and we call the constructor : `new <class name>(<arguments list>);`
- we can declare and affect the object in the same instruction

```
Character asterix = new Character("Astérix");
Character obelix = new Character("Obelix"),
   idefix = new Character(”Idéfix”),
   romain = new Character();
```

Ready to build a class

- we know how to create a class
- we know how to create objects
- to build a class, we need to define
  - instance variables : they can be other objects (so we know how to declare them and create them),
    or "primitive types" such as `int`, `double` (we'll talk about this soon).
  - instance methods : these methods access or modify the state of the object. (we'll talk about creating method soon).
  - class variables and methods if needed.
  - and we need to define at least one constructor (we already know how to do that !)

Write your first class :

1- Code a class that represent students. Each students has a name and 3 grades : one for each subject : bank, finance, insurance.

2- What *instance methods* would be usefull ?

Some basic knowledge

The basics

All about writing basic code without using the concept of object.

- Variables, operators, types
- Tests, loops
- Arrays
- methods

Instructions and comments

```java
// This is a comment
```

```java
/* this is a
 comment
 using different lines */
```

An instruction must satisfy the grammar of the langage `Java`.

Most of the instructions finish with a `;`

Elementary Types

These types are **not** objects, they are primitive/elementary types that can be used to build/work with objects.

The "number of bits" denotes the size (in bits) needed to encode one element of each type.

Elementary Types	number of bits	value interval
`boolean`	1	`true` and `false`
`byte`	8	an integer between -128 and 127
`short`	16	an integer between -2^{15}=-32768 et $2^{15}-1$=32767
`int`	32	an integer between -$2^{31} \approx -2.1 \cdot 10^9$ and $2^{31}-1 \approx 2.1 \cdot 10^9$
`long`	64	an integer between -$2^{63} \approx -9.2 \cdot 10^{18}$ and $2^{63}-1 \approx 9.2 \cdot 10^{18}$
`char`	16	unicode characters, there are 65536 codes
`float`	32	a floating point number following theIEEE norm
`double`	64	a floating point number following the IEEE norm

similar types are used for instance in VBA.

Variables : declaration and initialisation

- *simple declaration* :
  `<type> <nom>;`
- *declaration with affectation* :
  `<type> <name> = <value in the type> | <variable> | <expression>;`
- *multiple declarations* :
  <type> <name$_1$>, <name$_2$>, ..., <name$_k$>;
- *multiple declaration with partial affectation* :
  <type> <name$_1$>, <name$_2$>= <value in the type>, ..., <name$_k$>;

Cast : when types do not match

Here is the situation :

```
<type1> <nom1> = <valeur1>;
<type2> <nom2> = <nom1>;
<type2> <nom2> = <valeur1>;
```

- Implicit cast : when type$_2$ is « stronger » than type$_1$

  ```
  int i = 10;
  double x = 10;
  double y = i;
  ```

  an `int` « fits » inside a `double`.
- Explicit cast when `<type1>` is « strictly stronger » than `<type2>` : we need to tell the compiler to do something

  ```
  double x= 3.1416;
  int i = (int)x;
  ```

  We need to tell `Java` to « cut » the `double` to make it fit inside an `int`.

Unary operator

Operator	degree of priority	action	examples
`+`	1	positive sign	`+a; +7`
`-`	1	negative sign	`-a; -(a-b); -7`
`!`	1	logical negation	`!(a<b);`
`++`		affectation and increment by one	`n++;++n;`
`--`		increment by one then affectation	`n++; --i;`

Binary operator

Operator	degree of priority	action	examples
`*`	2	multiplication	`a * i`
`/`	2	division	`n/10`
`%`	3	remainder of integer division	`k%n`
`+`	3	addition	`1+2`
`-`	3	substraction	`x-5`
`<`	5	strictly smaller than	`i<n`
`<=`	5	smaller or equal to	`i <= n`
`>`	5	strictly greater	`i < n`
`>=`	5	greater or equal	`i >= n`
`==`	6	equality	`i==j`
`!=`	6	different	`i!=j`
`&`	7	conjunction (logical and)	`(i<j) & (i<n)`
`|`	9	disjunction (logical or)	`(i<j) | (i<n)`
`&&`	10	optimised conjunction	`(i<j) && (i<n)`
`||`	11	optimised disjunction	`(i<j) || (i<n)`
`=`		affectation	`x = 10; x=n;`
`+=, -=`		affectation and increment	`i += 2; j-=4`

Warning : do **not** use = for equality test!

Expression type

Is the following code correct ?

```java
int i = 5,j;
double x = 5.0;
j=i/2;
j=x/2;
```

Expression type

Is the following code correct ?

```java
int i = 5, j;
double x = 5.0;
j=i/2;
j=x/2;
```

```java
double x=2.75;
int y = (int) x * 2;
int z = (int) (x *2);
```

What are the values of `y` and `z` ?

Equality between object

```java
Character asterix = new Character("Astérix");
Character asterixBis = asterix;
Character asterixTer = new Character("Astérix");
if (asterix == asterixBis)
  System.out.println("Red");
else
  System.out.println("Green");
if (asterix == asterixTer)
  System.out.println("Blue");
else
  System.out.println("Yellow");
```

What is written in the output ?

Equality between object

```java
Character asterix = new Character("Astérix");
Character asterixBis = asterix;
Character asterixTer = new Character("Astérix");
if (asterix == asterixBis)
   System.out.println("Red");
else
   System.out.println("Green");
if (asterix == asterixTer)
   System.out.println("Blue");
else
   System.out.println("Yellow");
```

What is written in the output ?

- a variable is a *reference* to an object in memory and **not** the object !
- == is the equality between references : two references are equal if they refer to the same object in memory
- For testing the equality between **properties** of an object we use a special method `boolean equals(Object o)`.

Let's repeat again

- **Declaration of an object**

```
Character asterix;
```

  one tells that an object of a given type is known with a given name : The object named as `asterix` is of type `Character`

- **Creation of an object**

```
asterix = new Character("Astérix";)
```

  The variable `asterix` refers to an object in memory that is a `Character`

```
asterix = new Character("Idéfix");
```

  The variable `asterix` now refers to a different object that is also a `Character`.
  What happens to the previous object ?

Arrays

How to declare an array :

```
<type> [] line;
<type> [][] rectangle;
<type> [][][] cube;
```

How to create an array : you **must** tell the array **size** !

```
<type> [] line = new <type>[<taille1>];
<type> [][] rectangle = new <type>[<taille2>][<taille3>];
```

How to get the size of the array : `cube.length`

Warning : in computer Science,

- the first entry of an array is indexed by **`0`**.
- ➩ the last entry of an array is then **`length-1`**.

How to use the array : use brackets **`[]`** :

```
rectangle[3][4] + cube[1][2][5];
```

Examples

It is possible to initialise an array using a « list » notation :

primes :

2	3	5	7	11	13	17	19

triangle :

1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

```java
int[] premiers = {2, 3, 5, 7, 11, 13, 17, 19 };
int[][] triangle = {{1,1,1,1}, { 0,1,1,1},
        { 0, 0, 1, 1}, {0, 0, 0, 1} };
```

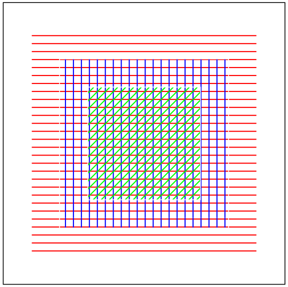
A 2-dimensional array is a 1-dimensional array of 1-dimensional array... so

```java
int[][][] cube = new int[3][4][5];
int[][] rectangle = cube[2];
int n1 = cube.length;
int n2 = cube[0].length;
int n3 = cube[0][0].length;
```

Instruction Blocks

A block gathers together instructions.
The variables that are declared **inside** a block are **only known** inside a block.
i.e. outside the block, the variable does not exist !

```java
int a,b=10;
 {
   int d=2*b;
   {
     int e=b+d;
     a=e*d;
     {
       int g= b+ d*e;
     }
   }
 }
```



a and b are known everywhere.
d only exists in the red part.
e only exists in the blue part.
g only exists in the green part

Conditionnals : `if` … `then` … `else`

```
if ( <boolean expression> )
   <block to execute
     when the condition is satisfied>
else
   <block to execute
     if the condition is not satisfied>
```

The **`else`** block is **optional**.

```
int gains,payment,withdraw,invest;
// some code that modify the gains
   ...
if (gains<0)
  payment = gains;
else if (gains > 10) {
  withdraw = 10;
  invest = gains-10;
}
else
  withdraw = gains;
```

Multiple choices

```
int choice;
...
// something is done with choice
...
switch(choice) {
  case 1:
    //instruction block for case 1
    ...
    break;
  case 2:
    //instruction block for case 2
    ...
    break;
  default
    // default instruction block
    ...
}
```

Inside a **`switch`** `we can use a variable with types` **`int`**`,` **`char`**`,` `and` **`String`**

Loop : for loop

```
for (<initialisation>
    <stopping condition> ;
    <update>)
  <instruction block>
```

What happens in that case ? Is this valid ?

```
for ( ; ; ) {
    // instructions
}
```

a classical example :

```
int n=10;
for (int i=0; i< n; i++ ) {
    // instructions
}
```

Another example

```java
int n=10;
for (int i=0, j=n; j< i; i++; j-- ){
    // instructions
}
```

Loop : `while` loop

```
while(<condition>)
   <instruction block>
```

The instruction block is execute as long as the condition is met.

Example for checking convergence of a series $u : n \rightarrow r^n$:

```
double epsilon = 0.0000001;
double r = 0.75, u=1;
while( u-u*r <= -epsilon && u - u* r >= epsilon)
    u = u * r;
```

Loop `do ... while`

```
do
    <Instruction block>
while(<condition>);
```

Warning ! Do **not** forget the semi column **;** right after the condition !

```
double epsilon = 0.0000001;
double r = 0.75, u=1;
do
    u = u * r;
while ( -epsilon >= u-u*r || u - u* r >= epsilon);
```

choosing a while loop or a do while loop is a matter of elegance, one usually comes easier than the other.

Choosing between a while loop or a for loop

- if we know exactly how many times we iterate : use a `for` loop
- otherwise, use a while loop.
- what is more expected ?

ex :

- search an element in an array ?
- search for the largest element in an array ?

Methods

It is the term used for *function* in a Object Oriented Programming Language.

Goal : Factorise/gather together a sequence of instruction that could be used multiple times.
The code becomes

- more readable (if a pertinent name is used !)
- shorter
- **important** If one wants/needs to modify the code, one just need to update the code in one location !

Method

```
<static or ∅> <public or private> <type of what is returned> <name>
     ( <parameter list>) {
   body of the method
}
```

- Choose an illustrative name !
- the **order** of the parameters is significant :
  `Java` does not match the parameters using names, it uses the order !
- If the method does not return something (it is a procedure), we use the return type `void`.
- when the method returns something, the instruction `return <result>` is used to terminate the method and to output `result`.

Example

```java
public static int max( int[] tab) {
  int m= tab[0];
  for (int i=1;i<tab.length; i++){
    if (tab[i] > m)
      m = tab[i];
  }
  return m;
}
```

Calling the method :

```java
int tab = {7, 12, 15, 9, 11, 17, 13};
int m = max(tab);
```

Overloading

We call **signature** the name of the method with its list of argument.

The signature of a method must be unique to avoid ambiguities.

➾ We can have two methods with the same name but different lists of parameters !

```java
public static double max( double[] tab) {
  double m= tab[0];
  for (int i=1;i<tab.length; i++){
    if (tab[i] > m)
      m = tab[i];
  }
  return m;
}
```

Passing arguments of primitive types

```
public int f(int n){
    n = 3 * n * n -2 *n + 1
   if (n > 0)
     return n;
   else
     return 0;
}
```

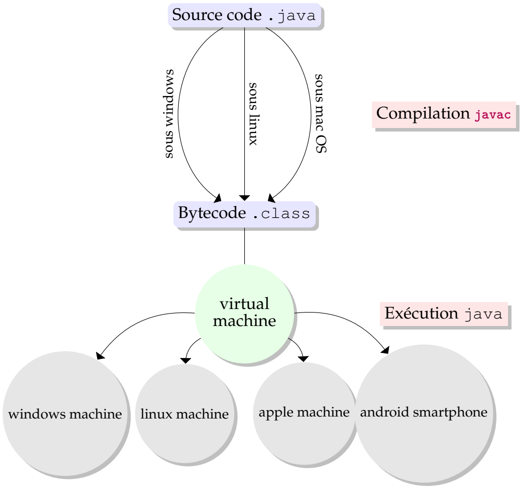
```
int i=13;
   int j= f(i);
```

What is the value of `i` ?

We pass the argument of primitive type **by value**.

Compilation, execution, virtual machine

`Java` is not only a langage and a library of classes
`Java` has tools for *generating* and *executing* code.



Compilation

A class `<MyClass>` is saved in a file `<MyClass>.java` : the name of the class **must match** the name of the file with the extension `.java`.

If the class is named `Character`, it **must** be stored in a file called `Character.java`.

To *compile*, we use a program called **`javac`** that translate your code into machine readable code.

The compiler *translates* your code into a langage that the virtual machine understands.

For `Java` it produces `bytecode`.

The result of the compilation is a file name `<MyClass>.class`

Compilation

Roughly, there are two stages in the compilation process :

- *syntaxic analysis* : we check the grammar of the code
- *semantic analysis* : translation of the code in bytecode
  and we check if everything is well known (other classes)

Execution

What is executed is a special method called `main`.
Each class can have one `main`.
If a method `main` is implemented in a class `MyClass`, we lauch the virtual `Java`machine, which runs the main :
`java MaClass`
(on linux or mac os, you can run this command)

The `main` method has a well specified signature

```java
public static void main(String[] args)
```

- `public` : it must be called from outside the class
- `static` : we have not yet been able to create an object !
- `void` : lthe method does not return anything (to whom should it return something ?)
- `String[] args` : when we start the execution, we can add some text, which will be accessible in this array of string. This is useful when we want to launch an application with some options.

Your turn : back to the student class!

- `String toString()` that returns a representation of the student
- a method to add each grade
- a method that compute the average of the grades. If one note is missing, write a message. As you must return a value, choose an appropriate one.
- a method that tells whether the student passes.

Use a `main` method to test your code.

PS : to write a message on the console, use the following instruction :
`System.out.println(<a string>)`

PPS : for Strings, the binary operator + appends the two strings