# Java 101 - Magistère BFA

## Lesson 1

Stéphane Airiau

Université Paris-Dauphine

## Today

### ⇨ The basics

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

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

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/2017.pdf`

# Instructions and comments

```
1 // This is a comment
```

```
1 /*this is a
2  comment
3  using different lines */
```

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

Most of the instructions finish with a `;`

## Elementary Types

| 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> <name1>, <name2>, ..., <namek>;`
- *multiple declaration with partial affectation* :
  `<type> <name1>, <name2>= <value in the type>, ..., <namek>;`

# 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

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

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

# Expression type

Is the following code correct ?

```
1 int i = 5,j;
2 double x = 5.0;
3 j=i/2;
4 j=x/2;
```

# Expression type

Is the following code correct ?

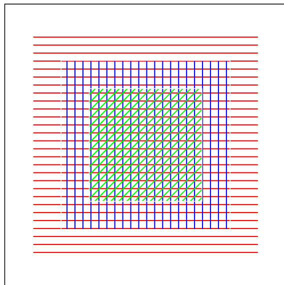
```
1 int i = 5,j;
2 double x = 5.0;
3 j=i/2;
4 j=x/2;
```

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

What are the values of y and z ?

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 !

```
1  int a,b=10;
2   {
3     int d=2*b;
4     {
5       int e=b+d;
5       a=e*d;
6       {
5         int g= b+ d*e;
6       }
6     }
7   }
```



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

```
1 if ( <boolean expression> )
2    <block to execute
3      when the condition is satisfied>
4 else
5    <block to execute
6      if the condition is not satisfied>
```

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

```
1  int gains,payment,withdraw,invest;
2  // some code that modify the gains
3     ...
4  if (gains<0)
5    payment = gains;
6  else if (gains > 10) {
7    withdraw = 10;
8    invest = gains-10;
9  }
10 else
11   withdraw = gains;
```

# Multiple choices

```
1  int choice;
2  ...
3  // something is done with choice
4  ...
5  switch (choice) {
6    case 1:
7      //instruction block for case 1
8      ...
9      break;
10   case 2:
11     //instruction block for case 2
12     ...
13     break;
14   default
15     // default instruction block
16     ...
17 }
```

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

# Loop : for loop

```
1  for (<initialisation>
2      <stopping condition> ;
3      <update>)
4    <instruction block>
```

What happens in that case ? Is this valid ?

```
1  for ( ; ; ) {
2      // instructions
3  }
```

a classical example :

```
0  int n=10;
1  for (int i=0; i< n; i++ ) {
2      // instructions
3  }
```

# Another example

```
0 int n=10;
1 for (int i=0, j=n; j< i; i++; j-- ){
2     // instructions
3 }
```

# 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 $u : n \to r^n$ :

```
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

```
1  public static <type of what is returned> <name>
2        ( <parameter list>) {
3     body of the method
4  }
```

`public` and `static` will become clear in the coming lectures

- 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

```
1  public static int max( int[] tab) {
2    int m= tab[0];
3    for (int i=1;i<tab.length; i++){
4      if (tab[i] > m)
5        m = tab[i];
6    }
7    return m;
8  }
```

## Calling the method :

```
1  int tab = {7, 12, 15, 9, 11, 17, 13};
2  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 !

```
1 public static double max( double[] tab) {
2   double m= tab[0];
3   for (int i=1;i<tab.length; i++){
4     if (tab[i] > m)
5       m = tab[i];
6   }
7   return m;
8 }
```

# Passing arguments of primitive types

```
1  public int f(int n){
2     n = 3 * n * n -2 *n + 1
3    if (n > 0)
4      return n;
5    else
6      return 0;
7  }
```

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

What is the value of `i` ?

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