L3. Object-oriented programming. Course 1

Marie-Pierre Béal (Rémi Forax's course)

Getting started in Java

A Brief History of Programming Languages

Languages

- In the 60s : COBOL, FORTAN, LISP, ALGOL
- Imperative and structured languages (70s) : C, Pascal
- Functional languages (80s and 90s) : ML, OCaml, Haskell
- Object-oriented languages (80s and 90s) : Smalltalk, C++, Objective C,
- Multiparadigm languages : Java, Python

A Brief History of Programming Languages

Imperative versus functional style

- **An imperative language** :
  - executes commands
  - modifies a state (a memory box)
- **A functional language** :
  - executes functions
  - the return value of a function depends only on the values of the parameters

How do we organize and write functions ?

How do we organize and write functions ?

- class
- encapsulation
- sub-typing
- late binding

Correspondence of concepts between imperative and functional styles

imperative	**functional**
Objet + method	lambda
Mutable (Collection)	Non mutable (String)
loop	Stream

The Java language

Java is

- statically typed
- multi-paradigm (imperative, functional, object-oriented, generic, declarative and reflexive)
- with encapsulation, subtyping, late binding

History of Java

Java Versions

- Java 1.0 (1995), Object-oriented
- Java 1.5 (2004), Parameterized types
- Java 1.8 (2014), Lambda
- Java 17 (2021), Record + sealed types
- Java 21 (2023) Pattern Matching
- Java 23 current, JEP 477 Implicitly Declared Classes and Instance Main Methods, JEP 476 Module Import Declarations

Created by James Gosling, Guy Steele et Bill Joy à SUN Microsystem in 1995. It is based on C (syntax) and Smalltalk (virtual machine).

Open source since 2006 `http://github.com/openjdk/jdk`

The Java platform

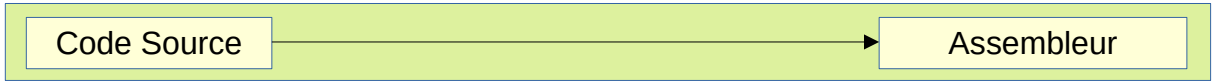
Write Once Run Anywhere

Execution environment

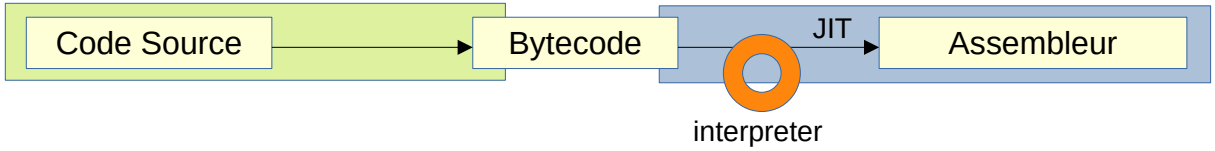
- Virtual Machine / Runtime
- Just In Time (JIT) compiler
- Garbage Collectors

Modèle d'exécution

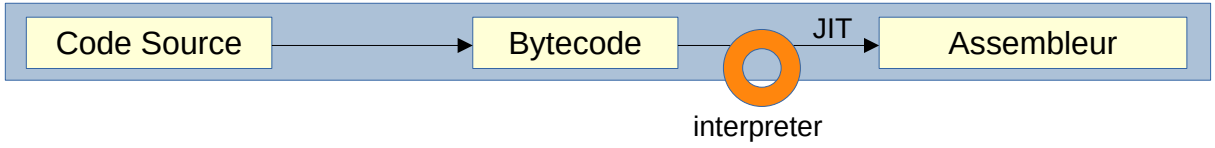
Modèle du C



Modèle de Java



Modèle de JavaScript



A la compilation

A l'execution

Java Reviews

Java is too verbose. Java is the kingdom of names

- Java prefers easy-to-read code
- Java considers each class as a library (easy to use for users)

Java is dead

- Java is backward compatible so it evolves slowly

Children of Java

- JavaScript 95
  - Browser Scripting (Web language)
- C#, 2001
  - Microsoft's Java
- Groovy, 2003
  - Untyped Java
- Scala, 2004
  - Fusion of object-oriented programming and functional programming
- Google Go, 2010
  - Google's Java, static compilation
- Kotlin, 2011
  - Scala made simpler
- Swift, 2014
  - Apple's Java (based on Objective C)

Getting started in Java

Getting started in Java

Java is rigid.

To allow easy reading

- Things are stored
  - The code is stored in a function called a method
  - Methods are stored in a unit called a class
- Code conventions
  - A file is written in CamelCase (uppercase at the beginning) and ends with the suffix `.java`
  - A method or a variable is written in camelCase (lowercase at the beginning)
  - Place the opening brace of a method at the end of the line, and align the closing brace with the start of the block
  - If a class has a name Foo (see lecture 3), then it is declared in the file `Foo.java`

First program

- In a file named `HelloWorld.java`, we write :

```
void main() {
  println("Hello World!");
}
```

- The entry point of the program is the main method
- `println` is a method stored in a class, the class `java.io.IO`.
- We can also write

```
void main() {
  IO.println("Hello World!");
}
```

Compile in memory and execute

- One can compile in memory and execute in one command java followed by the name of the source file.

  ```
  $ java --enable-preview HelloWorld.java
  Hello World!
  ```

  The result is on standard output.

Compile and Run

- We launch the compilation alone with the javac command followed by the name of the source file.

  ```
  $ javac --release 23 --enable-preview HelloWorld.java
  ```

  A `HelloWorld.class` file is created.

  ```
  $ ls
  HelloWorld.class HelloWorld.java
  $
  ```

- We launch the execution (the interpretation of the bytecode) by the Java virtual machine with the java command followed by the name of the file without the `.java` suffix.

  ```
  $ java  --enable-preview HelloWorld
  Hello World!
  ```

JShell

To quickly test or discover an API, there is the interactive REPL jshell window (with tab to complete)

```
$ jshell --enable-preview
|  Welcome to JShell -- Version 23
|  For an introduction type: /help intro
jshell> var a = 3
a ==> 3
jshell> IO.println(a)
3
jshell> /exit
|  Goodbye
```

Types et variables

Types

Java has two kinds of types

- Primitive types
  - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double` (in lowercase)
- Object types
  - `String`, `Date`, `Pattern`, `String[]`, etc (uppercase on the first letter)

Type variables

Primitive types are manipulated by their value

```
int i = 3;
int j = i;   // copy 3
```

We can also write

```
var  i = 3;
var  j = i;   // copy 3
```

Object types are manipulated by their memory address (reference)

```
String s = "hello";
String s2 = s;   // copy the address
```

There is a special null reference. It can be used as a value of any non-primitive type.

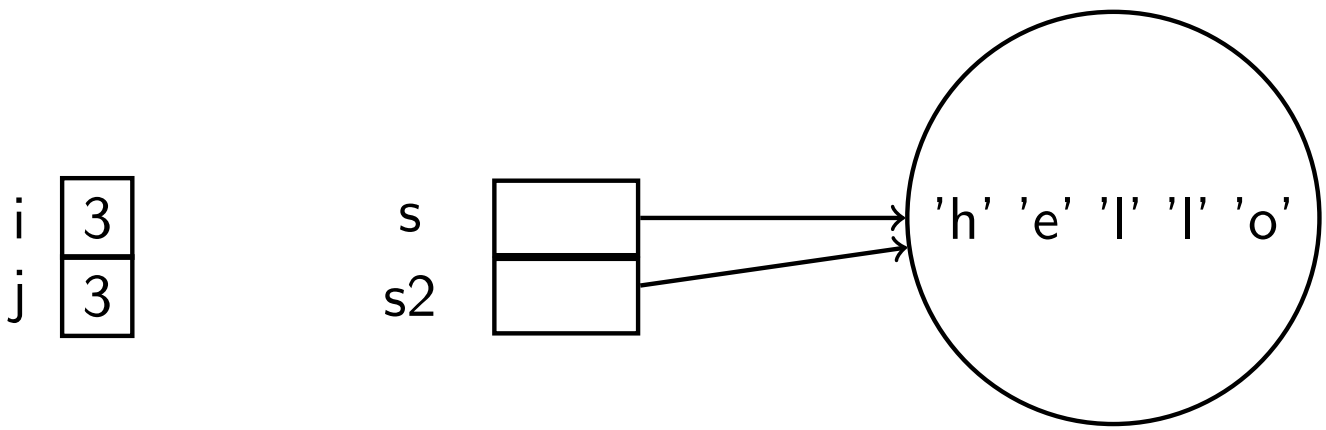
In memory

Primitive type

```
int i = 3;
int j = i;   // copy 3
```

Object type

```
String s = "hello";
String s2 = s;   // copy the address
```



In the bytecode variables are not manipulated by names but by numbers (0, 1, etc.) in order of appearance.

The operator ==

The == operator is used to test whether two memory cells have the same value ont la même valeur

```
var i = 3;
var j = 4;
i == j  // renvoie false
i == i  // renvoie true
```

Be careful with objects, == tests if it is the same reference (same address in memory)

```
var s = ...
var s2 = ...
s == s2  // tests if it is the same address in memory,
         // not the same content
```

Local variable

Declaration

```
Type nom;  // information for the compiler
           // disappears at runtime
```

```
Type nom = expression;
```

équivaut à

```
Type nom;            // information for the compiler
nom = expression;    // assignment at runtime
```

```
var nom = expression;
```

asks the compiler to calculate (infer) the type of expression, so equivalent to

```
Type(expression) nom = expression;
```

Primitive types and processor

Processors have 4 types for runtime operations (so on the stack), not for RAM storage (on the heap) : int 32bits, int 64bits, float 32bits and float 64bits. So boolean, byte, short, char are 32bit ints. So boolean, byte, short, char are 32bit ints.

The compiler disallows numeric operations on boolean. For other types, operations return an int.

```
short s = 3;
short s2 = s + s; // does not compile, the result is an int
```

Numerical types and processor

The types `byte`, `short`, `int`, `long`, `float` and `double` are signed.

There is no unsigned tyye except `char`.

We have specific operations for unsigned

Integer.compareUnsigned(int, int),
Integer.parseUnsignedInt(String),
Integer.toUnsignedString(int),
Byte.toUnsignedInt(byte),
etc

Integers and processor

The `int/long` are weird

Defined between `Integer.MIN_VALUE` et `Integer.MAX_VALUE`, otherwise we have an Overflow (goes into positive/negative).

```
jshell> Integer.MIN_VALUE
$1 ==> -2147483648
jshell> Integer.MAX_VALUE
$2 ==> 2147483647
```

Donc

- `Integer.MAX_VALUE + 1 == Integer.MIN_VALUE`
- `Integer.MIN_VALUE - 1 == Integer.MAX_VALUE`
- `- Integer.MIN_VALUE == Integer.MIN_VALUE`
- `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`
- et 1 / 0 throws an ArithmeticException

Floats and processor

`float/double` are weird too (differently)

- 0.1 is not representable so we have an approximate value
- Imprecision in the computations $0.1 + 0.2 != 0.3$
- 1. / 0. is `Double.POSITIVE_INFINITY`,
- -1. / 0. is `Double.NEGATIVE_INFINITY`,
- 0. / 0. is `Double.NaN` (Not a Number)
- `Double.NaN` is a number (actually, several) that is not equal to itself.
  - `Double.NaN == Double.NaN` returns false
  - `Double.isNaN(Double.NaN)` returns true
- In Java we manipulate floats with the `double` type, not the `float` type
- We never manipulate prices with floats

Record

Record

A record allows one to declare named tuples.
In a file named `PointTest.java file`, write :

```
record Point(int x, int y){}

void main() {
  var point1 = new Point(2, 3);
  var point2 = new Point(1, 4);
  println(point1);
  println(point2);
}
```

We use `new` to create an instance (an object). Here it reserves enough memory space to store two integers (the memory is managed by the garbage collector).

Record

```
$ java --enable-preview PointTest.java
Point[x=2, y=3]
Point[x=1, y=4]
$ ls
PointTest.java

$ javac --release 23 --enable-preview PointTest.java
$ ls
PointTest$Point.class    PointTest.java     PointTest.class

$ java  --enable-preview PointTest
Point[x=2, y=3]
Point[x=1, y=4]
```

Instance Methods

Inside a record, we can define methods (functions stored in a record)

```
record Point(int x, int y) {
  double distanceToOrigin() {
    return ...
  }
}

void main() {
  var point = new Point(2, 3);
  var distance = point.distanceToOrigin();
}
```

Instance methods : this explicit

"this" (like "self" in Python) is a parameter that allows access to the fields of the current object.

```
record Point(int x, int y) {
  double distanceToOrigin(Point this) {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}

void main() {
  var point = new Point(2, 3);
  var distance = point.distanceToOrigin();
}
```

"this" is the value of the object before the ".". For the `point.distanceToOrigin()` call, we call the `distanceToOrigin()` method of `Point` with this = point.

Instance methods : implicit this

There is no need to declare this, the compiler adds it automatically

```
record Point(int x, int y) {
  double distanceToOrigin() {
    return ...
  }
}

void main() {
  var point = new Point(2, 3);
  var distance = point.distanceToOrigin();
}
```

Note : It is very rare to see a method with an explicit this. Some people are unaware that the syntax exists.

implicit access to "this"

If a variable is accessed that is not declared, the compiler adds "this"

```
record Point(int x, int y) {
  double distanceToOrigin() {
    return Math.sqrt(x * x + y * y);
  }
}
```

```
$ java  --enable-preview PointTest.java
3.605551275463989
```

Accessors

In a record, the compiler adds accessor methods automatically. These are methods with the same names as the fields.

```
record Point(int x, int y){}

void main() {
  var point = new Point(2, 3);
  println(point.x());
}
```

```
$ java --enable-preview PointTest.java
2
```

Method equals

In a record, the compiler adds a method `equals` which returns a `boolean`

```
record Point(int x, int y){}

void main() {
  var point1 = new Point(2, 3);
  var point2 = new Point(1, 4);
  var point3 = new Point(2, 3);;
  println(point1.equals(point3));
  println(point1.equals(point2));
}
```

```
$ java --enable-preview PointTest.java
true
false
```

Methods `hashCode()` et `toString()`

In a record, the compiler also automatically adds methods `hashCode` and `toString`.

- The method `toString` returns a `String` representing the object.
- The method `toString` is the method called by println.
- The method `hashCode` returns an integer (almost randomly selected), used for example when putting an object in a collection using hashing. Two objects equal by `equals` have the same hashCode (see later).

Methods hashCode() et toString()

```
record Point(int x, int y){}

void main() {
  var point1 = new Point(2, 3);
  var point2 = new Point(2, 3);
  var point3 = new Point(4, 5);
  println(point1);
  println(point1.toString());
  println(point1.hashCode());
  println(point2.hashCode());
  println(point3.hashCode());
}
```

```
$ java --enable-preview PointTest.java
Point[x=2, y=3]
Point[x=2, y=3]
65
65
129
```

Arrays

Arrays

Arrays are object types.

- They can contain objects or primitive types
  - `String[]` ou `int[]`
- We use `new` to create them
  - Create an array based on a size
    - `String[] array = new String[16];`
    - `var array = new String[16];`
    - initialized with default value : false, 0, 0.0 or null
  - Create an array based on values
    - `var array = new int[] { 2, 46, 78, 34 };`

Arrays

Arrays have a fixed size (not fixed at compile time).
They have a length field which corresponds to their size

- `array.length` (be careful not `array.length()`)
- Indices range from 0 to `array.length - 1`

Arrays are mutable, we use `[` and `]`

```
var array = new int[12];
var value = array[4];
array[3] = 56;
```

We cannot go out of bounds

```
var array = new int[12];
array[25] = ...  // throw ArrayIndexOutOfBoundsException
array[-1] ... // throw ArrayIndexOutOfBoundsException
```

Loops on arrays

Java has a shorthand way to write a loop over an array (the `for(:)`).

- `for(;;)`

```
var array = new int[12];
for (var i = 0; i < array.length; i++) {
  var element = array[i];
  ...
}
```

- `for(:)`

```
var array = new int[12];
for (var element: array) {
  ...
}
```

`equals/hashCode/toString` for arrays

These methods also exist on arrays but do not have the expected behavior

- `equals()` returns true if the arrays have the same address in memory

```
var array1 = new int[] { 1, 2 };
var array2 = new int[] { 1, 2 };
array1.equals(array2);  // false
```

- `hashCode()` returns an almost random number, the same for each instance
- `toString()` returns hashCode + "@" + array type

The class `java.util.Arrays`

The class Arrays (package java.util) provides static utility methods.

To display an array, we will use for example

```
var numbers = new int[] { 4, 11, 24, 7, 18, 91, 6, 2, 55 };
println(Arrays.toString(numbers));
// [4, 11, 24, 7, 18, 91, 6, 2, 55]
```

```
void main() {
  var array1 = new int[] { 4, 11, 24, 7, 18, 91, 6, 2, 55 };
  var array2 = new int[] { 4, 11, 24, 7, 18, 91, 6, 2, 55 };
  println(Arrays.equals(array1, array2));  // true
  println(Arrays.hashCode(array1));  // -1095323397
  var array3 = new int[] { 4, 11, 24 };
  println(Arrays.hashCode(array3));  // 34000
}
```

Summary

The method that serves as the entry point is called `main`

```
void main(){
  println("Hello");
}
```

An object is instantiated (created) by calling new

```
record Train(int weight){}

void main() {
  var train = new Train(250);
}
```

Summary

A method stored in a record is an instance method.

An instance method is called on an object with the symbol "." after the object.

```
record Train(int weight){
  void method(){
    ...
  }
}

void main() {
  var train = new Train(250);
  train.method();
}
```