Object-oriented programming with Java – Part 1

Samuel Toubon

Ensai

Acknowledgment

This course is strongly inspired by Olivier Levitt's one, available at **formations.levitt.fr**

Agenda

- 4 parts
  - Part 1 & 2 : OOP with Java
  - Part 3 : How to use Java ?
  - Part 4 : How to deal with a real project ?
- 4 lessons (1.5h), each with a practical session (3h)
- A final exam (multiple choice, alone, on paper)

Table of contents

1. Reminder about OOP
2. Welcome to Java
3. The basics
4. A strong typing discipline
5. **static** and **final** keywords
6. **this** keyword (and how not to overuse it)

Reminder about OOP

How would you model this situation ? How would you implement it ?

A car has four wheels, each characterized with a unique id. Each car has a unique registration number, which can change, and a brand which cannot. At every time, a wheel belongs to only one car, but you could change the wheel of a car. You could destroy the car and still get back the wheels.

What if you should store thousands of such cars in a database ?

Reminder about OOP

What is a class ? An instance ?

Reminder about OOP

What is an attribute ? A method ?

What can be found inside a class ?

Table of contents

1. Reminder about OOP
2. Welcome to Java
3. The basics
4. A strong typing discipline
5. **static** and **final** keywords
6. **this** keyword (and how not to overuse it)

What is Java ?

- A language
- A **programming** language
- An **object-oriented** programming language
- A **compiled** object-oriented programming language (kind of, more on that later)

Why so many languages ?

https ://www.college-de-france.fr/site/gerard-berry/course-2015-11-04-16h00.htm

▸ Link

18'58

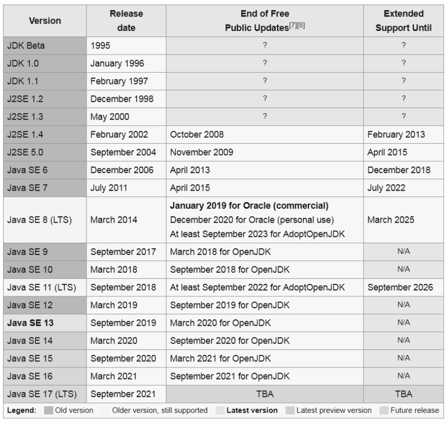
Why Java ?

- popular
- portable (desktop, servers, smartphones, more on that later)
- robust and secure
- simple
- open source
- fast (kind of, more on that later)
- INSEE-friendly : more than 9 out of 10 home-made apps running Java there

Java popularity (2019)

- ieee.org : Python, Java, C, C++, R
- tiobe.com : Java, C, Python, C++, C#
- tiobe.com : Javascript, Java, Python, PHP, C++

Java versions (source : wikipedia)



Version	Release date	End of Free Public Updates[7][8]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	**January 2019 for Oracle (commercial)** December 2020 for Oracle (personal use) At least September 2023 for AdoptOpenJDK	March 2025
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least September 2022 for AdoptOpenJDK	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

Legend: ▨ Old version □ Older version, still supported □ **Latest version** ▨ Latest preview version □ Future release

Table of contents

1. Reminder about OOP
2. Welcome to Java
3. The basics
4. A strong typing discipline
5. **static** and **final** keywords
6. **this** keyword (and how not to overuse it)

A simple class

```
public class Student {
    public String name = "Toubon";
    public String firstName = "Samuel";
}
```

A simple instance

```
Student alice = new Student();
alice.firstName = "Alice";
```

Naming conventions

- starts with a letter
- only includes letters, numbers, and underscores
- case sensitive !
- cannot be a language keyword (such as **while**)
- camelCase is used, i.e. variables start with a lowercase and words are separated with an uppercase
- there is a special rule for constants (more on that later)

A simple method

```
public class Student {
        public String name = "Toubon";
        public String firstName= "Samuel";

        public void sayHello() {
                System.out.println("Hello, my name is
                    "+firstName);
        }
}
```

the constructor, a special method

- it has the name of the class and no type of return
- it's used to initialize an instance
- Java provides by default an hidden void constructor to each class... which is disabled if you implement you own
- you can have several constructors for each class

```
public class Student {
    public String name = "Toubon";
    public String firstName = "Samuel";

    public Student(String name, String firstName) {
        this.name = name;
        this.firstName = firstName;
    }
}
```

```
Student s = new Student("Toto","titi");
```

main, another special method

```
public class Main {

        public static void main(String[] args) {
                Student alice = new Student();
                alice.firstName = "Alice";
                alice.sayHello();
        }

}
```

Notice the signature of the method, it has to be exactly this one !

Conditional blocks

```
if (booleanExpression1) {
    ...
} else if (booleanExpression2) {
    ...
} else {
    ...
}
```

```
switch (value) {
    case value1:
        ...
    break;
    case value2:
        ...
    break;
    default:
        ...
}
```

Loop blocks

```
while (booleanExpression1) {
    ...
}
```

Do... while exists, too.

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Operators

- Comparative operators : <, >, <=, >=, ==, !=
- Boolean operators : !, &&, ||

Table of contents

1. Reminder about OOP
2. Welcome to Java
3. The basics
4. A strong typing discipline
5. **static** and **final** keywords
6. **this** keyword (and how not to overuse it)

Primitive types

Primitive types are the most basic data types available within the Java language.

- Integer :

type	size (bytes)	minimum value	maximum value
byte	1	$-2^7 = -128$	$2^7 - 1 = 127$
short	2	$-2^{15} = -32\,768$	$2^{15} - 1 = 32\,767$
int	4	$-2^{31} = -2\,147\,483\,648$	$2^{31} - 1 = 2\,147\,483\,647$
long	8	$-2^{63} \approx -9 \cdot 10^{18}$	$2^{63} - 1 \approx 9 \cdot 10^{18}$

- Floating-point :

type	size (bytes)	amplitude	precision
int	4	limited	limited
long	8	less limited	less limited

- Boolean : **boolean** true or false
- Characters : **char** on 2 bytes, delimited with single quotes ' '.

String

- Not a primitive type but very mainstream.
- Delimited by double quotes : " ".

```
String hey = "Hello world :)";
```

- As a non-primitive type, the name **String** begins with a capital.
- All **String** variables are instances of the class **String** ! So we could write it this way :

```
String hey = new String("Hello world :)");
```

Types : example

```
int myInteger = 5;
float myFloat = 5.0f/8; //0.625 will be stored !
char myChar = ’a’;
String a = 15; //will fail !
int b = 3.5; //will fail !
```

Strong typing everywhere

```
public class Student {
    public String name;

    public void changeName(String newName) {
        name = newName;
    }
}
```

Table of contents

1. Reminder about OOP
2. Welcome to Java
3. The basics
4. A strong typing discipline
5. **static** and **final** keywords
6. **this** keyword (and how not to overuse it)

Let’s speak about attributes

- **static** means it’s attached to the class, not the instance
- **final** means it cannot change over time, i.e. once it has a value it keeps it forever, i.e. it is a constant

Game : I want to write a FrenchCitizen class. Can you find one example attribute for each of these empty cells ? What would be their types ?

FrenchCitizen	final	not final
static		
not static		

- NB : **final** can also be used for a simple "variable" inside a method, it’s not only for attributes !

Let’s speak about attributes : syntax

```
public class Car {
    public final static int NUMBER_OF_WHEELS = 4;

    public String name = "Model S";
}
```

```
Car.NUMBER_OF_WHEELS;  //we do not use camelCase on
    constants !

Car myCar = new Car();
myCar.name;
```

What about methods ?

- **static** means it’s attached to the class, not the instance (easy, right ?)
- **final** is trickier but not so useful, more on that later

```
public class Maths {

    public static int add(int a, int b) {
        return a + b;
    }
}
```

```
int total = Maths.add(2, 3);
```

Table of contents

1 Reminder about OOP

2 Welcome to Java

3 The basics

4 A strong typing discipline

5 **static** and **final** keywords

6 **this** keyword (and how not to overuse it)

this keyword

this refers to things related to the current instance, precisely :

- Used as a function, it refers to the constructor of the class of the instance.
- Used as a variable, it refers to the current instance.

We have already seen this example :

```
public class Student {
    public String name = "Toubon";
    public String firstName = "Samuel";

    public Student(String name, String firstName) {
        this.name = name;
        this.firstName = firstName;
    }
}
```

Useful this vs unuseful this

Idea : what if we change the names of the function parameters ?

```
public class Student {
    public String name = "Toubon";
    public String firstName = "Samuel";

    public Student(String lastName, String givenName) {
        this.name = lastName;
        this.firstName = givenName;
    }
}
```

this used as a function : example

```
class Counter {
    int position, step;

    Counter(int position; int step) {
        this.position = position;
        this.step = step;
    }

    Counter(int position) {
        this(position, 1);
    }
}
```

Object-oriented programming with Java – Part 2

Samuel Toubon

Ensai

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Motivation

The goals of encapsulation are :

- define which parts must be visible from outside and which should not
- be sure that only the authorized methods can change the value of some attributes
- have a clear distinction between the claimed behaviour and the implementation

Or to make it (over)simple :

- group relevant attributes in a class
- hide the implementation from outside the class
- allow only certain access via public methods

Visibility

4 levels of visibility in Java :

- public
- private
- protected, more on that later
- package (by default)

Each level can apply to a **class**, a **method**, or an **attribute**.

The good practice : every attribute should be put as private by default.

Visibility : the problem

```java
public class Pokemon {
    public int xp = 0;
    public int level = 1;
}
```

```java
Pokemon pokemon = new Pokemon();
pokemon.xp = 9999;
// pokemon.level is still 1
```

Getters & setters

These are functions to access/modify private attributes while protecting them against misusage.

```java
public class Pokemon {
    private int xp = 0;
    private int level = 1;

    public int getXp() {
        return xp;
    }

    public int getLevel() {
        return level;
    }

    public void setXp(int xp) {
        this.xp = xp;
        this.level = Level.relatedLevel(xp);
    }
}
```

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Inheritance

- Inheritance is used to define a class (sub class) based on the characteristics (attributes, methods) of another existing class (super class or base class).
- Most of the time, inheritance means there is an **is-a** relationship between these concepts. Dog is a kind of Animal, Car is a kind of Vehicle...
- There is no multiple inheritance between classes in Java.

Inheritance : syntax

```
public class Animal {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

Inheritance : syntax

```
public class Cat extends Animal {

    public void meow() {
        System.out.println("Miaouh");
    }
}


public class Dog extends Animal {

    public void bark() {
        System.out.println("Ouaf");
    }
}
```

Inheritance : syntax

```
Animal animal = new Animal();
animal.setName("Toto");

Cat cat = new Cat();
cat.setName("Kroquette");
cat.meow();

Dog dog = new Dog();
dog.setName("Medor");
dog.bark();
```

Let's take a break and think

- Remember the very first question of this course ? "A car has four wheels..." What is the difference between the solution we thought about then and inheritance ? Could we have used inheritance ?
- Oh, and what about this tricky thing about **final** on methods ? (And classes ?)
- Do we now know enough to understand **protected** ?

super

super keyword has two usages :

- used as a method, it refers to the constructor of the super class
- used with a dot, it refers to a method of the super class

```
public class Animal {
    private String name;
    public Animal(String name){
        this.name=name;
    }
}


public class Duck extends Animal {
        public Duck() {
            super("Donald"); //ducks default name is Donald
        }
        public Duck(String name) {
            super(name);
        }
}
```

super with a dot

```
public class Animal {
    private String name;
    public Animal(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }
}
public class Duck extends Animal {
        public Duck() {
            super("Donald");
        }
        public Duck(String name) {
            super(name);
        }
        public String getName(){
            return super.getName()+" the duck";
        }
}
```

A few more things

- We have learned that a class can inherit of at most one other class, i.e. there is no multiple inheritance.
- In reality, **Object** is the super class of any class which does not explicitly extends another. So a class inherits of at least one other class.
- To sum up, in Java, apart from **Object**, every class has exactly one super class.

A few more things

- **Object** provides a public **toString** method, so every class does. The sub class can redefine it or not. If not, the implementation of the super class applies.
- **Object** provides a public **equals** method, so every class does. The sub class can redefine it or not. If not, the implementation of the super class applies.
- **Trap !** Using == to compare two instances (including **String**s !) means we check whether they are the same instance physically stored in memory.

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Polymorphism

- Polymorphism is used to attach a different kind of behaviour to classes which look the same from the outside.

Abstract classes

What if we do not want people to be able to instantiate **Animal**s but only concrete **Cat**s and **Dog**s ?

```
public abstract class Animal {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

Abstract classes

Then we realize that Cats and Dogs do essentially the same thing (they kind of speak) each their fashion.

```
public abstract class Animal {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public abstract String speak();
}
```

Animals do not have a fashion to speak, right ?

Abstract classes : syntax

```
public class Cat extends Animal {

    public void speak() {
        System.out.println("Miaouh");
    }
}


public class Dog extends Animal {

    public void speak() {
        System.out.println("Ouaf");
    }
}
```

Hey, now Dog and Cat look the same from outside ! They both have a **speak** method with no argument and no return.

Abstract classes : syntax

Now, as Cat and Dog are both animal, we can create instances of these classes and type them as Animal. It would be very useful if we wanted to populate a set of Animal and make them speak no matter the details. More on that later.

```
Animal myCat = new Cat();
Animal myDog = new Dog();
myCat.speak();
myDog.speak();
```

Hint ! Java auto selects the more specialized version of the used method. So, even if **speak** had not been abstract, dogs would still have said "Ouaf" and cats "Miaouh".

Interfaces

What if we want to go further and separate interface from implementation ?

Meet Java **interfaces** :

- they are essentially a contract
- they declare methods
- they do not have attributes
- they do not hold implementation
- one cannot instantiate an interface

Interfaces

- an **interface** can be respected by zero, one or several classes with different implementations
- a class can respect zero, one or several contracts, i.e. implement several **interfaces**
- a class can both inherit from another class (abstract or not) and implement one or many **interfaces**

Remember a class that you define always inherit from another ? So the third item is obvious.

Interfaces : syntax

```
public interface Rectangle {
        public float getHeight();
        public float getWidth();
}


public interface Colored {
        public String getColor();
}
```

Interfaces : syntax

```
public class ColoredRectangle implements Rectangle , Colored
{
        private String color;
        private float height;
        private float width;

        public String getColor() {
                return color;
        }

        public float getHeight() {
                return height;
        }

        public float getWidth() {
                return width;
        }
}
```

Interfaces : syntax

Depending of the context, if we would like to handle a set of Rectangles, in which ColoredRectangle are a special case, we could write :

```
Rectangle a = new ColoredRectangle();
```

Or in the other case :

```
Colored a = new ColoredRectangle();
```

Upcasting and downcasting

- At runtime, Java will try to treat an instance of a class as an instance of another one.
- Upcasting is to give an actual instance and type it as a super class or interface that is implemented by its class. It's always possible.
- Downcasting is the contrary, i.e. to give an actual instance and type it as a subclass. **It might fail at runtime !**

Upcasting and downcasting : examples

Upcasting is always possible :

```
ColoredRectangle a = new ColoredRectangle();
Rectangle b = (Rectangle) a;
b.getHeight();
b.getWidth();
b.getColor(); //not possible , but the compiler will nicely
    warn you
```

b and a are the same instance, stored at the same place in the memory, but the compiler does not allow the same method calls.

Upcasting and downcasting : examples

Downcasting might fail !

```
//assume that a is a Rectangle you get from elsewhere

ColoredRectangle b = (ColoredRectangle) a; //might fail at
    the runtime !
b.getHeight();
b.getWidth();
b.getColor();
```

Depending of the specific class of a, wether it is a ColoredRectangle or not, Java might fail to downcast it. **Be sure** that a can only be a ColoredRectangle in this context if you write that !

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Motivation

- The basic idea is to have a convenient structure to store several instances of a shared type.
- This type could be a class, an abstract class or an interface.

The basics : tables

- They have a fixed size.
- Elements are identified by an integer.

- Definition :

```
int[] table = {1,2,3};
Animal[] animals = {animal1, animal2};
String[] strings = new String[10];
```

- Access :

```
int value = tableau[0];
String value2 = strings[1];
Animal value3 = animals[0];
```

- Size :

```
int size = value.length;
```

- Modification :

```
table[0] = 42;
```

The basics : tables

- Notice that a **matrix** (2-dimensional table) in no more in Java that a table of tables.
- A such defined matrix is not necessarily square...
- ... or even rectangle.
- There is no privileged dimension : one must choose what will be lines and columns.

```
matrix = new int[5][];
for (int row = 0 ; row < matrix.length ; row++) {
    matrix[row] = new int[10];
}
//or in short
matrix = new int[5][10];
```

Lists

- Their size can be modified after initialization.
- Elements are identified by an integer.

- List is an interface.
- There are several implementations like ArrayList or LinkedList.

- **Quizz :** which is the best ?

```
ArrayList<String> strings = new ArrayList<>();
ArrayList<String> strings = new List<>();
List<String> strings = new List<>();
List<String> strings = new ArrayList<>();
```

Lists

- Definition :

```
List<Animal> animals = new ArrayList<Animal>();
List<Animal> animals = Arrays.asList(animal1, animal2);
```

- Access :

```
Animal animal = animals.get(0);
```

- Size :

```
int size = animals.size();
```

- Modification :

```
animals.add(animal1);
animals.set(42,animal1);
```

Sets

- Their size can be modified after initialization.
- Elements are NOT identified by an integer.
- There is no order.
- Elements can not appear twice : no duplicate

- Set is an interface.
- There are several implementations like HashSet or TreeSet.

- **Quizz :** which is the best ?

```
HashSet<String> strings = new HashSet<>();
HashSet<String> strings = new Set<>();
Set<String> strings = new Set<>();
Set<String> strings = new HashSet<>();
```

Sets

- Definition :

```
Set<String> strings = new HashSet<>();
Set<Animal> animals = new HashSet<>(listOfAnimals);
```

- Access : see later.
- Size :

```
int size = animals.size();
```

- Modification :

```
animals.add(animal1);
```

Maps

- A key – value principle
- Their size can be modified after initialization.
- Elements are NOT identified by an integer but by a key
- There is no order.
- Keys can not appear twice.

- Map is an interface.
- There are several implementations like HashMap or LinkedHashMap.

- **Quizz :** which is the best ?

```
HashMap<User,Integer> scores = new HashMap<>();
HashMap<User,Integer> scores = new Map<>();
Map<User,Integer> scores = new Set<>();
Map<User,Integer> scores = new HashSet<>();
```

Maps

- Definition :

```
Map<User,Integer> scores = new HashMap<>();
Map<Animal,Boolean> zoo = new HashMap<>();
```

- Access :

```
Integer score = scores.get(user1);
```

- Size :

```
int size = scores.size();
```

- Modification :

```
scores.put(user1,42);
```

Summary

type	ordered	fixed size ?	key feature
table	yes	yes	indexed by an integer
list	yes	no	indexed by an integer
set	no	no	no duplicate
map	no	no	indexed by a unique key

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Motivation

- Iterators are just a means to browse a collection.
- In Java, they implement the **Iterator** interface which impose these methods :
  - **hasNext**
  - **next**
  - **remove** (tricky, some implementations do not fully support this one)
- We will not need to explicitly use these methods as Java provide a handier (implicit) way to benefit from them.

Iterate over tables and lists : with an integer

```
String[] table = {"toto","tata","titi"};
for (int i = 0; i < table.length; i++) {
    System.out.println(table[i]);
}


List<Integer> randomNumbers = Arrays.asList({ 4, 8, 15, 16,
    23, 42 });
for (int i = 0; i < randomNumbers.size(); i++) {
    System.out.println(randomNumbers.get(i));
}
```

Iterate over lists, sets and maps : with an iterator

```
for (String str : table){
    System.out.println(str);
}

for (Integer number : randomNumbers){
    System.out.println(number);
}

Set<Animal> animals = new HashSet<Animal>();
for (Animal animal : animals){
    System.out.println(animal.toString());
}

Map<Animal, Food> myMap = new HashMap<Animal, Food>();
for (Entry<Animal, Food> entry : myMap.entrySet()) {
        System.out.println(entry.getKey() + " = " +
            entry.getValue());
}
```

Containers + polymorphism = <3

```
Set<Animal> animals = new HashSet<Animal>();
animals.put(new Dog());
animals.put(new Cat());

for (Animal currentAnimal : animals){
    currentAnimal.speak();
}
```

Reminder : here, **Animal** can be a class (concrete or abstract) or even an interface. **Dog** and **Cat** are concrete classes, so we can use **new**.

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Motivation

- **enum** is a finite set of predefined elements.
- These elements are (by definition) **static** and **final**.
- They are used by the programmer to define a set which will not change during the lifespan of the application.

```
enum Suit {
    SPADES,
    HEARTS,
    DIAMONDS,
    CLUBS ;
}
```

Without enum

```
public class Suit {
    private String name;

    public Suit(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}


Suit spades = new Suit("spades");
Suit hearts = new Suit("hearts");
Suit diamonds = new Suit("diamonds");
Suit clubs = new Suit("clubs");
```

With enum

```
enum Suit {
    SPADES("spades"), HEARTS("hearts"),
        DIAMONDS("diamonds"), CLUBS("clubs");

    private final String name;

    private Suit(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}


Suit spades = Suit.SPADES;
Suit hearts = Suit.HEARTS;
```

Iterate over enum

```
for (Suit suit : Suit.values()) {
    System.out.println(suit.getName());
}
```

Table of contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Containers
5. Iterators
6. Enums
7. Checked exceptions handling

Motivation

- Exceptions are a way to handle unexpected scenarios. (It's the same as **raise** in Python.)
- In Java, exceptions are defined with classes and instances, like almost everything else.
- Some exceptions preexist in Java, and we can add our own.

Examples :

- A required file was not found.
- The program tried to divide by zero.
- The program tried to read the *n*th item of a table which size was *n*.

3-steps principle : 1) define the exception

The key idea is to inherit from the **Exception** class.

```
public class MyException extends Exception {

    private int number;

    public MyException(int number) {
        this.number = number;
    }

    public String getMessage() {
        return "Error "+number;
    }

}
```

3-steps principle : 2) throw the exception

Whenever the situation is not supposed to run this way (i.e. we have detected a condition was not satisfied).

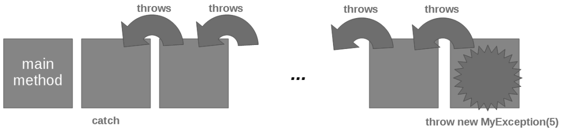
```
public void doSomething() throws MyException {
    // some important stuff

        if(problem) {
                throw new MyException(5);
        }

}
```

3-steps principle : 3) handle the exception

```
try {
    doSomething();
    //we know something wrong could happen
    //the doSomething method might throw a MyException
}
catch (MyException e) {
    //we will deal with this situation in that case
}
```

3-steps principle : 3) handle the exception

Or we can throw the exception again to the method which called us by using the **throws** keyword. I.e. we say we do not know how to deal with this situation and we declare it is calling method’s business to handle it.



Example

```
public static void test(int value) {
    System.out.print("A ");
    try {
        System.out.println("B ");
        if (value > 12) throw new MyException(value);
        System.out.print("C ");
    } catch (MyException e) {
        System.out.println(e);
    }
    System.out.println("D");
}
```

And finally ...

```
try {
    doSomething();
    //we know something wrong could happen
    //the doSomething method might throw a MyException
}
catch (MyException e) {
    //we will deal with this situation in that case
}
finally {
    //what we do in both cases
}
```

RuntimeException

- These exceptions are called "unchecked".
- We do not see them in the **trows** clause.
- They are often bugs which we could not have been handled by a catch clause.

Examples. All these exceptions inherit from RuntimeException :

- ArithmeticException
- ClassCastException
- IllegalArgumentException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

How to use Java ?

Samuel Toubon

Ensai

Table of contents

1 Compilation

2 Integrated development environment (IDE)

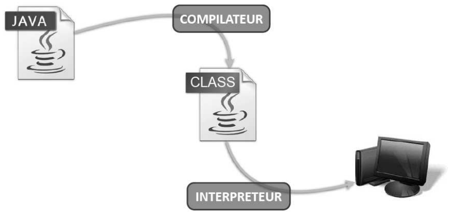
Interpreted vs compiled

- Programming languages are usually interpreted or compiled.
- Compiled languages run on a specific kind of architecture but are fast.
  - C++, ...
- Interpreted languages are portable but slow.
  - Python, PHP...

- Java tries to get the best of both worlds by introducing **bytecode** and **JVM**.

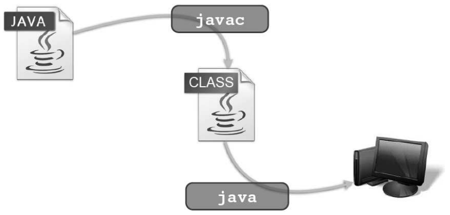
A 2-parts process

- Java source code are plain text **.java** files.
- They are compiled into bytecode, it produces **.class** files. → **Compilation time**
- **.class** files are interpreted by a JVM no matter the specific architecture. → **Runtime**



A 2-parts process

```
class HelloWorld {
    public static void main(String arg[]) {
        System.out.println("Hello world!");
    }
}
```



```
/path/to/javac HelloWorld.java
/path/to/java HelloWorld HelloWorld.class
```

Table of contents

1 Compilation

2 Integrated development environment (IDE)

Meet Eclipse

- Eclipse is a **integrated development environment (IDE)**.
- Wikipedia : *An IDE is a software application that provides comprehensive facilities to computer programmers for software development. An IDE consists of at least a source code editor, build automation tools, and a debugger.*
- Its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages.
- Eclipse also runs Git out of the box.
- Very rich software, not so easy to learn.

Useful shortcuts

- Auto-complete : CTRL+space
- Auto-indent : CTRL+SHIFT+F or CTRL+I
- Refactor : right-clic, refactor...
- Auto-import : CTRL+SHIFT+O
- many more...

Demo

- Auto-complete
- Export a project
- Import a project

How to deal with a real project?

Samuel Toubon

Ensai

Table of contents

1. A real project lifecycle
2. How to deliver something your client can execute ?
3. How to deal with dependencies ?
4. We want tests !

The actors

Who is involved ?

The steps

Who should do what ? When ?

Agile

What is agile software development ?

Table of contents

1. A real project lifecycle
2. How to deliver something your client can execute ?
3. How to deal with dependencies ?
4. We want tests !

Eclipse is for development

- An **IDE** is an integrated **development** environment
- Your client does not have Eclipse
- Your client does not know Eclipse
- Your client does not want Eclipse
- Your client does not know how to use Eclipse

Compile and run

Quick reminder about compilation

Meet JAR

- JAR stands for Java ARchive
- It's sort of a zip containing class files.
- A JAR file car be **runnable**, in which case it contains the name of the class containing the main method.

Meet JAR

Let's try that.

```
/path/to/java -jar jeanmichel.jar
/path/to/java Main -jar jeanmichel.jar
```

Table of contents

1. A real project lifecycle
2. How to deliver something your client can execute ?
3. How to deal with dependencies ?
4. We want tests !

Motivation

- **Dependencies** a.k.a. **libraries** are a way to reuse code from projects to projects.
- The main goal is use code already made by others not to reinvent the wheel.
- Focus only on what makes your project specific.

with JAR

Compilation time :

```
/path/to/javac -cp lib.jar Main.java
```

Runtime :

```
/path/to/java -jar lib.jar Main Main.class
```

with Eclipse

Let's try that.

Some problems

- How to handle dependencies of dependencies (=**transitive dependencies**) ?
- Which version should I use ? How to keep up to date ?
- What if two dependencies have the same dependency in different versions ?

Meet Maven

- Wikipedia : **Maven** is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software : how software is built, and its dependencies.
- Maven is an independent software but works well with Eclipse.
- Notice Ensai-specific configuration before starting.

Meet Maven

- Maven uses a single xml file to describe how your project should be built and what are its dependencies : **pom.xml**
- It has to be this exact name and present at the root of the project.
- Maven only works if you structure well your project using a specific tree.

So, two very important steps :

- have a well formed pom.xml
- have a accurate tree

Tree

```
my-app
|-- pom.xml
‘-- src
    |-- main
    |   ‘-- java
    |       ‘-- com
    |           ‘-- mycompany
    |               ‘-- app
    |                   ‘-- App.java
    ‘-- test
        ‘-- java
            ‘-- com
                ‘-- mycompany
                    ‘-- app
                        ‘-- AppTest.java
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.ensai.mygroup</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0</version>

</project>
```

Add a dependency

Check **mvnrepository.com** to see what is available.

```
<dependencies>
    <dependency>
        <groupId>groupId</groupId>
        <artifactId>artifactId</artifactId>
        <version>version</version>
    </dependency>
</dependencies>
```

Example

```
<dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-math3</artifactId>
        <version>3.6.1</version>
    </dependency>
</dependencies>
```

```
SimpleRegression regression = new SimpleRegression();
regression.addData(1, 2);
regression.addData(2, 3);
regression.addData(3, 4);
System.out.println(regression.getIntercept());
```

http://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/stat/regression/SimpleRegression.html

Table of contents

1 A real project lifecycle

2 How to deliver something your client can execute ?

3 How to deal with dependencies ?

4 We want tests !

Typology

30% to 40% of development time is occupied by tests.

Two very important things : **tools** and **processes**.

What kind of tests can you think of ?

Focus on unit tests in Java

- Unit tests in Java are just like in Python.
- The general principle is **given** a situation (somes variables), **when** I call this specific function, **then** I'm supposed to get this result.

Focus on unit tests in Java

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

Focus on unit tests in Java

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.5.2</version>
    <scope>test</scope>
</dependency>
```

Focus on unit tests in Java : example

```
public class Maths {
    public int addition(int a, int b) {
        return a+b;
    }
}
```

```
public class MathsTest {
    @Test
    public void testAddition() {
        //GIVEN
        int a = 1;
        int b = 2;
        //WHEN
        int c = new Maths().addition(a,b);
        //THEN
        Assert.assertEquals(3,c);
    }
}
```