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

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

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

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

Inheritance : syntax

```java
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** ?

École nationale de la statistique et de l'analyse de l'information

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

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

```java
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.

ENSAI École nationale de la statistique et de l'analyse de l'information

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 ?

```java
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.

```java
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

```java
public class Cat extends Animal {

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

```java
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

ENSAI École nationale de la statistique et de l'analyse de l'information

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

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

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

Interfaces : syntax

```java
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 **Rectangle**s, 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

École nationale de la statistique et de l'analyse de l'information

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);
```

ENSAI École nationale de la statistique et de l'analyse de l'information

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

```java
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

École nationale de la statistique et de l'analyse de l'information

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

```java
public class Suit {
    private String name;

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

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

```java
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;
```

ENSAI École nationale de la statistique et de l'analyse de l'information

Iterate over **enum**

```java
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.

École nationale de la statistique et de l'analyse de l'information

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;
    }

}
```

ENSAI École nationale de la statistique et de l'analyse de l'information

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).

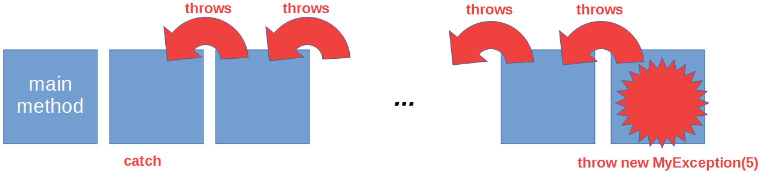
```
public void doSomething() throws MyException {
    // some important stuff

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

}
```

3-steps principle : 3) handle the exception

```java
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

```java
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