# JAVA - Labs information

# LAB01

**Java – elements of OOP (I)**

1. *Concepts of encapsulation, inheritance, and polymorphism*
2. Encapsulation consists in gathering data and methods in a structure while hiding the implementation of the object, i.e. preventing access to data by a means other than the proposed services (methods). Thus, encapsulation allows to protect the data of the object. So the user will not have direct access to the attributes of the class but will have to go through the methods (the interfaces).

To characterize this, in java, we use the private keyword in front of the attributes so that they are never modified by mistake and the public keyword in general in front of the methods so that access to the users is possible.

In our first example called “*HelloEncapsulation*”, we can see that the *propVal* attribute is private and methods are public. There are the getter method for read the attributes and the setter method for modify the attributes values.

1. The getter method is used to simply read the attributes (an accessor) and the setter method is used to modify the attribute values (a mutator).
2. The *this* keyword is a reference variable to reference the actual object in the class. This keyword is always used in the setter methods. It also allows to make the difference between the parameter and the constant attribute.

The *super* keyword is a reference variable to reference an immediate object of the parent class in a child class. This keyword is often used for the inheritance method.

1. Inheritance is a mechanism that allows you to create a new class from an existing class by using its properties and methods. Thus, to define a new class, it is enough to make it inherit from an existing class and to add new properties/methods to it.

With Java, the tree structure is strict, i.e. a given class can only have one superclass. On the other hand, a class can have an unlimited number of subclasses. When a class is instantiated, it inherits all the properties and methods of its superclass, which in turn inherits a possible superclass, etc. Thanks to this organizational system, it is easy to reuse existing components and add behaviour to them. Moreover, the modification of the superclass implies the automatic modification of all the subclasses. Moreover, the inheritance is transitive. It means that an object can have several types.

In Java, we use the *expends* keyword.

In our example, the RichDad class is the mother class (or superclass) and the RichDadsKid class is the daughter class. It means that the RichDadsKid class has all the attributes of RichDad class plus his own attributes.

1. **Ad hoc**

There also is **the overloading** in Java. It means that we can have several methods with the same name but with different parameters.

**Sub-type/inclusion**

1. Sub-type/inclusion polymorphism and inheritance

Polymorphism is the fact of referencing a child class with a referent declared as a parent class. In Java, we will have: TYPE OF THE MOTHER CLASS NAME OF THE OBJECT THAT WE WANT TO CREATE = new METHOD OF THE DAUGHTER CLASS. For example, Animal myDog = new Dog();

Moreover, if we have the same name of methods in the superclass and in the child class, then the method of the child class will be executed.

For example, let’s imagine that we have the meet method in both classes:

Animal myDog = new Dog();

myDog.meet(another one);

The method of the child class will be executed.

1. *Static members (variables/constants and methods)*
2. The *static* keyword can be used as well for a variable as for a class.

We use *static* for a variable that does not change for several objects. For example, the name of the company for the employees. It gets memory only once in the class at loading time, so it saves memory.

If a constant is not static, Java will allocate memory for that constant in each object of the class (i.e. one copy of the constant per object). If a constant is static, there will be only one copy of the constant for that class (i.e. one copy per class). Therefore, if the constant has only one value, it must be declared as static. If the constant can have a different value for each object, for example the time of creation of the object, it should not be declared static

If *static* is used for a method, the method belongs to the class and not to the object. So we can call a static method without an instance of a class. It can access static members and can modify their value.

Une méthode statique est utile lorsqu’il y a qu’un seul objet à créer, contrairement aux non-statique.

(If static is used for a class, the class can’t be instantiated or inherited, and all of its members are static in nature.)

1. Most static variables are declared public since they must be available for users of the class.
2. A static method cannot access instance variables and instance methods of a class because a static method can be called even if no object of the class has been instantiated. For the same reason, the *this* reference cannot be used in a static method.
3. Example of one application of static method: to calculate a total area of a shape.

*3) Constructors, factory methods, and singletons*

1. Here, there are the steps for the object initialisation process. First, we need to build our constructors in order to instantiate our objects in the main file. If we don’t do that, there will be a default constructor but once a constructor is made, the default one disappears. Here we have two constructors: public HelloObjectInit(double pdfd1) and public HelloObjectInit(). Then, we need to create the anonymous blocks and methods and static variables and static constants.

2. D9 extends D1 and D1 extends B1

3. Constructors and factory methods are two methods to instantiate objects. The constructor is called when an object of a class is created. It can be used to define the initial values of the attributes of an object. As for the factory method, it allows to instantiate objects whose type is derived from an abstract type. The exact class of the object is therefore not known by the caller.

*A factory method - a static method that returns new instances of the class*

1. **The singleton pattern is a method whose class can have only one object (an instance of the class) at a time**.

Application examples: a prof of the only one class, a god class.

1. *Immutable objects/classes and Java Records*
2. An immutable object is an object that will not change its internal state after creation. Therefore, to have an immutable object, we need to:

* Do not add any definition method (a definition method consists in modifying the internal value of a field)
* Declare all final and private fields
* If a field is a mutable object, create defensive copies for the getter methods
* If a mutable object passed to the constructor must be assigned to a field, create a defensive copy of it
* Do not allow subclasses to override methods

1. To do an immutable class, we need to have:

* The class should be marked as final
* All fields must be private and final
* Replace setters by constructor (to assign a value to a variable).

Therefore, in both cases, "immutable" means "will not change". For an object, it will not change after instantiation and for a class, it cannot be inherited.

1. They are very useful in multithreaded applications because they can be shared between threads without synchronization. In addition, they are consistent condition, automatic thread safety and simplicity.

*A Java Record is a special kind of Java class which has a concise syntax for defining immutable  
\* data-only classes.*

Examples of applications:

1. *Overriding hashCode, equals, and toString*
2. In Java, == only compares two references (i.e. for non-primitives), i.e. it checks whether the two operands refer to the same object (pointing to the same object).

However, the equals method can be overridden - so two distinct objects can still be equal (comparing the values).

Examples:

1. myAccount.equals(yourAccount) est true car ils ont le **même valeur**, mais
2. myAccount == yourAccount est false car ils ne sont pas les **même compte**.
3. This formula means that if 01.equals(02) method then hasCode(01) == hashCode(02), so, two unequal objects may have the same hash code.
4. The Object class is the parent class of all classes in Java by default. In other words, it is the highest class in Java.

The Object class is useful if you want to reference an object whose type you don't know. Note that the reference variable of the parent class can refer to the object of the child class, called upward conversion.

Let's take an example, there is a getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use the Object class reference to reference that object.

1. **The general contract of hashCode and equals is**:

Whenever called multiple times on the same object during the execution of a Java application, the hashCode method must always return the same integer, provided that no information used in equal comparisons of the object is changed. This integer does not need to remain consistent from one run of an application to another run of the same application.

If two objects are equal according to the equals(Object) method, calling the hashCode method on each of the two objects must produce the same integer result.

It is not mandatory that if two objects are not equal according to the equals(Java.lang.Object) method, calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects can improve the performance of hashtables.

hashcode()- Returns a unique integer value for the object at runtime. By default, the integer value is primarily derived from the object's memory address in the heap (but this is not always required). This hash code is used to determine the location of the compartment, when this object is to be stored in a data structure of type HashTable .

# Lab02

## Final variables, methods, and classes

* 1. **Final keywords:** element cannot be changed in the rest of the program

**Local Variables:** to create constant local variables

**Instance variable:** can be explicitly initialized only once

**Static constant:** the best way to create a constant variable in Java

**Methods:** cannot be redefined in a derived class (overriding)

**Classes:** cannot be derived, inherited

1. Benefits of using constants in programming: avoid errors, save memory, very clear

* Not only. To have an immutable class, we need to:
* No setter method
* All fields as private and primitive types
* Final class
* Yes

1. Because de class is not final, the get method is in public and not private
2. TO DO

## Enumeration classes (enum)

*An enum class is a class declared with abbreviated syntax that defines a small set of named class instances.  
A classic example of an enumerated type that defines a list of enumerated values corresponding to the seasons of the year.*

INSTANCE est l'élément enum unique réel qui est l'instance singleton

## Nested classes (une classe dans une autre)

Nested classes are divided into two categories: non-static and static. Non-static nested classes are called *inner classes*. Nested classes that are declared static are called *static nested classes*.

Une classe imbriquée est membre de sa classe englobante. Les classes imbriquées non statiques (classes internes) ont accès aux autres membres de la classe englobante, même si elles sont déclarées privées. Les classes imbriquées statiques n'ont pas accès aux autres membres de la classe englobante. En tant que membre de la OuterClass, une classe imbriquée peut être déclarée private, public, protectedou *empaqueter privé* . (Rappelez-vous que les classes externes peuvent uniquement être déclarées publicou *package private* .)

## Pourquoi utiliser des classes imbriquées ?

Les raisons impérieuses d'utiliser des classes imbriquées sont les suivantes :

* **C'est une façon de regrouper logiquement des classes qui ne sont utilisées qu'à un seul endroit** : si une classe n'est utile qu'à une seule autre classe, alors il est logique de l'intégrer dans cette classe et de garder les deux ensemble. L'imbrication de telles "classes d'assistance" rend leur package plus rationalisé.
* **Cela augmente l'encapsulation** : Considérez deux classes de niveau supérieur, A et B, où B a besoin d'accéder aux membres de A qui seraient autrement déclarés private. En cachant la classe B dans la classe A, les membres de A peuvent être déclarés privés et B peut y accéder. De plus, B lui-même peut être caché du monde extérieur.
* **Cela peut conduire à un code plus lisible et maintenable** : l'imbrication de petites classes dans des classes de niveau supérieur rapproche le code de l'endroit où il est utilisé.

Pour instancier une classe interne :

1. On instance la classe externe

OuterClass externalObject = new OuterClass();

1. Créer l’objet interne dans l’objet externe

OuterClass.InnerClass innerObject = externalObject.new InnerClass();

Il existe deux types particuliers de classes internes : [les classes locales](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html) et [les classes](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html)[anonymes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html) .

1. Classe locale : une classe dans une méthode. Elle a accès aux membres de sa classe englobante, aux variables locales mais seulement finales.
2. Classes anonymes : Il est possible de définir une classe interne, sans lui donner de nom par dérivation d’une super classe, ou par implémentation d’une interface.

Une classe anonyme peut être déclarée dans une méthode, comme une classe locale, ou comme classe membre.

La différence est qu'elle ne porte pas de nom, qu'elle utilise un mécanisme de déclaration qui se fusionne avec son instanciation.

Ex :

Comparator comparator = **new** Comparator() {

//methods…

}

//la classe Comparator n’existe pas

1. Une classe interne ne peut pas être utilisée dans une autre classe que dans celle où elle est imbriquée.

Yes

## Abstract Data Types

Une classe abstraite et une interface ne peuvent pas être instanciées.

Interface : implements keyword, utilisé pour implémenter une classe, toutes les méthodes sont abstraites, aucune méthode comporte du code

Classe abstraite et interface :

Une image contenant texte

Description générée automatiquement

1. abstract class E { abstract void m1() ; }
2. interface D { void m1(); }

# Lab03

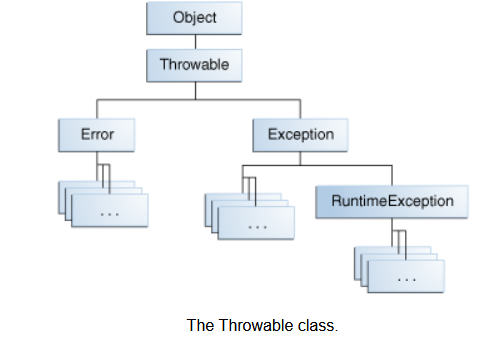
* 1. **Java exceptions hierarchy**

Une exception est un événement qui se produit pendant l'exécution d'un programme et qui perturbe le flux normal des instructions.

<https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>

Une image contenant texte

Description générée automatiquement



**Error class** : dynamic pb

**Exception class :** erreurs pas importantes qui ne posent pas de pb pour run correctement. Une sous-classe d'exception, RuntimeException, est réservée aux exceptions qui indiquent une utilisation incorrecte d'une API. Un exemple d'exception d'exécution est NullPointerException, qui se produit lorsqu'une méthode tente d'accéder à un membre d'un objet par le biais d'une référence nulle.

1. **Throws, throw, try-catch, try-finally, try-catch-finally**

**try (Statement stmt = con.createStatement())**

In Java exceptions under Error and RuntimeException classes are **unchecked exceptions**, everything else under throwable is checked.

*Unchecked exceptions do not need to be declared in a method or constructor's throws clause  
\* if they can be thrown by the execution of the method or constructor and propagate  
\* outside the method or constructor boundary.*

Checked: sont les exceptions qui sont vérifiées au moment de la compilation. Si un code d'une méthode lève une exception vérifiée, la méthode doit soit gérer l'exception, soit la spécifier à l'aide du mot-clé throws.

Unchecked : sont les exceptions qui ne sont pas vérifiées au moment de la compilation. En Java, les exceptions des classes Error et RuntimeException sont des exceptions non vérifiées, tout le reste de la catégorie throwable est vérifié. ON NE PEUT PAS LES CATCH

**throw** keyword is used inside a function. It is used when it is required to throw an Exception logically.

throw new IOException();

**throws** keyword is in the function signature. It is used when the function has some statements that can lead to some exceptions.

void Demo() throws ArithmeticException, NullPointerException

{

// Statements where exceptions might occur.

}

The try statement allows **you to define a block of code to be tested for errors while it is being executed**. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

La clause **finally** définit un bloc qui sera toujours exécuté, qu'une exception soit levée ou non.

<https://www.jmdoudoux.fr/java/dej/chap-exceptions.htm#exceptions-1>

1. **try-with-ressources and autoCloseable**

try-with-ressources: statement qui peut déclarer une ou plusieurs ressources. Une ressource est un objet qui doit être fermé après que le programme est fini avec lui. The try-with-ressources assure que chaque ressource soit bien fermée à la fin du statement.

autoCloseable: The AutoCloseable interface is used for resources that need to be closed after they are not needed anymore. The close() method from this interface is called automatically by JVM when exiting a [try-with-resources](https://frontbackend.com/java/using-try-with-resources-statement-in-java-7) block.

1. Simple file

**bufferedReader:** lit le texte à partir d'un flux d'entrée de caractères, en mettant les caractères en mémoire tampon (**buffering**) afin de permettre une lecture efficace des caractères, des tableaux et des lignes.

**bufferedWriter:** Écrit du texte dans un flux de sortie de caractères, en mettant les caractères en mémoire tampon afin de permettre l'écriture efficace de caractères uniques, de tableaux et de chaînes.

**fileReader:** lit les fichiers de caractères, étend InputStreamReader.

**fileWriter:** écriture dans une fichier, étend OutputStreamWriter

**printWriter:** Imprime des représentations formatées d'objets dans un flux de sortie de texte.

**fileInputStream:** est destiné à la lecture de flux d'octets bruts tels que des données d'image, étend InputStream

**fileOutputStram:** est destiné à l'écriture de flux d'octets bruts tels que des données d'image, étend OutputStream.

**dataInputStream:** permet à une application de lire des types de données Java primitifs à partir d'un flux d'entrée sous-jacent d'une manière indépendante de la machine., étend FilterInputStream et implémente DataInput.

**dataOutputStream:**  permet à une application d'écrire des types de données Java primitifs dans un flux de sortie de manière portable.

**objectInuputStream:** désérialise (sélectionne des données sépérées afin de les réunir au sein d’un méme flux) les données primitives et les objets précédemment écrits à l'aide d'un ObjectOutputStream, étend InputStream, implémente ObjetInput

**objectOutputStream:** writes primitive data types and graphs of Java objects to an OutputStream.

**files:** Cette classe se compose exclusivement de méthodes statiques qui opèrent sur des fichiers, des répertoires ou d'autres types de fichiers, étend la classe Object

**path:** Comme son nom l'indique, la Path classe est une représentation programmatique d'un chemin dans le système de fichiers. Un Path objet contient le nom de fichier et la liste de répertoires utilisés pour construire le chemin, et est utilisé pour examiner, localiser et manipuler des fichiers.

**file:** The File class from the java.io package, allows us to work with files.

1. **Java marker interfaces**

**Serialization:** est un mécanisme de conversion d’état d’objet en bytes stream (séquence de bytes)

**Cloneable :** les objets de la classe peut être clonés. La classe peut réécrire la méthode clone() héritée de la classe objet afin que ses instances puissent procéder à une copie de ses attributs vers une nouvelle instance (copie de l’objet par valeur).

## Lab04