Cours 2 : Java - POO

Mardi 1er février 2022

Mathéo Allard (mallard@oxyl.fr)

Ulysse Coscoy (ucoscoy@oxyl.fr)

Guillaume Da Silva (gdasilva@oxyl.fr)

Alexander van Dalen (avan-dalen@oxyl.fr)

Excilys
Développeurs de passion

SOMMAIRE

I. Plain Old Java Object
II. Enum
III. Héritage
IV. Classes Abstraites
V. Interfaces
VI. Polymorphisme
VII. Collections
VIII. Généricité
IX. Exceptions

I. Plain Old Java Object

A. L’encapsulation
B. Mise en pratique
C. Méthode de classe VS Méthode d’instance

L'encapsulation

L'encapsulation est le mécanisme qui permet de gérer **l'accessibilité** des attributs et des méthodes dans une classe.

4 niveaux d'accessibilité en Java :

Modificateur du membre	**private**	***aucun (default)***	**protected**	**public**
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui

Plain Old Java Object

Mise en pratique

```
class Personne {

 private String name, surname;
 private static int numberOfPersonne = 0;

 public Personne(String name, String surname) {
   this.name = name;
   this.surname = surname;
   numberOfPersonne++;
 }

 public Personne(String nom) {
   this(name, "Inconnu");
 }

 public static int getNumberOfPersonne() {
   return numberOfPersonne;
 }

 public String getName() {
   return name;
 }
 public String getSurname() {
   return surname;
 }

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

 public void setPrenom(String surname) {
   this.surname = surname;
 }

}
```

Plain Old Java Object

méthode d'instance

```
 // Dans la Classe Entreprise
public String getName() {
   return name;
}
```

Méthode de classe VS d'instance

méthode de classe

```
// Dans la Classe Entreprise
public static int getNumberEmployee() {
   return numberOfPersonne;
}
```

```
public class Entreprise {
    private String name;
    private static int numberOfEmployee;
    // Le code des constructeurs
    public static void main(String[] args) {
        Entreprise entreprise = new Entreprise();
        entreprise.getName(); // Accès à une méthode d'instance
        Entreprise.getNumberOfPersonne(); // Accès à une méthode de classe
    }
}
```

Méthode de classe VS d'instance

- Une instance d'une classe a accès au champ de classe (static)
- Les variables et méthodes d'instances sont inaccessibles via le nom de classe.

méthode d'instance

```
 // Dans la Classe Entreprise
public String getName() {
   numberOfPersonne += 1;
   Entreprise.getNumberOfPersonne(); // Warn
   return name;
}
```

méthode de classe

```
// Dans la Classe Entreprise
public static int getNumberEmployee() {
   name = name + "$"; // Compilation Err
   return numberOfPersonne;
}
```

II. Enum

Enum

Définition rapide : un ensemble fini de valeurs (constantes)

Intérêt : Plus de clarté dans le code

```
public enum FeuTricolore {
  VERT, ORANGE, ROUGE
}
```



```
public class FeuTricolore {
  public static final int VERT = 0;
  public static final int ORANGE = 1;
  public static final int ROUGE = 2;
}
```

Excilys
Développeurs de passion

Enum

Les **Enum** offrent [plus de possibilités](#) mais nous nous limiterons à l'essentiel.

FeuTricolore.java

```java
public enum FeuTricolore {
    VERT, ORANGE, ROUGE
}
```

TestEnum.java

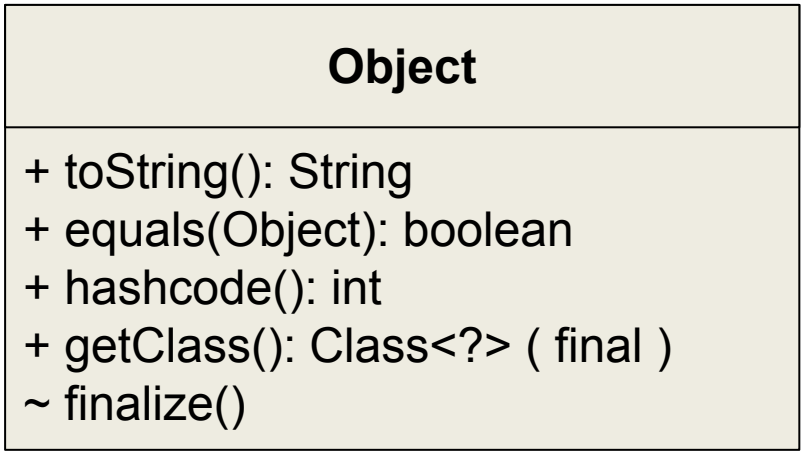
```java
public class TestEnum {
    Run | Debug
    public static void main(String[] args) {
        FeuTricolore vert = FeuTricolore.VERT;
        System.out.println(vert); // VERT
        System.out.println(vert.ordinal()); // 0
        for(FeuTricolore feu : FeuTricolore.values()) // VERT, ORANGE, ROUGE
            System.out.print(feu + ", ");
        System.out.println();
        switch(vert) {
            case ROUGE:
                System.out.println("STOP!");
                break;
            default:
                System.out.println("Passez...");
        }
    }
}
```

III. Héritage

A. La classe Object
B. Hériter d'une classe
C. *super*

La classe Object

Classe mère de toutes les Classes



Hériter d'une classe

mot clé : *extends*

contrainte : pas d'héritage multiple

MaClasse.java

```
1   class MaClasse extends Object {}
```

intérêt : accès et extensions des classes parentes

/! redéfinition de méthodes ≠ surcharge de méthodes

```
class MaClasse extends Object {
    @Override
    public String toString() {
        return "toto";
    }
}
```

Le mot clé *super*

Classe *Mère*

```
class Maman {
    protected String s;
    public Maman(String s) {
        this.s = s;
    }
    public String toString() {
        return s;
    }
}
```

Classe *Fille*

```
class Fille extends Maman {
    public Fille(String s) {
        super(s);
    }
    public String toString(String s) {
        return super.toString() + s;
    }
    @Override
    public String toString() {
        return super.toString() + s;
    }
```

IV. Classes Abstraites

A. Syntaxe
B. Cadre d'utilisation

Une **classe abstraite** est une classe incomplète, elle doit être marquée du mot-clé **abstract**.

- Une classe abstraite est une classe qui **ne** peut **pas** être instanciée.
- Une classe abstraite peut contenir des méthodes déjà implémentées.
- Une classe abstraite peut contenir des méthodes **non** implémentées.
- Pour pouvoir construire un objet à partir d'une classe abstraite, il faut créer une classe non abstraite qui hérite de cette classe et qui implémente **toutes** les méthodes **non** implémentées.

Classe abstraite

Cadre d'utilisation

```
abstract class Vehicule {
 private String brand, name;
 public String getName() {
   return name;
 }
 public String getBrand() {
   return brand;
 }
 public void setName(String name) {
   this.name = name;
 }

 public void setBrand(String brand) {
   this.brand = brand;
 }

 public abstract void start();
}
```

Classe abstraite

Cadre d'utilisation

```
class Car extends Vehicule {
 public void start() {
   // TODO...
 }

 public static void main(String[] args) {
   Car v1 = new Car();

   Vehicule v2 = new Vehicule() {

     public void start() {
       // ....
     }

   };

 }
}
```

Excilys
Développeurs de passion

V. Interfaces

A. Syntaxe
B. Cadre d'utilisation
C. Interface fonctionnelle
D. Multi-Héritage

Interfaces

Syntaxe

CanDive.java

```
public interface CanDive {
    public static int maxDepth = 5000;
    public void dive();
    public void floatBack();
    default void showMaxDepth() {
        System.out.println("Max depth: " + maxDepth);
    }
}
```

```
[excilys@Excilys] ~/excilys $ javac CanDive.java
```

- Ne peut pas être instanciée directement
- Une interface peut être vide
- Une interface ne peut pas être private, protected ou final
- Uniquement des champs **static final**
- Les méthodes sont **public abstract**
- Autorise des méthodes **default**

Cadre d'utilisation

```java
public interface CanDive {
    public void dive();
    public void floatBack();
}
```

```java
CanDive = new CanDive();

   Exception in thread "main" java.lang.Error:
   Unresolved compilation problems:
       CanDive cannot be resolved
       Cannot instantiat                    anDive
       at Subma                    ubmarine.java:15)
```

Don't do this

```java
public class Submarine implements CanDive {

    public void dive() {
        System.out.println("Is going down to the deep ...");
    }

    public void floatBack() {
        System.out.println("Is going up to the surface ...");
    }

    public static void main(String[] args) {
        Submarine submarine = new Submarine();
        submarine.dive();
        submarine.floatBack();
    }
}
```

Interface fonctionnelle

Java 8 introduit le concept d'interface fonctionnelle qui permet de définir une interface qui ne contient qu'**une seule** méthode abstraite.

```
@FunctionalInterface
public interface Addition {
 public abstract int addition(int x, int y);
}
```

```
class Mathematique {

  public static int getResultat(Addition interfaceAddition, int x, int y) {
      return interfaceAddition.addition(x, y);
  }

  public static void main(String[] args) {
      if (args.length < 2) System.exit(1);
      int x = Integer.parseInt(args[0]);
      int y = Integer.parseInt(args[1]);
      Addition interfaceAddition = (a, b) -> a + b;
      int resultat = getResultat(interfaceAddition, x, y);
  }
}
```

Multi-Héritage

```java
interface A { default void fA(){} }
interface B { void fB(); }
class AB implements A, B {
    public void fB(){};
}
//...
Run | Debug
public static void main(String[] args) {
    AB ab = new AB();
    A a = ab;
    B b = ab;
    a.fA(); b.fB();
    ab.fA(); ab.fA(); ab.fB();
}
```

Une interface pouvant étendre une interface...

VI. Polymorphisme

A. Principe
B. Exemples
C. Les classes *Wrapper*

polymorphisme d'héritage ≠ surcharge

Principe :

Pouvoir associer l'instance d'un objet à une variable d'un type "*lié*" (ancêtre selon l'arborescence de l'héritage, interfaces implémentées).

```
Run | Debug
 8      public static void main(String[] args) {
 9          Object o = "s";
10          System.out.println(o.getClass().getName());
11      }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
java.lang.String
```

Une variable de type Object référençant un objet de type String

Principe

Principe :

L'instance est manipulée selon le type de la variable.

Les méthodes d'instances sont toujours invoquées selon le type de l'instance.

~~*o.length()*~~

```
Run | Debug
public static void main(String[] args) {
    Object o = "s";
    System.out.println(o.getClass().getName());
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
java.lang.String
```

Une variable de type Object référençant un objet de type String

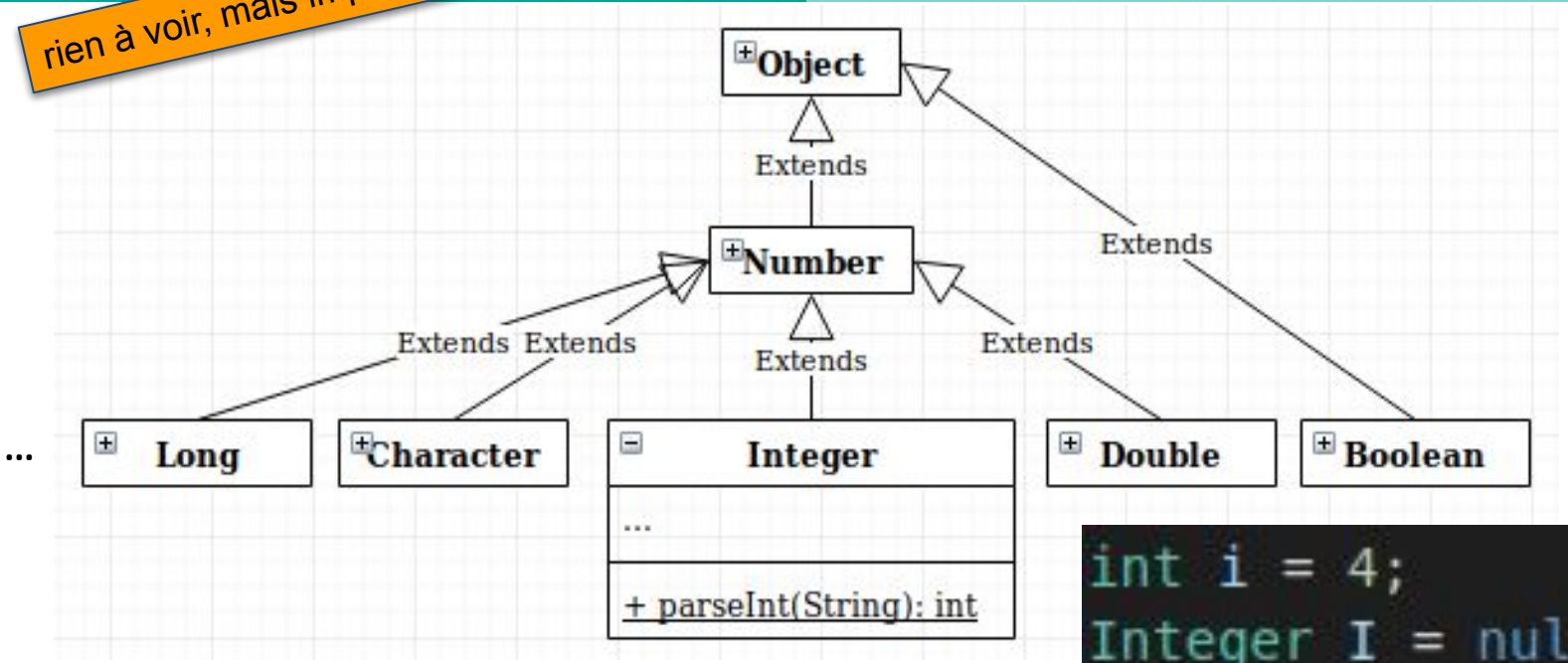
Exemples

```
1   class Animal {
2       String nom;
3       Animal(String nom) { this.nom = nom; }
4       String getNom() { return nom; }
5       public String toString() { return this.getNom(); }
6   }
7   class Girafe extends Animal {
8       Girafe() { super("Girafe"); }
9       @Override String getNom() { return nom.toLowerCase(); }
10  }
11  class Lion extends Animal {
12      Lion() { super("Lion"); }
13      String getNom() { return super.nom.toUpperCase(); }
14  }
15  Animal[] animals = { new Lion(), new Girafe() };
16  for(Animal animal : animals)
17      System.out.println(animal);
```

```
LION
girafe
```

rien à voir, mais important

Les classes *Wrapper*




```
int i = 4;
Integer I = null;
I = i;
i = I;
Long L = i;
```

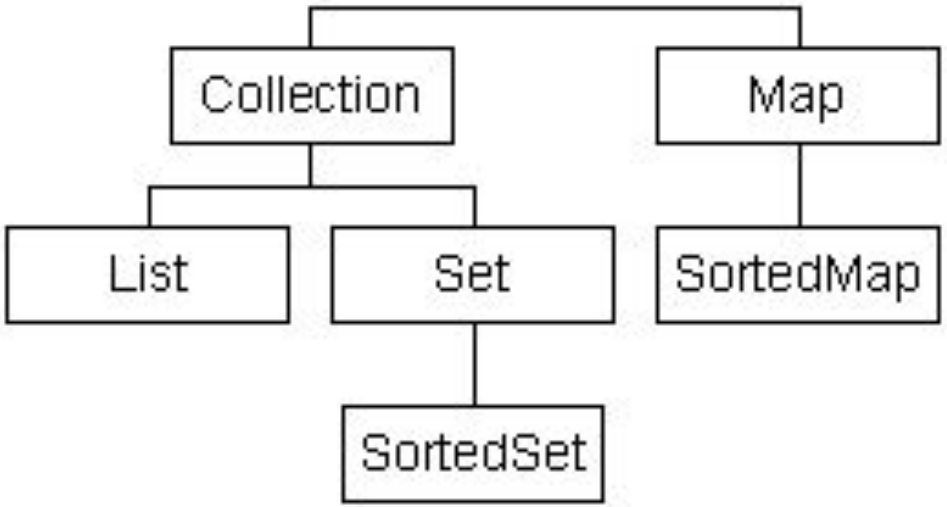
VII. Généricité

Généricité

```cpp
#include <iostream>
template <class T>
T min (T a, T b)
{
    return ((a < b)? a : b);
}

int main()
{
    std::cout << min(10, 20) << std::endl;
    std::cout << min(10.0, 25.0) << std::endl;
    std::cout << min('x', 'f') << std::endl;
}
```

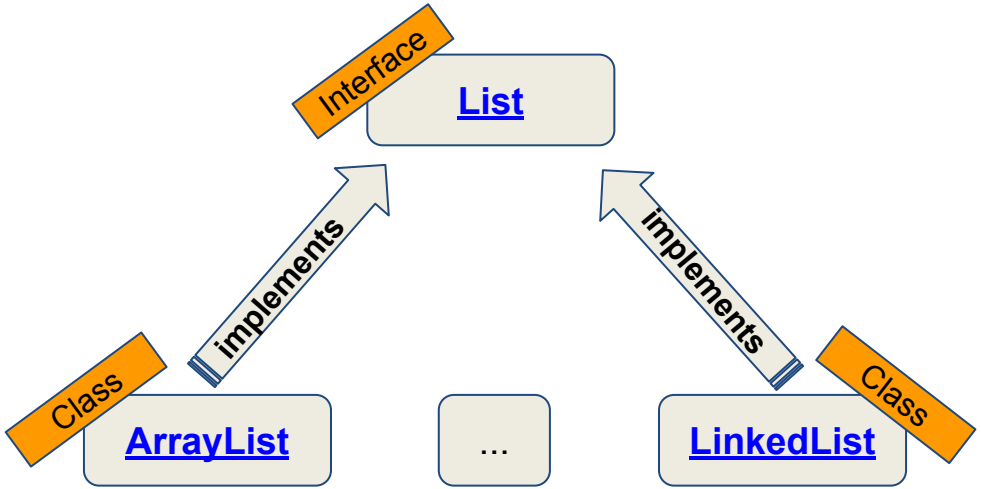
```java
class MyArrayList<T> {
    T[] array;
    final static int CAPACITY = 10;
    MyArrayList(){
        array = (T[])new Object[CAPACITY];
    }
    void set(T value, int index) {
        if(index >= 0 && index < CAPACITY)
            array[index] = value;
    }
    T get(int index) {
        return array[index];
    }
    public static void main(String[] args) {
        MyArrayList<String> l= new MyArrayList<>();
        l.set("toto", 0);
        System.out.println(l.get(0));
    }
}
```

VIII. Collections



List

Une liste est une collection **ordonnée** d'éléments qui permet d'avoir des **doublons**.
L'accès à un élément d'une liste se fait via un index (int).



List - ArrayList

Création d'un ArrayList :

```
En Java : ArrayList<Integer> alist = new ArrayList<Integer>();
```

```
En C++ : vector<int> g1;
```

Quelques Méthodes :

```
class ArrayListExample {

 public static void main(String[] args) {
      ArrayList<String> alist=new ArrayList<>();
      alist.add("Steve"); // Boolean
      alist.add(1, "John"); // Void
      alist.contains("John"); // Boolean
      alist.isEmpty(); // Boolean
      alist.size(); // Entier
      alist.remove("Jhon"); // Boolean
      alist.remove(0); // Élément retiré
 }
}
```

List - LinkedList

Implémentation **doublement chaînée** de l'interface List

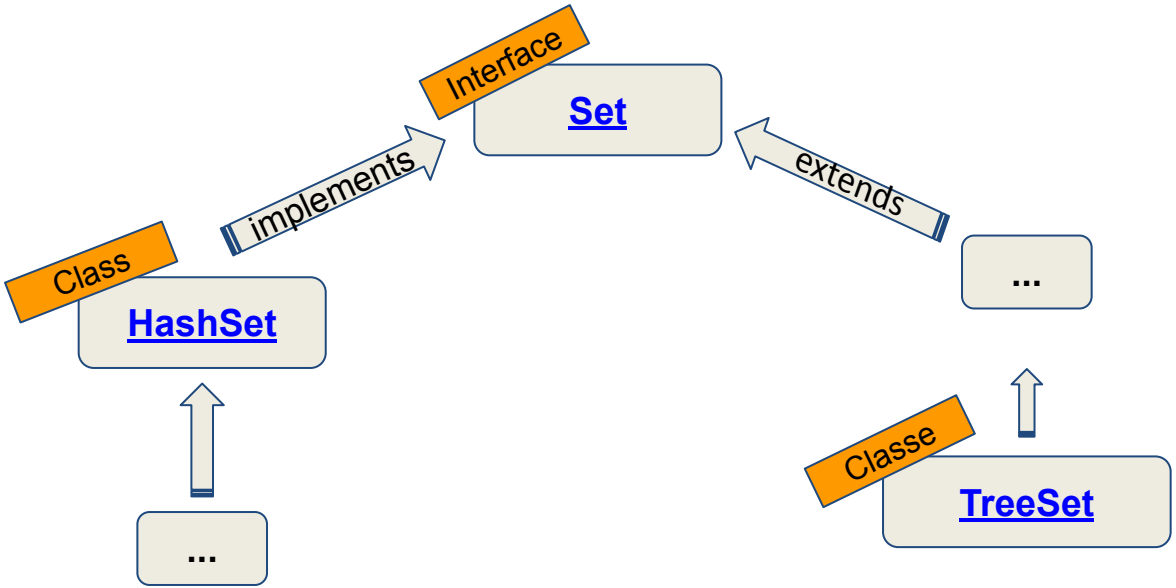
```
En C++ : list <int> listInt;
```

Quelques Méthodes :

```
class LinkedListExample {

 public static void main(String[] args) {
     LinkedList<String> linkedlist = new LinkedList<String>();
     linkedlist.add("Item1");
     System.out.println("Linked List Content: " +linkedlist);

     linkedlist.addFirst("First Item");
     linkedlist.addLast("Last Item");

     Object firstvar = linkedlist.get(0);
     linkedlist.set(0, "Changed first item");

     linkedlist.removeFirst();
     linkedlist.removeLast();

     linkedlist.add(0, "Newly added item");
     linkedlist.remove(2);
 }
}
```

Set

L'interface **Set** permet de créer des ensembles d'éléments uniques.



Set - HashSet

Une implémentation de Set basée sur le principe des hashtables.

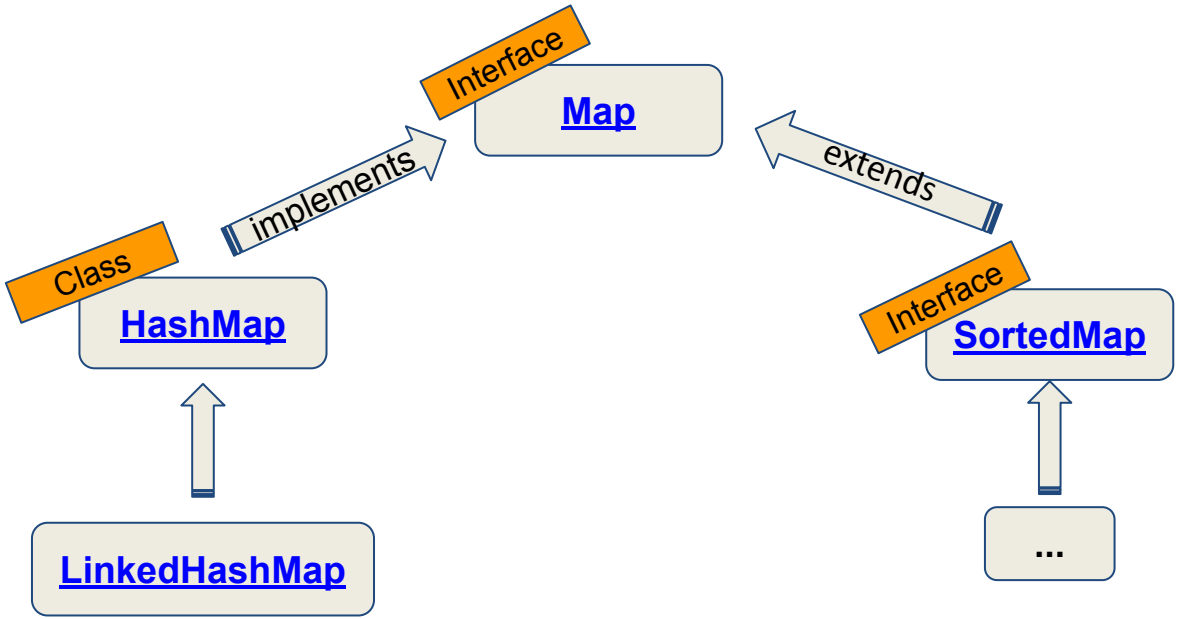
```
Set<Integer> naturalNumber = new HashSet<>();
countries.add(1);       // boolean
countries.size();       // int
countries.remove(1);    // boolean
countries.clear();      // void
countries.isEmpty();    // boolean
```

- L'ordre d'insertion n'est pas préservé.
- Autorise l'élément **null**
- Utilise la méthode hashCode()

Il existe aussi la classe TreeSet qui, elle, préserve l'ordre des insertions.

Map

L'interface Map permet de créer des **associations** [clés : valeur]. Une Map ne peut pas contenir deux clés identiques. Une clé est associée à **une seule** valeur.
Il existe différentes implémentations de l'interface Map :



Map

HashMap

Une implémentation de Map basée sur le principe des hashtables.

```
Map<String, String> book = new HashMap<>();
book.put("Eric", "0628272625");
book.size();
book.get("Eric");
book.remove("Eric");
book.clear();
book.isEmpty();
```

```
Collection<String> values = book.values();
for(String value : values) {
    System.out.print(value + " ");
}
```

```
Set<String> keys = book.keySet();
for(String key : keys) {
    System.out.print(key + " ");
}
```

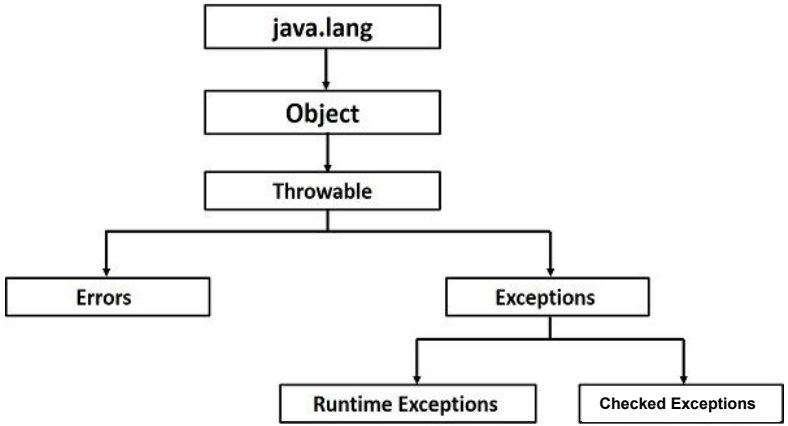
```
System.out.println(book);
```

```
output : {Sophie=0629303132, Eric=0628272625}
```

IX. Exception

A. Syntaxe
B. Cadre d’utilisation

Exception



Syntaxe

throws : A mettre dans la signature d'une méthode

throw : permet de retourner une exception

```
try {
    // code
} catch (Exception e) {
    // traitement de l'erreur
} finally {
    // Faire autre chose
}
```

Syntaxe

```
// Dans une méthode :
public void checkIfSorted() throws Exception {
    throw new Exception("Avec un message");
}
```

```
public void f(){
    try {
        checkIfSorted();
    } catch (Exception e) {
        System.out.println(e);
    } finally {
        // Le bloc finally est optionnel
        // Le code ici sera toujours exécuté
    }
}
```

Toute méthode qui retourne une exception héritant de la classe Exception (ainsi que Exception elle même) doit spécifier dans sa signature la classe d'exception retournée grâce au mot clé throws.

Cependant les exceptions qui héritent de RuntimeException (ainsi que RuntimeException elle même) n'ont pas besoin d'être spécifiées dans la signature de la fonction, ni d'être rattrapées.

Cadre d'utilisation

- Il est possible de rattraper une exception dans un bloc catch via le nom de cette exception, ou via le nom d'une de ses classes mères

- Les blocs catch doivent être placés par niveau de précision

Cadre d'utilisation

```
public class MonException extends Exception {
    public MonException() {
        super();
    }
    public MonException(String message) {
        super(message);
    }
}
```

Cadre d'utilisation

```
// Dans une classe quelconque
public void methode() throws Exception {
    throw new MonException("Avec un message");
}
```

Cadre d'utilisation

```
// Dans cette même classe quelconque
public void f(){
    try {
        methode();
        // Lance une exception "MonException"
        // A dans sa signature le type "Exception"
    } catch (MonException e) {
        System.out.println(e);
    } catch (Exception e) {
        System.out.println(e);
    }
}
```