Chapitre 4

Etude détaillée de la notion de classe
et d'interface Java

1. Classe et autres caractéristiques
2. La notion d’interface
3. Généricité

1 Classe et autres caractéristiques

2 La notion d’interface

3 Généricité

Classe et autres caractéristiques

Rappel : dans le chapitre précédent, nous avons étudié :

- Comment développer une classe avec ses membres : attributs et méthodes
- Comment créer un objet à partir d'une classe : instanciation et le mot clé *new*
- Les variables d'instances et de classe
- Les modificateurs de visibilité qui influencent l'accessibilité des membres d'une classe

A venir : dans ce chapitre nous étudions d'autres caractéristiques enrichissant la maîtrise de ce concept :

- Les constructeurs d'objets
- Le mot clé null
- Le destructeur d'objets
- La notion de référence d'objets
- La portée des variables dans un classe
- Les méthodes et classes abstraites

1 Constructeur (1/3)

Constructeur par défaut
(constructeur sans paramètres)

Deux autres constructeurs
(constructeur avec paramètres)

NB : on peut définir autant de constructeurs que l'on veut

```
public class Personne {

    private String nom;
    private String prenom;
    private int age;
    private String statutMatrimonial;

    public Personne() { }

    public Personne(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
    }

    public Personne(String nom, String prenom, int age, String statutMatrimonial) {
      this.nom = nom;
      this.prenom = prenom;
      this.age = age;
      this.statutMatrimonial = statutMatrimonial;
    }

    //méthodes
}
```

2 Constructeur (2/3)

➔ Lors de l'instanciation d'un objet, l'opérateur ***new*** invoque une méthode spéciale appelée **constructeur**

Exemples :

```
Personne pers = new Personne();1
Personne patrick = new Personne("Durant", "Partick");2
Personne leo = new Personne("Carré", "Léo", 21, "Célibataire");3
```

➔ Le rôle d'un constructeur est de permettre à la JVM de créer et charger un objet dans la mémoire (la RAM) de l'ordinateur lors de l'instanciation d'une classe. Il n'est donc appelé qu'une seule fois lors de la création de l'objet

➔ Sert à initialiser les données membres de la classe et à réaliser toute sorte d'actions que le concepteur de la classe souhaite voir s'exécuter lors de la création de l'objet. Exemples des constructeurs avec paramètres de la slide 5 et les instanciations 2 et 3

➔ Un constructeur :

- porte le même nom que la classe dans laquelle il est défini
- n'a pas de type de retour (même pas *void*)
- peut avoir des paramètres, *cf.* 2 et 3
- n'est pas obligé d'être défini lorsqu'il n'est pas nécessaire; dans ce cas, Java octroie à la classe un constructeur par défaut qui ne prend pas de paramètres (*cf.* constructeur par défaut slide 5) et qui s'instancie comme en 1
- si plusieurs constructeurs sont définis dans une classe, il faut explicitement définir un constructeur sans paramètres si l'on veut créer des objets le plus simplement possible comme sur l'exemple 1

2 Constructeur (3/3)

➔ Deux modificateurs de visibilité

- public : le constructeur peut être appelé de n'importe où dans les autres classes qui l'instancie
- private : le constructeur ne peut être appelé que par un autre constructeur de la classe

Exemple :

```
private Personne(String nom, String prenom) {
  this.nom = nom;
  this.prenom = prenom;
}

public Personne(String nom, String prenom, int age, String statutMatrimonial) {
  this(nom, prenom); 1
  this.age = age;
  this.statutMatrimonial = statutMatrimonial;
}
```

➔ Lorsqu'on appelle un constructeur dans un autre, alors le constructeur appelé doit être la toute première ligne dans le corps du du constructeur appelant. On utilise le mot clé **this(…)** (*cf.* 1) pour faire référence au constructeur appelé. Java réussi à identifier le bon constructeur appelé en analysant les paramètres passés dans this() pour voir le constructeur qui correspond lors de l'instanciation d'une classe.

➔ On peut aussi utiliser le mot clé **this** sous forme de notation pointée pour indiquer que l'on fait référence à une variable d'instance de la classe. Exemple : `this.age`, `this.nom`, `this.prenom`, `this.statutMatrimonial`

2 Le mot clé null

➔ Lorsqu'on développe un programme,
- on peut être amené à indiquer que l'on ne connait pas encore la valeur réelle d'un objet encours de déclaration
- on peut être amené à créer un objet et ne plus en avoir besoin après utilisation

Question : comment fait-on pour répondre à ce type de besoin ?

Solution :

➔ Java propose le mot clé **null** à affecter aux objets uniquement (et non aux types primitifs -> on y reviendra dans le chapitre sur les types de données) pour indiquer qu'un objet a une valeur nulle (ce qui signifie : objet non utilisé)

```
public class Personne {

    private String nom = "Carré";
    private String prenom = "Léo";
    private int age = 21;
    private String statutMatrimonial;

    //constrcteurs
    //méthodes
}
```

```
Personne leo = new Personne("Carré", "Léo", 21, null);
```

=> à la création de l'objet leo, on indique au constructeur d'affecter la valeur *null* au *statutMatrimonial*, car à cet instant on ne connait pas sa valeur réelle

```
Personne patrick = null;
```

=> ici, on déclare un objet patrick de type personne tout en disant qu'il est null pour l'instant

3 Destructeur

➔ Lors de la création d'un objet, Java invoque le *constructeur* de la classe pour créer et charger l'objet dans la mémoire de l'ordinateur :

- Le nombre d'objets présents en mémoire est donc proportionnel au nombre d'objets créer dans le programme
- Plus les objets sont créés par un programme, plus la mémoire se rempli
- **Question :** la mémoire totale allouée à un programme étant limitée, comment libérer au fur et à mesure l'espace mémoire utilisé par ce dernier afin d'éviter un dépassement de capacité pendant qu'il s'exécute et donc son plantage ?

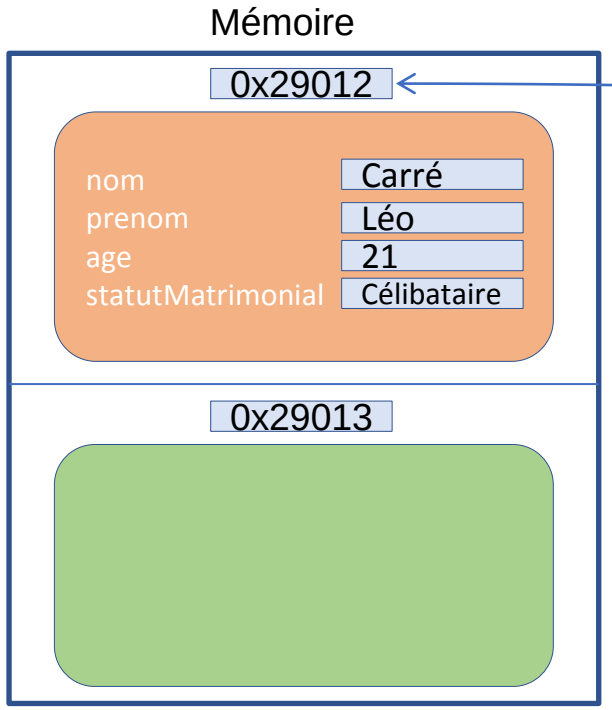
Solution :

➔ Java ne propose pas une gestion manuelle de la destruction des objets. Il n'existe pas de destructeur à proprement parler que l'on défini comme les constructeurs d'une classe. Java utilise le **Garbage Collector**.

➔ Garbage Collector ?

- Aussi appelé « ramasse-miettes », c'est un programme incorporé dans la JVM qui permet de repérer et de libérer automatiquement de la mémoire, tout objet qui n'est plus utilisé (et donc ayant la valeur null) dans le programme
- Lorsqu'il est invoqué par la JVM, il fait appel à une méthode public void finalize(){…}, que Java injecte implicitement dans toute classe que nous définissons. Le développeur peut explicitement redéfinir cette méthode dans sa classe, mais en général on en a pas besoin

4 Notion de référence d'objets (1/2)

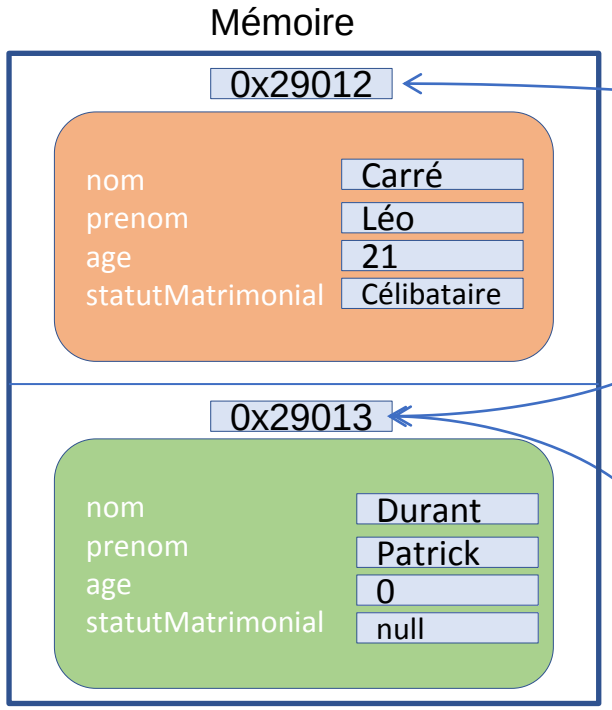



```java
public class Personne {

    private String nom;
    private String prenom;
    private int age;
    private String statutMatrimonial;

    public Personne() { }

    public Personne(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
    }

    public Personne(String nom, String prenom, int age, String statutMatrimonial) {
      this.nom = nom;
      this.prenom = prenom;
      this.age = age;
      this.statutMatrimonial = statutMatrimonial;
    }

    //méthodes
}
```

```java
Personne leo = new Personne("Carré", "Léo", 21, "Célibataire");
```

Lorsqu'on instancie une classe, la variable créée (*i.e* `Leo`) que nous appelons abusivement **objet**, est en réalité une référence vers l'adresse contenant l'**objet** effectif créé et chargé en mémoire. C'est cela la notion de référence d'objet

4 Notion de référence d'objets (2/2)



- Les objets d'une même classe créés chacun avec l'opérateur *new*, ont des références différentes et sont donc indépendants

```
Personne leo = new Personne("Carré", "Léo", 21, "Célibataire");

Personne patrick = new Personne("Durant", "Patrick");
```

- Lorsqu'on affecte un objet *obj1* à un autre *obj2* (i.e, *obj2 = obj1*), on réalise en fait un changement d'adresse (et donc de référence) de *obj2*. Par conséquent, obj1 et obj2 sont équivalents quelque soit leurs valeurs initiales internes, car ils pointent désormais à la même adresse

```
Personne leo = new Personne("Carré", "Léo", 21, "Célibataire");

Personne patrick = new Personne("Durant", "Patrick");
leo = patrick;
```

5 La portée des variables dans une classe

```
public class Personne {

    public String nom;
    public String prenom;
    public int age;
    public String statutMatrimonial;

    public String getNomComplet (String nom, String prenom) {

      String nomComplet = null;

      nomComplet = prenom+nom;
      return nomComplet;
    }

    public int getAge () {
      return this.age;
    }

    public void changeStatut (String newStatut) {
      this.statutMatrimonial = newStatut;
    }

    public static void main(String[] args) {
      System.out.println("It is " + (args != null && args.length >= 2));
    }
}
```

Toute variable d'instance ou de classe est aussi appelée **variable globale** de la classe, car elle peut directement être utilisée dans toutes les méthodes de la classe. Exemple : les attributs age et statutMatrimonial sont directement utilisés dans les méthodes *getAge*() et *changeStatut*()

Tout paramètre de méthode ou variable déclarée dans la méthode est appelé(e) **variable locale**. Sa portée (son utilisation) est limitée uniquement dans le corps de la méthode. Exemple : on ne peut pas utiliser la variable *nomComplet* déclarée dans la méthode *getNomComplet*() en dehors de celle-ci.

Pour information, une variable déclarée dans le corps d'une méthode doit obligatoirement être initialisée (*cf.* `String nomComplet = null;`)

6 Définition méthode et classe abstraite

```
public abstract class Animal {

    private String nom;

    public abstract boolean isDomestique();

    public String getNom(){ return this.nom; }

}
```

➔ Java permet de définir des méthodes sans corps ; de telles méthodes sont appelées : **méthodes abstraites**
Une méthode abstraite doit obligatoirement être publique et doit nécessairement être redéfinie dans une sous-classe (que nous étudierons dans le chapitre sur l'héritage) de la classe abstraite qui la déclare. On utilise le mot clé *abstract* dans sa signature pour faire sa déclaration. Exemple de la méthode abstraite *isDomestique()*

➔ Une **classe abstraite** est une classe particulière Java, portant le mot clé *abstract,* non instanciable, qui a deux fonctions :
- définir des méthodes qui seront implémentées plus tard dans ses sous-classes. Exemple de la méthode *isDomestique()*
- définir des méthodes déjà codés et qui peuvent directement être réutilisées. Exemple de la méthode *getNom()*

=> Son principal intérêt sera évoqué quand l'on abordera le chapitre sur l'héritage

1 Classe et autres caractéristiques

2 La notion d’interface

3 Généricité

Définition d'une interface

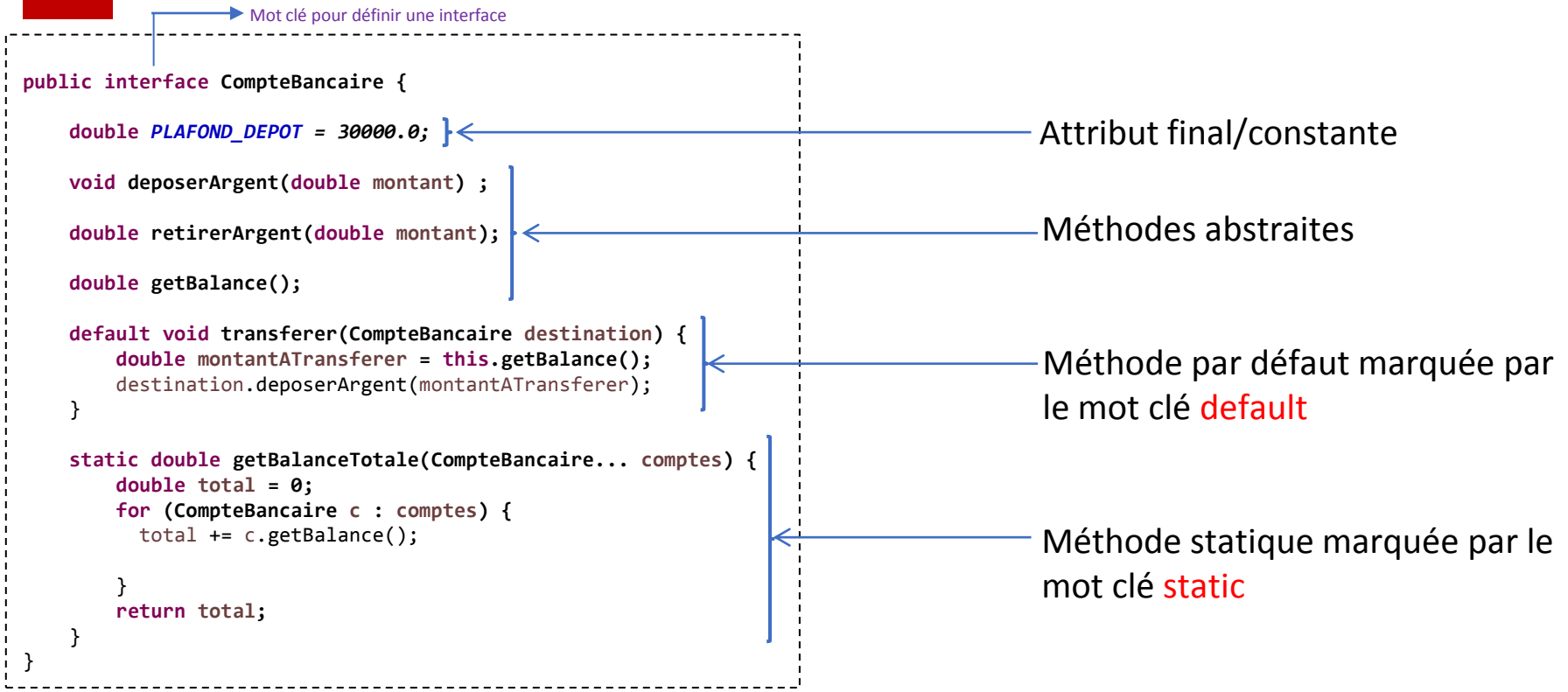
Classe particulière Java :

- disposant de :
  - méthodes abstraites
  - méthodes statiques
  - méthodes par défaut
- dont tous les attributs sont finaux (mot clé final) et sont donc des constantes
- qui ne peut être instanciée comme une classe « ordinaire »
- qui n'est exploitable que lorsqu'on l'**implémente** par une ou plusieurs classes « ordinaire »

Son intérêt est de pouvoir regrouper en son sein un ensemble de méthodes et de constantes transverses qui peuvent être utilisées par plusieurs classes dans un programme.

(Nous verrons également l'intérêt de cette notion dans le chapitre sur l'héritage)

Exemple interface



Interface – considérations par défaut

- Java ajoute implicitement le modificateur public et les mots clés static et final aux attributs déclarés dans une interface
- Java ajoute implicitement de même le modificateur public et le mot clé abstract à toute méthode abstraite de l'interface
- Java ajoute implicitement de même le modificateur public aux méthodes statiques et par défaut

```
public interface CompteBancaire {

    double PLAFOND_DEPOT = 30000.0;

    void deposerArgent(double montant) ;

    double retirerArgent(double montant);

    double getBalance();

    default void transferer(CompteBancaire destination) {
        double montantATransferer = this.getBalance();
        destination.deposerArgent(montantATransferer);
    }

    static double getBalanceTotale(CompteBancaire... comptes) {
        double total = 0;
        for (CompteBancaire c : comptes) {
          total += c.getBalance();

        }
        return total;
    }
}
```

équivalent

```
public interface CompteBancaire {

    public static final double PLAFOND_DEPOT = 30000.0;

    public abstract void deposerArgent(double montant) ;

    public abstract double retirerArgent(double montant);

    public abstract double getBalance();

    public default void transferer(CompteBancaire destination) {
        double montantATransferer = this.getBalance();
        destination.deposerArgent(montantATransferer);
    }

    public static double getBalanceTotale(CompteBancaire... comptes) {
        double total = 0;
        for (CompteBancaire c : comptes) {
          total += c.getBalance();

        }
        return total;
    }
}
```

Interface fonctionnelle

Interface ayant toutes les caractéristiques suscitées, mais ne disposant que d'une et une seule méthode abstraite

➔ Son intérêt réside dans les expressions lambda que nous aborderons au chapitre 14

➔ Sa définition est marquée par l'ajout de l'annotation @FunctionalInterface sur l'interface en déclaration

```
@FunctionalInterface
public interface MonIterface {

    //constantes

    void monUniqueMethodeAbstraite(double montant) ;

    //éventuelles méthodes statiques ou par défaut

}
```

1 Classe et autres caractéristiques

2 La notion d’interface

3 Généricité

Généricité (1/3)

Problématique

Supposons qu'on ait besoin d'une classe dont une ou plusieurs de ses méthodes effectuent les mêmes opérations quelque soit le type de ses attributs

Exemples :

classe contenant une méthode qui effectue en même temps :

- une somme arithmétique, si les attributs sont des nombres entiers ou réels
- une concaténation, si les attributs sont des chaînes de caractères

➔ **Impossible de répondre à une telle problématique avec une seule classe si on n'utilise pas le concept de généricité. Car la seule solution serait de définir une classe pour chaque type de données des attributs (entiers, réels, chaîne de caractères), et donc 3 classes**

Généricité (2/3)

Qu'est-ce donc ?

- Mécanisme permettant de définir un type de données extrêmement abstrait/générique (qui n'est pas du tout précis) pour certains attributs lors de l'implémentation d'une classe

- Ce type de données générique se verra préciser sa vraie valeur (Entier ?, Réels ? Chaîne de caractères ? etc) uniquement au moment de l'instanciation de la classe
  ➔ **On choisit donc pour chaque instance le type que l'on souhaite utiliser**.

- En général, on représente un type générique avec une lettre majuscule : T, U, V, R, S …

- S'applique aussi bien sur les **classes** que sur les **interfaces**

Généricité (3/3)

Exemple de Syntaxe :

```
public class MaClasseGenerique<T> {

   private T var1;
   private T var2;

   //constructeurs

   public T somme() {
     T res = null;
     /*effectuer somme ou concatenation
         de var1 et var2 et affecter le résultat
           à la variable locale res */
     return res;
   }

   //autres méthodes...
}
```

Exemples d'instanciations et utilisation :

- MaClasseGenerique<Integer> obj1 = new MaClasseGenerique<Integer>(23, 74); => obj1.somme(); -> retourne : **97**
- MaClasseGenerique<String> obj2 = new MaClasseGenerique<String>("Hello", "World"); => obj2.somme(); -> retourne : **Hello World**


SUCCESS