La programmation Orientée Objet

SOMMAIRE

- Concepts et Définitions
- Classes et objets en Java
- L’héritage
- Classes abstraites et Interfaces
- Les packages

CHAPITRE 1

CONCEPTS

CONCEPT D'OBJET

- ➢ **Qu'est-ce qu'un objet ?**
  - ✓ **Le monde qui nous entoure est composé d'objets**
  - ✓ **Ces objets ont tous deux caractéristiques**
    1) un état
    2) un comportement
- ➢ **Exemples d'objets du monde réel :**
  - ✓ **chien**
    1) état : nom, couleur, race, poids....
    2) comportement : manger, aboyer, renifler...
  - ✓ **Bicyclette**
    1) état : nombre de vitesses, vitesse courante, couleur
    2) comportement : tourner, accélérer, changer de vitesse

L'APPROCHE OBJET

- **L'approche objet :**
  - ✓ Programmation dirigée par les données et non par les traitements
    - les procédures existent toujours mais on se concentre d'abord sur les entités que l'on va manipuler avant de se concentrer sur la façon dont on va les manipuler
  - ✓ Notion d'encapsulation
    - les données et les procédures qui les manipulent sont regroupés dans une même entité (appelée une ***classe)***.
- **Un objet informatique**
  - ✓ maintient son état dans des variables (appelées ***champs*** ou ***attributs***)
  - ✓ implémente son comportement à l'aide de procédures (appelées ***méthodes***)
  - ✓ **objet informatique = regroupement logiciel d'attributs et de méthodes**
- **Cycle de vie**
  - ✓ construction (en mémoire)
  - ✓ Utilisation (changements d'état par affectations, comportements par exécution de méthodes)
  - ✓ destruction

L'ENCAPSULATION

- **Notion d'encapsulation :**
  - ✓ les données et les procédures qui les manipulent sont regroupées dans une même entité, l'objet.
  - ✓ Les détails d'implémentation sont cachés, le monde extérieur n'ayant accès aux données que par l'intermédiaire d'un ensemble d'opérations constituant l'*interface* de l'objet.
  - ✓ Le programmeur n'a pas à se soucier de la représentation physique des entités utilisées et peut raisonner en termes d'abstractions.
- **Programmation dirigée par les données :**
  - ✓ pour traiter une application, le programmeur commence par définir les classes d'objets appropriées, avec leurs opérations spécifiques.
  - ✓ chaque entité manipulée dans le programme est un représentant (ou instance) d'une de ces classes.
  - ✓ L'univers de l'application est par conséquent composé d'un ensemble d'objets qui détiennent, chacun pour sa part, les clés de leur comportement.

L'INSTANCIATION

- **<u>Instanciation</u> : concrétisation d'une classe en un objet « concret ».**
  - ✓ Dans nos programmes Java nous allons définir des classes et instancier ces classes en des objets qui vont interagir. Le fonctionnement du programme résultera de l'interaction entre ces objets « instanciés ».
  - ✓ En Programmation Orientée Objet, on décrit des classes et l'application en elle-même va être constituée des objets instanciés, à partir de ces classes, qui vont communiquer et agir les uns sur les autres.

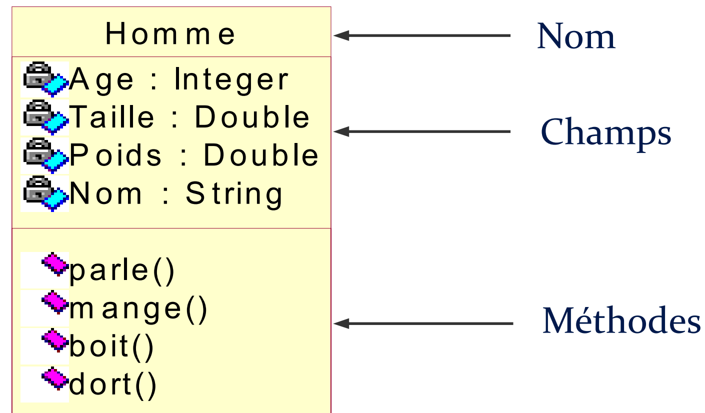
- **<u>Instance :</u>**
  - ✓ représentant physique d'une classe
  - ✓ obtenu par moulage du dictionnaire des variables et détenant les valeurs de ces variables.
  - ✓ Son comportement est défini par les méthodes de sa classe
  - ✓ Par abus de langage « instance » = « objet »

- **<u>Exemple :</u>**
  - ✓ si nous avons une classe voiture, alors votre voiture est une instance particulière de la classe voiture.
  - ✓ **Classe** = concept, description
  - ✓ **Objet** = représentant ***concret*** d'une classe

LA CLASSE (1/2)

- La « classification » d'un univers qu'on cherche à modéliser est sa distribution systématique en diverses catégories, d'après des critères précis
- En informatique, **la classe est un modèle** décrivant les caractéristiques communes et le comportement d'un ensemble d'objets : la classe est un moule et l'objet est ce qui est moulé à partir de cette classe
- Mais l'état de chaque objet est indépendant des autres
  - ✓ Les objets sont des représentations dynamiques (appelées ***instances***) du modèle défini au travers de la classe
  - ✓ Une classe permet d'instancier plusieurs objets
  - ✓ Chaque objet est instance d'une seule classe
- **<u>Classe</u>** : description d'une famille d'objets ayant une même structure et un même comportement. Elle est caractérisée par :
  - ✓ Un *nom*
  - ✓ Une *composante statique* : des champs (ou attributs) nommés ayant une valeur. Ils caractérisent l'état des objets pendant l'exécution du programme
  - ✓ Une *composante dynamique* : des méthodes représentant le comportement des objets de cette classe. Elles manipulent les champs des objets et caractérisent les actions pouvant être effectuées par les objets.

LA CLASSE (2/2)

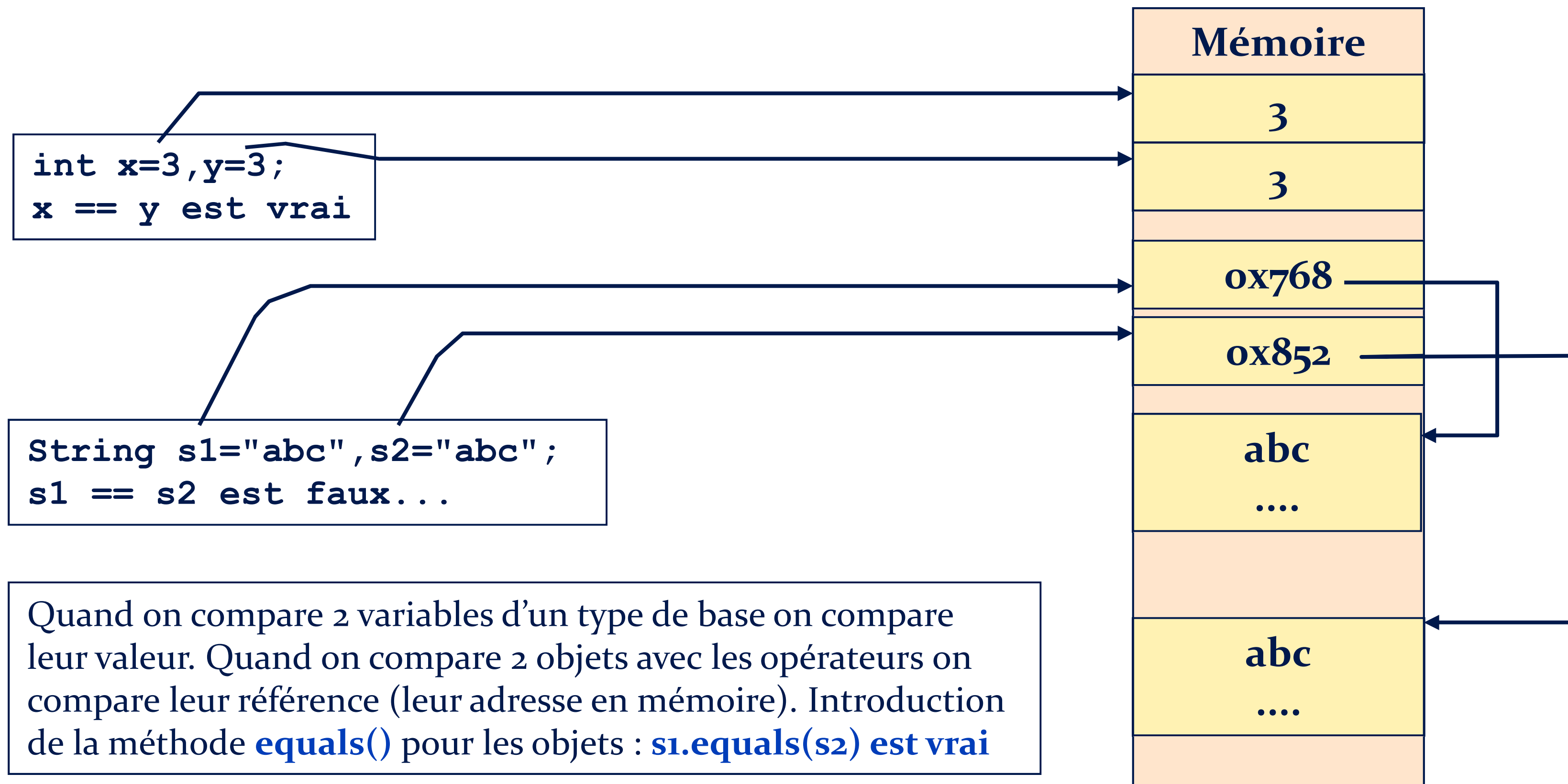


Une classe représentée avec la notation UML
(Unified Modeling Language)

LES RÉFÉRENCES (1/2) : OBJETS ET TABLEAUX ≠ TYPES DE BASE

- La ***référence*** est, en quelque sorte, un pointeur pour lequel le langage assure une manipulation transparente, comme si c'était une valeur (**pas de déréférencement**).
- Par contre, du fait qu'une référence n'est pas une valeur, c'est au programmeur de prévoir l'allocation mémoire nécessaire pour stocker effectivement l'objet (utilisation du **new**).
- Lorsqu'une variable est d'un type de base, la variable **contient** la valeur.
- Lorsqu'une variable est d'un type objet ou tableau, ce n'est pas l'objet ou le tableau lui-même qui est stocké dans la variable mais une **référence vers cet objet ou ce tableau** (on retrouve la notion d'adresse mémoire ou du pointeur en C).

LES RÉFÉRENCES (2/2) :
OBJETS ET TABLEAUX ≠ TYPES DE BASE



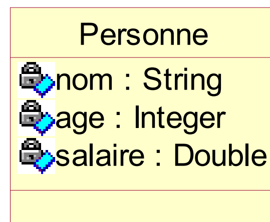
Quand on compare 2 variables d'un type de base on compare leur valeur. Quand on compare 2 objets avec les opérateurs on compare leur référence (leur adresse en mémoire). Introduction de la méthode **equals()** pour les objets : **s1.equals(s2) est vrai**

CHAPITRE 2

CLASSES ET OBJETS EN JAVA

CLASSE ET OBJET EN JAVA

Du modèle à …



… la classe Java et de la classe à …

```
class Personne
{
  String nom;
  Integer age;
  Double salaire;
}
```

… des instances de cette classe

```
Personne jean, pierre;
jean = new Personne();
pierre = new Personne();
```

- L'opérateur d'instanciation en Java est new :
  - ✓ **MaClasse monObjet = new MaClasse();**
- En fait, **new** va réserver l'espace mémoire nécessaire pour créer l'objet « monObjet » de la classe « MaClasse »
- Le **new** ressemble beaucoup au **malloc** du C

SYNTAXE DE DÉFINITION D'UNE CLASSE

- Exemple : Une classe définissant un point

```
public class Point
{
        private double x; // abscisse du point
        private double y; // ordonnée du point
        // initialisation de la classe
        public Point (double xBase, double yBase) {
                x = xBase;
                y = yBase;
        }
        // translation de point de (dx,dy)
        void translate (double dx, double dy) {
                x = x + dx;
                y = y + dy;
        }
        // calcule la distance du point à l'origine
        double distance() {
                return Math.sqrt(x*x + y*y);
        }
```

Nom de la Classe

Attributs

Méthodes

LES ATTRIBUTS

Personne.java

```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public Personne pere;
}
```

Application.java

```
public class Application
{
  public static void main(String args[])
  {
   Personne jean = new Personne();
   jean.nom = "Valjean";
   jean.prenom = "Jean";
  }
}
```

- Les attributs sont les variables de la classe
- Contrairement aux variables, les attributs d'une classe, s'ils ne sont pas initialisés, se voient affecter automatiquement une valeur par défaut. Cette valeur vaut **0** pour les variables numériques, **false** pour les booléens, et **null** pour les références.
- Portée des variables et attributs :
  - ✓ Les variables ne sont connues qu'à l'intérieur du bloc dans lequel elles sont déclarées.
  - ✓ En cas de conflit de nom, c'est toujours la variable la plus locale qui est considérée comme étant la variable désignée par cette partie du programme.

ACCÈS AUX ATTRIBUTS

Personne.java

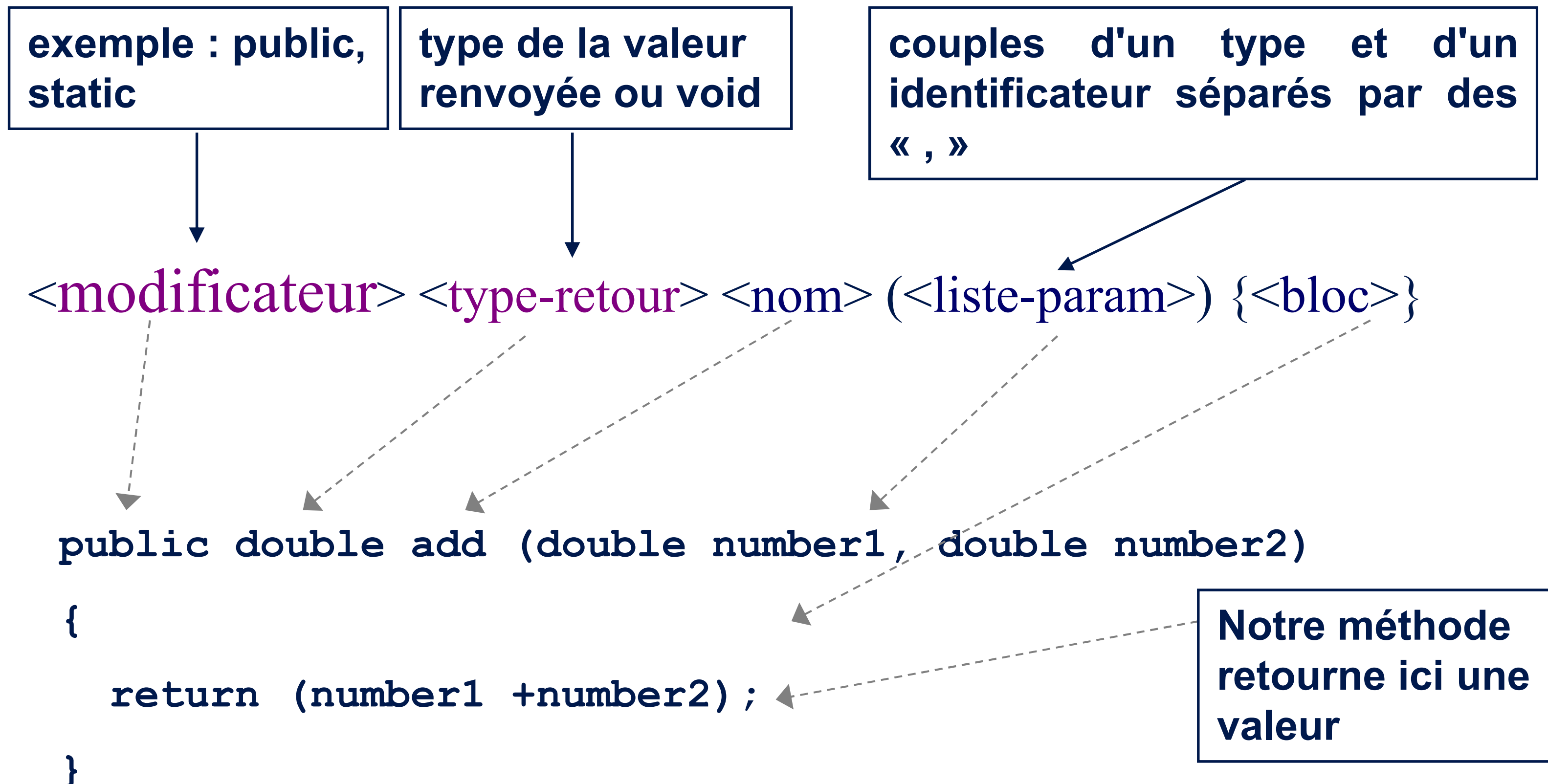
```
public class Personne
{
  private String nom;
  private String prenom;
  private int age;

  public void setNom(String unNom)
  {
    if (unNom.equals(""))
      nom = null;
    else
      nom = unNom;
  }
  public String getNom()
  {
    return (nom);
  }
}
```

Application.java

```
public class Application
{
  public static void main(String args[])
  {
   Personne jean = new Personne();
   jean.setNom( "Valjean");
   System.out.println(jean.getnom());
  }
}
```

- Dans une classe, les attributs s'utilisent comme des variables.
- En théorie, on empêche leur accès de l'extérieur, on utilise des méthodes spécifiques pour cela :
  - ✓ Les **accesseurs**, ou **getter**, récupèrent le contenu de l'attribut (**getNom** dans l'exemple)
  - ✓ Les **mutateurs**, ou **setter**, y affectent une valeur (**setNom** dans l'exemple)
- Cela permet de contrôler les échanges et/ou effectuer d'autres actions en même temps

LES MÉTHODES (1/2)



LES MÉTHODES (2/2)

- **La notion de méthodes dans les langages objets :**
  - ✓ Proches de la notion de procédure ou de fonction dans les langages procéduraux.
  - ✓ La méthode c'est avant tout le regroupement d'un ensemble d'instructions suffisamment générique pour pouvoir être réutilisées
  - ✓ Comme pour les procédures ou les fonctions (au sens mathématique) on peut passer des paramètres aux méthodes et ces dernières peuvent renvoyer des valeurs (par `return`).
- **Mode de passage des paramètres :**
  - ✓ Java n'implémente qu'un seul mode de passage des paramètres : **le passage par valeur.**
  - ✓ Conséquences :
    - l'argument passé à une méthode ne peut être modifié,
    - si l'argument est une instance, c'est sa référence qui est passée par valeur. Ainsi, le contenu de l'objet peut être modifié, mais pas la référence elle-même (ceci est donc valable aussi pour les tableaux).
  - ✓ Nombre variable de paramètres avec les varargs (transformé en un tableau par le compilateur :

```
public void maFonction (String... args) {
          //le vararg "args" est ensuite disponible sous forme de tableau
          dans la méthode
          for(String arg:args){
                    System.out.println(arg);
          }
L'appel se fait alors comme suit :

maFonction( "param1", "param2", "param3", "param4", "param5" );
```

SURCHARGE DE MÉTHODES

- **Dans une même classe, plusieurs méthodes peuvent posséder le même nom, pourvu qu'elles diffèrent en nombre et/ou type de paramètres (dans l'ordre).**

```
public class Point {
    int x;
    int y;
    String nom;
    // initialisation de la classe
    public Point (String nom, int x, int y) { … }
    public Point (int x, int y, String nom) { … } // ordre de param. ≠ → OK
    public Point (int x, int y) { … } // nombre de paramètres ≠ → OK
    public Point (int y, int x) { … } // même nombre et ordre de param. → PAS BON
}
```

- ✓ On parle de **surdéfinition**, **surcharge**, **polymorphisme statique** ou **overloading** en anglais.
- ✓ Le choix de la méthode à utiliser est fonction des paramètres passés à l'appel.
- ✓ Ce choix est réalisé de façon statique, c'est-à-dire à la compilation et non à l'exécution.
- ✓ Très souvent, les constructeurs sont surchargés (plusieurs constructeurs prenant des paramètres différents et initialisant de manières différentes les objets). Cela permet d'initialiser avec un nombre de paramètres limités, les autres étant initialisés à des valeurs par défaut.
- ✓ Voir l'exemple ci-après pour les constructeurs.

LES CONSTRUCTEURS (1/4)

- **L'appel de *new* pour créer un nouvel objet déclenche, dans l'ordre :**
  - ✓ L'allocation mémoire nécessaire au stockage de ce nouvel objet et l'initialisation par défaut de ses attributs,
  - ✓ L'initialisation explicite des attributs, s'il y a lieu,
  - ✓ L'exécution d'un constructeur défini ou celui par défaut.
- **Un *constructeur* est une méthode d'initialisation.**
- Lorsque l'initialisation explicite n'est pas possible (par exemple lorsque la valeur initiale d'un attribut est demandée dynamiquement à l'utilisateur), il est possible de réaliser l'initialisation au travers d'un constructeur.
- Le constructeur est une méthode :
  - ✓ de même nom que la classe,
  - ✓ sans type de retour.
- Toute classe possède au moins un constructeur. Si le programmeur ne l'écrit pas, il en existe un par défaut, sans paramètres, de code vide.

LES CONSTRUCTEURS (2/4)

Personne.java

```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public Personne(String unNom,
                  String unPrenom,
                  int unAge)
  {
    nom=unNom;
    prenom=unPrenom;
    age = unAge;
  }
 }
```

Définition d'un Constructeur. Le constructeur par défaut (Personne()) n'existe plus.
Le code ci-dessous occasionnera une erreur à la compilation.

Application.java

```
public class Application
{
  public static void main(String args[])
  {
    Personne jean = new Personne();
    jean.setNom("Jean") ;
  }
}
```

Va donner une erreur à la compilation

LES CONSTRUCTEURS (3/4)

- Pour une même classe, il peut y avoir plusieurs constructeurs, de signatures différentes (surcharge).
- L'appel de ces constructeurs est réalisé avec le new auquel on fait passer les paramètres :
  - ✓ p1 = new Personne("Pierre", "Richard", 56);
- Déclenchement du « bon » constructeur :
  - ✓ Il se fait en fonction des paramètres passés lors de l'appel (nombre et types). C'est le mécanisme de « lookup ».
- **Attention**
  - ✓ Si le programmeur crée un constructeur (même si c'est un constructeur avec paramètres), le constructeur par défaut n'est plus disponible. Attention aux erreurs de compilation !

LES CONSTRUCTEURS (4/4)

Personne.java

```
public class Personne
{
  private String nom;
  private String prenom;
  private int age;

  public Personne()
  {
    nom=null; prenom=null; age = 0;
  }
  public Personne(String unNom,
                  String unPrenom,
                  int unAge)
  {
    nom=unNom;
    prenom=unPrenom;
    age = unAge;
  }
}
```

Redéfinition d'un Constructeur sans paramètres.

On définit plusieurs constructeurs qui se différencient uniquement par leurs paramètres (on parle de leur signature).

Application.java

```
public class Application
{
  public static void main(String args[])
  {
   Personne anonyme = new Personne();
   Personne jean = new Personne("Jean", "Valjean", 18);
  }
}
```

DESTRUCTION D’OBJETS (1)

- Java n'a pas repris à son compte la notion de destructeur telle qu’elle existe en C++ par exemple. Ainsi, le programmeur n’a plus à gérer directement la destruction des objets, ce qui était une importante source d'erreurs (on parlait de fuite de mémoire ou « memory leak » en anglais).
- C'est le ramasse-miettes (ou ***Garbage Collector*** - GC en anglais) qui s'occupe de collecter les objets qui ne sont plus référencés.
- Le ramasse-miettes fonctionne en permanence dans un thread de faible priorité (en « tâche de fond », parallèlement à l’exécution des programmes de la JVM). Il est basé sur le principe du compteur de références.
- Il peut être "désactivé" en lançant l'interpréteur java avec l'option **-noasyncgc**.
- Inversement, le ramasse-miettes peut être lancé par une application avec l'appel **System.gc();**
- la méthode **finalize()** permet au programmeur d'indiquer ce qu'il faut faire juste avant de détruire un objet (fermer une base de données ou un fichier, couper une connexion réseau, etc.).

CONTRÔLE D'ACCÈS (1)

- **Chaque attribut et chaque méthode d'une classe peut être :**
  - ✓ **private** : visible uniquement depuis les instances de sa classe. En d'autres termes, son nom peut être utilisé uniquement dans l'écriture d'une méthode de sa classe. Il est alors privé, noté par le signe « – » en UML.
  - ✓ **protected :** visible uniquement depuis les instances de sa classe et des classes filles (voir l'héritage plus loin). En d'autres termes, son nom peut être utilisé uniquement dans l'écriture d'une méthode de sa classe et dans celles de toutes les classes dérivées. Il est alors privé , noté par le signe « # » en UML.
  - ✓ **public** : visible depuis les instances de toutes les classes d'une application. En d'autres termes, son nom peut être utilisé dans l'écriture d'une méthode de ces classes. Il est alors public , noté par le signe « + » en UML.
- **En toute rigueur, il faudrait toujours que :**
  - ✓ les attributs ne soient pas visibles,
  - ✓ Les attributs ne devraient pouvoir être lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels (accesseurs et mutateurs).
  - ✓ les méthodes "utilitaires" (utiles uniquement aux autres méthodes de la classe) ne soient pas visibles,
  - ✓ seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.
  - ✓ C'est la notion ***d'encapsulation***

VARIABLES DE CLASSE

UneClasse.java

```
public class UneClasse
{
  // déclaration de la variable de classe
  public static int compteur=0;
  public UneClasse()
  {
   compteur++;
  }
  protected void finalize()
  {
   super.finalize() ;
   compteur-- ;
  }
}
```

Application.java

```
public class Application
{
  public static void main(String args[])
  {
   UneClasse uc1= new UneClasse();
   System.out.println(UneClasse.compteur);
   UneClasse uc2= new UneClasse();
   System.out.println(uc2.compteur);
  }
}
```

- Avec le mot clé **static**, on peut définir un attribut dont la valeur est partagée par toutes les instances d'une classe. On parle alors de variable de classe. Elle est stockée une seule fois, pour toutes les instances d'une classe. On la spécifie par « +#- » en UML.
- On y accède depuis une méthode de la classe comme pour tout autre attribut, via une instance de la classe ou à l'aide du nom de la classe.

MÉTHODES DE CLASSE

UneClasse.java

```
public class UneClasse
{
  public int unAttribut;
  public static void main(String args[])
  {
    unAttribut = 5;     // ERREUR de compilation
  }
}
```

La méthode main est une méthode de classe donc elle ne peut pas accéder à un attribut non lui-même attribut de classe.

- Il peut être nécessaire de disposer d'une méthode qui puisse être appelée sans instance de la classe. C'est une méthode de classe. On la spécifie par « +#- » en UML.
- On utilise là aussi le mot réservé **static**.
- Puisqu'une méthode de classe peut être appelée sans même qu'il n'existe d'instance, une méthode de classe ne peut pas accéder à des attributs non statiques. Elle ne peut accéder qu'à ses propres variables et à des variables de classe.
- Autres exemples, dont beaucoup dans les API de Java :
  - ✓ **System.out.println();**
  - ✓ **Math.sin(x);**
  - ✓ **String.valueOf(i);**

L'AUTORÉFÉRENCE : THIS (1)

- Le mot réservé **this**, utilisé dans une méthode, désigne la référence de l'instance à laquelle le message a été envoyée (donc celle sur laquelle la méthode est « exécutée »).
- Il est utilisé principalement :
  - ✓ lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode,
  - ✓ pour lever une ambiguïté,
  - ✓ dans un constructeur, pour appeler un autre constructeur de la même classe.

```
class Personne {
  public String nom;
  Personne (String nom)
  {
    this.nom=nom;
  }
}
```

Pour lever l'ambiguïté sur le mot « nom » et déterminer si c'est le nom du paramètre ou de l'attribut.

```
public MaClasse(int a, int b) {...}

public MaClasse (int c)
   {
      this(c,0);
   }
public MaClasse ()
   {
      this(10);
   }
```

Appelle le constructeur MaClasse(int a, int b) (pour `this(c,0);`)

Appelle le constructeur MaClasse(int c) (pour `this(10);`)

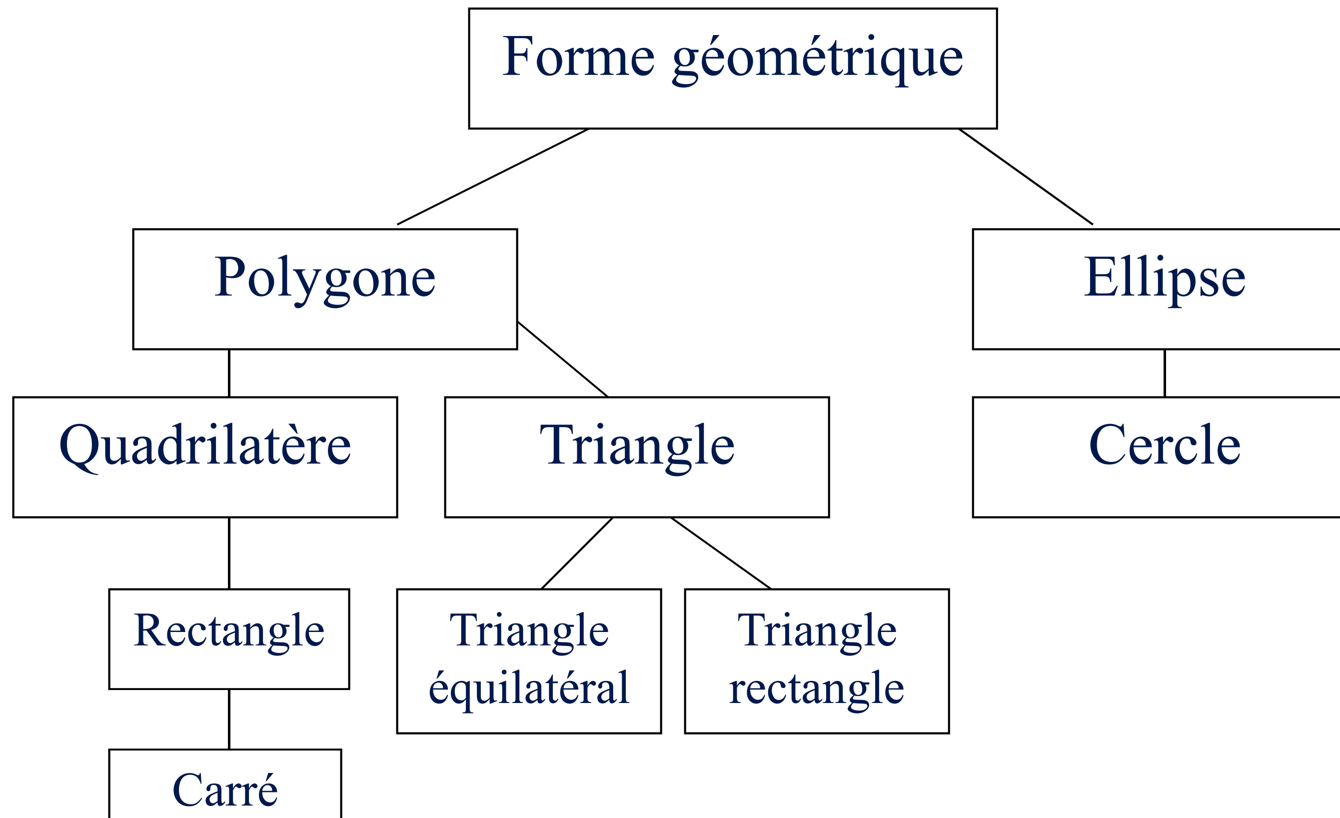
CHAPITRE 3

L'HÉRITAGE

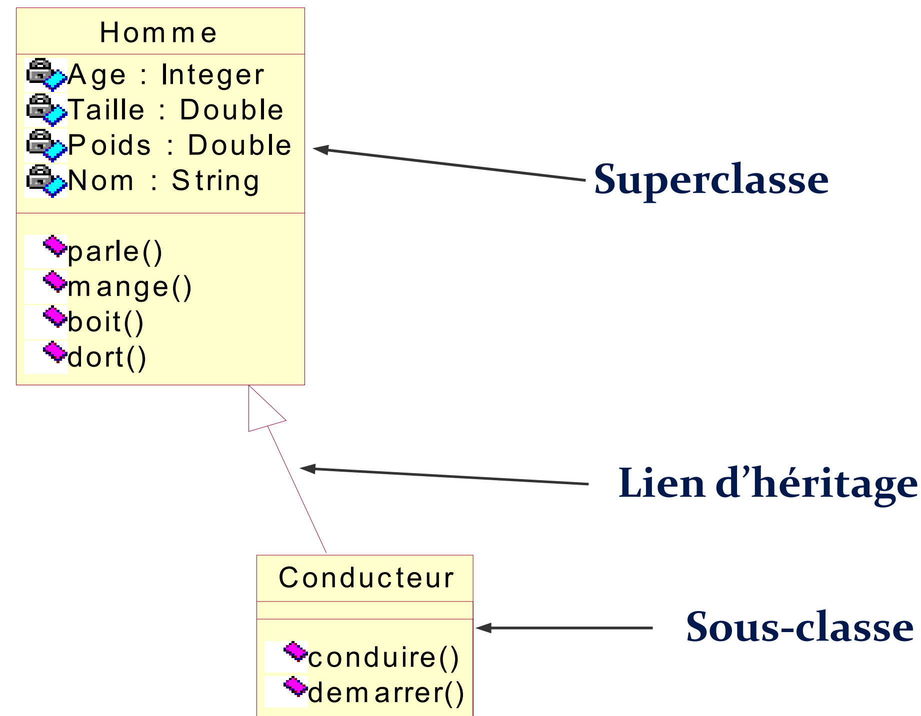
L'HÉRITAGE : DÉFINITION

- La modélisation du monde réel nécessite une classification des objets qui le composent.
- **Classification** = distribution systématique en catégories selon des critères précis.
- **Classification** = hiérarchie de classes (comme celle des éléments chimiques).
- **Héritage** = mécanisme permettant le partage et la réutilisation de propriétés entre les objets. La relation d'héritage est une relation de généralisation / spécialisation.
- La classe parente est la ***superclasse*** et la classe qui hérite est la ***sous-classe*.**
- On parle de ***généralisation*** quand on regroupe les éléments communs de plusieurs classes en une superclasse et de ***spécification*** quand on crée des sous-classes pour différencier les éléments particuliers d'une classe.
- Java implémente le mécanisme d'héritage simple qui permet de "factoriser" de l'information grâce à une relation de généralisation / spécialisation entre deux classes.
- Pour le programmeur, il s'agit d'indiquer, dans la sous-classe, le nom de la superclasse dont elle hérite.
- Par défaut toutes classes Java hérite de la classe **Object** qui est la **superclasse**
- L'héritage multiple n'existe pas en Java.
- Mot réservé : **extends**

L'HÉRITAGE : EXEMPLE



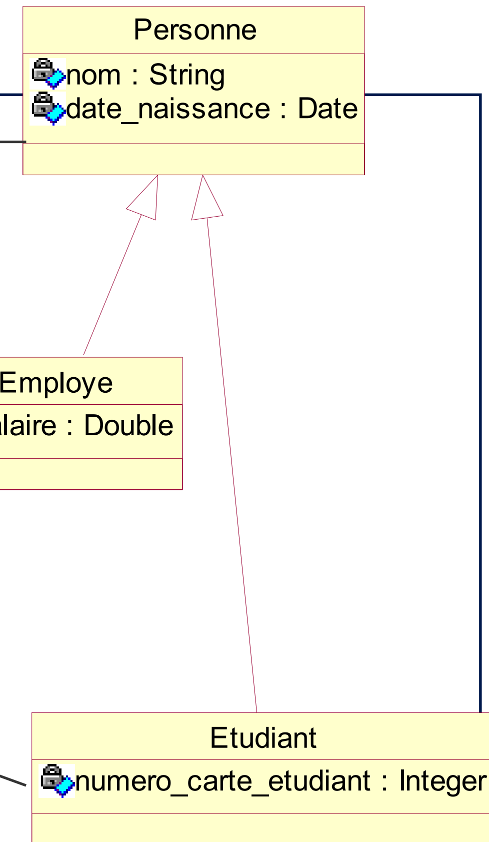
L'HÉRITAGE : REPRÉSENTATION GRAPHIQUE



Représentation avec UML d'un héritage (simple)

L'HÉRITAGE AVEC JAVA (1/3)

```
class Personne {
  protected String nom;
  protected Date date_naissance;
  // ...
}

class Employe extends Personne {
  protected float salaire;
  // ...
}

class Etudiant extends Personne {
  protected int numero_carte_etudiant;
  // ...
}

public static void main(String args[]) {
  Personne p = new Employe(); // Bon, même si les spécificités d'Employe
  p.salaire = 5;              // ne seront pas accessibles (ci-contre)
  Employe e1 = (Employe) p;   // Dans ce cas, on cast p en Employe
  e1.salaire = 5;             // et là ça marche
  Employe e2 = new Personne(); // ne marche pas car un objet Personne ne
                 // pourra pas contenir les spécificités d'un objet Employe
}
```



L'HÉRITAGE AVEC JAVA (2/3)

```
public class Employe extends Personne
{
 protected float salaire;
 public Employe () {}
 public Employe (String nom,
                 String prenom,
                 int anNaissance,
                 float salaire)
 {
   super(nom, prenom, anNaissance);
   this.salaire = salaire;
 }
}
```

```
public class Personne
{
   protected String nom, prenom;
   protected int anNaissance;
   public Personne()
   {
     nom=""; prenom="";
   }
   public Personne(String nom,
                   String prenom,
                   int anNaissance)
   {
     this.nom=nom;
     this.prenom=prenom;
     this.anNaissance=anNaissance;
   }
}
```

Appel explicite à ce constructeur avec le mot clé super, qui fait référence à la classe mère.

L'HÉRITAGE AVEC JAVA (3/3)

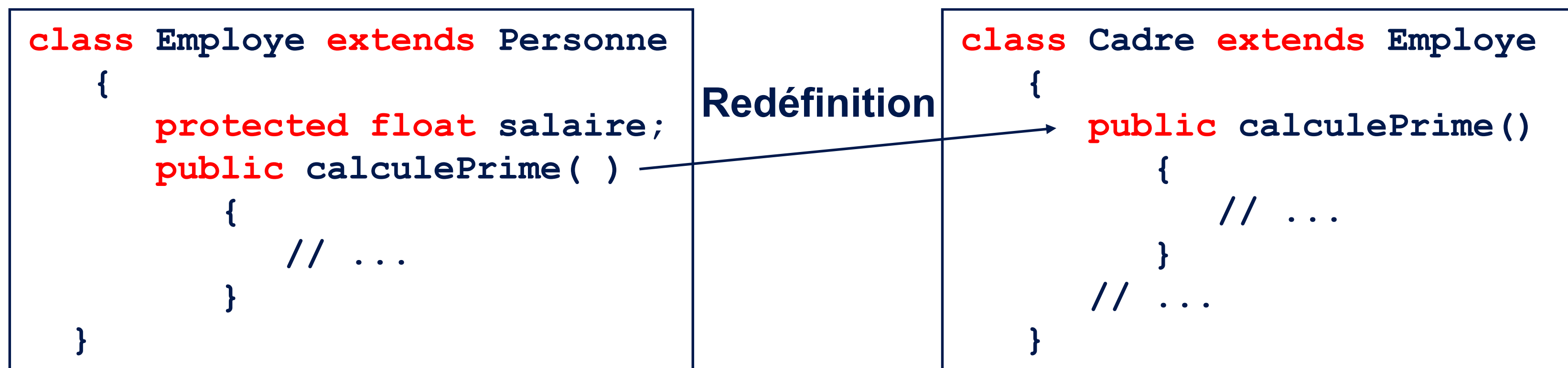
```
public class Personne
{
   protected String nom, prenom;
   protected int anNaissance;
   protected Personne()
   {
     nom=""; prenom="";
   }
   public Personne(String nom,
                   String prenom,
                   int anNaissance)
   {
     this.nom=nom;
     this.prenom=prenom;
     this.anNaissance=anNaissance;
   }
}
```

```
public class
Object
{
  public Object()
  {
    … / ...
  }
}
```

Appel par défaut dans le constructeur de *Personne* au constructeur par défaut de la superclasse de *Personne*, qui est Object

REDÉFINITION DE MÉTHODES

- Une sous-classe peut **redéfinir** des méthodes existant dans une de ses superclasses (directe ou indirectes), à des fins de spécialisation.
  - ✓ Le terme anglophone est "**overriding**". On parle aussi de **masquage**.
  - ✓ **La méthode redéfinie doit avoir la même signature.**

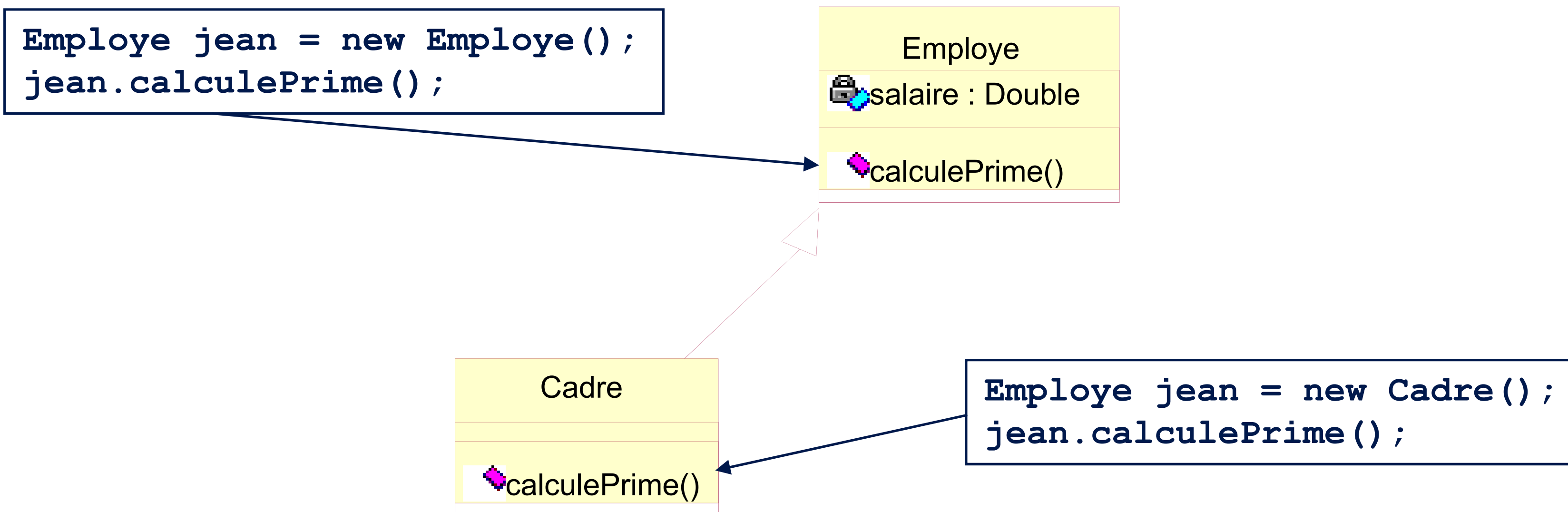


RECHERCHE DYNAMIQUE DES MÉTHODES (1/2)

- **Le polymorphisme (dynamique) :**
  - ✓ Capacité pour une entité de prendre plusieurs formes.
  - ✓ En Java, toute variable désignant un objet est potentiellement polymorphe, à cause de l'héritage.
  - ✓ On parle de *Polymorphisme dit « d'héritage »*
- **Le mécanisme de "lookup" dynamique :**
  - ✓ C'est le déclenchement de la méthode la plus spécifique d'un objet, c'est-à-dire celle correspondant au type réel de l'objet, déterminé à l'exécution uniquement (et non le type de la référence, seul type connu à la compilation, qui peut être plus générique).
  - ✓ Cette dynamicité permet d'écrire du code plus générique.

RECHERCHE DYNAMIQUE DES MÉTHODES (2/2)

- Exemple : jean est de type *Employe*
  - ✓ Si *jean* est créé comme un objet *Employe*, il exécute la méthode *calculPrime* d'*Employe*
  - ✓ Si *jean* est créé comme un objet *Cadre*, il exécute la méthode *calculPrime* de *Cadre*



OPÉRATEUR INSTANCEOF

- L'opérateur instanceof confère aux instances une capacité d'introspection : il permet de savoir si une instance est instance d'une classe donnée.
- Il renvoie une valeur booléenne

```
if ( ... )
  Personne jean = new Etudiant();
else
  Personne jean = new Employe();

//...

if (jean instanceof Employe)
  // discuter affaires
else
  // proposer un stage
```

FORÇAGE DE TYPE / TRANSTYPAGE (1/2)

- Lorsqu'une référence du type d'une classe désigne une instance d'une sous-classe, il est nécessaire de forcer le type de la référence pour accéder aux attributs spécifiques à la sous-classe.
- Si ce n'est pas fait, le compilateur ne peut déterminer le type réel de l'instance, ce qui provoque une erreur de compilation.
- On utilise également le terme de transtypage, en anglais « casting »
- Similaire au « cast » en C

FORÇAGE DE TYPE / TRANSTYPAGE (2/2)

```
class Personne
   {
      private String nom;
      private Date date_naissance;
      // ...
   }

class Employe extends Personne
   {
      public float salaire;
      // ...
   }

Personne jean = new Employe ();
float i = jean.salaire; // Erreur de compilation
float j = ( (Employe) jean ).salaire;  // OK
```

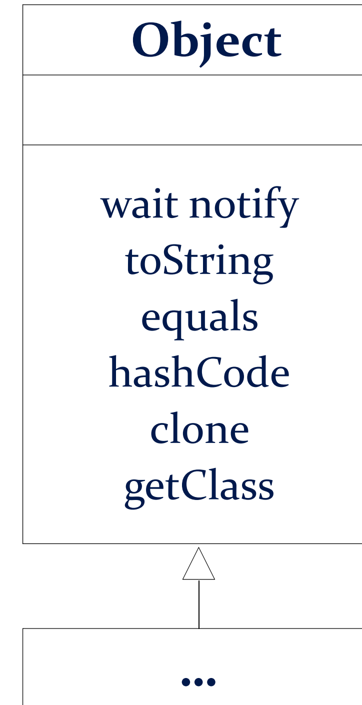
A ce niveau, pour le compilateur, dans la variable « jean » c'est un objet de la classe Personne, donc qui n'a pas d'attribut « salaire ».

On « force » le type de la variable « jean »
pour pouvoir accéder à l'attribut « salaire ».
On peut le faire car c'est bien un objet Employe qui est dans cette variable

LA CLASSE OBJECT

- La classe **Object** est la classe mère de toutes les classes
- Une référence de type Object permet de référencer tout objet, ce qui est très utile pour des test avec **instanceof.**
- La méthode **toString**, à redéfinir pour tout objet, permet d'en visualiser son état.
- Il est nécessaire de redéfinir **equals** pour tester l'égalité d'état des objets, car ce n'est pas le même pour **Object**, qui ne considère pas que le contenu des attributs.
- **hashCode** permet de ranger les objets dans une table de hachage (dans les collections par exemple).
- La méthode **clone** duplique un objet mais la classe **doit** implémenter l'interface **Cloneable**, sinon une exception **CloneNotSupportedException** est déclenchée.

  Les objets référencés ne sont pas dupliqués, donc **clone** doit être redéfinie pour le faire, sinon les références pointeront sur les mêmes objets !



CHAPITRE 4

CLASSES ABSTRAITES ET INTERFACES

CLASSES ABSTRAITES

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthode sans la définir (c'est-à-dire sans en donner le code). La définition du code est dans ce cas laissée aux sous-classes.
- Une telle classe est appelée ***classe abstraite***.
- Elle doit être marquée avec le mot réservé **abstract**.
- Toutes ses méthodes qui ne sont pas définies doivent elles aussi être marquées **abstract**.
- Une classe abstraite ne peut pas être instanciée, elle ne peut qu'être héritée (c'est son but).
- Par contre, il est possible de déclarer et d'utiliser des variables du type de la classe abstraite.
- Tant qu'une sous-classe d'une classe abstraite ne définit pas TOUTES les méthodes abstraites de ses superclasses, elle est abstraite elle aussi.

```
public abstract class Polygone
{
  private int nombreCotes = 3;

  public abstract void dessine (); // méthode non définie

  public int getNombreCotes()
  {
    return(nombreCotes);
  }
}
```

INTERFACES : DÉFINITION

- Java ne permettant pas l'héritage multiple, pour définir qu'une certaine catégorie de classes doit implémenter un ensemble de méthodes, on peut regrouper les déclarations de ces méthodes dans une ***interface***.
- Une interface ne peut pas être instanciée, elle ne peut qu'être implémentée (c'est son but) ou référencée.
- Le but est de décrire le fait que de telles classes pourront ainsi être manipulées de manière identique, même si elles n'ont pas de lien d'héritage.
- Exemple :
  - ✓ Tous les appareils électriques peuvent être allumés ou éteint
  - ✓ Tous les objets comestibles peuvent être mangés
  - ✓ Tous les objets lumineux éclairent
- Les classes désirant appartenir à la catégorie ainsi définie
  - ✓ *déclareront qu'elles implémentent cette interface,*
  - ✓ *fourniront le code spécifique des méthodes déclarées dans cette interface.*
- Cela peut être vu comme un contrat entre la classe et l'interface
  - ✓ *la classe s'engage à implémenter les méthodes définies dans l'interface*

INTERFACES : IMPLÉMENTATION

- Implémentation de l'interface :
  - ✓ Mot réservé : **interface**
  - ✓ Dans un fichier nomInterface.java, on définit la liste de toutes les méthodes de l'interface

```
interface nomInterface {
type_retour methode1(paramètres);
type_retour methode2(paramètres);
 … }
```

  - ✓ Les méthodes d'une interface sont publiques et abstraites (abstract est facultatif) : elles seront écrites spécifiquement dans chaque classe implémentant l'interface
  - ✓ Les champs d'une interface sont par défaut static et final;
- Implémentation de l'interface :
  - ✓ Mot réservé : **implements**
  - ✓ La classe doit expliciter le code de chaque méthode définie dans l'interface

```
class MaClasse implements nomInterface {

…
type_retour methode1(paramètres)

   {code spécifique à la méthode1 pour cette classe};

 … }
```

INTERFACES : EXEMPLE

```
interface Electrique
{
  void allumer();
  void eteindre();
}
```

```
class Radio implements Electrique
{
  // ...
  void allumer() {
    System.out.println("bruit");
  }
  void eteindre() {
    System.out.println("silence");
  }
}
```

```
// …

Radio maRadio = new Radio();
Ampoule monAmpoule = new Ampoule();
Electrique c;
Boolean sombre;

// …

if(sombre == true)
   c = monAmpoule;
else
   c = maRadio;

c.allumer();
…
c.eteindre();

// …
```

```
class Ampoule implements Electrique
{
  // ...
  void allumer() {
    System.out.println(« jour");
  }
  void eteindre() {
    System.out.println(« nuit");
  }
}
```

INTERFACES ET HÉRITAGE MULTIPLE

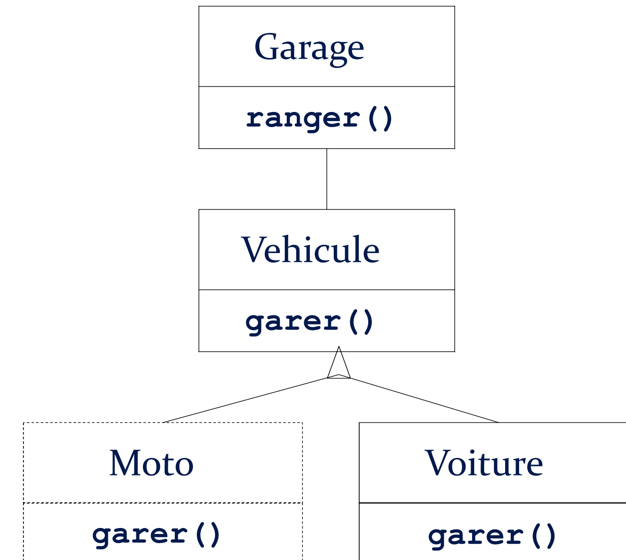
- Une classe ne peut hériter que d'une seule classe mais peut implémenter plusieurs interfaces.
- Une variable peut être définie selon le type d'une interface
- Une classe peut implémenter plusieurs interfaces différentes
- L'opérateur **instanceof** peut être utilisé sur les interfaces
- Exemple :

```
interface Electrique
…
interface Lumineux
…
class Ampoule extends Lumiere implements Electrique, Lumineux
…
Electrique e;
Object o = new Ampoule();
if (o instanceof Electrique) {
  e=(Electrique)o;
  e.allumer();
}
```

INTERFACES COMME FONCTIONS RÉFLEXES

- Permet de faire des appels à des classes entre elles de manière réflexive. Exemple :

```
class Voiture implements Vehicule { ... }
class Moto    implements Vehicule { ... }

// ...

class Garage {
  Vehicule vehicule ;
  public void setVehicule( Vehicule véhicule )
  {
    this.vehicule = vehicule;
    vehicule.garage = this ;
  }
  public void ranger() {
    vehicule.garer();
  }
}

// ...

Garage garage = new Garage();
Voiture voiture = new Voiture();
garage.setVehicule( voiture );
garage.ranger() ;
// ...
Moto moto = new Moto();
garage.setVehicule(moto);
garage.ranger();
```



INTERFACES : CONCLUSION

- C'est un moyen d'écrire du code générique
- C'est une solution au problème de l'héritage multiple
- C'est un outil pour concevoir des applications réutilisables
- **Cependant** :
  - ✓ Une interface ne dispose que de champs static et final
  - ✓ Une interface doit être déclarée publique pour pouvoir être utilisée à l'extérieur du package où elle a été écrite
  - ✓ Une interface peut étendre une autre interface
  - ✓ Une classe abstraite peut avoir une implémentation partielle.

CHAPITRE 5

LES PACKAGES

LES PAQUETAGES : DÉFINITION

- Un paquetage (« package » en anglais) permet de regrouper un ensemble de classes.
- Les classes d'un même package sont compilées dans un répertoire portant le nom du package. On peut même créer une arborescence de package.
- Le mot clef **package** doit apparaître en premier dans un fichier source.
- Sans déclaration d'accès, donc par défaut, les attributs et les méthodes sont déclarés **package friendly**. Ils sont alors visibles et accessibles dans tous les fichiers du package mais pas dans les autres.
- Le mot clef **import** permet d'importer un package dans un fichier et de rendre accessible ses éléments. Cela permet notamment d'accéder aux API standards.
- Les répertoires de packages peuvent être construit de manière arborescente. On peut alors importer un ensemble de package avec le caractère * : **import java.io.*;**
- En fonctionnant par package, on peut, et c'est recommandé !, ne mettre qu'une seule classe par fichier.

```
// déclaration de notre package
package figuresGeometriques;
// import des packages de méthodes d'E/S
import java.io.*;

public class Polygone
{
  int perimetre = 1000;
}
```

```
package figuresGeometriques;

public class Essai
{
  public static void main(String [] argv)
  {
    Polygone pol = new Polygone();
    pol.perimetre = pol.perimetre + 100;
  }
}
```