Chapitre 3:

La Programmation Orientée Objet avec Java

Pourquoi l'objet?

- L'homme perçoit son monde comme un ensemble d'objets
  - Personnes, animaux, plantes, machines, etc.
- Dans notre cerveau chaque objet est identifié par des caractéristiques bien connues (des propriétés ou des actions)
  - Propriétés : animal, joli, fidèle, bruyant,
  - Actions : mange, boit, aboie, court, etc.
- Problème en programmation habituelle
  - La logique de nos programmes est plus proche au fonctionnement de la machine qu'au fonctionnement du cerveau du concepteur !

Programmation procédurale

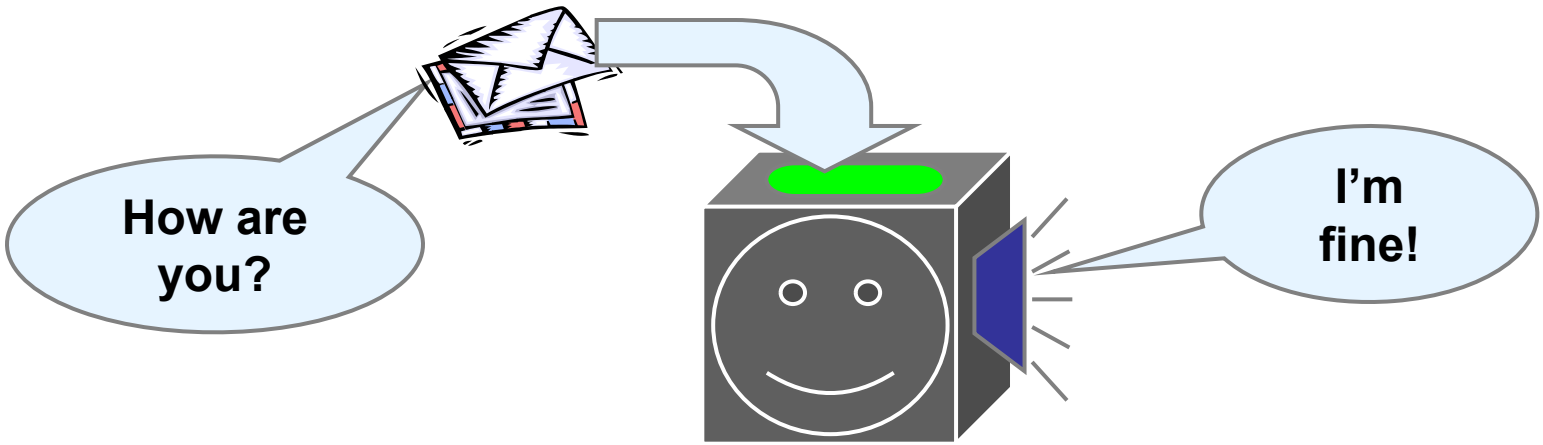
- Les données constituent des ensembles globaux traités globalement
- Les traitements sont regroupés dans quelques grandes fonctions qui peuvent être imbriquées et faire appel les unes aux autres
- Les données sont donc partagées collectivement et les fonctions sont interpénétrées
- On a donc pensé aux traitements et à leur enchaînement AVANT de penser aux données

➔ Approche « TOP-DOWN »

- Le programme devient de plus en plus complexe à mesure que la complexité et interdépendances des traitements s'intensifient
- Difficulté de réutiliser les fonctions dans un autre contexte
- Si on modifie une entité, il faut vérifier tout le code

Programmation Objet

- Placer les entités, objets ou acteurs du problème à la base de la conception
- Etudier les traitements comme des interactions entre les différentes entités
- Penser aux données AVANT de penser aux traitements
- Penser à un programme en modélisant le monde tel qu'il nous apparaît
- Qu'est-ce qu'un objet?
  - ➔ Une boîte noire qui reçoit et envoie des messages



Programmation Objet

- Que contient cette boîte?
  - Du code ➔ Traitements composés d'instructions ➔ **Comportements**
  - Des données ➔ L'information traitée par les instructions et qui caractérise l'objet ➔ **Etats** ou **Attributs**

  ➔Les traitements sont conçus pour une boîte en particulier et ne peuvent pas s'appliquer à d'autres boîtes

  ➔Données et traitements sont donc indissociables
- Cette boite peut être réutilisée en totalité dès qu'on a besoin d'un objet de même type : **Réutilisation**

Pourquoi une boite noire ?

- L'utilisateur d'un objet ne devrait jamais avoir à plonger à l'intérieur de la boîte
- Toute l'utilisation et l'interaction avec l'objet s'opère par messages
- Les messages définissent l'interface de l'objet donc la façon d'interagir avec les autres
- Il faut et il suffit de connaître l'interface des messages pour pouvoir exploiter l'objet à 100%, sans jamais avoir à connaître le contenu exact de la boîte ni les traitements qui l'animent
- Les utilisateurs d'un objet ne sont pas menacés si le concepteur de l'objet en change les détails ou la mécanique interne
- On parle du concept clé de la programmation objet : **l'encapsulation**

L'encapsulation

- Un objet est semblable à une variable améliorée :
  - elle stocke des données qui décrivent son «état»;
  - mais qui possède aussi un ensemble de fonctions ou méthodes «comportement»,
- L'ensemble des services (méthodes) proposées par un objet est appelé l'interface de cet objet.
- Un objet est encapsulé par son interface :
  - la seule manière d'interagir (demander un service) avec cet objet est d'invoquer une des méthodes de son interface.
- L'encapsulation d'un objet par son interface permet de masquer son contenu : Il s'agit de montrer uniquement ce qui est nécessaire pour son utilisation :
  - Les attributs sont généralement considérés comme données privées.
  - Les méthodes constituent l'interface d'interaction avec un objet d'une classe. Elle sont donc accessibles (publiques).

Quelques exemples

Classe	États	Comportements
Chien	Nom, race, âge, couleur	Aboyer, chercher le baton, mordre, faire le beau
Compte	N°, type, solde, ligne de crédit	Retrait, virement, dépôt, consultation du solde
Téléphone	N°, marque, sonnerie, répertoire, opérateur	Appeler, Prendre un appel, Envoyer SMS, Charger
Voiture	Plaque, marque, couleur, vitesse	Tourner, accélérer, s'arrêter, faire le plein, klaxonner

Instanciation

- Classe = modèle décrivant le *contenu* et le *comportement* des futurs objets de la classe = ensemble d'objets
  - Le contenu = les attributs
  - Le comportement = les méthodes
- La classe **Voiture** peut être décrite par :
  - **Attributs :** Plaque, marque, couleur, vitesse
  - **Méthodes :** Tourner, accélérer, s'arrêter, faire le plein, klaxonner
- La classe détermine tout ce que peut contenir un objet et tout ce qu'on peut faire de cet objet
- Classe = Moule, Définition, ou Structure d'un d'objet
- Objet = Instance d'une classe
- Le processus de création d'un objet à partir d'une classe est appelé instanciation d'un objet ou création d'instance d'une classe.

Quelques exemples

- La classe « chien » définit:
  - Les attributs d'un chien (nom, race, couleur, âge…)
  - Les comportements d'un chien (Aboyer, chercher le baton, mordre…)
- Il peut exister dans le monde plusieurs objets (ou instances) de chien

Classe	Objets
Chien	**Mon chien: Bill, Teckel, Brun, 1 an** **Le chien de mon voisin: Hector, Labrador, Noir, 3 ans**
Compte	**Mon compte à vue: N° 210-1234567-89, Courant, 1.734 DT,** **Mon compte épargne: N° 083-9876543-21, Epargne, 27.000 DT**
Voiture	**Ma voiture: ABC-123, VW Polo, grise, 0 km/h** **La voiture que je viens de croiser: 102TU5968, Porsche, noire, 170 km/h**

Classe vs Objet

- On définit les classes par :

```
class nomClasse {
   type_donnees données
   définition des méthodes
}
```

- Par exemple :

```
class Automobile {
   String genre;
   String immatriculation;
   int NbPlaces;
   String propriétaire;
   void s_arreter(){ ... }
   void          avancer(float
   nbmetres) { ... }
}
```

• En Java il faut construire explicitement les objets (**new**)

• Déclaration d'une référence :

```
Automobile aut1;
```

• Initialisation (Allocation) :

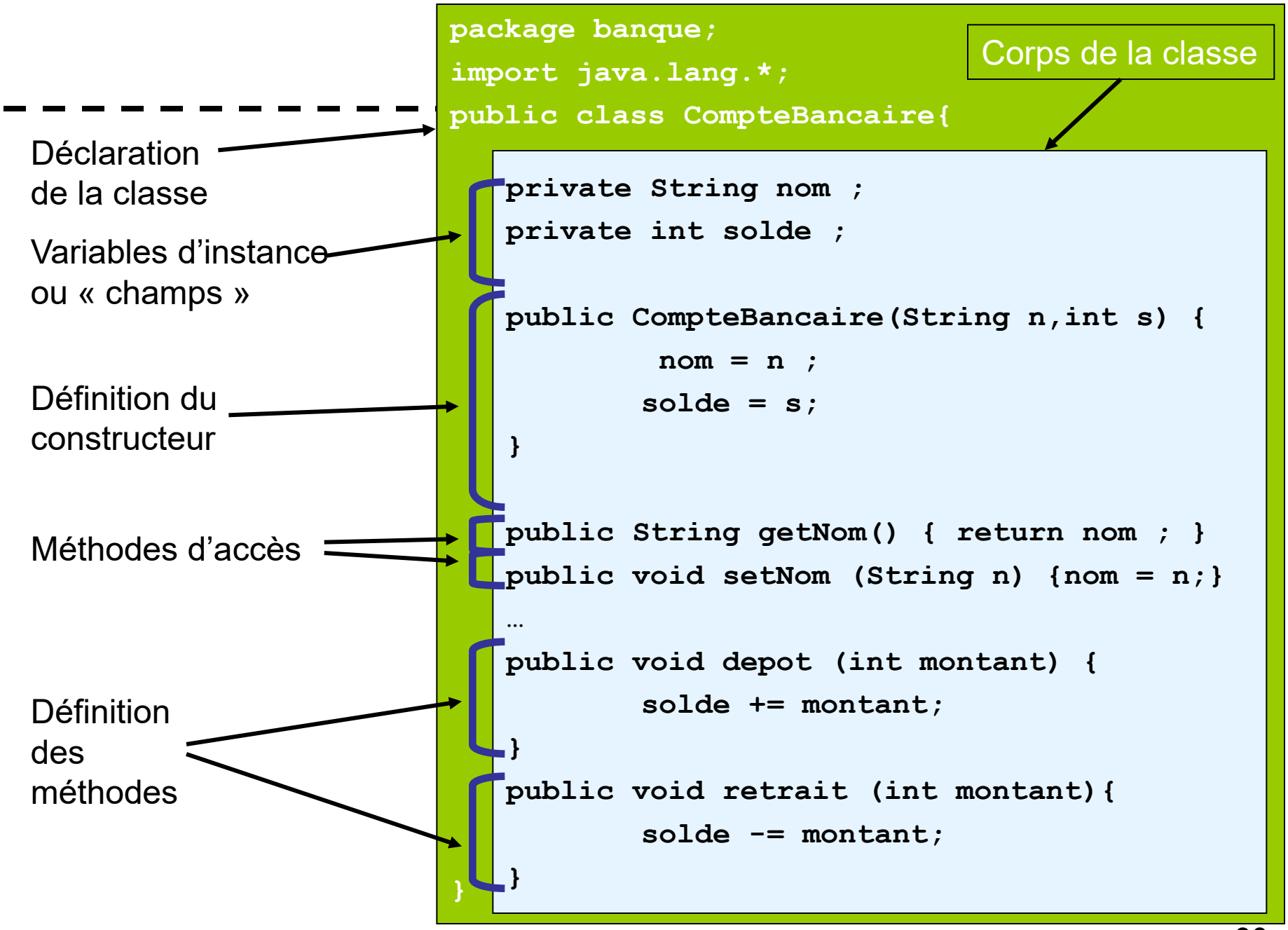
```
aut1 = new Automobile();
```

La référence aut1 est initialisée, on l'utilise alors pour lancer toute méthode sur l'objet (i.e. pour envoyer tout message à l'objet) par :

```
aut1.avancer(4.5);
```

III.1 Composition de classe dans Java

Composition, attributs, méthodes constructeurs, …


package banque;
import java.lang.*;
public class CompteBancaire{
Corps de la classe
Déclaration de la classe
Variables d'instance ou « champs »
Définition du constructeur
Méthodes d'accès
Définition des méthodes
private String nom ;
private int solde ;
public CompteBancaire(String n,int s) {
nom = n ;
solde = s;
}
public String getNom() { return nom ; }
public void setNom (String n) {nom = n;}
…
public void depot (int montant) {
solde += montant;
}
public void retrait (int montant){
solde -= montant;
}
}

La classe en Java

- Une classe est un moule pour créer des objets
- Une description de la boîte noire : ce qu'il contient, ce qu'il peut faire
- Un type d'objet
- Une classe est composée de :
  - Variables, ou champs
    - qui donnent l'état des instances
  - Constructeurs
    - qui permettent de créer des nouvelles instances de la classe (objets)
  - Accesseurs et modificateurs (get et set)
    - qui permettent de lire et modifier les variables d'instance
  - Méthodes
    - qui indiquent les types de messages qui peuvent être envoyés aux instances

Les attributs

- Les attributs d'une classe sont les éléments qui composent les objets de cette classe ou les caractéristiques de ces objets

- Tous les attributs sont dotés d'un type défini au niveau de la classe (autre classe ou type de base)

- Les attributs sont vus comme des variables globales internes à un objet

- La portée d'un attribut est l'objet dans lequel il est défini

- L'attribut d'un objet garde sa valeur d'une méthode à l'autre tant qu'aucune méthode ne le modifie

Les attributs

- Il existe deux types de variables dans une classe :
  - **« d'Instance »,** représentent l'état interne d'un objet et contenues:
    - dans une classe
    - dans aucune méthode
  - **« Locales »,** contenues dans une méthode et utilisées par cette méthode pendant son exécution

- Lorsqu'on utilise un attribut dans une méthode, il ne faut pas rappeler son type
- Ne jamais avoir une variable dans une méthode qui porte le même nom qu'un attribut
- Si un paramètre porte le même nom qu'un attribut, attention aux confusions

Les attributs

Attention !

```
class Voiture
{
    double longueur;

     public void demarrer()
        {
          int longueur = 3;

          if (longueur<2) ...
        }
}
```

```
class Voiture
{
    double longueur;

     public Voiture(double
        longueur)
     {
       this.longueur = longueur;
     }
}
```

Ou bien :

```
     public Voiture(double lon)
            {
              longueur = lon;
            }
```

Le constructeur

- Un constructeur est une méthode automatiquement appelée au moment de la création de l'objet.
- Un constructeur est utile pour procéder a toutes les initialisations nécessaires lors de la création de la classe.
- Le constructeur porte le même nom que le nom de la classe et n'a pas de valeur 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.
- Pour une même classe, il peut y avoir plusieurs constructeurs, de signatures différentes

Le constructeur

- L'appel de ces constructeurs est réalisé avec le new auquel on fait passer les paramètres.
- L'appel de new déclenche l'allocation mémoire nécessaire au stockage du nouvel objet créé et l'initialisation de ses attributs,
- 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.

Le constructeur

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

```
Personne p1 = new Personne();
```

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

```
Personne p2 =
new Personne("Anne", "Brun", 36);
```

Accesseurs et modificateurs

- Les accesseurs et modificateurs sont des méthodes particulières qui servent à accéder à des données privées, i.e déclarées en private
- Il existe deux types de méthodes:
  - ✓ les accesseurs en lecture, dit aussi *accesseur (get)*
  - ✓ les accesseurs en modification, dit aussi *modificateur (set)*
- Pour les accesseurs en lecture, on utilise la notation suivante:
  - ✓ **toto.getNom()**
  - ✓ la méthode getNom() n'a pas de paramètres d'entrée
- Pour les accesseurs en modification, on utilise la notation suivante :
  - ✓ **toto.setAge(unAge)**
  - ✓ la méthode setAge(unAge) a généralement un paramètre d'entrée qui servira à affecter la nouvelle valeur à l'attribut qu'on veut modifier

Les méthodes

- Définition du comportement interne/externe de l'objet, ce qu'il peut faire.
- Les méthodes *public* définissent l'interface
- il existe des méthodes private qui servent de "sous-programmes" utilitaires aux autres méthodes de classe
- Java n'implémente qu'un seul mode de passage des paramètres à une méthode : 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.

Les méthodes

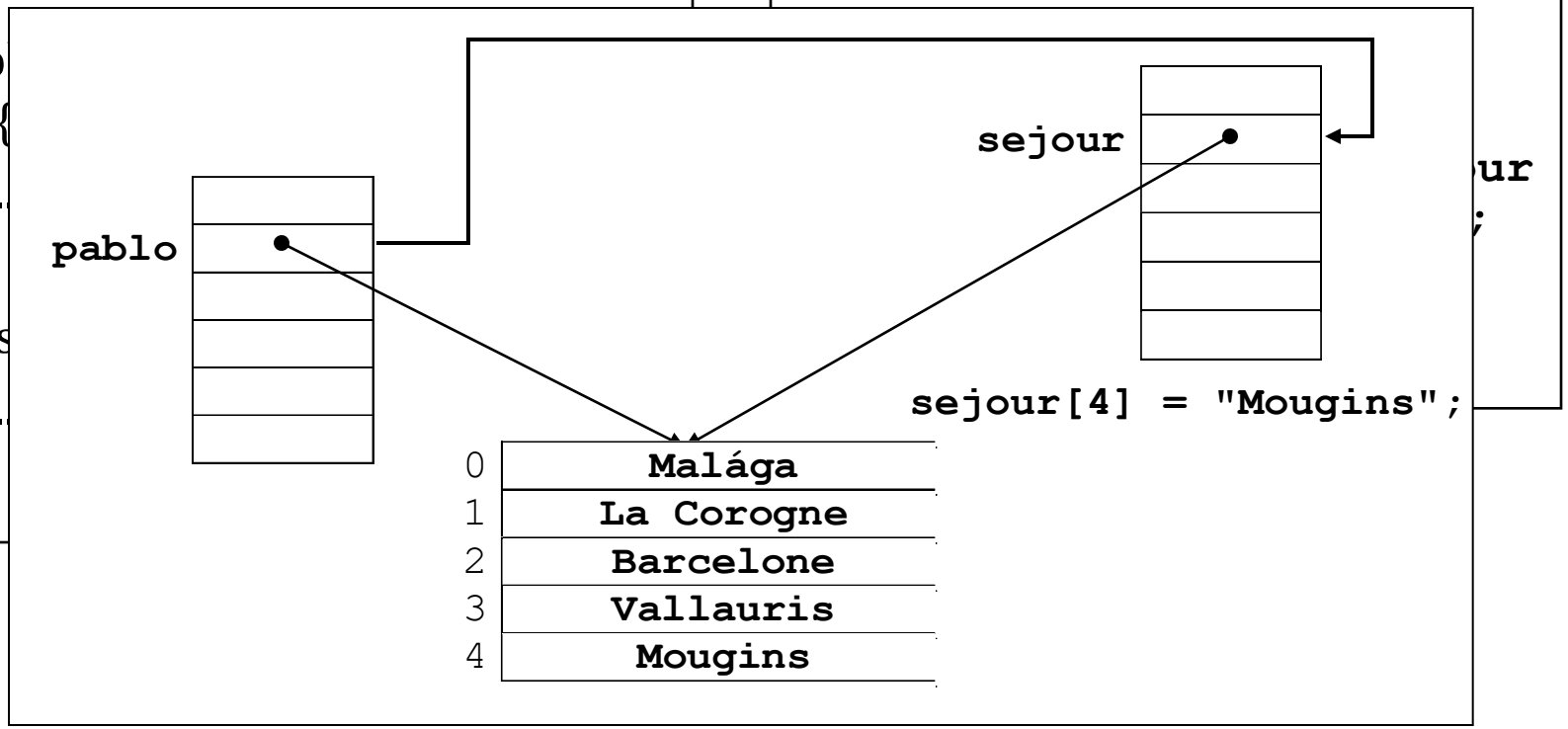
- Passage de paramètres
  - Quand on passe un objet en paramètres, c'est une référence sur cet objet qui est passée et copiée
  - Donc les paramètres passés peuvent être modifiés à l'intérieur de la méthode
  - Donc si l'objet est modifié dans la méthode, les modifications seront visibles de l'extérieur
  - Solution : faire une copie des paramètres avant l'appel

Les méthodes

```
String[] pablo = {"Málaga", "La
    Corogne", "Barcelone",
    "Vallauris", "Cannes"};

pub
    {
    .
    .
    .
}
```

```
System.out.println("Séjour
   final : " + pablo[4]);
-> Cannes
```



Les méthodes

- **Surcharge d'une méthode**
  - en Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a la même signature qu'une méthode existante

    public double calculerMoyenne()

    public double calculerMoyenne(double [] desNotes)
  - en Java, il est interdit de surcharger une méthode en changeant le type de retour
  - autrement dit, deux méthodes ne peuvent avoir la même signature sans avoir le même type de retour
  - exemple : il est interdit d'avoir ces 2 méthodes dans une même classe :

    int calculerMoyenne()

    double calculerMoyenne(double [] desNotes)

III.2 L’accès aux membres d’une classe Java

modificateurs de visibilités, les variables et méthodes de classe, etc.

Les modificateurs d'accès

Dans un objet l'accès et l'utilisation d'un attribut ou d'une méthode est contrôlé :

- **public** : tout le monde peut y accéder, entre autre un utilisateur le « client programmeur ».
  - Définit l'interface de la classe.
- **private** : accès interne à l'objet
  - Selon le principe d'encapsulation, les attributs doivent être « private »
- **protected** : public pour les classes qui héritent (nous y reviendrons plus tard)
- **Aucun modificateur** c'est-à-dire que c'est publique dans son package (on l'appelle un attribut ou une méthode amie)

Les modificateurs d'accès

- Préservation de la sécurité des données
  - Les données privées sont simplement inaccessibles de l'extérieur
  - Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques
- Préservation de l'intégrité des données
  - La modification directe de la valeur d'une variable privée étant impossible, seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place des mécanismes de vérification et de validation des valeurs de la variable
- Cohérence des systèmes développés en équipes
  - Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de connaître la signature. Leur code est donc indépendant de l'implémentation des méthodes

Les modificateurs d'accès

- En java, les modificateurs d'accès sont utilisés pour protéger l'accessibilité des variables et des méthodes.
- Les accès sont contrôlés en respectant le tableau suivant:

Mot-clé	classe	package	sous classe	world
`private`	**Y**			
`protected`	**Y**	**Y**	**Y**	
`public`	**Y**	**Y**	**Y**	**Y**
`[aucun]`	**Y**	**Y**		

Seul les membres publics sont visibles depuis le monde extérieur.

Une classe a toujours accès à ses membres.

Les classes d'un même package protègent uniquement leurs membres privés (à l'intérieur du package)

Une classe fille (ou dérivée) n'a accès qu'aux membres publics et protected de la classe mère.

Les modificateurs d'accès

- **Pour déclarer une classe,** il faut respecter la syntaxe :

[*modifiers*] class ClassName [extends SuperClassName] [implements InterfaceName]

- *modifiers* représente un des mots clés : public, abstract ou final
  - ✓ le mot clé public signifie que la classe peut être utilisée par tous et qu'elle est donc visible de l'extérieur
  - ✓ le mot clé abstract signifie que la classe est composée de méthodes abstraites (i.e des méthodes abstract : sans implémentation)
  - ✓ le mot clé final signifie qu'on ne peut pas hériter de cette classe
- l'écriture extends SuperClassName permet de spécifier que la classe ClassName dérive (hérite) de SuperClassName
- l'écriture implements InterfaceName permet de spécifier que la classe ClassName implémente les méthodes *abstract* contenues dans InterfaceName

Les modificateurs d'accès

- **Pour déclarer une variable**, il faut respecter la syntaxe :

[*access*] [static] [final] [transient] [volatile] *Type* variableName

- *access* sert à spécifier la visibilité de la variable, i.e si elle est public, private ou protected
- le mot clé static permet de spécifier que la variable est une variable de classe
- le mot clé final signifie que la variable est une constante
- le mot clé transient signifie que la variable n'est pas persistante pour l'objet
- le mot clé volatile signifie que la variable peut être modifiée par des threads concurrents et ce de manière asynchrone

Les modificateurs d'accès

- **Pour écrire une méthode**, il faut respecter la syntaxe :

  [*access*] [static] [abstract] [final] [native] [synchronized] *returnType* methodeName([param])
- *access* sert à spécifier la visibilité de la méthode
- le mot clé static signifie qu'il s'agit d'une méthode de classe
- le mot clé abstract signifie que la méthode n'est pas implémentée (i.e. qu'elle n'a pas de code).
- le mot clé final signifie que la méthode ne peut pas être polymorphée, i.e qu'en cas d'héritage on ne peut pas la redéfinir.
- le mot clé native est utile lorsque l'on veut utiliser des fonctions écrites dans un autre langage que Java.
- le mot clé synchronized sert dans le cas où on veut utiliser des threads. Il permet à 2 ou plusieurs méthodes de synchroniser leurs accès aux informations.

Les variables et méthodes de classe

- **Variables de classe**
  - le mot clé static signifie que la variable est une variable de classe
  - les variables de classe sont partagées par toutes les instances d'une classe
  - une variable de classe peut être initialisée lors de sa déclaration. Dans ce cas cette variable est initialisée une seule fois quand la classe est chargée en mémoire (i.e dès que l'on fait appel à la classe)
  - l'utilisation des 2 mots clés public static, est réservé aux constantes

    ```
    public static final String LE_NOM = "inconnu";
    ```
  - une constante peut être utilisée depuis une autre classe. Pour cela il faut pécéder le nom de la variable par le nom de la classe

    ```
    String leNom = Etudiant.LE_NOM
    ```
  - une variable déclarée en private static est partagé par toutes les instances et n'est visible que dans sa propre classe

    ```
    private static int leNombreEtudiants = 0
    ```

Les variables et méthodes de classe

- **Méthodes de classe**
  - le mot clé static signifie que la méthode est une méthode de classe
  - une méthode ne peut être déclarée en static que si elle exécute une action qui est indépendante d'une instance de la classe (i.e un objet spécifique)
  - exemple :

```
public static int nombreEtudiants() {
    return leNombreEtudiants;
}
```

  - pour appeler une méthode static depuis une autre classe, il faut précéder le nom de la méthode par le nom de la classe :

```
int leNombre = Etudiant.nombreEtudiants()
```

  - une méthode de classe doit avoir une signature différente de celle d'une méthode d'instance

Les variables et méthodes de classe

- Variables d'instance
  - Déclarées par ***accès type nom***
  - Référencées par ***objet.nom*** (toto.leNom)
  - Une copie pour chaque instance (objet)
- Variables de classe
  - Déclarées par ***accès* `static` *type nom***
  - Référencées par ***Classe.nom***
  - Une seule copie pour toutes les instances

- Méthodes d'instance
  - Déclarées par ***accès retour nom(…)***
  - Référencées par ***objet.nom(…)***
    - toto.getLeNom()
- Méthodes de classe
  - Déclarées par ***accès* `static` *retour nom(…)***
  - Référencées par ***Classe.nom(…)***

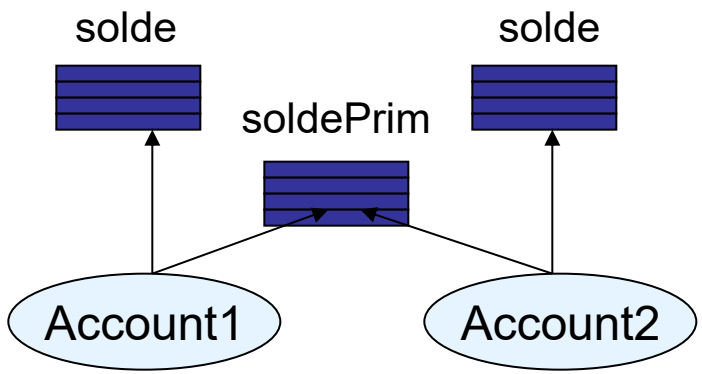
Les variables et méthodes de classe

- Chaque objet a sa propre “mémoire” de ses variables d’instance
- Le système alloue de la mémoire aux variables de classe dès qu’il rencontre la classe. Chaque instance possède la même valeur d’une variable de classe

variable d’instance

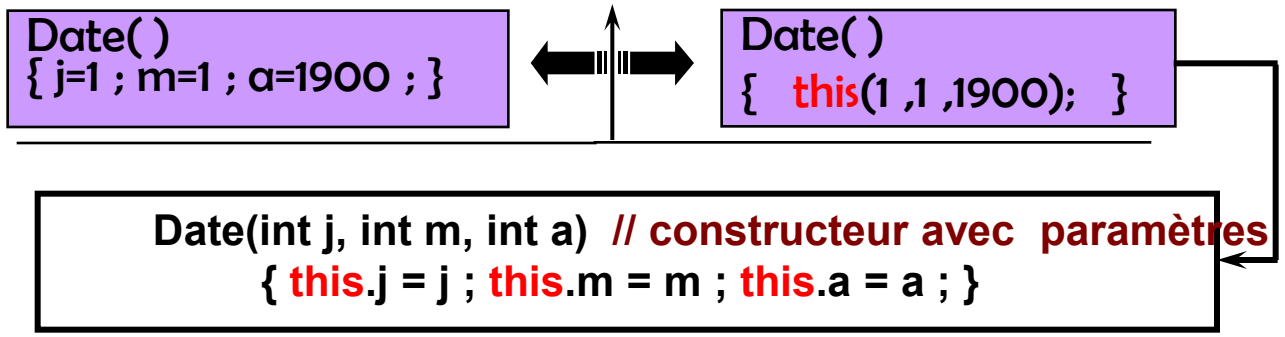
variable de classe

```
class BankAccount {
        int solde;
        static int soldePrim;
        void deposit(int amount){
                solde+=amount;
                soldePrim+=amount;
        }
}
```

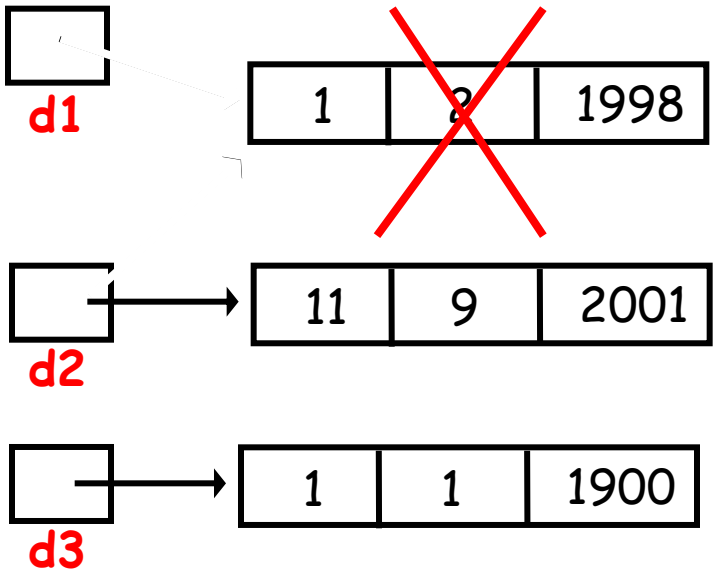


Le mot-clé this

- Le mot-clé this représente une référence sur l'objet courant.
  - celui qui est en train d'exécuter la méthode dans laquelle se trouvent les instructions concernées.
- this peut être utile :
  - this peut servir à lever des ambiguïtés entre des noms d'attributs et des noms d'arguments de méthodes
    - Lorsqu'une variable locale (ou un paramètre) "cache", en portant le même nom, un attribut de la classe.
  - Pour déclencher un constructeur depuis un autre constructeur.



La destruction des objets




```
Date d1;
```

```
d1 = new Date(1,2,1998);
```

```
Date d2;
```

```
d2 = d1;
```

```
Date d3 = new Date( );
```

```
d1 = null ;
```

```
d2=new Date(11,9,2001 );
```

La destruction des objets

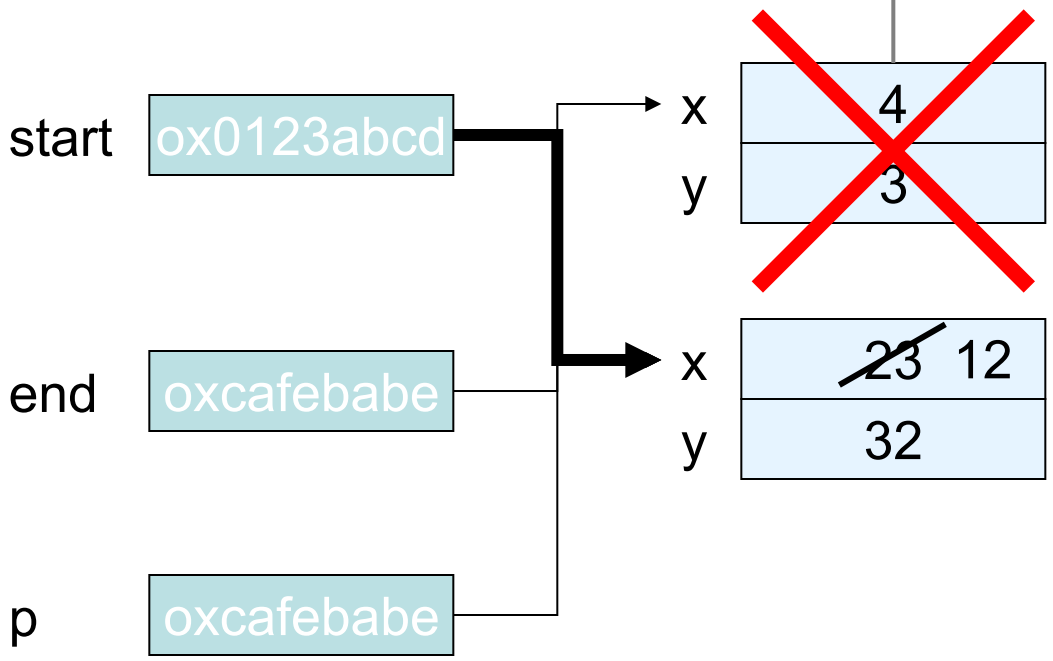
- La destruction des objets est prise en charge par le *garbage collector (GC).*
- Le GC détruit les objets pour lesquels il n'existe plus de référence.
- La récupération de mémoire peut être aussi invoquée explicitement pas le programmeur à des moments bien précis avec la commande System.gc().
- Certaines situations nécessitent un nettoyage spécial que le GC ne peut pas effectuer :
  - Par exemple, certains fichiers ont été ouverts pendant la durée de vie de l'objet et vous voulez vérifier qu'ils sont correctement fermés quand l'objet est détruit.
- Pour cela, la méthode spéciale finalize(), peut être définie.
- Cette méthode (si elle est présente) est appelée par le GC lorsque l'objet est détruit pour assurer un nettoyage spécial.

```
class Point{
 int x=7, y=10;
 Point(int x,int y){this.x=x;this.y=y;}
 void move(int dx,int dy){x+=dx;y+=dy;}
}
```

Il n'y a désormais plus de référence vers l'ancien Point « start », il sera donc détruit par le Garbage Collector

Assignation d'un type de référence

- `Point start=new Point(4,3);`
- `Point end=new Point(23,32);`
- `Point p=end;`
- `p.x=12;`
- `start=p;`



La destruction des objets

```
public class Cercle {
...
void finalize() { System.out.println("Je suis garbage collecte"); }
}
...
Cercle c1;
if (condition) {
Cercle c2 = new Cercle(); // c2 référence une nouvelle instance
c1 = c2;
}
// La référence c2 n'est plus valide mais il reste une référence,c1,sur l'instance
c1=null; // L'instance ne possède plus de référence. Elle n'est plus accessible.
…// A tout moment le gc peut detruire l'objet.
System.gc();
```

Résultat : Affichage (je suis grabage collecte)

Les opérateurs sur les références

- Les seuls opérateurs sur les références sont des opérateurs logiques :
  - **==** : permet de tester si deux références désignent le même objet.
  - **!=** : permet de tester si deux références ne désignent pas le même objet.
  - equals: permet de comparer le contenu de deux références.
  - **instanceof** : permet de tester si l'objet référencé est une instance d'une classe donnée ou d'une de ses sous-classes

```
if ( d1 instanceof Date ) {
    ... Traitement
}
```

Les opérateurs sur les références

- Utilisation de '==' :
  - Il y'a égalité que si d1 et d2 désignent la même zone mémoire représentant une date.

```
Date d1 = new Date(1,2,2000) ;
Date d2 = new Date(1,2,2000) ;
if(d1 == d2) . . . .                    // il n'y a pas égalité

Date d1 = new Date(1,2,2000) ;
Date d2 = d1 ;
if(d1 == d2) . . . .                    // il y a égalité
```

- Pour comparer deux objets il faut utiliser la méthode 'equals':

```
Date d1 = new Date(1,2,2000) ;
Date d2 = new Date(1,2,2000) ;
if(d1.equals(d2)). . . .// il y a égalité si « equals » est définie
```