Définition de classe en Java

Fichier Point.java

```
public class Point {

   private int x, y;

   // Constructeurs
   public Point(int abs, int ord) { this.x = abs; this.y = ord; }
   public Point() { this.x = this.y = 0; }
   public Point(Point aPoint) { this.x = aPoint.x; this.y = aPoint.y; }

   // Accesseurs
   public void setX(int abs) { this.x = abs; }
   public void setY(int ord) { this.y = ord; }
   public int getX() { return this.x; }
   public int getY() { return this.y; }

   public void deplace(int deltaX, int deltaY) { this.x += deltaX; this.y += deltaY; }

   @Override
   public String toString() { return "(" + this.x + ", " + this.y + ")"; }

  /* Méthode de test de la classe. */
   public static void main(String[] args) {
     Point p1 = new Point();
     Point p2 = new Point(10, 15);

     p1.deplace(5, 20);
     System.out.println("p1 : " + p1 + " p2 : " + p2);
    } // main
} // class Point
```


Les constructeurs

- Les constructeurs sont des méthodes portant le même nom que la classe, ils ne renvoient rien (pas même `void`) car un constructeur renvoie toujours implicitement une instance de la classe (l'objet `this`).
- Si on ne définit *aucun* constructeur, Java crée un *constructeur public par défaut*, qui ne fait rien (les attributs seront initialisés avec les valeurs par défaut pour leur types), ou qui appelle le constructeur par défaut de la super-classe dans les situations d'héritage. (Corollaire : pour empêcher l'instanciation d'une classe, il suffit de créer un constructeur *protected* ou *private*, voir plus loin).
- Une classe peut avoir un nombre quelconque de constructeurs, pourvu qu'ils puissent être différenciés par leurs paramètres.
- L'exemple précédent définit 3 constructeurs : un constructeur « général », un constructeur « par défaut » et un constructeur « de copie ».

Les constructeurs

- Lorsqu'il y a plusieurs constructeurs, ou plusieurs méthodes de même nom, les règles qui s'appliquent sont celles de la *surcharge* des méthodes (*method overloading*) et on rappelle que les paramètres par défaut n'existent pas en Java.
- Le mot-clé `this` (qui désigne l'instance de classe) permet également d'appeler un constructeur *à partir d'un autre constructeur* en utilisant la syntaxe `this(...)`. Le constructeur par défaut aurait donc pu être écrit ainsi :

```
public Point() { this(0, 0); } // Appel de Point(abs, ord).
```

- Idem pour le constructeur de copie :

```
public Point(Point pt) { this(pt.x, pt.y); } // Appel de  Point(abs, ord).
```

- À la différence de C++, on notera qu'il faut répéter pour chaque méthode (et chaque attribut) le type de sa visibilité.
- La méthode `main()` est particulière : elle est automatiquement appelée en premier lorsque l'on « exécute » la classe à partir d'une JVM.

Attributs et méthodes de classe

- Au lieu de décrire une instance particulère, un attribut peut s'appliquer à la classe entière (donc être partagé par toutes les instances de cette classe). On parle alors d'*attribut de classe*.
- De même, une méthode peut porter sur toute une classe (donc sur toutes les instances de cette classe). On parle alors de *méthode de classe*.
- Les types de visibilité des attributs et méthodes d'instance s'appliquent également aux attributs et méthodes de classe (voir plus loin).
- Pour déclarer un attribut ou une méthode de classe, il suffit d'utiliser le mot-clé `static` :

Attributs et méthodes statiques

```
protected static int nbPoints = 0; // Nbre de points créés}
...
public static void main(String[] args) {
...
}
```

Attributs et méthodes de classe

- Bien qu'ici, l'initialisation de l'attribut ne soit pas techniquement nécessaire, elle ajoute de la lisibilité. Il faudrait, évidemmment, rajouter une instruction `nbPoints++;` dans les constructeurs... (mais cet exemple est piégé... voir plus loin pourquoi).
- Une méthode de classe ne s'appliquant pas à une instance particulière, elle ne connaît pas `this`. Elle ne peut pas faire référence à des attributs d'instances, ni à des méthodes d'instance.
- On appelle une méthode de classe en utilisant la syntaxe

```
UneClasse.methodeStatique(...);
```

Exemples d'appels de méthodes statiques

```
Math.sqrt(42);
System.out.println(...);
```


Attributs et méthodes de classe

- Dans le dernier exemple, `out` est un attribut (car il n'est pas suivi de parenthèses) appliqué à la classe `System`, c'est donc un *attribut de classe* (ce qui est normal, car la sortie standard est la même pour tout le monde...) de type `PrintStream`.
- La méthode `println()` est une *méthode d'instance* de la classe `PrintStream`. Elle s'applique ici à l'objet `out`.
- Notez que sans la documentation de l'API, cette sémantique est dure à deviner...
- Certaines classes (`Math`, `System`, ...) ne fournissent que des méthodes de classe publiques et des constantes.

Attributs de classe particuliers : les constantes statiques

- Une classe peut définir une ou plusieurs *constantes statiques*, qui seront donc partagées par toutes les instances, et qui suivent les règles de visibilité classique des attributs.
- Une constante statique est un attribut de classe déclaré comme `final`. Par définition, une constante ne peut être modifiée après sa déclaration (si elle a une visibilité publique, elle pourra être consultée en dehors de la classe, mais jamais modifiée).
- Exemple :

```
class Point {
 ...
 public static final int ORIG_ABS = 0;
 ...
}
```

- On pourra y accéder à l'extérieur de la classe par `Point.ORIG_ABS`.

Importations statiques

L'instruction `import static` permet de réaliser des importations statiques, c'est-à-dire d'importer des membres statiques afin d'alléger l'écriture.

Importations statiques

```
import static Point.ORIG_ABS;
import static java.lang.Math.*;
 ...
if (x > ORIG_ABS) ...

if (val > sqrt(42)) ...
```

La deuxième instruction `import` importe tous les membres statiques (constantes et méthodes) de la classe `java.lang.Math`.

Remarque

- Une importation statique ne peut concerner qu'une classe appartenant à un paquetage.
- Vous ne pouvez importer statiquement un membre d'une classe que vous avez définie que si celle-ci appartient à un paquetage, pas au paquetage par défaut.

Initialisateurs statiques

- Lorsqu'une simple initialisation ne suffit pas, on peut utiliser des *initialisateurs* : dans le cas des attributs d'instance, les *constructeurs* jouent généralement ce rôle (voir remarque plus loin), pour les attributs de classe, on utilisera des *initialisateurs statiques*.
- Un initialisateur est appelé automatiquement par Java, il n'a donc pas besoin de nom.
- À la différence d'un constructeur, on ne peut pas lui passer de paramètre. Comme un constructeur, il ne renvoie rien.
- Un initialisateur statique est appelé *une seule fois* : au chargement de la classe, avant que le moindre objet ne soit créé.

Initialisateurs statiques

- La syntaxe d'un initialisateur statique est :

```
static {
  ...    // code quelconque ne connaissant pas this
}
```

- Une classe peut posséder plusieurs initialisateurs statiques : ils seront exécutés dans l'ordre de leur apparition.
- En pratique, les initialisateurs statiques servent pour les initialisations complexes (tableaux), ou lorsqu'une classe définit des méthodes `native` (écrites en C) : l'initialisateur peut alors effectuer les appels `System.load()` et `System.loadLibrary()` pour charger la bibliothèque native implémentant ces méthodes.

Résumé sur l'initialisation des attributs

- Les **attributs de classe** sont créés et initialisés *au premier chargement de la classe*.
- Les **attributs d'instance** sont créés et initialisés *lorsque l'objet concerné est créé*.
- Dans les deux cas, l'ordre d'initialisation est le suivant :
  - Java fournit une valeur par défaut pour chaque type primitif.
  - Si le programmeur fournit une valeur d'initialisation, celle-ci remplace la valeur par défaut.
  - S'il y a des initialisateurs, ils sont exécutés dans l'ordre de leur apparition.
- Dans le cas de la création d'un objet, appel du constructeur par défaut produit automatiquement par Java (si la classe n'en définit pas) ou du constructeur correspondant à l'appel à `new`.

Destruction des objets

- À la différence de C++, il n'y a pas de *destructeurs* en Java. Un ramasse-miettes (*Garbage Collector*) s'occupe de supprimer automatiquement les objets qui ne sont plus accessibles.
- L'avantage du GC est que le programmeur n'a plus à se soucier de libérer les emplacements qu'il a réservés, ce qui diminue donc le risque potentiel des *fuites mémoire*.
- Le GC fonctionne comme un thread de faible priorité. Le point fondamental est qu'*on ne peut pas prévoir l'instant de son exécution*. Même si l'objet est supprimé « logiquement » (parce qu'on est sorti de sa portée locale, par exemple), le GC ne s'exécute pas aussitôt.

Destruction des objets : finalisateurs

- Le GC libère les ressources mémoire utilisées par les objets.
- Si un objet utilise d'autres ressources (fichiers, sockets, connexion à une BD, etc.), elles ne seront pas libérées par le GC...
- Pour ce faire, une classe peut définir une méthode `void finalize()`, qui s'occupera de libérer ces ressources (fermer les fichiers, les sockets, etc.) et qui sera appelée *avant la suppression de l'objet* par le GC.
- L'interpréteur Java peut se terminer *avant* que le GC se soit occupé de tous les objets inutilisés : certains finalisateurs peuvent donc ne jamais être appelés...
- Java ne garantit pas quand le GC aura lieu, ni dans quel ordre les objets seront supprimés. On ne peut donc pas prévoir le moment où un finalisateur sera appelé, ni dans quel ordre.

Visibilité des attributs et méthodes

Java fournit 4 niveaux de visibilité (par ordre décroissant de permissivité) :

- La visibilité *publique* (mot-clé `public`) : l'attribut ou la méthode est directement accessible partout en dehors de la classe.
- La visibilité *protégée* (mot-clé `protected`) : l'attribut ou la méthode n'est accessible qu'à l'intérieur de la classe, des classes héritant de celle-ci (quels que soient les paquetages auxquelles elles appartiennent) et des classes du même paquetage de la classe..
- La visibilité de *paquetage* (défaut) : l'attribut ou la méthode est accessible par toutes les classes du même paquetage. C'est un peu l'équivalent des classes *friend* en C++, mais pour un paquetage.
- La visibilité *privée* (mot-clé `private`) : l'attribut ou la méthode n'est accessible qu'à l'intérieur de la classe.

Visibilité des attributs et méthodes

Le tableau suivant résume les droits d'accès en fonction des types de visibilité :

Accessible dans :	Type de visibilité			
	public	**protected**	**défaut**	**private**
La même classe	Oui	Oui	Oui	Oui
Toute classe du même paquetage	Oui	Oui	Oui	Non
Sous-classe d'un autre paquetage	Oui	Oui	Non	Non
Autre classe, dans un autre paquetage	Oui	Non	Non	Non

Remarques sur la visibilité des attributs et méthodes

- On n'utilisera une visibilité *public* que pour les méthodes et les constantes faisant partie de l'API publique de la classe. Les attributs très souvent utilisés seront non publics et l'on fournira des accesseurs publics (*getters* et *setters*) pour les manipuler en toute sécurité.
- On utilisera la visibilité *protected* pour les attributs et méthodes qui ne sont pas nécessaires à la manipulation de la classe, mais qui peuvent servir à d'éventuelles classes dérivées, faisant partie d'un autre paquetage.
- On utilisera la visibilité par défaut pour les attributs et méthodes que l'on veut cacher à l'extérieur du paquetage, mais que l'on veut partager entre les classes d'un même paquetage.
- On utilisera la visibilité *private* pour les attributs et méthodes qui ne servent qu'à la classe et qui doivent être masqués pour l'extérieur.

Remarques sur la visibilité par défaut et privée

- Comme une classe définie en dehors de tout paquetage est considérée comme appartenant à un *paquetage par défaut*, tous ses attributs et méthodes à visibilité par défaut seront accessibles par les autres classes de ce paquetage (i.e. par toutes les classes n'appartenant pas à un paquetage).
- Il est donc conseillé d'éviter cette visibilité par défaut lorsque l'on ne définit pas un paquetage avec le mot-clé `package`.
- Les visibilités s'appliquent *à la classe*, pas à l'objet. Ceci signifie que, dans une classe `A`, on a accès à tous les membres privés de toutes les instances de `A` :

```
class Truc {
  private int champ;

  public Truc(int val) { champ = val; }
  public Truc(Truc truc) {  // Constructeur de copie
     champ = truc.champ;    // champ est privé à la classe, pas à l'instance
  }

  @Override
  public String toString() { return "" + champ; }
}
```


Accesseurs

- Une bonne pratique de conception consiste à créer des champs privés et à fournir (ou non) des *méthodes accesseurs* en lecture et/ou en écriture.
- Les accesseurs en écriture, notamment, pourront vérifier la cohérence des valeurs fournies.
- En Java, il est d'usage d'appeler ces méthodes `getX()` et `setX()`, où `X` est le nom du champ concerné.
- Certains IDE permettent de créer automatiquement les squelettes de ces méthodes.

Exemple

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

 public Personne(String n, String p, int a) { ... }

 public String getNom()    { return nom; }
 public String getPrenom() { return prenom; }
 public int    getAge()    { return age; }

 public void setNom(String nouveau) { nom = nouveau; }
 public void setAge(int a) { if (a > 0 && age < 110) age = a; }
}
```

Copies et références

Rappels :

- Une déclaration de variable d'un type primitif crée un emplacement pour stocker la valeur de cette variable. Cet emplacement est initialisé avec la valeur par défaut pour le type.
- Une déclaration d'une instance de classe *ne crée pas* un objet mais une *référence* (un pointeur), initialisé à `null`.
- Le passage des paramètres a toujours lieu *par copie* : les paramètres réels ne seront jamais modifiés (mais un objet référencé peut l'être).

Conséquences

- Un objet n'est pas créé par sa déclaration. Toute tentative d'y accéder lèvera une exception `NullPointerException`.
- L'affectation (=) et la comparaison (`==`, `!=`) d'objets n'a pas les effets attendus (voir les méthodes `clone()` et `equals()`).
- Un objet n'est pas défini tant que l'on n'a pas utilisé l'opérateur `new`, qui crée l'objet en appelant un de ses constructeurs.
- Une méthode renvoyant une valeur qui n'est pas d'un type primitif renvoie donc, en fait, une référence... Si cette référence désigne un attribut de la classe, et qu'il est d'un type modifiable, cet attribut pourra ensuite être modifié indirectement via cette référence, quelle que soit sa protection... Le principe d'encapsulation est donc violé.
- Si une méthode doit renvoyer un objet modifiable, il est fortement conseillé de cloner l'attribut et de renvoyer ce clone.

Mauvais Exemple

```
class Machin {
  private int[] truc = {10, 9, 8, 7};   // champ référence privé

  public int[] danger() { return truc; }  // Non !!!

  @Override
  public String toString() {
    StringBuilder result = new StringBuilder();
    for (int elt : truc) {
      result.append(elt + ", ");
    }
    return result.toString();
  }
}

public class Programme {
   public static void main(String[] args) {
     Machin machin = new Machin();
     System.out.println(machin);   // 10, 9, 8, 7,
     machin.danger()[2] = 100;
     System.out.println(machin);   // 10, 9, 100, 7, !!!
   }
}
```


Exemple correct

```
class Machin {
  private int[] truc = {10, 9, 8, 7};   // champ référence privé

  public int[] danger() { return truc.clone(); }  // Oui !

  (...)
}

public class Programme {
   public static void main(String[] args) {
     Machin machin = new Machin();
     System.out.println(machin);   // 10, 9, 8, 7,
     machin.danger()[2] = 100;     // Ici, machin.danger renvoie un clone de truc, pas truc lui-même
     System.out.println(machin);   // 10, 9, 8, 7,
   }
}
```