Programmation objet en Java

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://spiralconnect.univ-lyon1.fr` module "ASPE - Bases de la POO - Java P1 et P2"
48H prévues ≈ 43H de cours+TDs/TPs, 1H - interros, et 2 DS de promos de 2H

Évaluation : 50% Contrôle continu + 50% DS promo + Bonus/Malus TP

Plan

Java est un langage quasi pur POO

- Une **classe** généralise la notion de type.
- Un **objet** généralise la notion de variable ; chaque objet comporte son propre jeux de données.
- En POO pure, on réalise l'**encapsulation des données** : on ne peut accèder aux *champs* d'un objet qu'en recourant aux **méthodes** (≈ fonctions) prévues à cet effet.

Java est un langage quasi pur POO

- Une **classe** généralise la notion de type.
- Un **objet** généralise la notion de variable ; chaque objet comporte son propre jeux de données.
- En POO pure, on réalise l'**encapsulation des données** : on ne peut accèder aux *champs* d'un objet qu'en recourant aux **méthodes** (≈ fonctions) prévues à cet effet.

Java est un langage quasi pur POO

- Une **classe** généralise la notion de type.
- Un **objet** généralise la notion de variable ; chaque objet comporte son propre jeux de données.
- En POO pure, on réalise l'**encapsulation des données** : on ne peut accèder aux *champs* d'un objet qu'en recourant aux **méthodes** ($\approx$ fonctions) prévues à cet effet.

Java est un langage quasi pur POO

Nous allons répondre aux questions suivantes :

- *Comment définir une classe pour représenter des données ?*
- *Comment instancier/fabriquer un objet à partir du modèle/moule de la classe ?*
- *Comment initialiser automatiquement les champs d'un objet dès sa création* (**pour que l'objet soit automatiquement dans un état initial valide**) *?*

Important : on doit écrire une classe CMonType avant de pouvoir instancier un objet de type CMonType.

Java est un langage quasi pur POO

Nous allons répondre aux questions suivantes :

- *Comment définir une classe pour représenter des données ?*
- *Comment instancier/fabriquer un objet à partir du modèle/moule de la classe ?*
- *Comment initialiser automatiquement les champs d'un objet dès sa création* (**pour que l'objet soit automatiquement dans un état initial valide**) *?*

Important : on doit écrire une classe CMonType avant de pouvoir instancier un objet de type CMonType.

Java est un langage quasi pur POO

Nous allons répondre aux questions suivantes :

- *Comment définir une classe pour représenter des données ?*
- *Comment instancier/fabriquer un objet à partir du modèle/ moule de la classe ?*
- *Comment initialiser automatiquement les champs d'un objet dès sa création* (**pour que l'objet soit automatiquement dans un état initial valide**) *?*

Important : on doit écrire une classe CMonType avant de pouvoir instancier un objet de type CMonType.

Java est un langage quasi pur POO

Nous allons répondre aux questions suivantes :

- *Comment définir une classe pour représenter des données ?*
- *Comment instancier/fabriquer un objet à partir du modèle/ moule de la classe ?*
- *Comment initialiser automatiquement les champs d'un objet dès sa création* (**pour que l'objet soit automatiquement dans un état initial valide**) *?*

Important : on doit écrire une classe CMonType avant de pouvoir instancier un objet de type CMonType.

```
public class Point2D {
 ... // Qu'est-ce qu'on va mettre à l'intérieur?
}
```

- Des **champs données** pour représenter les 2 coordonnées x et y d'un objet point 2D ;
  on appelle les champs données d'un objet les **attributs d'instance** de la classe ;
- Des **fonctions pour manipuler ces données**, propres à la classe Point2D ;
  on appelle ces fonctions définies pour manipuler les données d'un objet les **méthodes d'instance** de la classe.

```
public class Point2D {
 ... // Qu'est-ce qu'on va mettre à l'intérieur?
}
```

- Des **champs données** pour représenter les 2 coordonnées x et y d'un objet point 2D ;
  on appelle les champs données d'un objet les **attributs d'instance** de la classe ;
- Des **fonctions pour manipuler ces données**, propres à la classe Point2D ;
  on appelle ces fonctions définies pour manipuler les données d'un objet les **méthodes d'instance** de la classe.

```
public class Point2D {
 ... // Qu'est-ce qu'on va mettre à l'intérieur?
}
```

- Des **champs données** pour représenter les 2 coordonnées x et y d'un objet point 2D ;
  on appelle les champs données d'un objet les **attributs d'instance** de la classe ;
- Des **fonctions pour manipuler ces données**, propres à la classe Point2D ;
  on appelle ces fonctions définies pour manipuler les données d'un objet les **méthodes d'instance** de la classe.

```
public class Point2D {
 ... // Qu'est-ce qu'on va mettre à l'intérieur?
}
```

- Des **champs données** pour représenter les 2 coordonnées x et y d'un objet point 2D ;
  on appelle les champs données d'un objet les **attributs d'instance** de la classe ;
- Des **fonctions pour manipuler ces données**, propres à la classe Point2D ;
  on appelle ces fonctions définies pour manipuler les données d'un objet les **méthodes d'instance** de la classe.

```
public class Point2D {
 ... // Qu'est-ce qu'on va mettre à l'intérieur?
}
```

- Des **champs données** pour représenter les 2 coordonnées x et y d'un objet point 2D ;
  on appelle les champs données d'un objet les **attributs d'instance** de la classe ;
- Des **fonctions pour manipuler ces données**, propres à la classe Point2D ;
  on appelle ces fonctions définies pour manipuler les données d'un objet les **méthodes d'instance** de la classe.

Choix des attributs et des méthodes

Première proposition :

```
public class Point2D {
   // Les attributs : [mode d'accès] [type] [identificateur]
   private int x ; // private => non-accessible à l'extérieur de la classe
   private int y ; // private => encapsulation des données!!

   // Les méthodes : [mode d'accès] [type de retour] [identificateur]([liste arg formels])
   public void initialise(int abs, int ord) // public => accessible à tout le monde
   {                                        // (l'extérieur de la classe + l'intérieur)
      x = abs ;
      y = ord ;
   }

   public void affiche() // accessible à l'extérieur et l'intérieur de la classe
   {
      System.out.println( "Je suis un point 2D de coordonnées " + x + " " + y ) ;
   }
}
```

Choix des attributs et des méthodes

Meilleure proposition : **méthodes puis attributs** :

```
public class Point2D {
   // Les méthodes
   public void initialise(int abs, int ord)
   {
      x = abs ;
      y = ord ;
   }

   public void affiche()
   {
      System.out.println( "Je suis un point 2D de coordonnées " + x + " " + y ) ;
   }

   // Les attributs : on les place généralement à la suite des méthodes
   private int x ;
   private int y ;
}
```

Remarques :

- On peut déclarer une méthode privée (private) : elle ne sera accessible qu'aux autres méthodes de la classe. *Intérêt ?* Avoir une méthode de service dédiée à la classe.
- On peut déclarer un attribut public : cela est fortement déconseillé car il faut respecter le principe de l'encapsulation des données.

Remarques :

- On peut déclarer une méthode privée (private) : elle ne sera accessible qu'aux autres méthodes de la classe. *Intérêt ?* Avoir une méthode de service dédiée à la classe.
- On peut déclarer un attribut public : cela est fortement déconseillé car il faut respecter le principe de l'encapsulation des données.

Remarques :

- On peut déclarer une méthode privée (private) : elle ne sera accessible qu'aux autres méthodes de la classe. *Intérêt ?* Avoir une méthode de service dédiée à la classe.
- On peut déclarer un attribut public : cela est fortement déconseillé car il faut respecter le principe de l'encapsulation des données.

Remarques :

- On peut déclarer une méthode privée (private) : elle ne sera accessible qu'aux autres méthodes de la classe. *Intérêt ?* Avoir une méthode de service dédiée à la classe.
- On peut déclarer un attribut public : cela est fortement déconseillé car il faut respecter le principe de l'encapsulation des données.

Remarques :

- On peut déclarer une méthode privée (private) : elle ne sera accessible qu'aux autres méthodes de la classe. *Intérêt ?* Avoir une méthode de service dédiée à la classe.
- On peut déclarer un attribut public : cela est fortement déconseillé car il faut respecter le principe de l'encapsulation des données.

Remarques :

- On peut déclarer une méthode privée (private) : elle ne sera accessible qu'aux autres méthodes de la classe. *Intérêt ?* Avoir une méthode de service dédiée à la classe.
- On peut déclarer un attribut public : cela est fortement déconseillé car il faut respecter le principe de l'encapsulation des données.

Remarques :

- Nous avons introduit 2 modes d'accès pour les membres d'une classe : `private` et `public` ;
  il en existe 2 autres : un mode *protected* (cf. cours sur l'héritage) et un mode *package-private* (en l'absence de mot d'accès, cf. cours sur les paquetages).
- En C/C++ on distingue la déclaration d'une fonction/d'une méthode de sa définition : cette distinction n'existe pas en Java !

Remarques :

- Nous avons introduit 2 modes d'accès pour les membres d'une classe : `private` et `public` ;
  il en existe 2 autres : un mode *protected* (cf. cours sur l'héritage) et un mode *package-private* (en l'absence de mot d'accès, cf. cours sur les paquetages).
- En C/C++ on distingue la déclaration d'une fonction/d'une méthode de sa définition : cette distinction n'existe pas en Java !

Remarques :

- **Un attribut n'est pas une variable au sens usuel :** un attribut est attaché à un objet tandis qu'une variable est attachée à une méthode ou fonction. Les mots-clés d'accessibilité ne s'appliquent pas aux variables.
- Si j'ai besoin de faire des calculs, je déclare des variables locales à ma méthode et non pas des attributs !

Remarques :

- **Un attribut n'est pas une variable au sens usuel** : un attribut est attaché à un objet tandis qu'une variable est attachée à une méthode ou fonction. Les mots-clés d'accessibilité ne s'appliquent pas aux variables.
- Si j'ai besoin de faire des calculs, je déclare des variables locales à ma méthode et non pas des attributs !

Remarques :

- **Un attribut n'est pas une variable au sens usuel** : un attribut est attaché à un objet tandis qu'une variable est attachée à une méthode ou fonction. Les mots-clés d'accessibilité ne s'appliquent pas aux variables.
- Si j'ai besoin de faire des calculs, je déclare des variables locales à ma méthode et non pas des attributs !

Remarques :

- **Un attribut n'est pas une variable au sens usuel** : un attribut est attaché à un objet tandis qu'une variable est attachée à une méthode ou fonction. Les mots-clés d'accessibilité ne s'appliquent pas aux variables.
- Si j'ai besoin de faire des calculs, je déclare des variables locales à ma méthode et non pas des attributs !

La classe Point2D **va permettre d'instancier des objets de type** Point2D. Elle permettra de créer autant d'instances d'objet de type Point2D que l'on veut !

Que réalise la déclaration suivante en Java ?

```
Point2D p;
```

Contrairement à la déclaration d'un type primitif, l'instruction précédente ne va pas réserver la mémoire pour un objet de type Point2D, mais seulement un emplacement pour stocker une **référence** à un objet de type Point2D.

Une référence $\approx$ un pointeur "caché".

La classe Point2D **va permettre d'instancier des objets de type** Point2D. Elle permettra de créer autant d'instances d'objet de type Point2D que l'on veut !

Que réalise la déclaration suivante en Java ?

```
Point2D p;
```

Contrairement à la déclaration d'un type primitif, l'instruction précédente ne va pas réserver la mémoire pour un objet de type Point2D, mais seulement un emplacement pour stocker une **référence** à un objet de type Point2D.

Une référence ≈ un pointeur "caché".

La classe Point2D **va permettre d'instancier des objets de type** Point2D. Elle permettra de créer autant d'instances d'objet de type Point2D que l'on veut !

Que réalise la déclaration suivante en Java ?
Point2D p ;

Contrairement à la déclaration d'un type primitif, l'instruction précédente ne va pas réserver la mémoire pour un objet de type Point2D, mais seulement un emplacement pour stocker une **référence** à un objet de type Point2D.

Une référence ≈ un pointeur "caché".

La classe Point2D **va permettre d'instancier des objets de type** Point2D. Elle permettra de créer autant d'instances d'objet de type Point2D que l'on veut !

Que réalise la déclaration suivante en Java ?
Point2D p ;

Contrairement à la déclaration d'un type primitif, l'instruction précédente ne va pas réserver la mémoire pour un objet de type Point2D, mais seulement un emplacement pour stocker une **référence** à un objet de type Point2D.

Une référence ≈ un pointeur "caché".

La classe Point2D **va permettre d'instancier des objets de type** Point2D. Elle permettra de créer autant d'instances d'objet de type Point2D que l'on veut !

Que réalise la déclaration suivante en Java ?
Point2D p ;

Contrairement à la déclaration d'un type primitif, l'instruction précédente ne va pas réserver la mémoire pour un objet de type Point2D, mais seulement un emplacement pour stocker une **référence** à un objet de type Point2D.

Une référence $\approx$ un pointeur "caché".

L'opérateur unaire new

Syntaxe : `new UnNomDeClasse();`

L'opérateur `new` **réserve la mémoire** (*allocation dynamique*) **pour un objet du type de la classe passée en argument et retourne une référence vers cet objet.**

Ensuite il faut récupérer cette référence pour manipuler l'objet :

```
Point2D ref = new Point2D();
```

L'opérateur unaire new

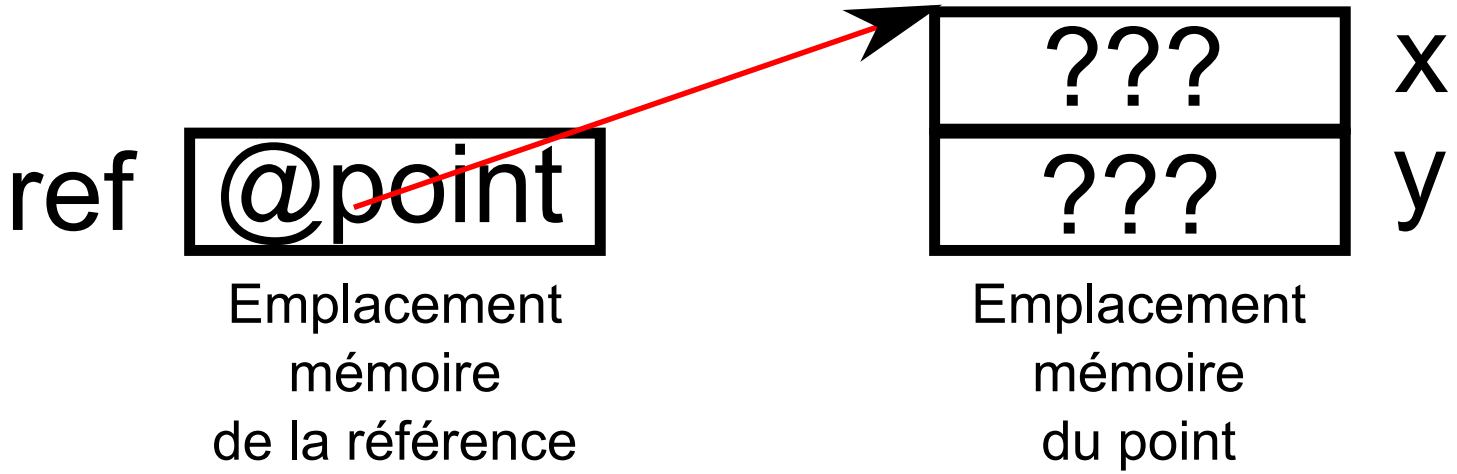
Syntaxe : `new UnNomDeClasse();`

L'opérateur new **réserve la mémoire** (*allocation dynamique*) **pour un objet du type de la classe passée en argument et retourne une référence vers cet objet**.

Ensuite il faut récupérer cette référence pour manipuler l'objet :
`Point2D ref = new Point2D();`

Qu'est-ce qu'une référence en Java ?

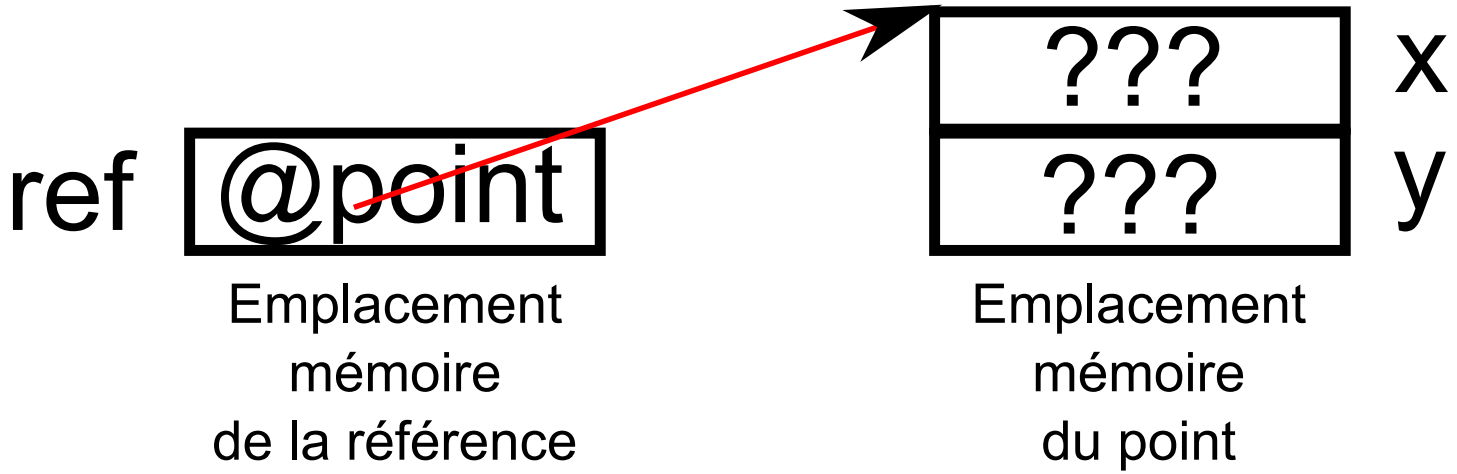
```
Point2D ref = new Point2D();
```



A ce stade je ne connais pas les valeurs de x et y, je connais uniquement la valeur de la référence.

Qu'est-ce qu'une référence en Java ?

```
Point2D ref = new Point2D();
```



A ce stade je ne connais pas les valeurs de x et y, je connais uniquement la valeur de la référence.

Où résident dans la mémoire les *références* et les *objets* créés via l'opérateur new ?

Les références dans la **pile** associée au processus (RAM, allocation automatique).

Où résident dans la mémoire les *références* et les *objets* créés via l'opérateur new ?

Les références dans la **pile** associée au processus (RAM, allocation automatique).

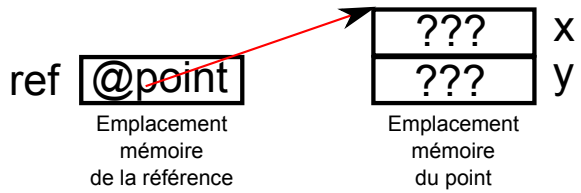
Les objets dans le **tas** associé au processus (RAM, allocation dynamique via l'opérateur new).

Comment utiliser l'objet alloué via sa référence ?

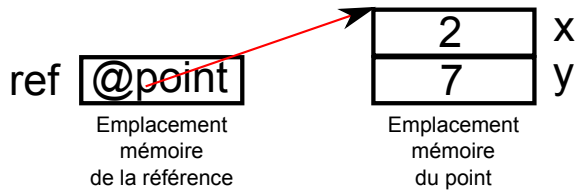
```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref ;
      ref = new Point2D() ;
      ref.initialise(2,7) ; ref.affiche() ;

      Point2D ref2 = new Point2D() ;
      ref2.initialise(-5,4) ; ref2.affiche() ;
   }
}
```

Attention : les 2 points créés ont des emplacements mémoires différents !

ref = new Point2D() ;



ref.initialise(2,7) ;

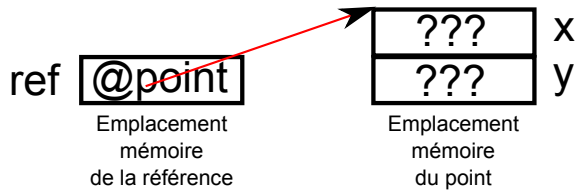


Comment utiliser l'objet alloué via sa référence ?

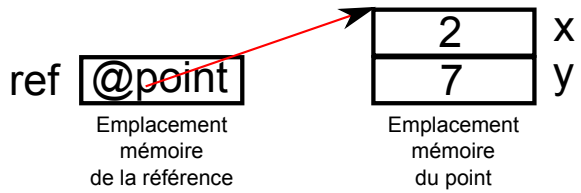
```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref ;
      ref = new Point2D() ;
      ref.initialise(2,7) ; ref.affiche() ;

      Point2D ref2 = new Point2D() ;
      ref2.initialise(-5,4) ; ref2.affiche() ;
   }
}
```

Attention : les 2 points créés ont des emplacements mémoires différents !

ref = new Point2D() ;



ref.initialise(2,7) ;



Comment utiliser l'objet alloué via sa référence ?

ref2.initialise(-5,4) ne changera pas le contenu référencé par ref.

Quelle est la sortie console du programme précédent ?

Comment utiliser l'objet alloué via sa référence ?

`ref2.initialise(-5,4)` ne changera pas le contenu référencé par `ref`.

Quelle est la sortie console du programme précédent ?

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche commune :

- 1 fichier source *MaClasse.java* par classe *MaClasse* ;
- **1 seul fichier à exécuter par la JVM**, qui doit contenir une méthode publique `main` ; la classe contenant la méthode `main` doit être publique ;
- si une classe `A` dépend d'une classe `B`, alors `B.java` doit être compilée avant `A.java` ; le compilateur `javac` doit avoir accès aux fichiers compilés.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche commune :

- 1 fichier source *MaClasse.java* par classe *MaClasse* ;
- **1 seul fichier à exécuter par la JVM**, qui doit contenir une méthode publique `main` ; la classe contenant la méthode `main` doit être publique ;
- si une classe `A` dépend d'une classe `B`, alors `B.java` doit être compilée avant `A.java` ; le compilateur `javac` doit avoir accès aux fichiers compilés.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche commune :

- 1 fichier source *MaClasse.java* par classe *MaClasse* ;
- **1 seul fichier à exécuter par la JVM**, qui doit contenir une méthode publique `main` ; la classe contenant la méthode `main` doit être publique ;
- si une classe `A` dépend d'une classe `B`, alors `B.java` doit être compilée avant `A.java` ; le compilateur `javac` doit avoir accès aux fichiers compilés.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche pour Point2D.java et TestPoint2D.java :

1. javac -sourcepath . TestPoint2D.java
2. java -classpath . TestPoint2D

Grâce à l'option -sourcepath pour spécifier le répertoire contenant les fichiers des classes Java, javac va trouver dans le répertoire "." les classes dont dépend TestPoint2D et les compiler si elles ne le sont pas déjà.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche pour Point2D.java et TestPoint2D.java :

1. javac -sourcepath . TestPoint2D.java
2. java -classpath . TestPoint2D

Grâce à l'option -sourcepath pour spécifier le répertoire contenant les fichiers des classes Java, javac va trouver dans le répertoire "." les classes dont dépend TestPoint2D et les compiler si elles ne le sont pas déjà.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche pour Point2D.java et TestPoint2D.java si les fichiers .class doivent être dans le dossier ./out :

1. javac -d ./out -sourcepath . TestPoint2D.java
2. java -classpath ./out TestPoint2D

Comment mettre en oeuvre un programme avec plusieurs classes ?

Démarche pour Point2D.java et TestPoint2D.java si les fichiers .class doivent être dans le dossier ./out :

1. javac -d ./out -sourcepath . TestPoint2D.java
2. java -classpath ./out TestPoint2D

Comment mettre en oeuvre un programme avec plusieurs classes ?

Remarques :

- une classe est soit publique soit rien ; *ne rien mettre signifie que la classe n'est accessible qu'au sein du même paquetage (package-private)* ;
- on peut avoir plusieurs classes dans un même fichier source, mais **UNE SEULE doit être PUBLIQUE** ;
  c'est pourquoi, pour pouvoir réutiliser plus souvent une classe, **on écrira dans la grand majorité des cas 1 classe par fichier**, cette dernière étant publique.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Remarques :

- une classe est soit publique soit rien ; *ne rien mettre signifie que la classe n'est accessible qu'au sein du même paquetage (package-private)* ;
- on peut avoir plusieurs classes dans un même fichier source, mais **UNE SEULE doit être PUBLIQUE** ;
  c'est pourquoi, pour pouvoir réutiliser plus souvent une classe, **on écrira dans la grand majorité des cas 1 classe par fichier**, cette dernière étant publique.

Comment mettre en oeuvre un programme avec plusieurs classes ?

Remarques :

- une classe est soit publique soit rien ; *ne rien mettre signifie que la classe n'est accessible qu'au sein du même paquetage (package-private)* ;
- on peut avoir plusieurs classes dans un même fichier source, mais **UNE SEULE doit être PUBLIQUE** ;
  c'est pourquoi, pour pouvoir réutiliser plus souvent une classe, **on écrira dans la grand majorité des cas 1 classe par fichier**, cette dernière étant publique.

Pourquoi la méthode main de test n'a pas été écrite dans Point2D ?

2 raisons :

- une méthode d'une classe a accès aux attributs privés de la classe, ce qu'on cherche à éviter dans un test (*on doit rester dans les conditions usuelles d'utilisation*) ;
- les tests unitaires doivent être conçus de façon indépendante (*les tests ont leurs propres dépendances*, e.g. JUnit).

Pourquoi la méthode main de test n'a pas été écrite dans Point2D ?

2 raisons :

- une méthode d'une classe a accès aux attributs privés de la classe, ce qu'on cherche à éviter dans un test (*on doit rester dans les conditions usuelles d'utilisation*) ;
- les tests unitaires doivent être conçus de façon indépendante (*les tests ont leurs propres dépendances*, e.g. JUnit).

Intérêt des constructeurs

Actuellement pour initialiser un objet de type Point2D, le programmeur doit utiliser la méthode initialise.

S'il oublie de le faire, l'objet en question contiendra des valeurs inconnues. En fait pas tout à fait...

La notion de constructeur permet d'automatiser l'initialisation des objets directement après leur instanciation.

Intérêt des constructeurs

Actuellement pour initialiser un objet de type Point2D, le programmeur doit utiliser la méthode initialise.

S'il oublie de le faire, l'objet en question contiendra des valeurs inconnues. En fait pas tout à fait...

La notion de constructeur permet d'automatiser l'initialisation des objets directement après leur instanciation.

Intérêt des constructeurs

Actuellement pour initialiser un objet de type Point2D, le programmeur doit utiliser la méthode initialise.

S'il oublie de le faire, l'objet en question contiendra des valeurs inconnues. En fait pas tout à fait...

La notion de constructeur permet d'automatiser l'initialisation des objets directement après leur instanciation.

Intérêt des constructeurs

Actuellement pour initialiser un objet de type Point2D, le programmeur doit utiliser la méthode initialise.

S'il oublie de le faire, l'objet en question contiendra des valeurs inconnues. En fait pas tout à fait...

La notion de constructeur permet d'automatiser l'initialisation des objets directement après leur instanciation.

Caractéristiques des constructeurs

Un constructeur est une méthode *sans valeur de retour* portant le même nom que la classe.

```
public class MaClasse {
   public MaClasse(...) { ... }
   ...
}
```

Remarques :

- choisir le type de retour `void` pour un constructeur engendre une erreur de compilation ;
- une classe peut avoir 0, 1 ou plusieurs constructeurs (*leurs signatures étant alors différentes*) ;

Caractéristiques des constructeurs

Un constructeur est une méthode *sans valeur de retour* portant le même nom que la classe.

```
public class MaClasse {
   public MaClasse(...) { ... }
   ...
}
```

Remarques :

- choisir le type de retour void pour un constructeur engendre une erreur de compilation ;
- une classe peut avoir 0, 1 ou plusieurs constructeurs (*leurs signatures étant alors différentes*) ;

Caractéristiques des constructeurs

Un constructeur est une méthode *sans valeur de retour* portant le même nom que la classe.

```
public class MaClasse {
   public MaClasse(...) { ... }
   ...
}
```

Remarques :

- choisir le type de retour void pour un constructeur engendre une erreur de compilation ;
- une classe peut avoir 0, 1 ou plusieurs constructeurs (*leurs signatures étant alors différentes*) ;

Caractéristiques des constructeurs

Remarques (fin) :

- la syntaxe `new Classe()` n'est autorisée que si la classe `Classe` ne possède pas de constructeur ou si elle possède un constructeur sans argument ;
- un constructeur ne peut pas être appelé depuis un objet (déjà construit !) ;
  `ref.Point2D()` engendre une erreur de compilation.

Caractéristiques des constructeurs

Remarques (fin) :

- la syntaxe `new Classe()` n'est autorisée que si la classe `Classe` ne possède pas de constructeur ou si elle possède un constructeur sans argument ;
- un constructeur ne peut pas être appelé depuis un objet (déjà construit !) ;

`ref.Point2D()` engendre une erreur de compilation.

Caractéristiques des constructeurs

Remarques (fin) :

- la syntaxe `new Classe()` n'est autorisée que si la classe `Classe` ne possède pas de constructeur ou si elle possède un constructeur sans argument ;
- un constructeur ne peut pas être appelé depuis un objet (déjà construit !) ;
  `ref.Point2D()` engendre une erreur de compilation.

Initialisation des champs d'un objet

Contrairement aux variables locales, les champs d'un objet sont toujours initialisés *par défaut*. Les boolean à false, les char, entier ou flottant à 0 et les références sur des objets à null.

La création d'un objet entraîne, dans cet ordre :

1. l'initialisation des champs avec une valeur par défaut,
2. l'initialisation explicite des champs déclarés avec une *valeur explicite* ;
3. et finalement l'initialisation selon les instructions du corps du constructeur appelé.

Il est préférable d'effectuer les initialisations dans le constructeur.

Initialisation des champs d'un objet

Contrairement aux variables locales, les champs d'un objet sont toujours initialisés *par défaut*. Les boolean à false, les char, entier ou flottant à 0 et les références sur des objets à null.

La création d'un objet entraîne, dans cet ordre :

1. l'initialisation des champs avec une valeur par défaut,
2. l'initialisation explicite des champs déclarés avec une *valeur explicite* ;
3. et finalement l'initialisation selon les instructions du corps du constructeur appelé.

Il est préférable d'effectuer les initialisations dans le constructeur.

Initialisation des champs d'un objet

Contrairement aux variables locales, les champs d'un objet sont toujours initialisés *par défaut*. Les boolean à false, les char, entier ou flottant à 0 et les références sur des objets à null.

La création d'un objet entraîne, dans cet ordre :

1. l'initialisation des champs avec une valeur par défaut,
2. l'initialisation explicite des champs déclarés avec une *valeur explicite* ;
3. et finalement l'initialisation selon les instructions du corps du constructeur appelé.

Il est préférable d'effectuer les initialisations dans le constructeur.

Initialisation des champs d'un objet

Contrairement aux variables locales, les champs d'un objet sont toujours initialisés *par défaut*. Les boolean à false, les char, entier ou flottant à 0 et les références sur des objets à null.

La création d'un objet entraîne, dans cet ordre :

1. l'initialisation des champs avec une valeur par défaut,
2. l'initialisation explicite des champs déclarés avec une *valeur explicite* ;
3. et finalement l'initialisation selon les instructions du corps du constructeur appelé.

Il est préférable d'effectuer les initialisations dans le constructeur.

Initialisation des champs d'un objet

Contrairement aux variables locales, les champs d'un objet sont toujours initialisés *par défaut*. Les boolean à false, les char, entier ou flottant à 0 et les références sur des objets à null.

La création d'un objet entraîne, dans cet ordre :

1. l'initialisation des champs avec une valeur par défaut,
2. l'initialisation explicite des champs déclarés avec une *valeur explicite* ;
3. et finalement l'initialisation selon les instructions du corps du constructeur appelé.

Il est préférable d'effectuer les initialisations dans le constructeur.

Initialisation des champs d'un objet

Contrairement aux variables locales, les champs d'un objet sont toujours initialisés *par défaut*. Les boolean à `false`, les char, entier ou flottant à `0` et les références sur des objets à `null`.

La création d'un objet entraîne, dans cet ordre :

1. l'initialisation des champs avec une valeur par défaut,
2. l'initialisation explicite des champs déclarés avec une *valeur explicite* ;
3. et finalement l'initialisation selon les instructions du corps du constructeur appelé.

Il est préférable d'effectuer les initialisations dans le constructeur.

Initialisation des champs d'un objet

```
public class TestA {
   public static main(String args[])
   {
      A a = new A() ;
      a.affiche() ;   // => donnez le résultat
   }
}
class A { // TestA et A sont dans le même fichier => seule TestA est publique,
          // A est package-private
   // Les méthodes
   public A() // public => accessible à l'extérieur de la classe et du package de la classe
   {
      np = n * p ;
      n = 5 ;
   }

   public void affiche() // public => accessible à l'extérieur de la classe et du package
   {
      System.out.println( "n = " + n + ", p = " + p + ", np = " + np) ;
   }

   // Les attributs : on les place généralement à la suite des méthodes
   private int n = 20, p =10 ; // initialisation explicite
   private int np ;            // pas d'initialisation explicite
}
```

Initialisation d'un champs "final" d'un objet

final **impose une initialisation unique *sans prendre en compte l'initialisation par défaut*.**

```
class A {

   public A(int mm)
   {
      n = 5 ;
      m = mm ;
   }

   private final int p = 10 ; // initialisation explicite => OK
   private final int n ;      // initialisation au niveau du constructeur => OK
   private final int m = 2 ;  // initialisation explicite et au niveau du constructeur => ERREUR
   private final int l ;      // pas d'initialisation => ERREUR DE COMPILATION
}
```

Règle : un champ final doit être initialisé soit explicitement, soit au plus tard au niveau du constructeur.

Initialisation d'un champs "final" d'un objet

final **impose une initialisation unique *sans prendre en compte l'initialisation par défaut*.**

```
class A {

   public A(int mm)
   {
      n = 5 ;
      m = mm ;
   }

   private final int p = 10 ; // initialisation explicite => OK
   private final int n ;      // initialisation au niveau du constructeur => OK
   private final int m = 2 ;  // initialisation explicite et au niveau du constructeur => ERREUR
   private final int l ;      // pas d'initialisation => ERREUR DE COMPILATION
}
```

Règle : **un champ final** doit être initialisé soit explicitement, soit au plus tard au niveau du constructeur.

La notion de contrat

Une classe doit rendre un certain nombre de *services*, tout en garantissant que ces services continueront de réaliser ce qu'ils sont sensés faire au cours des évolutions successives de la classe.

Un service est caractérisé par

- la *signature* de sa méthode publique,
- et le *comportement* de cette méthode.

L'encapsulation des données permet au concepteur de la classe de changer de représentation des données (comme bon lui semble) de manière transparente à l'utilisateur d'un service.

La notion de contrat

Une classe doit rendre un certain nombre de *services*, tout en garantissant que ces services continueront de réaliser ce qu'ils sont sensés faire au cours des évolutions successives de la classe.

Un service est caractérisé par

- la *signature* de sa méthode publique,
- et le *comportement* de cette méthode.

L'encapsulation des données permet au concepteur de la classe de changer de représentation des données (comme bon lui semble) de manière transparente à l'utilisateur d'un service.

La notion de contrat

Une classe doit rendre un certain nombre de *services*, tout en garantissant que ces services continueront de réaliser ce qu'ils sont sensés faire au cours des évolutions successives de la classe.

Un service est caractérisé par

- la *signature* de sa méthode publique,
- et le *comportement* de cette méthode.

L'encapsulation des données permet au concepteur de la classe de changer de représentation des données (comme bon lui semble) de manière transparente à l'utilisateur d'un service.

La notion de contrat

Une classe doit rendre un certain nombre de *services*, tout en garantissant que ces services continueront de réaliser ce qu'ils sont sensés faire au cours des évolutions successives de la classe.

Un service est caractérisé par

- la *signature* de sa méthode publique,
- et le *comportement* de cette méthode.

L'encapsulation des données permet au concepteur de la classe de changer de représentation des données (comme bon lui semble) de manière transparente à l'utilisateur d'un service.

La notion de contrat

Au lieu de contrat, on parle souvent d'**abstraction des données**.

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

G. Booch, Object-Oriented Design With Applications, Benjamin/-Cummings, Menlo Park, California, 1991.

La notion de contrat

Au lieu de contrat, on parle souvent d'**abstraction des données**.

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

G. Booch, Object-Oriented Design With Applications, Benjamin/-Cummings, Menlo Park, California, 1991.

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

La notion de contrat

Si on a besoin d'une classe pour représenter un point 2D, le concepteur peut choisir soit les *coordonnées cartésiennes*, soit les *coordonnées polaires* pour le représenter, et il peut changer de représentation au cours du temps.

Serait-il souhaitable que la classe Point2D possède les méthodes suivantes ?

- public int getX() { ... } **OUI**
- public int getY() { ... } **OUI**
- public void setX(int abs) { ... } **NON**
- public void setY(int ord) { ... } **NON**
- public void setPosition(int abs, int ord) { ... } **OUI**

Vocabulaire

1. Les *méthodes d'accès* dont le nom est de la forme getXXXX sont appelées des *getters* ou *accesseurs*.
2. Les *méthodes d'altération* dont le nom est de la forme setXXXX sont appelées des *setters* ou *mutateurs*.

Attention : il n'est pas judicieux de prévoir systématiquement un setter pour chaque membre privé d'une classe (cf. slide précédent).
=> la représentation interne doit être cachée à l'utilisateur de la classe (pour laisser la liberté de changer de représentation).

Vocabulaire

1. Les *méthodes d'accès* dont le nom est de la forme getXXXX sont appelées des *getters* ou *accesseurs*.
2. Les *méthodes d'altération* dont le nom est de la forme setXXXX sont appelées des *setters* ou *mutateurs*.

Attention : il n'est pas judicieux de prévoir systématiquement un setter pour chaque membre privé d'une classe (cf. slide précédent).
=> la représentation interne doit être cachée à l'utilisateur de la classe (pour laisser la liberté de changer de représentation).

Vocabulaire

1. Les *méthodes d'accès* dont le nom est de la forme getXXXX sont appelées des *getters* ou *accesseurs*.
2. Les *méthodes d'altération* dont le nom est de la forme setXXXX sont appelées des *setters* ou *mutateurs*.

Attention : il n'est pas judicieux de prévoir systématiquement un setter pour chaque membre privé d'une classe (cf. slide précédent).
=> la représentation interne doit être cachée à l'utilisateur de la classe (pour laisser la liberté de changer de représentation).

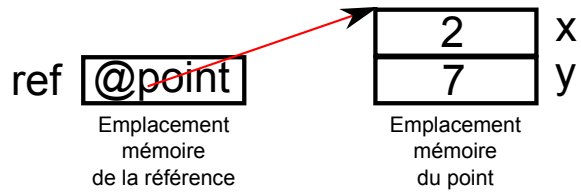
Plan

On manipule des références...

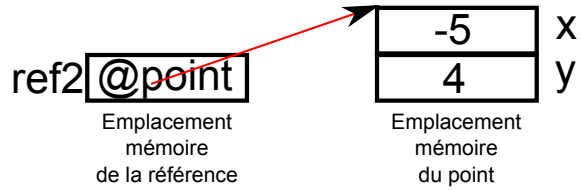
```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;

      Point2D ref2 = new Point2D(-5,4) ;
   }
}
```

Les 2 points créés ont des emplacements mémoires différents !

ref = new Point2D(2,7) ;



ref2 = new Point2D(-5,4) ;

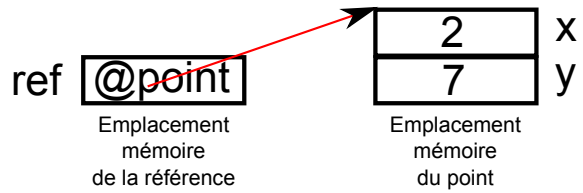


On manipule des références...

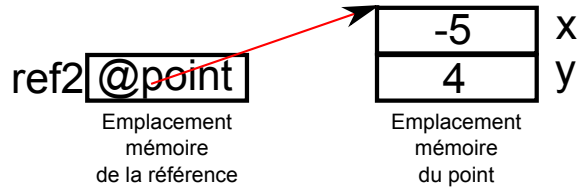
```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;

      Point2D ref2 = new Point2D(-5,4) ;
   }
}
```

Les 2 points créés ont des emplacements mémoires différents !

ref = new Point2D(2,7) ;

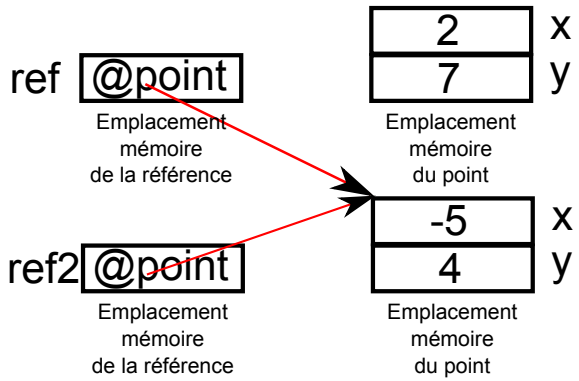


ref2 = new Point2D(-5,4) ;



On manipule des références...

```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;

      Point2D ref2 = new Point2D(-5,4) ;

      ref = ref2 ; // affectation de référence
   }
}
```

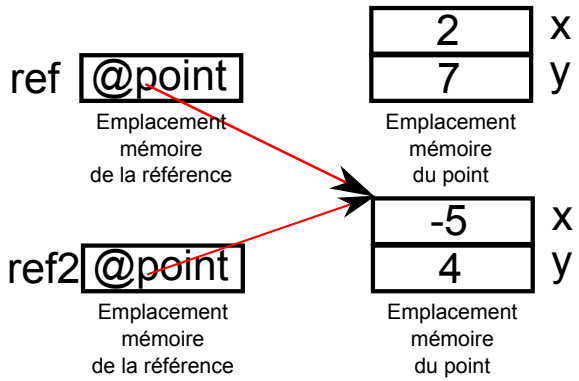


La référence contenue dans ref2 est copiée dans ref.

ref et ref2 désignent le même objet et non pas 2 objets différents de même "valeur".

On manipule des références...

```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;

      Point2D ref2 = new Point2D(-5,4) ;

      ref = ref2 ; // affectation de référence
   }
}
```



La référence contenue dans ref2 est copiée dans ref.
ref et ref2 désignent le même objet et non pas 2 objets différents de même "valeur".

2 comportements différents

- **Une référence non initialisée** (sans valeur) **ne peut pas être utilisée** sinon le programme ne compilera pas.
- **Une référence initialisée à** `null` **ne peut pas être utilisée** sinon le programme lancera une *exception* à l'exécution (`NullPointerException`). Et si l'exception n'est pas traitée, l'exécution du programme s'arrêtera.

  La gestion des exceptions sera traitée dans un autre cours.

2 comportements différents

- **Une référence non initialisée** (sans valeur) **ne peut pas être utilisée** sinon le programme ne compilera pas.
- **Une référence initialisée à** null **ne peut pas être utilisée** sinon le programme lancera une *exception* à l'exécution (NullPointerException). Et si l'exception n'est pas traitée, l'exécution du programme s'arrêtera.

La gestion des exceptions sera traitée dans un autre cours.

2 comportements différents

- **Une référence non initialisée** (sans valeur) **ne peut pas être utilisée** sinon le programme ne compilera pas.
- **Une référence initialisée à** `null` **ne peut pas être utilisée** sinon le programme lancera une *exception* à l'exécution (`NullPointerException`). Et si l'exception n'est pas traitée, l'exécution du programme s'arrêtera.

La gestion des exceptions sera traitée dans un autre cours.

2 comportements différents

- **Une référence non initialisée** (sans valeur) **ne peut pas être utilisée** sinon le programme ne compilera pas.
- **Une référence initialisée à** null **ne peut pas être utilisée** sinon le programme lancera une *exception* à l'exécution (NullPointerException). Et si l'exception n'est pas traitée, l'exécution du programme s'arrêtera.

La gestion des exceptions sera traitée dans un autre cours.

Les opérateurs binaires == et != s'appliquent aux références

- a == b est vrai uniquement si a et b font référence à un seul et même objet (mêmes adresses).
- a != b est vrai si a et b font référence à 2 objets bien distincts en mémoire (adresses différentes).

Les opérateurs binaires == et != s'appliquent aux références

- a == b est vrai uniquement si a et b font référence à un seul et même objet (mêmes adresses).
- a != b est vrai si a et b font référence à 2 objets bien distincts en mémoire (adresses différentes).

Recopie explicite d'un objet dans un nouvel objet

On souhaite parfois obtenir un nouvel objet, copie d'un autre objet de même type, tel que l'objet original et sa copie aient des emplacements mémoires bien distincts.

1. Créer un nouvel objet via l'opérateur new.
2. Copier tous les membres données de l'objet original dans le nouvel objet.

Recopie explicite d'un objet dans un nouvel objet

On souhaite parfois obtenir un nouvel objet, copie d'un autre objet de même type, tel que l'objet original et sa copie aient des emplacements mémoires bien distincts.

1. Créer un nouvel objet via l'opérateur new.
2. Copier tous les membres données de l'objet original dans le nouvel objet.

Recopie explicite d'un objet dans un nouvel objet

Problème : Même si la classe de l'objet est munie des *getters* et *setters* adéquates, *il sera difficile voire impossible de garantir que la copie des valeurs des membres soit indépendante de la représentation des données*.

Solution : munir la classe en question d'une méthode de copie/de clonage pour qu'un utilisateur extérieur n'ait pas à connaître l'implémentation actuelle.

Recopie explicite d'un objet dans un nouvel objet

Problème : Même si la classe de l'objet est munie des *getters* et *setters* adéquates, *il sera difficile voire impossible de garantir que la copie des valeurs des membres soit indépendante de la représentation des données*.

Solution : munir la classe en question d'une méthode de copie/de clonage pour qu'un utilisateur extérieur n'ait pas à connaître l'implémentation actuelle.

Exemple

```
public class Point2D {
   // Les méthodes
   public Point2D(int abs, int ord) // constructeur
   {
      x = abs ;
      y = ord ;
   }

   public void affiche()
   {
      System.out.println( "Je suis un point 2D de coordonnées " + x + " " + y ) ;
   }

   public Point2D copie() // méthode de clonage qui renvoie une référence (copie)
   {                      // sur un nouvel objet avec des coordonnées identiques
      Point2D ref = new Point2D(x, y) ;
      return ref ;
   }

   // Les attributs
   private int x, y ;
}
```

Copie superficielle et copie profonde d'un objet

- **Copie superficielle** : on se contente de recopier la valeur de tous les champs, même pour les références sur des objets.
- **Copie profonde** : on recopie la valeur des champs de type primitif et pour une référence sur un objet on fait une recopie explicite dans un nouvel objet via une méthode de clonage.

Copie superficielle et copie profonde d'un objet

- **Copie superficielle** : on se contente de recopier la valeur de tous les champs, même pour les références sur des objets.
- **Copie profonde** : on recopie la valeur des champs de type primitif et pour une référence sur un objet on fait une recopie explicite dans un nouvel objet via une méthode de clonage.

Libération de la mémoire allouée dynamiquement

Il n'existe aucun opérateur pour libérer la mémoire allouée en Java.

Java a opté pour un mécanisme de libération automatique de la mémoire connu sous de nom de **ramasse-miettes** (*Garbage Collector* en anglais). Son fonctionnement est le suivant :

- Il connaît à un instant t le nombre exact de références à un objet donné.
- Lorsqu'il n'existe plus de référence vers un objet, alors cet objet peut être détruit et il est ajouté à la liste des candidats au ramasse-miettes.
- La libération effective de la mémoire pour un candidat au ramasse-miettes aura lieu au moment le plus opportun.

Libération de la mémoire allouée dynamiquement

Il n'existe aucun opérateur pour libérer la mémoire allouée en Java.

Java a opté pour un mécanisme de libération automatique de la mémoire connu sous de nom de **ramasse-miettes** (*Garbage Collector* en anglais). Son fonctionnement est le suivant :

- Il connaît à un instant t le nombre exact de références à un objet donné.
- Lorsqu'il n'existe plus de référence vers un objet, alors cet objet peut être détruit et il est ajouté à la liste des candidats au ramasse-miettes.
- La libération effective de la mémoire pour un candidat au ramasse-miettes aura lieu au moment le plus opportun.

Libération de la mémoire allouée dynamiquement

Il n'existe aucun opérateur pour libérer la mémoire allouée en Java.

Java a opté pour un mécanisme de libération automatique de la mémoire connu sous de nom de **ramasse-miettes** (*Garbage Collector* en anglais). Son fonctionnement est le suivant :

- Il connaît à un instant t le nombre exact de références à un objet donné.
- Lorsqu'il n'existe plus de référence vers un objet, alors cet objet peut être détruit et il est ajouté à la liste des candidats au ramasse-miettes.
- La libération effective de la mémoire pour un candidat au ramasse-miettes aura lieu au moment le plus opportun.

Libération de la mémoire allouée dynamiquement

Il n'existe aucun opérateur pour libérer la mémoire allouée en Java.

Java a opté pour un mécanisme de libération automatique de la mémoire connu sous de nom de **ramasse-miettes** (*Garbage Collector* en anglais). Son fonctionnement est le suivant :

- Il connaît à un instant t le nombre exact de références à un objet donné.
- Lorsqu'il n'existe plus de référence vers un objet, alors cet objet peut être détruit et il est ajouté à la liste des candidats au ramasse-miettes.
- La libération effective de la mémoire pour un candidat au ramasse-miettes aura lieu au moment le plus opportun.

Plan

```
public class Point2D {
   // Les méthodes
   public Point2D(int abs, int ord) // constructeur
   {
      x = abs ;
      y = ord ;
   }
   public void affiche()
   {
      System.out.println( "Je suis un point 2D de coordonnées " + x + " " + y ) ;
   }
   public Point2D copie() // méthode de clonage
   {
      Point2D ref = new Point2D(x, y) ;
      return ref ;
   }
   public double distanceAOrigine() { return Math.sqrt( x*x + y*y ) ; }

   // Les getters
   public int getX() { return x ; }
   public int getY() { return y ; }

   // les setters
   public void setPosition(int abs, int ord) { x = abs ; y = ord ; } // public

   // Les attributs
   private int x, y ;
}
```

```
public class Point2D {
   // Les méthodes
   public Point2D(int abs, int ord) // constructeur usuel
   {
      x = abs ;
      y = ord ;
   }
   ...
   public Point2D(Point2D autre) // constructeur par copie
   {
      x = autre.x ;
      y = autre.y ;
   }
   ...
   public Point2D copie() // méthode de clonage
   {
      Point2D ref = new Point2D(x, y) ;
      return ref ;
   }
   ...
   // Les attributs
   private int x, y ;
}
```

Différences

- Le **constructeur par copie** construit un nouvel objet à partir d'un objet existant de même type : tous les champs/attributs ne sont pas forcément identiques, par exemple une clé primaire (e.g. un numéro de série) sera différente pour l'objet de référence et l'objet nouvellement construit.
- La **méthode de clonage** duplique l'objet courant : tous les champs/attributs sont forcément identiques.
- *Cas d'utilisation* : La méthode de clonage s'utilise afin d'éviter la modification de l'objet courant, tandis que le constructeur par copie sert à construire des nouveaux objets à partir d'un objet modèle.

Différences

- Le **constructeur par copie** construit un nouvel objet à partir d'un objet existant de même type : tous les champs/attributs ne sont pas forcément identiques, par exemple une clé primaire (e.g. un numéro de série) sera différente pour l'objet de référence et l'objet nouvellement construit.
- La **méthode de clonage** duplique l'objet courant : tous les champs/attributs sont forcément identiques.
- *Cas d'utilisation* : La méthode de clonage s'utilise afin d'éviter la modification de l'objet courant, tandis que le constructeur par copie sert à construire des nouveaux objets à partir d'un objet modèle.

Différences

- Le **constructeur par copie** construit un nouvel objet à partir d'un objet existant de même type : tous les champs/attributs ne sont pas forcément identiques, par exemple une clé primaire (e.g. un numéro de série) sera différente pour l'objet de référence et l'objet nouvellement construit.
- La **méthode de clonage** duplique l'objet courant : tous les champs/attributs sont forcément identiques.
- *Cas d'utilisation* : La méthode de clonage s'utilise afin d'éviter la modification de l'objet courant, tandis que le constructeur par copie sert à construire des nouveaux objets à partir d'un objet modèle.

Différences

- Le **constructeur par copie** construit un nouvel objet à partir d'un objet existant de même type : tous les champs/attributs ne sont pas forcément identiques, par exemple une clé primaire (e.g. un numéro de série) sera différente pour l'objet de référence et l'objet nouvellement construit.
- La **méthode de clonage** duplique l'objet courant : tous les champs/attributs sont forcément identiques.
- *Cas d'utilisation* : La méthode de clonage s'utilise afin d'éviter la modification de l'objet courant, tandis que le constructeur par copie sert à construire des nouveaux objets à partir d'un objet modèle.

Différences

- Le **constructeur par copie** construit un nouvel objet à partir d'un objet existant de même type : tous les champs/attributs ne sont pas forcément identiques, par exemple une clé primaire (e.g. un numéro de série) sera différente pour l'objet de référence et l'objet nouvellement construit.
- La **méthode de clonage** duplique l'objet courant : tous les champs/attributs sont forcément identiques.
- *Cas d'utilisation* : La méthode de clonage s'utilise afin d'éviter la modification de l'objet courant, tandis que le constructeur par copie sert à construire des nouveaux objets à partir d'un objet modèle.

En grande partie comme en C/C++

- La portée d'une variable est limitée au bloc {...} dans lequel elle est déclarée.
- La portée d'un paramètre est limitée au bloc de sa méthode.

Nouveautés en Java :

- Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc {...} englobant.
- Une variable locale n'a pas le droit de porter le même nom qu'un des arguments de sa méthode.

En grande partie comme en C/C++

- La portée d'une variable est limitée au bloc {...} dans lequel elle est déclarée.
- La portée d'un paramètre est limitée au bloc de sa méthode.

Nouveautés en Java :

- Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc {...} englobant.
- Une variable locale n'a pas le droit de porter le même nom qu'un des arguments de sa méthode.

En grande partie comme en C/C++

- La portée d'une variable est limitée au bloc {...} dans lequel elle est déclarée.
- La portée d'un paramètre est limitée au bloc de sa méthode.

Nouveautés en Java :

- Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc {...} englobant.
- Une variable locale n'a pas le droit de porter le même nom qu'un des arguments de sa méthode.

En grande partie comme en C/C++

- La portée d'une variable est limitée au bloc {...} dans lequel elle est déclarée.
- La portée d'un paramètre est limitée au bloc de sa méthode.

Nouveautés en Java :

- Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc {...} englobant.
- Une variable locale n'a pas le droit de porter le même nom qu'un des arguments de sa méthode.

En grande partie comme en C/C++

```
void f(int m)
{
   int n ;
   int m ; // INTERDIT
   ...
   for(...)
   {
      int p ;
      int n ; // INTERDIT
   }

   if(...)
   {
      int p ; // OK
   }

   {
      int p ; // OK
   }
}
```

Arguments effectifs VS paramètres formels

- Les paramètres formels sont des sortes de variables locales initialisées avec les valeurs des arguments effectifs fournis au moment de l'appel.
- Un paramètre formel peut être déclaré avec l'attribut `final`, dans ce cas le compilateur s'assurera que ce paramètre n'est pas modifié dans le corps de la méthode.

Arguments effectifs VS paramètres formels

- Les paramètres formels sont des sortes de variables locales initialisées avec les valeurs des arguments effectifs fournis au moment de l'appel.
- Un paramètre formel peut être déclaré avec l'attribut `final`, dans ce cas le compilateur s'assurera que ce paramètre n'est pas modifié dans le corps de la méthode.

Arguments effectifs et convertion dans le type des paramètres formels

Java n'autorise que des conversions implicites non-dégradantes, autrement dit dans un type qui peut représenter l'information initiale.

```
public class TestPoint2D {
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;
      int n=1; byte b=2; long q=3 ;
      ref.setPosition(n, b) ; // OK
      ref.setPosition(n, q) ; // ERREUR
   }
}
class Point2D { ...
   public void setPosition(int abs, int ord) { x = abs ; y = ord ; }
   ...
}
```

Mode de transmission des méthodes

Passage *par valeur* ou *par copie*

En passage par valeur, la méthode reçoit une copie de l'argument effectif ; elle ne travaille que sur cette copie et cela n'a aucune conséquence sur l'argument effectif.

En Java, la transmission *d'un argument à une méthode* et *celle de son résultat (*return*)* ont toujours lieu par valeur.

Cela permet d'utiliser des *expressions* comme argument effectif ou comme valeur de retour !

Le passage par adresse est impossible en Java, ce qui contribue largement à sa sécurité.

Mode de transmission des méthodes

Passage *par valeur* ou *par copie*

En passage par valeur, la méthode reçoit une copie de l'argument effectif ; elle ne travaille que sur cette copie et cela n'a aucune conséquence sur l'argument effectif.

En Java, la transmission *d'un argument à une méthode* et *celle de son résultat (*return*)* ont toujours lieu par valeur.

Cela permet d'utiliser des *expressions* comme argument effectif ou comme valeur de retour !

Le passage par adresse est impossible en Java, ce qui contribue largement à sa sécurité.

Mode de transmission des méthodes

Passage *par valeur* ou *par copie*

En passage par valeur, la méthode reçoit une copie de l'argument effectif ; elle ne travaille que sur cette copie et cela n'a aucune conséquence sur l'argument effectif.

En Java, la transmission *d'un argument à une méthode* et *celle de son résultat (*return*)* ont toujours lieu par valeur.

Cela permet d'utiliser des *expressions* comme argument effectif ou comme valeur de retour !

Le passage par adresse est impossible en Java, ce qui contribue largement à sa sécurité.

Ordre d'évaluation des arguments effectifs

En Java, les arguments effectifs sont évalués dans leur ordre d'apparition, donc de gauche à droite.

```
int n=5; double a=2.5, b=3.5 ;

f(n++, n, a, a=b, a, b) ; // quelles sont les valeurs des arguments effectifs?
```