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 "S2 - M2103 - Java et POO"
48H prévues $\approx$ 39H de cours-TDs + 6H de TPs, 1H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

Les champs d'objet

Jusqu'à présent les attributs (ou champs de données) sont propres aux objets, i.e. que chaque objet a son propre exemplaire des attributs en mémoire :

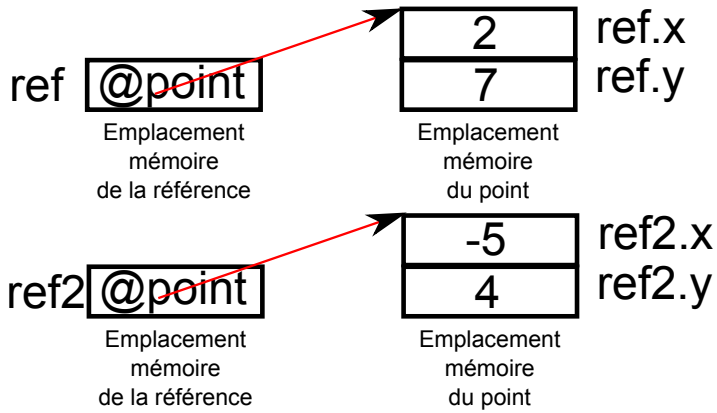
s'il y a plusieurs objets, alors il y a plusieurs exemplaires des attributs en mémoire.

Les champs d'objet

Jusqu'à présent les attributs (ou champs de données) sont propres aux objets, i.e. que chaque objet a son propre exemplaire des attributs en mémoire :
s'il y a plusieurs objets, alors il y a plusieurs exemplaires des attributs en mémoire.

Les champs d'objet

```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;
      Point2D ref2 = new Point2D(-5,4) ;
   }
}

class Point2D {
   public Point2D(int abs, int ord){
      x = abs ;
      y = ord ;
   }
   ...
   private int x, y ;
}
```



Les champs de classe

Il est possible de déclarer en Java des attributs propres à la classe, i.e. que tous les objets issus de la classe partageront le même attribut (1 seul exemplaire en mémoire même si plusieurs objets).

On parle alors de champs *de classe* ou de champs *statiques*.

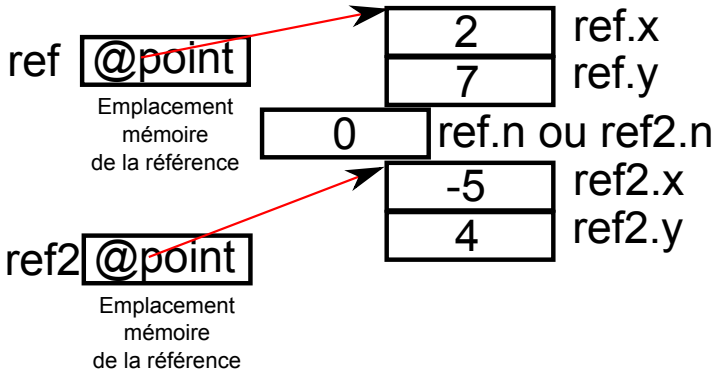
Les champs de classe

Il est possible de déclarer en Java des attributs propres à la classe, i.e. que tous les objets issus de la classe partageront le même attribut (1 seul exemplaire en mémoire même si plusieurs objets).

On parle alors de champs *de classe* ou de champs *statiques*.

Les champs de classe

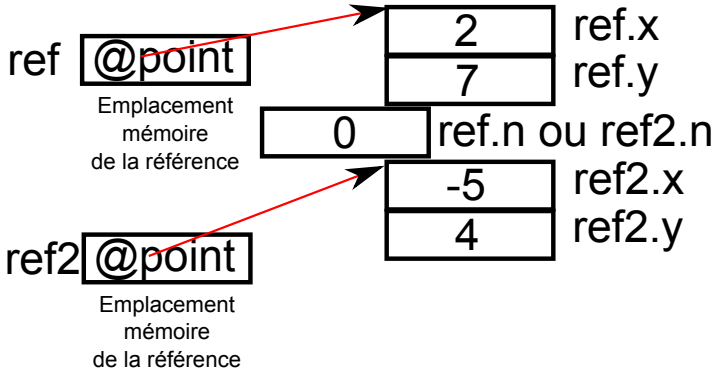
```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;
      Point2D ref2 = new Point2D(-5,4) ;
   }
}

class Point2D {
   public Point2D(int abs, int ord){
      x = abs ;
      y = ord ;
   }
   ...
   private int x, y ;
   static int n = 0 ; // pas privé pour le moment
}
```



Un champ statique existe et est unique indépendamment de la création d'objet : *la notation* Point2D.n *est autorisée (pas privé).*

Les champs de classe

```
public class TestPoint2D
{
   public static void main(String args[])
   {
      Point2D ref = new Point2D(2,7) ;
      Point2D ref2 = new Point2D(-5,4) ;
   }
}

class Point2D {
   public Point2D(int abs, int ord){
      x = abs ;
      y = ord ;
   }
   ...
   private int x, y ;
   static int n = 0 ; // pas privé pour le moment
}
```



Un champ statique existe et est unique indépendamment de la création d'objet : *la notation* Point2D.n *est autorisée (pas privé)*.

Les champs de classe

Il est possible de déclarer privé un champ statique (de classe), mais alors la notation $\mathrm{Point2D.n}$ ne serait plus autorisée à l'extérieur.

En fait, il nous faudrait utiliser une méthode.

Ce qui aura le plus de sens sera d'utiliser une méthode statique (de classe), indépendante de la création d'un objet, afin de manipuler des attributs statiques privés.

Les champs de classe

Il est possible de déclarer privé un champ statique (de classe), mais alors la notation $\mathrm{Point2D.n}$ ne serait plus autorisée à l'extérieur.

En fait, il nous faudrait utiliser une méthode.

Ce qui aura le plus de sens sera d'utiliser une méthode statique (de classe), indépendante de la création d'un objet, afin de manipuler des attributs statiques privés.

Les champs de classe

Il est possible de déclarer privé un champ statique (de classe), mais alors la notation $\mathrm{Point2D.n}$ ne serait plus autorisée à l'extérieur.

En fait, il nous faudrait utiliser une méthode.

Ce qui aura le plus de sens sera d'utiliser une méthode statique (de classe), indépendante de la création d'un objet, afin de manipuler des attributs statiques privés.

Les méthodes de classe

En Java, on peut définir des méthodes *de classe* ou *statiques*, qui peuvent être appelées indépendamment de la création d'un objet de la classe.

C'est le cas de la méthode main.

Les méthodes de classe

En Java, on peut définir des méthodes *de classe* ou *statiques*, qui peuvent être appelées indépendamment de la création d'un objet de la classe.

C'est le cas de la méthode `main`.

Les méthodes de classe

Une méthode de classe peut manipuler uniquement des attributs/méthodes de classe.

Bien entendu elle n'aura pas le droit de manipuler des membres non-statiques (liés à une instance).

Les méthodes de classe

Une méthode de classe peut manipuler uniquement des attributs/méthodes de classe.

Bien entendu elle n'aura pas le droit de manipuler des membres non-statiques (liés à une instance).

Les méthodes de classe

```
class Point2D {
   public Point2D(int abs, int ord){
      x = abs ;
      y = ord ;
   }
   public static void f(){
      x = 5 ; // ERREUR
      n = 2 ; // OK
   }
   public static int getN(){ return n ; } // Si getN n'était pas static on n'aurait pas d'erreur
   ...
   private int x, y ;
   private static int n = 0 ;
}
public class TestPoint2D {
   public static void main(String args[])
   {
      Point2D ref ; // constructeur pas nécessaire si seuls membres statiques appelés
      ref.f() ;     // autorisé mais à éviter
      Point2D.f() ; // autorisé et à privilégier: appel de la méthode f de la classe Point2D
      System.out.println( " n = " + Point2D.getN() ) ;
   }
}
```

Les champs et méthodes de classe : intérêts ?

- Disposer d'informations collectives : e.g. le nombre d'objets construits.
- Disposer d'informations sur la classe elle-même : e.g. le nom de la classe.
- Regrouper des constantes communes : e.g. PI.
- Regrouper des fonctions indépendantes de la création d'objet : e.g. les fonctions cosinus, sinus etc.

Les champs et méthodes de classe : intérêts ?

- Disposer d'informations collectives : e.g. le nombre d'objets construits.
- Disposer d'informations sur la classe elle-même : e.g. le nom de la classe.
- Regrouper des constantes communes : e.g. PI.
- Regrouper des fonctions indépendantes de la création d'objet : e.g. les fonctions cosinus, sinus etc.

Les champs et méthodes de classe : intérêts ?

- Disposer d'informations collectives : e.g. le nombre d'objets construits.
- Disposer d'informations sur la classe elle-même : e.g. le nom de la classe.
- Regrouper des constantes communes : e.g. PI.
- Regrouper des fonctions indépendantes de la création d'objet : e.g. les fonctions cosinus, sinus etc.

Les champs et méthodes de classe : intérêts ?

- Disposer d'informations collectives : e.g. le nombre d'objets construits.
- Disposer d'informations sur la classe elle-même : e.g. le nom de la classe.
- Regrouper des constantes communes : e.g. PI.
- Regrouper des fonctions indépendantes de la création d'objet : e.g. les fonctions cosinus, sinus etc.

Les attributs et méthodes : BILAN

Les attributs et méthodes d'instance (associés aux objets) :

- De tels attributs ne sont munipulables que dans des méthodes d'instance (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode d'instance peut utiliser des membres d'instance et des membres de classe.

Les attributs et méthodes de classe (mot-clé static) :

- De tels attributs sont munipulables dans des méthodes d'instance et de classe (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode de classe peut utiliser seulement des membres de classe.

Les attributs et méthodes : BILAN

Les attributs et méthodes d'instance (associés aux objets) :

- De tels attributs ne sont munipulables que dans des méthodes d'instance (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode d'instance peut utiliser des membres d'instance et des membres de classe.

Les attributs et méthodes de classe (mot-clé static) :

- De tels attributs sont munipulables dans des méthodes d'instance et de classe (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode de classe peut utiliser seulement des membres de classe.

Les attributs et méthodes : BILAN

Les attributs et méthodes d'instance (associés aux objets) :

- De tels attributs ne sont munipulables que dans des méthodes d'instance (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode d'instance peut utiliser des membres d'instance et des membres de classe.

Les attributs et méthodes de classe (mot-clé static) :

- De tels attributs sont munipulables dans des méthodes d'instance et de classe (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode de classe peut utiliser seulement des membres de classe.

Les attributs et méthodes : BILAN

Les attributs et méthodes d'instance (associés aux objets) :

- De tels attributs ne sont munipulables que dans des méthodes d'instance (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode d'instance peut utiliser des membres d'instance et des membres de classe.

Les attributs et méthodes de classe (mot-clé static) :

- De tels attributs sont munipulables dans des méthodes d'instance et de classe (public ou private) si le principe de l'encapsulation est respecté.
- Une méthode de classe peut utiliser seulement des membres de classe.

- **Un constructeur d'une classe ne peut pas initialiser des champs statiques** : l'initialisation des champs statiques doit se faire avant celle des champs des objets instanciés.
  - donc on est limité aux initialisations par défaut et explicites des champs.
- Java offre la possibilité d'**initialiser les champs statiques avec des blocs statiques définis dans la classe** :

```
private static int n;
static {
n = Clavier.lireInt();
...
}
```

Attention : la déclaration des membres statiques utilisés dans ce bloc doit le précéder !

- **Un constructeur d'une classe ne peut pas initialiser des champs statiques** : l'initialisation des champs statiques doit se faire avant celle des champs des objets instanciés.
  - donc on est limité aux initialisations par défaut et explicites des champs.
- Java offre la possibilité d'**initialiser les champs statiques avec des blocs statiques définis dans la classe** :

```
private static int n;
static {
n = Clavier.lireInt();
...
}
```

Attention : la déclaration des membres statiques utilisés dans ce bloc doit le précéder !

- **Un constructeur d'une classe ne peut pas initialiser des champs statiques** : l'initialisation des champs statiques doit se faire avant celle des champs des objets instanciés.
  - donc on est limité aux initialisations par défaut et explicites des champs.
- Java offre la possibilité d'**initialiser les champs statiques avec des blocs statiques définis dans la classe** :

```
private static int n;
static {
n = Clavier.lireInt();
...
}
```

Attention : la déclaration des membres statiques utilisés dans ce bloc doit le précéder !

- **Un constructeur d'une classe ne peut pas initialiser des champs statiques** : l'initialisation des champs statiques doit se faire avant celle des champs des objets instanciés.
  - donc on est limité aux initialisations par défaut et explicites des champs.
- Java offre la possibilité d'**initialiser les champs statiques avec des blocs statiques définis dans la classe** :

```
private static int n;
static {
n = Clavier.lireInt();
...
}
```

Attention : la déclaration des membres statiques utilisés dans ce bloc doit le précéder !

- **Un constructeur d'une classe ne peut pas initialiser des champs statiques** : l'initialisation des champs statiques doit se faire avant celle des champs des objets instanciés.
  - donc on est limité aux initialisations par défaut et explicites des champs.
- Java offre la possibilité d'**initialiser les champs statiques avec des blocs statiques définis dans la classe** :

```
private static int n;
static {
n = Clavier.lireInt();
...
}
```

Attention : la déclaration des membres statiques utilisés dans ce bloc doit le précéder !

Définition

Surdéfinition ou surcharge

On parle de *surdéfinition*/*surcharge* lorsqu'un même identifiant possède plusieurs significations possibles entre lesquelles on choisit on fonction du contexte.

Exemple : la signification de $a + b$ dépend des types de a et b.

En Java, les méthodes d'une même classe peuvent être surdéfinies : la **signature** d'une méthode (**identifiant+liste des types des paramètres formels**) sera utilisée pour choisir la bonne méthode parmi les surdéfinitions.

Définition

Surdéfinition ou surcharge

On parle de *surdéfinition*/*surcharge* lorsqu'un même identifiant possède plusieurs significations possibles entre lesquelles on choisit on fonction du contexte.

Exemple : la signification de $a + b$ dépend des types de a et b.

En Java, les méthodes d'une même classe peuvent être surdéfinies : la **signature** d'une méthode (**identifiant+liste des types des paramètres formels**) sera utilisée pour choisir la bonne méthode parmi les surdéfinitions.

Définition

Surdéfinition ou surcharge

On parle de *surdéfinition*/*surcharge* lorsqu'un même identifiant possède plusieurs significations possibles entre lesquelles on choisit on fonction du contexte.

Exemple : la signification de $a + b$ dépend des types de a et b.

En Java, les méthodes d'une même classe peuvent être surdéfinies : la **signature** d'une méthode (**identifiant+liste des types des paramètres formels**) sera utilisée pour choisir la bonne méthode parmi les surdéfinitions.

Définition

Surdéfinition ou surcharge

On parle de *surdéfinition*/*surcharge* lorsqu'un même identifiant possède plusieurs significations possibles entre lesquelles on choisit on fonction du contexte.

Exemple : la signification de $a + b$ dépend des types de a et b.

En Java, les méthodes d'une même classe peuvent être surdéfinies : la **signature** d'une méthode (**identifiant+liste des types des paramètres formels**) sera utilisée pour choisir la bonne méthode parmi les surdéfinitions.

Exemple

```
class Point2D {
   public Point2D(int abs, int ord){ x = abs ; y = ord ; }
   public void deplace(int dx, int dy){ x += dx ; y += dy ; System.out.println( "2 int " ) ; }
   public void deplace(int dx){ x += dx ; System.out.println( "1 int " ) ; }
   public void deplace(short dx){ x += dx ; System.out.println( "1 short " ) ; }

   private int x, y ;      // Les attributs d'instance
}

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

      ref.deplace(1, 3) ; // void deplace(int dx, int dy)
      ref.deplace(5) ;    // void deplace(int dx)

      short p = 8 ;
      ref.deplace(p) ;    // void deplace(short dx)

      byte b = 4 ;
      b=2;
      ref.deplace(b) ;    // void deplace(short dx)
   }
}
```

Exemple : cas d'ambiguité

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

   public void deplace(int dx, byte dy){ x += dx ; y += dy ;
                                         System.out.println( "1 int, 1 byte " ) ; }
   public void deplace(byte dx, int dy){ x += dx ; y += dy ;
                                         System.out.println( "1 byte, 1 int " ) ; }

   private int x, y ;      // Les attributs d'instance
}

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

      int a = 2;
      byte b = 4 ;
      ref.deplace(a, b) ; // void deplace(int dx, byte dy)
      ref.deplace(b, a) ; // void deplace(byte dx, int dy)

      ref.deplace(b,b) ;  // ERREUR AMBIGUITE: les 2 méthodes deplace sont possibles
      ref.deplace(2*b,b) ;// OK, car pour une op arithmétique byte est converti en int
   }
}
```

Règles pour choisir la bonne méthode

Voici ce que fait le compilateur pour trouver la bonne méthode :

1. rechercher les méthodes avec le *bon identifiant*, le *bon nombre d'arguments* et pour lesquelles *chaque type d'argument effectif est* ***compatible*** *avec le type de l'argument formel correspondant*.
2. conserver parmi les méthodes trouvées celles *accessibles*.
3. ensuite, plusieurs cas de figure :
   - Si la liste des méthodes est vide : *erreur de compilation*.
   - Si la liste des méthodes est réduite à une seule méthode : *cette méthode est sélectionnée*.
   - Si la lite comporte ≥ 2 méthodes : *recherche de* ***la*** *meilleure méthode* : celle avec les types les plus proches des arguments effectifs dont les types sont compatibles avec au moins 1 autre méthode de la liste.

Règles pour choisir la bonne méthode

Voici ce que fait le compilateur pour trouver la bonne méthode :

1. rechercher les méthodes avec le *bon identifiant*, le *bon nombre d'arguments* et pour lesquelles *chaque type d'argument effectif est* ***compatible*** *avec le type de l'argument formel correspondant*.
2. conserver parmi les méthodes trouvées celles *accessibles*.
3. ensuite, plusieurs cas de figure :
   - Si la liste des méthodes est vide : *erreur de compilation*.
   - Si la liste des méthodes est réduite à une seule méthode : *cette méthode est sélectionnée*.
   - Si la lite comporte ≥ 2 méthodes : *recherche de* ***la*** *meilleure méthode* : celle avec les types les plus proches des arguments effectifs dont les types sont compatibles avec au moins 1 autre méthode de la liste.

Règles pour choisir la bonne méthode

Voici ce que fait le compilateur pour trouver la bonne méthode :

1. rechercher les méthodes avec le *bon identifiant*, le *bon nombre d'arguments* et pour lesquelles *chaque type d'argument effectif est* ***compatible*** *avec le type de l'argument formel correspondant*.
2. conserver parmi les méthodes trouvées celles *accessibles*.
3. ensuite, plusieurs cas de figure :
   - Si la liste des méthodes est vide : *erreur de compilation*.
   - Si la liste des méthodes est réduite à une seule méthode : *cette méthode est sélectionnée*.
   - Si la lite comporte ≥ 2 méthodes : *recherche de* ***la*** *meilleure méthode* : celle avec les types les plus proches des arguments effectifs dont les types sont compatibles avec au moins 1 autre méthode de la liste.

Règles pour choisir la bonne méthode

Voici ce que fait le compilateur pour trouver la bonne méthode :

1. rechercher les méthodes avec le *bon identifiant*, le *bon nombre d'arguments* et pour lesquelles *chaque type d'argument effectif est* ***compatible*** *avec le type de l'argument formel correspondant*.
2. conserver parmi les méthodes trouvées celles *accessibles*.
3. ensuite, plusieurs cas de figure :
   - Si la liste des méthodes est vide : *erreur de compilation*.
   - Si la liste des méthodes est réduite à une seule méthode : *cette méthode est sélectionnée*.
   - Si la lite comporte ≥ 2 méthodes : *recherche de la meilleure méthode* : celle avec les types les plus proches des arguments effectifs dont les types sont compatibles avec au moins 1 autre méthode de la liste.

Règles pour choisir la bonne méthode

Voici ce que fait le compilateur pour trouver la bonne méthode :

1. rechercher les méthodes avec le *bon identifiant*, le *bon nombre d'arguments* et pour lesquelles *chaque type d'argument effectif est* ***compatible*** *avec le type de l'argument formel correspondant*.
2. conserver parmi les méthodes trouvées celles *accessibles*.
3. ensuite, plusieurs cas de figure :
   - Si la liste des méthodes est vide : *erreur de compilation*.
   - Si la liste des méthodes est réduite à une seule méthode : *cette méthode est sélectionnée*.
   - Si la lite comporte ≥ 2 méthodes : *recherche de* ***la*** *meilleure méthode* : celle avec les types les plus proches des arguments effectifs dont les types sont compatibles avec au moins 1 autre méthode de la liste.

Remarques

- Le type de retour d'une méthode n'intervient pas dans son choix.
- On peut surdéfinir des méthodes de classe.
- Le mot clé `final` devant le type d'un argument formel n'est pas discriminant (le processus de recherche n'en tient pas compte).
- C'est grâce à la surdéfinition de méthodes qu'une classe peut avoir plusieurs constructeurs.

Remarques

- Le type de retour d'une méthode n'intervient pas dans son choix.
- On peut surdéfinir des méthodes de classe.
- Le mot clé final devant le type d'un argument formel n'est pas discriminant (le processus de recherche n'en tient pas compte).
- C'est grâce à la surdéfinition de méthodes qu'une classe peut avoir plusieurs constructeurs.

Remarques

- Le type de retour d'une méthode n'intervient pas dans son choix.
- On peut surdéfinir des méthodes de classe.
- Le mot clé `final` devant le type d'un argument formel n'est pas discriminant (le processus de recherche n'en tient pas compte).
- C'est grâce à la surdéfinition de méthodes qu'une classe peut avoir plusieurs constructeurs.

Remarques

- Le type de retour d'une méthode n'intervient pas dans son choix.
- On peut surdéfinir des méthodes de classe.
- Le mot clé `final` devant le type d'un argument formel n'est pas discriminant (le processus de recherche n'en tient pas compte).
- C'est grâce à la surdéfinition de méthodes qu'une classe peut avoir plusieurs constructeurs.

Exemple avec les droits d'accès

```
class A {
   private void f(int i){ System.out.println( "1 int de valeur " + i ) ; }
   public void f(float g){ System.out.println( "1 float de valeur " + g ) ; }
   public void bidon(){ int n = 1; float p = 2.4f ; f(n) ; f(p) ; }
}

public class TestA {
   public static void main(String args[]){
      A a = new A() ;
      int n = 1 ; float p = 2.4f ; a.f(n) ; a.f(p) ;

      System.out.println( "Dans bidon : ") ;
      a.bidon() ;
   }
}
```

Quelle est la sortie ?

Exemple avec les droits d'accès

```
class A {
   private void f(int i){ System.out.println( "1 int de valeur " + i ) ; }
   public void f(float g){ System.out.println( "1 float de valeur " + g ) ; }
   public void bidon(){ int n = 1; float p = 2.4f ; f(n) ; f(p) ; }
}

public class TestA {
   public static void main(String args[]){
      A a = new A() ;
      int n = 1 ; float p = 2.4f ; a.f(n) ; a.f(p) ;

      System.out.println( "Dans bidon : ") ;
      a.bidon() ;
   }
}
```

Quelle est la sortie ?

Pourquoi transmettre des objets en argument ?

- Pour comparer 2 objets du même type.
- Pour déclarer une méthode de clonage statique (pas utile si non-statique).
- Pour utiliser cet objet dans un algorithme.

Pourquoi transmettre des objets en argument ?

- Pour comparer 2 objets du même type.
- Pour déclarer une méthode de clonage statique (pas utile si non-statique).
- Pour utiliser cet objet dans un algorithme.

Pourquoi transmettre des objets en argument ?

- Pour comparer 2 objets du même type.
- Pour déclarer une méthode de clonage statique (pas utile si non-statique).
- Pour utiliser cet objet dans un algorithme.

L'unité de l'encapsulation est la classe et non l'objet

```
class Point2D {
   ...
   public boolean estEgal(Point2D autre)
   {
      return x==autre.x && y==autre.y ; // autorisé en Java: au sein de la classe Point2D, on
                                        // peut accéder aux membres privés de toutes les
                                        // instances de la classe Point2D (pas celles des autres
                                        // classes)
   }
   ...
   private int x, y ;
}

public class TestPoint2D {
   public static void main(String args[]){
      Point2D pt = new Point2D(5, 7) ;
      Point2D pt2 = new Point2D(5, 7) ;

      if( pt.estEgal(pt2) )
         System.out.println( " Les 2 points ont les mêmes coordonnées. ") ;
      else
         System.out.println( " Les 2 points n'ont pas les mêmes coordonnées. ") ;
   }
}
```

On copie une référence d'un objet : on peut donc modifier l'objet référencé

```
class Point2D {
   ...
   public static void echange(Point2D p1, Point2D p2)
   {
      Point2D tmp = new Point2D(0, 0) ;
      tmp.x = p1.x ; tmp.y = p1.y ;
      p1.x = p2.x  ; p1.y = p2.y ;
      p2.x = tmp.x ; p2.y = tmp.y ;
   }
   ...
   private int x, y ;
}

public class TestPoint2D {
   public static void main(String args[]){
      Point2D pt = new Point2D(5, 7) ;
      Point2D pt2 = new Point2D(2, -3) ;
      pt.affiche() ; pt2.affiche() ;
      Point2D.echange(pt, pt2) ;
      pt.affiche() ; pt2.affiche() ;
   }
}
```

Remarques

- L'implémentation de la méthode estEgal n'est pas symétrique.
- On peut également modifier un objet dont une référence est retournée par une méthode (copie de la référence).

Remarques

- L'implémentation de la méthode estEgal n'est pas symétrique.
- On peut également modifier un objet dont une référence est retournée par une méthode (copie de la référence).

Utilisation

this **désigne la référence de l'objet courant.**

```
class A {
   ...
   public void f(...)
   {
      // ici "this" désigne la référence de l'objet ayant appelé la méthode f
   }
   ...
   public static void g(...)
   {
      // ici "this" n'est pas utilisable, car g peut être appelée directement via la classe sans
      // passer par un objet (et this n'existe que pour un objet) ; plus généralement "this"
      // ne peut s'utiliser que dans un contexte non-statique
   }
}
```

Utilisation

```
class A {
   ...
   public void f(...)
   {
      System.out.println( this.i ) ; // OK
   }
   ...
   private int i ;
}
```

Intérêts de this ?

- Obtenir des implantations plus symétriques pour des méthodes de comparaison d'objet.
- Ajouter la référence de l'objet courant à une liste ou un tableau pour un traitement séquentiel de plusieurs objets.
- Appel d'un contructeur de la classe au sein d'un autre constructeur de la classe.

Intérêts de this ?

- Obtenir des implantations plus symétriques pour des méthodes de comparaison d'objet.
- Ajouter la référence de l'objet courant à une liste ou un tableau pour un traitement séquentiel de plusieurs objets.
- Appel d'un contructeur de la classe au sein d'un autre constructeur de la classe.

Intérêts de this ?

- Obtenir des implantations plus symétriques pour des méthodes de comparaison d'objet.
- Ajouter la référence de l'objet courant à une liste ou un tableau pour un traitement séquentiel de plusieurs objets.
- Appel d'un contructeur de la classe au sein d'un autre constructeur de la classe.

Des implantations plus symétriques

```
class Point2D {
   ...
   public boolean estEgal(Point2D autre)
   {
      return this.x==autre.x && this.y==autre.y ;
   }
   ...
   private int x, y ;
}
```

Appel d'un contructeur de la classe au sein d'un autre constructeur

```
class Point2D {
   public Point2D(int abs, int ord) {
      x = abs ; y = ord ;
      System.out.println("2 arg") ;
   }

   public Point2D() {
      this(0, 0) ; // this(...) doit être obligatoirement la 1ère instruction
      System.out.println("0 arg") ;
   }
   ...
   private int x, y ;
}
```

Plan

Champ référence à un objet ou objet membre

En Java, une classe peut avoir des attributs d'instance :

- de type *scalaire* ou simple ;
- de type *référence sur un objet* dont le type est celui d'une classe (autre ou même).

C'est un premier point intéressant de la POO : **représenter des objets plus complexes à partir d'objets plus simples**.

Champ référence à un objet ou objet membre

En Java, une classe peut avoir des attributs d'instance :

- de type *scalaire* ou simple ;
- de type *référence sur un objet* dont le type est celui d'une classe (autre ou même).

C'est un premier point intéressant de la POO : **représenter des objets plus complexes à partir d'objets plus simples**.

Champ référence à un objet ou objet membre

En Java, une classe peut avoir des attributs d'instance :

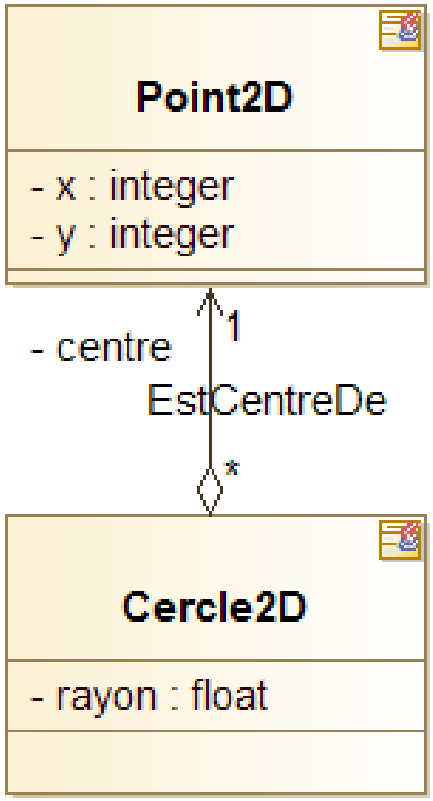
- de type *scalaire* ou simple ;
- de type *référence sur un objet* dont le type est celui d'une classe (autre ou même).

C'est un premier point intéressant de la POO : **représenter des objets plus complexes à partir d'objets plus simples**.

Classe Cercle2D utilisant un point 2D

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

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

   private int x, y ;       // Les attributs d'instance
}

class Cercle2D {
   public Cercle2D(..., float r){ ... } // constructeur

   public void affiche(){ ... }

   private Point2D centre ; // Les attributs d'instance
   private float rayon ;
}
```

En UML ça donne quoi ?



Outils gratuits de modélisation UML (de génération de code) : modelio `https://www.modelio.org/` et ArgoUML `http://argouml.tigris.org/`

En UML ça donne quoi ?

Un membre privé de type référence sur un objet "se traduit" en UML par une *association binaire*.

Définitions

Association binaire

Une association binaire entre 2 classes A et B est une abstraction des liens existants entre les objets de type A et ceux de type B.

Multiplicité d'une classe dans une association

La multiplicité d'une classe dans une relation binaire est le nombre d'instances de la classe pouvant être relié à une instance de l'autre classe.

Définitions

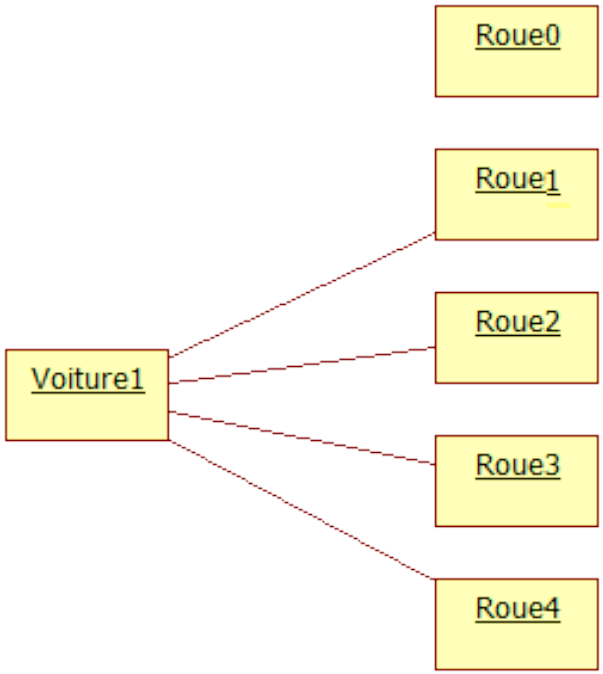
Association binaire

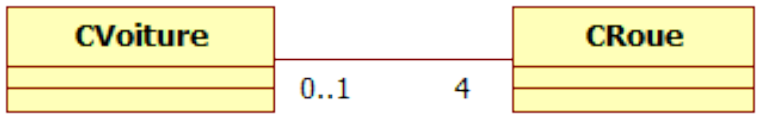
Une association binaire entre 2 classes A et B est une abstraction des liens existants entre les objets de type A et ceux de type B.

Multiplicité d'une classe dans une association

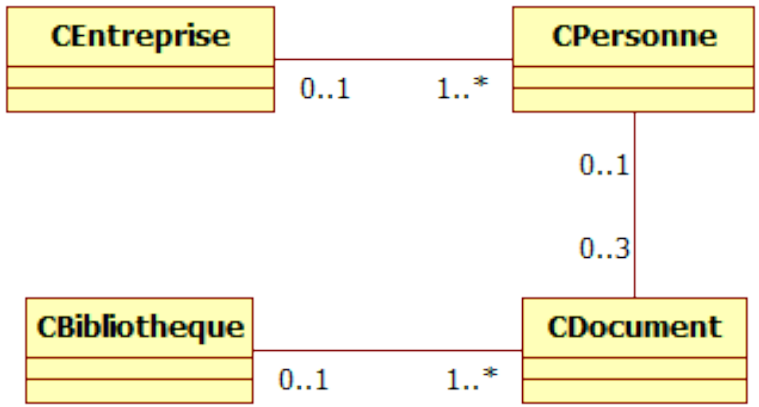
La multiplicité d'une classe dans une relation binaire est le nombre d'instances de la classe pouvant être relié à une instance de l'autre classe.

Application



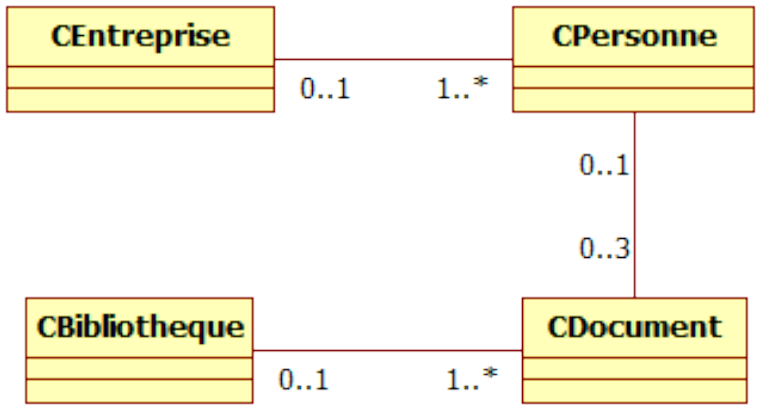


Application



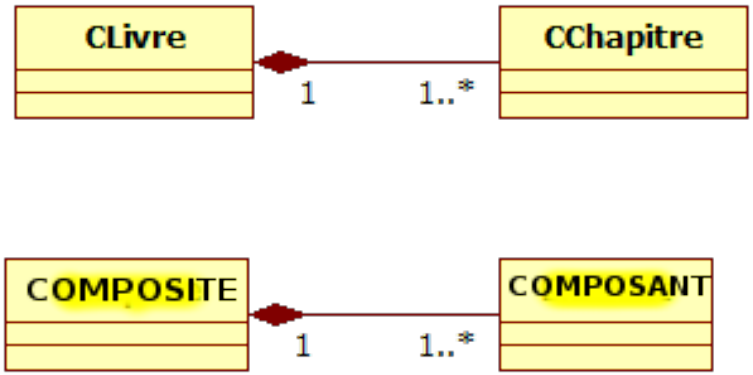
Conséquences au niveau du code ?

Application



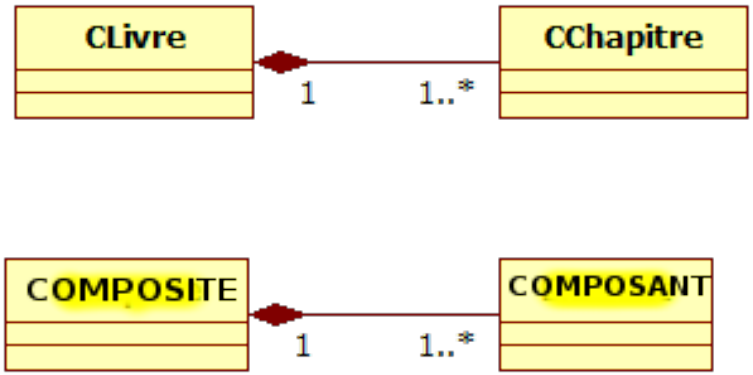
Conséquences au niveau du code ?

Composition



Un *composant* appartient à au plus un *composite* (le conteneur d'objet de l'autre classe). Le *composite* possède le *composant*.
La destruction du *composite* entraîne la destruction des *composants*.
La destruction d'un *composant* n'entraîne pas la destruction du *composite*.

Composition

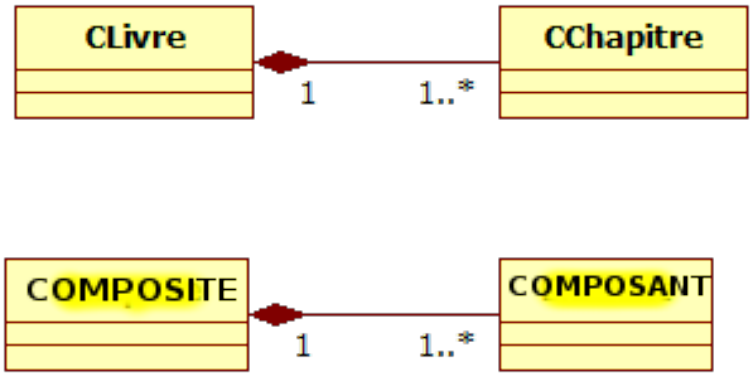


Un *composant* appartient à au plus un *composite* (le conteneur d'objet de l'autre classe). Le *composite* possède le *composant*.

La destruction du *composite* entraîne la destruction des *composants*.

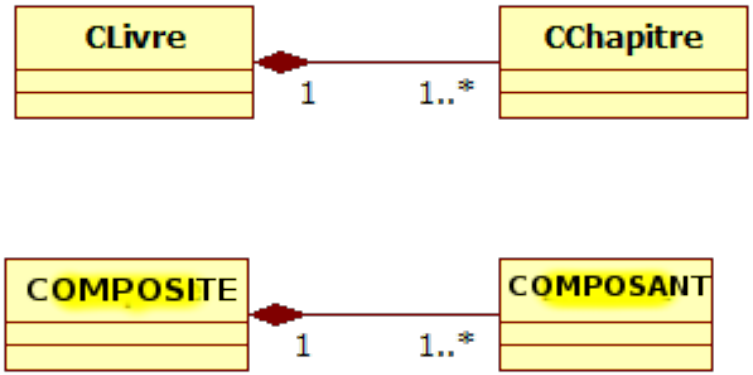
La destruction d'un *composant* n'entraîne pas la destruction du *composite*.

Composition



Un *composant* appartient à au plus un *composite* (le conteneur d'objet de l'autre classe). Le *composite* possède le *composant*.
La destruction du *composite* entraîne la destruction des *composants*.
La destruction d'un *composant* n'entraîne pas la destruction du *composite*.

Composition

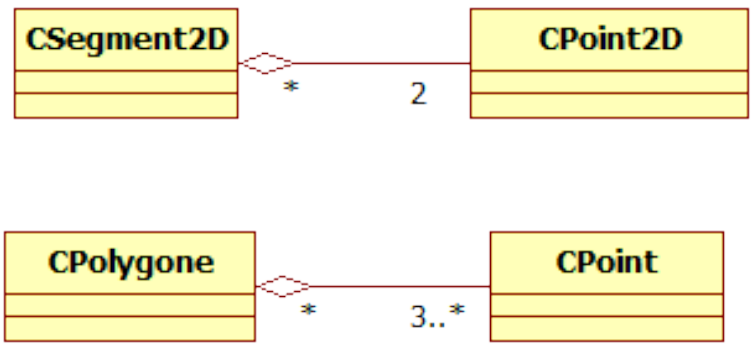


Un *composant* appartient à au plus un *composite* (le conteneur d'objet de l'autre classe). Le *composite* possède le *composant*.
La destruction du *composite* entraîne la destruction des *composants*.
La destruction d'un *composant* n'entraîne pas la destruction du *composite*.

Classe Cercle2D utilisant une composition

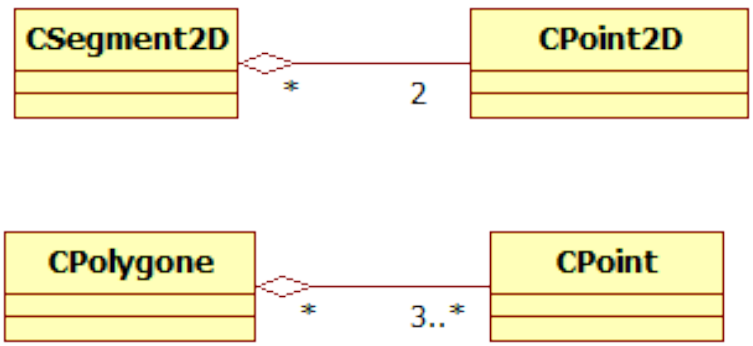
```
...
class Cercle2D {
   public Cercle2D(int abs, int ord, float r){
      centre = new Point2D(abs, ord) ; // l'instance de Point2D ne peut pas être
      rayon = r ;                      // partagée
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " ) ;
      centre.affiche();
   }

   private Point2D centre ; // Les attributs d'instance
   private float rayon ;
}
```

Agrégation



Simple regroupement de parties dans un tout, **la destruction du tout n'entraînant pas la destruction des parties**. Le tout "utilise" une partie.
Une partie peut être utilisée dans plusieurs tout.

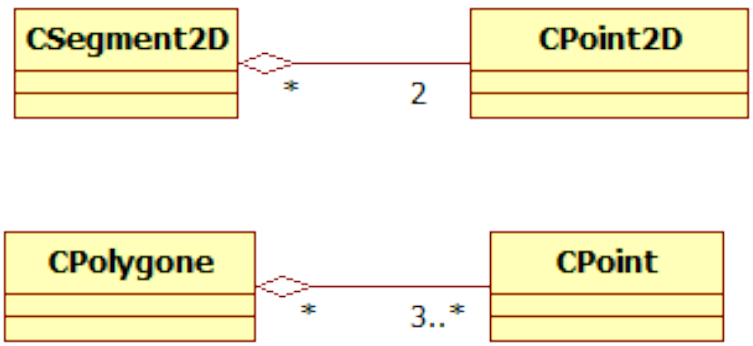
Agrégation



Simple regroupement de parties dans un tout, **la destruction du tout n'entraînant pas la destruction des parties**. Le tout "utilise" une partie.

Une partie peut être utilisée dans plusieurs tout.

Agrégation



Simple regroupement de parties dans un tout, **la destruction du tout n'entraînant pas la destruction des parties**. Le tout "utilise" une partie.
Une partie peut être utilisée dans plusieurs tout.

Classe Cercle2D utilisant une agrégation

```
...
class Cercle2D {
   public Cercle2D(Point2D ref, float r){
      centre = ref ;        // l'instance de Point2D est partagée
      rayon = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " ) ;
      centre.affiche();
   }

   private Point2D centre ; // Les attributs d'instance
   private float rayon ;
}
```

Remarque : l'affichage laisse à désirer. *Solution ?* Avoir des Getters getX et getY pour afficher à sa manière le centre.

Classe Cercle2D utilisant une agrégation

```
...
class Cercle2D {
   public Cercle2D(Point2D ref, float r){
      centre = ref ;        // l'instance de Point2D est partagée
      rayon = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " ) ;
      centre.affiche();
   }

   private Point2D centre ; // Les attributs d'instance
   private float rayon ;
}
```

Remarque : l'affichage laisse à désirer. *Solution ?* Avoir des Getters getX et getY pour afficher à sa manière le centre.

Classe Cercle2D utilisant une agrégation

```
...
class Cercle2D {
   public Cercle2D(Point2D ref, float r){
      centre = ref ;        // l'instance de Point2D est partagée
      rayon = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " ) ;
      centre.affiche();
   }

   private Point2D centre ; // Les attributs d'instance
   private float rayon ;
}
```

Remarque : l'affichage laisse à désirer. *Solution ?* Avoir des Getters getX et getY pour afficher à sa manière le centre.

Classe Cercle2D utilisant une agrégation

```
...
class Cercle2D {
   public Cercle2D(Point2D ref, float r){
      centre = ref ;        // l'instance de Point2D est partagée
      rayon = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " ) ;
      centre.affiche();
   }

   private Point2D centre ; // Les attributs d'instance
   private float rayon ;
}
```

Remarque : l'affichage laisse à désirer. *Solution ?* Avoir des Getters getX et getY pour afficher à sa manière le centre.

Plan

Définitions

Classe interne

Une classe est dite *interne* lorsque sa définition est située à l'intérieur de la définition d'une autre classe.

```
class UneClasse {
    ...
    class SaClasseInterne {
        ...
    }
    ...
}
```

Remarques

La définition de la classe SaClasseInterne :

- n'implique pas l'existance d'un membre de type référence sur SaClasseInterne au sein de la classe UneClasse ;
- permet d'instancier des objets de type SaClasseInterne au sein de la classe UneClasse ;
- implique une relation privilégiée entre les objets "externes" de type UneClasse et ceux "internes" de type SaClasseInterne : un objet interne créé au sein d'un objet externe a accès à tous les membres (même privés) de son objet externe et vice versa.

Remarques

La définition de la classe SaClasseInterne :

- n'implique pas l'existance d'un membre de type référence sur SaClasseInterne au sein de la classe UneClasse ;
- permet d'instancier des objets de type SaClasseInterne au sein de la classe UneClasse ;
- implique une relation privilégiée entre les objets "externes" de type UneClasse et ceux "internes" de type SaClasseInterne : un objet interne créé au sein d'un objet externe a accès à tous les membres (même privés) de son objet externe et vice versa.

Remarques

La définition de la classe SaClasseInterne :

- n'implique pas l'existance d'un membre de type référence sur SaClasseInterne au sein de la classe UneClasse ;
- permet d'instancier des objets de type SaClasseInterne au sein de la classe UneClasse ;
- implique une relation privilégiée entre les objets "externes" de type UneClasse et ceux "internes" de type SaClasseInterne : un objet interne créé au sein d'un objet externe a accès à tous les membres (même privés) de son objet externe et vice versa.

Remarques

La définition de la classe SaClasseInterne :

- n'implique pas l'existance d'un membre de type référence sur SaClasseInterne au sein de la classe UneClasse ;
- permet d'instancier des objets de type SaClasseInterne au sein de la classe UneClasse ;
- implique une relation privilégiée entre les objets "externes" de type UneClasse et ceux "internes" de type SaClasseInterne : un objet interne créé au sein d'un objet externe a accès à tous les membres (même privés) de son objet externe et vice versa.

Remarques

Une classe interne ne peut pas contenir de membre statique.

Une instance de SaClasseInterne est obligatoirement créée par une instance de UneClasse (sauf si la classe interne est autonome/statique).

Remarques

Une classe interne ne peut pas contenir de membre statique.

Une instance de SaClasseInterne est obligatoirement créée par une instance de UneClasse (sauf si la classe interne est autonome/statique).

Application : classe Cercle2D utilisant une classe Centre interne

```
class Cercle2D {
   public Cercle2D(int abs, int ord, float r){
      c = new Centre(abs, ord) ;
      this.r = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " ) ;
      c.affiche();
   }

   private Centre c ; // Les attributs d'instance
   private float r ;

   class Centre {
      public Centre(int x, int y){ this.x = x ; this.y = y ; }
      public void affiche(){ System.out.println( x + ", " + y ) ; }

      private int x, y ;
   }
}
```

Application : classe Cercle2D utilisant une classe Centre interne

```
class Cercle2D {
   public Cercle2D(int abs, int ord, float r){
      c = new Centre(abs, ord) ;
      this.r = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " + c.x + ",
      " + c.y ) ;
   }

   private Centre c ; // Les attributs d'instance
   private float r ;

   class Centre {
      public Centre(int x, int y){ this.x = x ; this.y = y ; }

      private int x, y ;
   }
}
```

Plus besoin de getter et de setter pour la classe Centre...

Application : classe Cercle2D utilisant une classe Centre interne

```
class Cercle2D {
   public Cercle2D(int abs, int ord, float r){
      c = new Centre(abs, ord) ;
      this.r = r ;
   }

   public void affiche(){
      System.out.println( "Je suis un cercle 2D de rayon " + rayon + " et de centre " + c.x + ",
      " + c.y ) ;
   }

   private Centre c ; // Les attributs d'instance
   private float r ;

   class Centre {
      public Centre(int x, int y){ this.x = x ; this.y = y ; }

      private int x, y ;
   }
}
```

Plus besoin de getter et de setter pour la classe Centre...

Plan

Un package Java regroupe de manière logique des *classes* et des *interfaces* (classes particulières).

Remarques

- La notion de paquetage est **proche de la notion de bibliothèque** dans d'autres langages (et surtout de la notion de namespace en C++).
- Les paquetages **facilitent le développement logiciel** : le problème d'avoir 2 classes de même nom (générant une erreur) est limité aux seules classes d'un même paquetage.

Remarques

- La notion de paquetage est **proche de la notion de bibliothèque** dans d'autres langages (et surtout de la notion de namespace en C++).
- Les paquetages **facilitent le développement logiciel** : le problème d'avoir 2 classes de même nom (génèrant une erreur) est limité aux seules classes d'un même paquetage.

Utilisation d'une classe d'un package monPackage :
soit on la préfixe avec monPackage., soit on utilise la directive import en début de fichier.

Exemples :

- import java.util.Random ;
  On se donne le droit d'utiliser la classe Random du package java.util ;
- import java.util.* ;
  On se donne le droit d'utiliser toutes les classes du package java.util.

Utilisation d'une classe d'un package monPackage :
soit on la préfixe avec monPackage., soit on utilise la directive import en début de fichier.
Exemples :

- import java.util.Random ;
  On se donne le droit d'utiliser la classe Random du package java.util ;
- import java.util.* ;
  On se donne le droit d'utiliser toutes les classes du package java.util.

Utilisation d'une classe d'un package monPackage :
soit on la préfixe avec monPackage., soit on utilise la directive import en début de fichier.
Exemples :

- import java.util.Random ;
  On se donne le droit d'utiliser la classe Random du package java.util ;
- import java.util.* ;
  On se donne le droit d'utiliser toutes les classes du package java.util.

Remarques

- Utiliser tout un package via `import monPackage.*` ; ne pénalise ni le temps de compilation ni la taille du bytecode.
- Importer 2 paquetages définissant une classe de même nom engendre une erreur de compilation.

Remarques

- Utiliser tout un package via `import monPackage.*` ; ne pénalise ni le temps de compilation ni la taille du bytecode.
- Importer 2 paquetages définissant une classe de même nom engendre une erreur de compilation.

Les classes standards de Java sont structurées en packages :

- **java.lang** : classes fournissant les éléments de base du langage (Object, Number, Double..., System, String, Math ...) ; java.lang est le seul paquetage dont l'emploi ne doit jamais être déclaré ;
- **java.util** : classes utilitaires : http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html ;
- **java.awt** : classes graphiques : http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html ;
- et plein d'autres : **java.io** : http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html, **java.net** : http://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html etc.

Les classes standards de Java sont structurées en packages :

- **java.lang** : classes fournissant les éléments de base du langage (Object, Number, Double..., System, String, Math ...) ; java.lang est le seul paquetage dont l'emploi ne doit jamais être déclaré ;
- **java.util** : classes utilitaires : `http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html` ;
- **java.awt** : classes graphiques : `http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html` ;
- et plein d'autres : **java.io** : `http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html`, **java.net** : `http://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html` etc.

Les classes standards de Java sont structurées en packages :

- **java.lang** : classes fournissant les éléments de base du langage (Object, Number, Double..., System, String, Math ...) ; java.lang est le seul paquetage dont l'emploi ne doit jamais être déclaré ;
- **java.util** : classes utilitaires : `http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html` ;
- **java.awt** : classes graphiques : `http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html` ;
- et plein d'autres : **java.io** : `http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html`, **java.net** : `http://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html` etc.

Les classes standards de Java sont structurées en packages :

- **java.lang** : classes fournissant les éléments de base du langage (Object, Number, Double..., System, String, Math ...) ; java.lang est le seul paquetage dont l'emploi ne doit jamais être déclaré ;
- **java.util** : classes utilitaires : `http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html` ;
- **java.awt** : classes graphiques : `http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html` ;
- et plein d'autres : **java.io** : `http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html`, **java.net** : `http://docs.oracle.com/javase/7/docs/api/java/net/package-su` `html` etc.

Pour créer son propre package Java :

- Il faut ajouter package *monPackage ;* au début de chaque fichier source devant composer le package.
- Il faut s'assurer qu'il n'existe pas de classe homonyme au sein du package.

Remarques :

- Lorsqu'on ne spécifie pas de package dans un fichier source, les classes de ce fichier appartiennent au package par défaut.
- Toutes les classes d'un même fichier source appartiennent obligatoirement au même package.
- Souvent les EDI imposent des contraintes quand à la localisation des fichiers sources d'un même package.

Pour créer son propre package Java :

- Il faut ajouter package *monPackage ;* au début de chaque fichier source devant composer le package.
- Il faut s'assurer qu'il n'existe pas de classe homonyme au sein du package.

Remarques :

- Lorsqu'on ne spécifie pas de package dans un fichier source, les classes de ce fichier appartiennent au package par défaut.
- Toutes les classes d'un même fichier source appartiennent obligatoirement au même package.
- Souvent les EDI imposent des contraintes quand à la localisation des fichiers sources d'un même package.

Pour créer son propre package Java :

- Il faut ajouter package *monPackage ;* au début de chaque fichier source devant composer le package.
- Il faut s'assurer qu'il n'existe pas de classe homonyme au sein du package.

Remarques :

- Lorsqu'on ne spécifie pas de package dans un fichier source, les classes de ce fichier appartiennent au package par défaut.
- Toutes les classes d'un même fichier source appartiennent obligatoirement au même package.
- Souvent les EDI imposent des contraintes quand à la localisation des fichiers sources d'un même package.

Pour créer son propre package Java :

- Il faut ajouter package *monPackage ;* au début de chaque fichier source devant composer le package.
- Il faut s'assurer qu'il n'existe pas de classe homonyme au sein du package.

Remarques :

- Lorsqu'on ne spécifie pas de package dans un fichier source, les classes de ce fichier appartiennent au package par défaut.
- Toutes les classes d'un même fichier source appartiennent obligatoirement au même package.
- Souvent les EDI imposent des contraintes quand à la localisation des fichiers sources d'un même package.

Pour créer son propre package Java :

- Il faut ajouter package *monPackage ;* au début de chaque fichier source devant composer le package.
- Il faut s'assurer qu'il n'existe pas de classe homonyme au sein du package.

Remarques :

- Lorsqu'on ne spécifie pas de package dans un fichier source, les classes de ce fichier appartiennent au package par défaut.
- Toutes les classes d'un même fichier source appartiennent obligatoirement au même package.
- Souvent les EDI imposent des contraintes quand à la localisation des fichiers sources d'un même package.

Convention de nommage des packages Java

- **Tout en minuscule sauf la première lettre des mots suivants qui peut être en majuscule.**
- **Utiliser seulement [a-z], [A-Z], [0-9] et le point '.'** :
  Ne pas utiliser de tiret '-', d'underscore '_', d'espace, ou d'autres caractères ($, *, accents, ...).
- Exemples :
  com.sun.eng
  com.apple.quicktime.v2
  edu.cmu.cs.bovik.cheese
  com.mycompany.framework.myFrameworkA.model.feature
  com.mycompany.framework.myFrameworkA.ui.wicket

Un peu de lecture sur le sujet :
http://stepaheadsoftware.blogspot.com.au/2012/06/java-packa
html

Convention de nommage des packages Java

- **Tout en minuscule sauf la première lettre des mots suivants qui peut être en majuscule.**
- **Utiliser seulement [a-z], [A-Z], [0-9] et le point '.' :**
  Ne pas utiliser de tiret '-', d'underscore '_', d'espace, ou d'autres caractères ($, *, accents, ...).
- Exemples :
  com.sun.eng
  com.apple.quicktime.v2
  edu.cmu.cs.bovik.cheese
  com.mycompany.framework.myFrameworkA.model.feature
  com.mycompany.framework.myFrameworkA.ui.wicket

Un peu de lecture sur le sujet :
http://stepaheadsoftware.blogspot.com.au/2012/06/java-packa
html

Convention de nommage des packages Java

- **Tout en minuscule sauf la première lettre des mots suivants qui peut être en majuscule.**
- **Utiliser seulement [a-z], [A-Z], [0-9] et le point '.'** :
  Ne pas utiliser de tiret '-', d'underscore '_', d'espace, ou d'autres caractères ($, *, accents, ...).
- Exemples :
  com.sun.eng
  com.apple.quicktime.v2
  edu.cmu.cs.bovik.cheese
  com.mycompany.framework.myFrameworkA.model.feature
  com.mycompany.framework.myFrameworkA.ui.wicket

Un peu de lecture sur le sujet :
http://stepaheadsoftware.blogspot.com.au/2012/06/java-packa
html

Convention de nommage des packages Java

- **Tout en minuscule sauf la première lettre des mots suivants qui peut être en majuscule.**
- **Utiliser seulement [a-z], [A-Z], [0-9] et le point '.'** :
  Ne pas utiliser de tiret '-', d'underscore '_', d'espace, ou d'autres caractères ($, *, accents, ...).
- Exemples :
  com.sun.eng
  com.apple.quicktime.v2
  edu.cmu.cs.bovik.cheese
  com.mycompany.framework.myFrameworkA.model.feature
  com.mycompany.framework.myFrameworkA.ui.wicket

Un peu de lecture sur le sujet :
`http://stepaheadsoftware.blogspot.com.au/2012/06/java-packa`
`html`