Le livre de Java premier langage

Anne Tasso

Éditions Eyrolles

ISBN : 2-212-09156-7

2000

8

Les principes du concept d'objet

Au cours du chapitre précédent, nous avons examiné comment mettre en place des objets à l'intérieur d'un programme Java. Cette étude a montré combien la structure générale des programmes se trouvait modifiée par l'emploi des objets.

En réalité, les objets sont beaucoup plus qu'une structure syntaxique. Ils sont régis par des principes essentiels, qui constituent les fondements de la programmation objet. Dans ce chapitre, nous étudions avec précision l'ensemble de ces principes.

Nous déterminons d'abord (*section « La communication objet »*) les caractéristiques d'une donnée `static` et évaluons leurs conséquences sur la construction des objets en mémoire. Nous analysons également la technique du passage de paramètres par référence. Nous observons qu'il est possible, avec la technologie objet, qu'une méthode transmette plusieurs résultats à une autre méthode.

Nous expliquons ensuite (*section « Les objets contrôlent leur fonctionnement »*), le concept d'encapsulation des données, et nous examinons pourquoi et comment les objets protègent leurs données.

Enfin, nous définissons (*section « L'héritage »*) la notion d'héritage entre classes. Nous observons combien cette notion est utile puisqu'elle permet de réutiliser des programmes tout en apportant des variations dans le comportement des objets héritants.

La communication objet

En définissant un type ou une classe, le développeur crée un modèle, qui décrit les fonctionnalités des objets utilisés par le programme. Les objets sont créés en mémoire à partir de ce modèle, par copie des données et des méthodes.

Cette copie est réalisée lors de la réservation des emplacements mémoire grâce à l'opérateur `new`, qui initialise les données de l'objet et fournit, en retour, l'adresse où se trouvent les informations stockées.

La question est de comprendre pourquoi l'interpréteur réalise cette copie en mémoire, alors que cela lui était impossible auparavant.

Les données `static`

La réponse à cette interrogation se trouve dans l'observation des différents programmes proposés dans ce manuel (*voir les chapitres 6, « Fonctions, notions avancées », et 7, « Les classes et les objets »*). Comme nous l'avons déjà constaté (*voir, au chapitre précédent, la section « Construire et utiliser ses propres classes »*), le mot-clé `static` n'est plus utilisé lors de la description d'un type, alors qu'il était présent dans tous les programmes précédant ce chapitre.

C'est donc la présence ou l'absence de ce mot-clé qui fait que l'interpréteur construise ou non des objets en mémoire.

Lorsque l'interpréteur rencontre le mot-clé `static` devant une variable ou une méthode, il réserve un seul et unique emplacement mémoire pour y stoker la valeur ou le pseudo-code associés. Cet espace mémoire est communément accessible pour tous les objets du même type.

Lorsque le mot-clé `static` n'apparaît pas, l'interpréteur réserve, à chaque appel de l'opérateur `new`, un espace mémoire pour y charger les données et les pseudo-codes décrits dans la classe.

Exemple : compter des cercles

Pour bien comprendre la différence entre une donnée `static` et une donnée non `static`, nous allons modifier la classe `Cercle`, de façon qu'il soit possible de connaître le nombre d'objets `Cercle` créés en cours d'application.

Pour ce faire, l'idée est d'écrire une méthode `créer()`, qui permette, d'une part, de saisir des valeurs `x`, `y` et `r` pour chaque cercle à créer et, d'autre part, d'incrémenter un compteur de cercles.

La variable représentant ce compteur doit être indépendante des objets créés, de sorte que sa valeur ne soit pas être réinitialisée à zéro à chaque création d'objet. Cette variable doit cependant être accessible pour chaque objet de façon qu'elle puisse s'incrémenter de `1` à chaque appel de la méthode `créer()`.

Pour réaliser ces contraintes, le compteur de cercles doit être une variable de classe, c'est-à-dire une variable déclarée avec le mot-clé `static`. Examinons tout cela dans le programme suivant.

```
public class Cercle  {
 public int x, y, r ; // position du centre et rayon
 public static int nombre; // nombre de cercle

  public void créer()  {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   System.out.print(" Rayon            :  ");
```

```
    r = Lire.i();
    nombre ++;
   }
   // et toutes les autres méthodes de la classe Cercle définies au
   // chapitre précédent
} // Fin de la classe Cercle
```

Les données définies dans la classe `Cercle` sont de deux sortes : les variables d'instance, `x`, `y` et `r`, et la variable de classe, `nombre`. Seul le mot-clé `static` permet de différencier leur catégorie.

Grâce au mot-clé `static`, la variable de classe `nombre` est un espace mémoire commun, accessible pour tous les objets créés. Pour faire appel à cette variable, il suffit de l'appeler par son nom véritable (*voir, au chapitre 6, « Fonctions, notions avancées », la section « Variable de classe »*), c'est-à-dire `nombre`, si elle est utilisée dans la classe `Cercle`, ou `Cercle.nombre`, si elle est utilisée en dehors de cette classe.

Exécution de l'application `CompterDesCercles`

Pour mieux saisir la différence entre les variables d'instance (non `static`) et les variables de classe (`static`), observons comment fonctionne l'application `CompterDesCercles`.

```
public class CompterDesCercles  {
 public static void main(String [] arg)
 {
  Cercle A  = new Cercle();
  A.créer();
  System.out.println("Nombre de cercle : " + Cercle.nombre);

  Cercle B = new Cercle();
  B.créer();
  System.out.println("Nombre de cercle : " + Cercle.nombre);
 }
} // Fin de la classe CompterDesCercles
```

Dans ce programme, deux objets de type `Cercle` sont créés à partir du modèle défini par le type `Cercle`. Chaque objet est un représentant particulier, ou une instance, de la classe `Cercle`, de position et de rayon spécifiques.

Lorsque l'objet `A` est créé en mémoire, grâce à l'opérateur `new`, les données `x`, `y` et `r` sont initialisées à `0` au moment de la réservation de l'espace mémoire. La variable de classe `nombre` est elle aussi créée en mémoire, et sa valeur est également initialisée à `0`.

Lors de l'exécution de l'instruction `A.créer();`, les valeurs des variables `x`, `y` et `r` de l'instance `A` sont saisies au clavier (`x = Lire.i(), …`). La variable de classe `nombre` est incrémentée de `1` (`nombre++`). Le nombre de cercles est alors de `1` (*voir l'objet* `A`, *décrit à la Figure 8-1*).

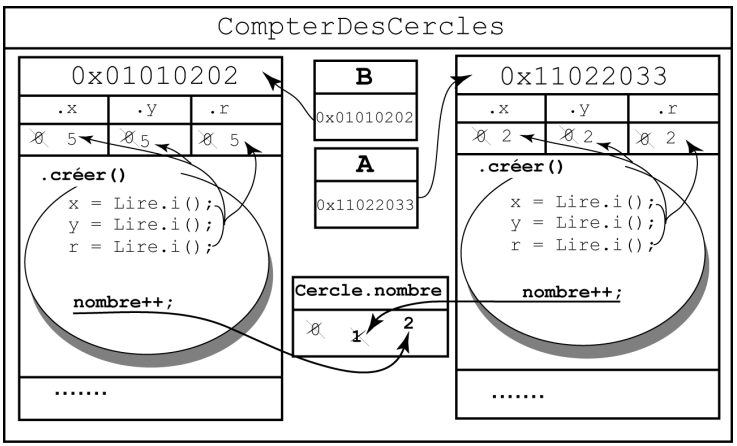


Figure 8-1.
La variable de classe `Cercle.nombre` *est créée en mémoire, avec l'objet* `A`*. Grâce au mot-clé* `static`*, il y a, non pas réservation d'un nouvel espace mémoire (pour la variable* `nombre`*) lors de la création de l'objet* `B`*, mais préservation de l'espace mémoire ainsi que de la valeur stockée.*

De la même façon, l'objet `B` est créé en mémoire grâce à l'opérateur `new`. Les données `x`, `y` et `r` sont, elles aussi, initialisées à `0`.

Pour la variable de classe `nombre`, en revanche, cette initialisation n'est pas réalisée. La présence du mot-clé `static` fait que la variable de classe `nombre`, qui existe déjà en mémoire, ne peut être réinitialisée directement par l'interpréteur.

Il y a donc, non pas réservation d'un nouvel emplacement mémoire, mais préservation du même emplacement mémoire, avec conservation de la valeur calculée à l'étape précédente, soit `1`.

Après saisie des données `x`, `y` et `r` de l'instance `B`, l'instruction `nombre++` fait passer la valeur de `Cercle.nombre` à `2` (*voir l'objet* `B` *décrit à la Figure 8-1*).

N'existant qu'en un seul exemplaire, la variable de classe `nombre` permet le comptage du nombre de cercles créés. L'incrémentation de cette valeur est réalisée indépendamment de l'objet, la variable étant commune à tous les objets créés.

Le passage de paramètres par référence

La communication des données entre les objets passe avant tout par l'intermédiaire des variables d'instance. Nous l'avons observé à la section précédente, lorsqu'une méthode appliquée à un objet modifie les valeurs de plusieurs données de cet objet, cette modification est visible en dehors de la méthode et de l'objet lui-même.

Il existe cependant une autre technique qui permette la modification des données d'un objet : le passage de **paramètres par référence**.

Ce procédé est utilisé lorsqu'on passe en paramètre d'une méthode, non plus une simple variable (de type `int`, `char` ou `double`), mais un objet. Dans cette situation, l'objet étant défini par son adresse (référence), la valeur passée en paramètre n'est plus la valeur réelle de la variable mais l'adresse de l'objet.

Grâce à cela, les modifications apportées sur l'objet passé en paramètre et réalisées à l'intérieur de la méthode sont visibles en dehors même de la méthode.

Échanger la position de deux cercles

Pour comprendre en pratique le mécanisme du passage de paramètres par référence, nous allons écrire une application qui échange la position des centres de deux cercles donnés.

Pour cela, nous utilisons le mécanisme d'échange de valeurs (*voir le chapitre 1, « Stocker une information »*), en l'appliquant à la coordonnée `x` puis à la coordonnée `y` des centres des deux cercles à échanger.

Examinons la méthode `échanger()`, dont le code ci-dessous s'insère dans la classe `Cercle`.

✔ Voir, au chapitre 7, *« Les classes et les objets »*, la section *« La classe descriptive du type* `Cercle` *»*.

```
public void échanger(Cercle autre)    {        // Échange la position d'un
  int tmp;                    // cercle avec celle du cercle donné en paramètre
  tmp = x;                    // échanger la position en x
  x = autre.x;
  autre.x = tmp;
  tmp = y;                    // échanger la position en y
  y = autre.y;
  autre.y = tmp;
}
```

Pour échanger les coordonnées des centres de deux cercles, la méthode `échanger()` doit avoir accès aux valeurs des coordonnées des deux centres des cercles concernés.

Si, par exemple, la méthode est appliquée au cercle `B` (`B.échanger()`), ce sont les variables d'instance `x` et `y` de l'objet `B` qui sont modifiées par les coordonnées du centre du cercle `A`. La méthode doit donc connaître les coordonnées du cercle `A`. Pour ce faire, il est nécessaire de passer ces valeurs en paramètres de la fonction.

La technique consiste à passer en paramètres, non pas les valeurs `x` et `y` du cercle avec lequel l'échange est réalisé, mais un objet de type `Cercle`. Dans notre exemple, ce paramètre s'appelle `autre`. C'est le paramètre formel de la méthode représentant n'importe quel cercle, et il peut donc représenter, par exemple, le cercle `A`.

Le fait d'échanger les coordonnées des centres de deux cercles revient à échanger les coordonnées du couple (`x`, `y`) du cercle sur lequel on applique la méthode (`B.x`, `B.y`) avec les coordonnées (`autre.x`, `autre.y`) du cercle passé en paramètre de la méthode (`A.x`, `A.y`).

Examinons maintenant comment s'opère effectivement l'échange en exécutant l'application suivante :

```
public class EchangerDesCercles    {
 public static void main(String [] arg)    {
  Cercle A  = new Cercle();
  A.créer();
  System.out.println("Le cercle A : ");
  A.afficher();

  Cercle B = new Cercle();
  B.créer();
  System.out.println("Le cercle B : ");
```

```
  B. afficher() ;

  B.échanger(A) ;
  System.out.println("Apres echange, ") ;
  System.out.println("Le cercle A : ") ;
  A.afficher() ;
  System.out.println("Le cercle B : ") ;
  B.afficher() ;
 }
}
```

Exécution de l'application `EchangerDesCercles`

Nous supposons que l'utilisateur ait saisi les valeurs suivantes, pour le cercle `A`

```
Position en x :  2
Position en y :  2
Rayon         :  2
Le cercle A :
Centre en 2, 2
Rayon : 2
```

et pour le cercle `B`

```
Position en x :  5
Position en y :  5
Rayon         :  5
Le cercle B :
Centre en 5, 5
Rayon : 5
```

L'instruction `B.échanger(A)` échange les coordonnées (`x`, `y`) de l'objet `B` avec celles de l'objet `A`. C'est donc le pseudo-codede l'objet `B` qui est interprété, comme illustré à la Figure 8-2.

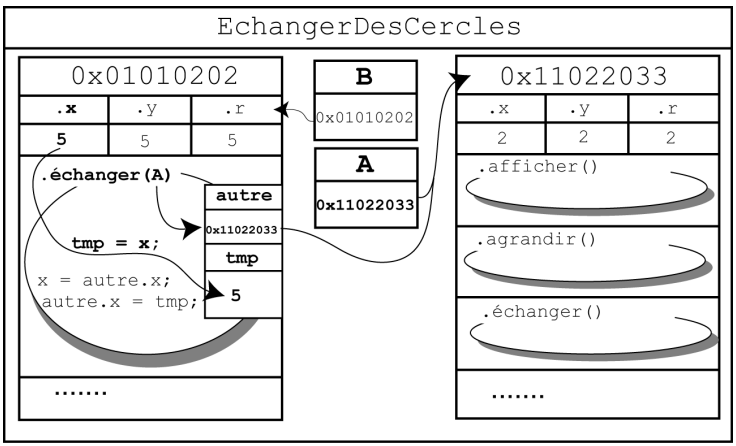


Figure 8-2.
L'instruction `B.échanger(A)` *fait appel à la méthode* `échanger()` *de l'objet* `B`. *Les données* `x`, `y` *et* `r` *utilisées par cette méthode sont celles de l'objet* `B`.

Examinons le tableau d'évolution des variables déclarées pour le pseudo-code de l'objet `B`.

instruction	tmp	x	y	autre
`valeurs initiales`	-	5	5	0x11022033

- À l'entrée de la méthode, la variable `tmp` est déclarée sans être initialisée.
- La méthode est appliquée à l'objet `B`. Les variables `x` et `y` de l'instance `B` ont pour valeurs respectives `5` et `5`.
- L'objet `autre` est simplement déclaré en paramètre de la fonction `échanger(Cercle autre)`. L'opérateur `new` n'étant pas appliqué à cet objet, aucun espace mémoire supplémentaire n'est alloué.

  Comme `autre` représente un objet de type `Cercle`, il ne peut contenir qu'une adresse et non pas une simple valeur numérique. Cette adresse est celle du paramètre effectivement passé lors de l'appel de la méthode.

  Pour notre exemple, l'objet `A` est passé en paramètre de la méthode `(B.échanger(A))`. La case mémoire de la variable `autre` prend donc pour valeur l'adresse de l'objet `A`.

instruction	tmp	x	autre	autre.x (A.x)
`tmp = x ;`	5	5	0x11022033	2
`x = autre.x ;`	5	2	0x11022033	2
`autre.x = tmp ;`	5	2	0x11022033	5

- La variable `tmp` prend ensuite la valeur de la coordonnée `x` de l'objet `B`, soit `5`.

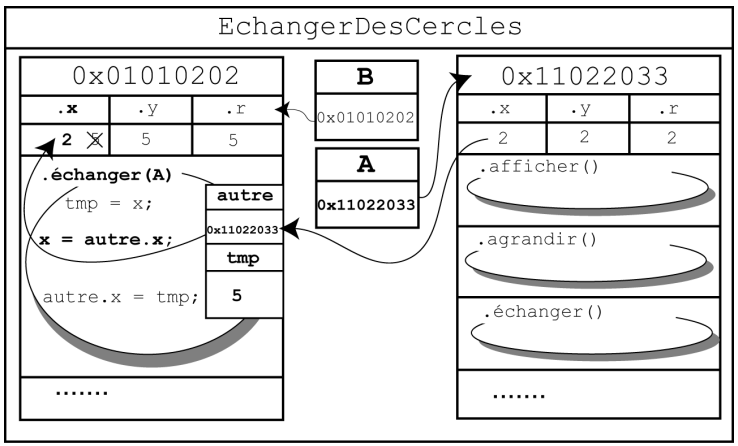


Figure 8-3.
L'objet `autre` *est le paramètre formel de la méthode* `échanger()`. *En écrivant* `B.échanger(A)`, *l'objet* `autre` *stocke la référence mémorisée en* `A`. *De cette façon,* `autre.x` *représente également* `A.x`. *La variable* `x` *de l'instance* `B` *prend la valeur de* `A.x` *grâce à l'instruction* `x = autre.x`.

- Lorsque l'instruction `x = autre.x` est exécutée, la coordonnée `x` de l'objet `B` prend la valeur de la coordonnée `x` de l'objet `autre.x`. Puisque `autre` correspond à l'adresse de l'objet `A`, le fait de consulter le contenu de `autre.x` revient, en réalité, à consulter le contenu de `A.x` (*voir Figure 8-3*). La variable d'instance `A.x` contenant la valeur `2`, `x` (`B.x`) prend la valeur `2`.
- Pour finir, l'échange sur les abscisses, `autre.x`, prend la valeur stockée dans `tmp`. Comme `autre` et `A` correspondent à la même adresse, modifier `autre.x`, c'est aussi modifier `A.x` (*voir Figure 8-4*). Une fois exécuté `autre.x = tmp`, la variable `x` de l'instance `A` vaut par conséquent `5`.

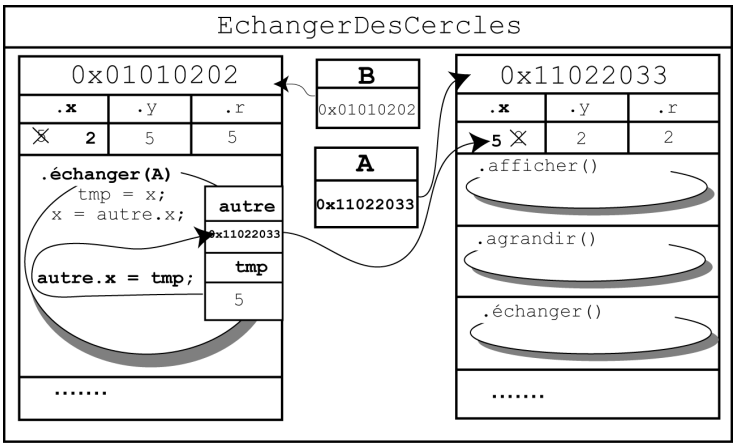


Figure 8-4.
`autre` *et* `A` *définissent la même référence, ou adresse. C'est pourquoi le fait de modifier* `autre.x` *revient aussi à modifier* `A.x`. *Ainsi, l'instruction* `autre.x = tmp` *fait que* `A.x` *prend la valeur stockée dans* `tmp`.

L'ensemble de ces opérations est ensuite réalisé sur la coordonnée `y` des cercles `B` et `A` *via* `autre`.

instruction	tmp	y	autre	autre.y (A.y)
`tmp = y ;`	5	5	0x11022033	2
`y = autre.y ;`	5	2	0x11022033	2
`autre.y = tmp ;`	5	2	0x11022033	5

L'exécution finale du programme a pour résultat

```
Apres echange,
Le cercle A :
Centré en 5, 5
Rayon : 2
Le cercle B :
Centre en 2, 2
Rayon : 5
```

Au final, nous constatons, à l'observation des tableaux d'évolution des variables, que les données `x` et `y` de `B` ont pris la valeur des données `x` et `y` de `A`, soit `2` pour `x` et `2` pour `y`. Parallèlement, le cercle `A` a été transformé par l'intermédiaire de la référence stockée dans `autre` et a pris les coordonnées `x` et `y` du cercle `B`, soit `5` pour `x` et `5` pour `y`.

En résumé, grâce à la technique du passage de paramètres par référence, tout objet passé en paramètre d'une méthode voit, en sortie de la méthode, ses données transformées par la méthode. Cette transformation est alors visible pour tous les objets de l'application.

Les objets contrôlent leur fonctionnement

L'un des objectifs de la programmation objet est de simuler, à l'aide d'un programme informatique, la manipulation des objets réels par l'être humain. Les objets réels forment un tout, et leur manipulation nécessite la plupart du temps un outil, ou une interface, de communication.

Par exemple, quand nous prenons un ascenseur, nous appuyons sur le bouton d'appel pour ouvrir les portes ou pour nous rendre jusqu'à l'étage désiré. L'interface de communication est ici le bouton d'appel. Nul n'aurait l'idée de prendre la télécommande de sa télévision pour appeler un ascenseur.

De la même façon, la préparation d'une omelette nécessite de casser des œufs. Pour briser la coquille d'un œuf, nous pouvons utiliser l'outil couteau. Un marteau pourrait être également utilisé, mais son usage n'est pas vraiment adapté à la situation.

Comme nous le constatons à travers ces exemples, les objets réels sont manipulés par l'intermédiaire d'interfaces **appropriées**. L'utilisation d'un outil inadapté fait que l'objet ne répond pas à nos attentes ou qu'il se brise définitivement.

Tout comme nous manipulons les objets réels, les applications informatiques manipulent des objets virtuels, définis par le programmeur. Cette manipulation nécessite des outils aussi bien adaptés que nos outils réels. Sans contrôle sur le bien-fondé d'une manipulation, l'application risque de fournir de mauvais résultats ou, pire, de cesser brutalement son exécution.

La notion d'encapsulation

Pour réaliser l'adéquation entre un outil et la manipulation d'un objet, la programmation objet utilise le concept d'**encapsulation**.

Par ce terme, il faut entendre que les données d'un objet sont protégées, tout comme le médicament est protégé par la fine pellicule de sa capsule. Grâce à cette protection, il ne peut y avoir transformation involontaire des données de l'objet.

L'encapsulation passe par le contrôle des données et des comportements de l'objet. Ce contrôle est établi à travers la protection des données (*voir la section suivante*), l'accès contrôlé aux données (*voir la section « Les méthodes d'accès aux données »*) et la notion de constructeur de classe (*voir la section « Les constructeurs »*).

La protection des données

Le langage Java fournit les niveaux de protection suivants pour les membres d'une classe (données et méthodes) :

- **Protection** `public`**.** Les membres (données et méthodes) d'une classe déclarés `public` sont accessibles pour tous les objets de l'application. Les données peuvent être modifiées par une méthode de la classe, d'une autre classe ou depuis la fonction `main()`.
- **Protection** `private`**.** Les membres de la classe déclarés `private` ne sont accessibles que pour les méthodes de la même classe. Les données ne peuvent être initialisées ou modifiées que par l'intermédiaire d'une méthode de la classe. Les données ou méthodes ne peuvent être appelées par la fonction `main()`.
- **Protection** `protected`**.** Tout comme les membres privés, les membres déclarés `protected` ne sont accessibles que pour les méthodes de la même classe. Ils sont aussi accessibles par les fonctions membres d'une sous-classe (*voir la section « L'héritage »*).

Par défaut, lorsque les données sont déclarées sans type de protection, leur protection est `public`. Les données sont alors accessibles depuis toute l'application.

Protéger les données d'un cercle

Pour protéger les données de la classe `Cercle`, il suffit de remplacer le mot-clé `public` précédant la déclaration des variables d'instance par le mot `private`. Observons la nouvelle classe, `CerclePrive`, dont les données sont ainsi protégées.

```
public class CerclePrive
{
 private int x, y, r ; // position du centre et rayon

 public void afficher() {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }

 public double périmètre() {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }
 public void déplacer(int nx, int ny) {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }

 public void agrandir(int nr) {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }
} // Fin de la classe CerclePrive
```

Les données `x`, `y` et `r` de la classe `CerclePrive` sont protégées grâce au mot-clé `private`. Étudions les conséquences d'une telle protection sur la phase de compilation de l'application `FaireDesCerclesPrives`.

```
public class FaireDesCerclesPrives
{
 public static void main(String [] arg)
 {
  CerclePrive A  = new CerclePrive();
  A.afficher();
  System.out.println(" Entrez le rayon : ");
  A.r = Lire.i() ;
  System.out.println(" Le cercle est de rayon : " + A.r) ;
 }
}
```

Compilation de l'application `FaireDesCerclesPrives`

Les données `x`, `y` et `r` de la classe `CerclePrive` sont déclarées privées. Par définition, ces données ne sont donc pas accessibles en dehors de la classe où elles sont définies.

Or, en écrivant dans la fonction `main()` l'instruction `A.r = Lire.i() ;`, le programmeur demande d'accéder, depuis la classe `FaireDesCerclesPrives`, à la valeur de `r`, de façon à la modifier. Cet accès est impossible, car `r` est défini en mode `private` dans la classe `CerclePrive` et non dans la classe `FaireDesCerclesPrives`.

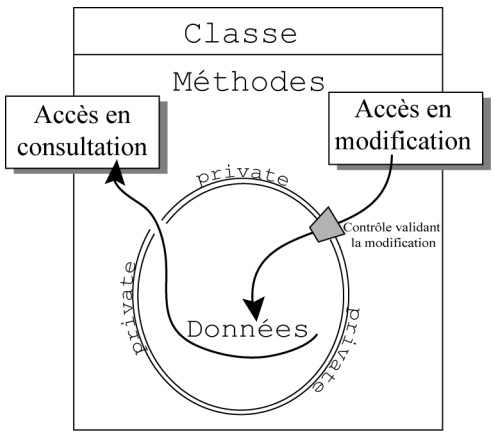
C'est pourquoi le compilateur détecte l'erreur `Variable x in class CerclePrive not accessible from class FaireDesCerclesPrives`.

Les méthodes d'accès aux données

Lorsque les données sont totalement protégées, c'est-à-dire déclarées `private` à l'intérieur d'une classe, elles ne sont plus accessibles depuis une autre classe ou depuis la fonction `main()`. Pour connaître ou modifier la valeur d'une donnée, il est nécessaire de créer, à l'intérieur de la classe, des méthodes d'accès à ces données.

Figure 8-5.

Lorsque les données d'un objet sont protégées, l'objet possède ses propres méthodes, qui permettent soit de consulter la valeur réelle de ses données, soit de modifier les données. La validité de ces modifications est contrôlée par les méthodes définies dans la classe.



Les données privées ne peuvent être consultées ou modifiées que par des méthodes de la classe où elles sont déclarées.

De cette façon, grâce à l'accès aux données par l'intermédiaire de méthodes appropriées, l'objet permet, non seulement la consultation de la valeur de ses données, mais aussi l'autorisation ou non, suivant ses propres critères, de leur modification.

Les méthodes d'une classe réalisent les modes d'accès suivants :

- **Accès en consultation.** La méthode fournit la valeur de la donnée mais ne peut la modifier. Ce type de méthode est aussi appelé **accesseur** en consultation.
- **Accès en modification.** La méthode modifie la valeur de la donnée. Cette modification est réalisée après validation par la méthode. On parle aussi d'accesseur en modification.

Contrôler les données d'un cercle

Dans l'exemple suivant, nous prenons pour hypothèse que le rayon d'un cercle ne puisse jamais être négatif ni dépasser la taille de l'écran. Ces conditions doivent être vérifiées pour toutes les méthodes qui peuvent modifier la valeur du rayon d'un cercle.

Comme nous l'avons déjà remarqué (*voir, au chapitre 7, « Les classes et les objets », la section « Quelques observations »*), les méthodes `afficher()` et `périmètre()` ne font que consulter le contenu des données `x`, `y` et `r`.

Les méthodes `déplacer()`, `agrandir()` et `créer()`, en revanche, modifient le contenu des données `x`, `y` et `r`. La méthode `déplacer()` n'ayant pas d'influence sur la donnée `r`, seules les méthodes `agrandir()` et `créer()` doivent contrôler la valeur du rayon, de sorte que cette dernière ne puisse être négative ou supérieure à la taille de l'écran. Examinons la classe `CercleControle` suivante, qui prend en compte ces nouvelles contraintes :

```
public class CercleControle    {
 private int x, y, r ; // position du centre et rayon
  public void créer()      {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   do  {
     System.out.print(" Rayon            :  ");
     r = Lire.i();
   } while ( r < 0 || r >  600);
  }

 public void afficher()     {    //Affichage des données de la classe
  System.out.println(" Centre en " + x + ", " + y);
  System.out.println(" Rayon : " + r);
 }

 public void agrandir(int nr)     {
   if (r + nr < 0) r = 0;
     else if ( r + nr > 600) r = 600;
     else r = r + nr;
```

```
  }
} // Fin de la classe CercleControle
```

La méthode `créer()` contrôle la valeur du rayon lors de sa saisie, en demandant de saisir une valeur pour le rayon tant que la valeur saisie est négative ou plus grande que `600` (taille supposée de l'écran). Dès que la valeur saisie est comprise entre `0` et `600`, la fonction `créer()` cesse son exécution. À la sortie de cette fonction, nous sommes certains que le rayon est compris entre `0` et `600`.

De la même façon, la méthode `agrandir()` autorise que la valeur du rayon soit augmentée de la valeur passée en paramètre, à condition que cette augmentation ne dépasse pas la taille de l'écran ou que la diminution n'entraîne pas un rayon négatif, si la valeur passée en paramètre est négative. Dans ces deux cas, la valeur du rayon est forcée respectivement à la taille de l'écran ou à `0`.

Exécution de l'application `FaireDesCerclesControles`

Pour vérifier que tous les objets `Cercle` contrôlent bien la valeur de leur rayon, examinons l'exécution de l'application suivante :

```
public class FaireDesCerclesControles        {
 public static void main(String [] arg)     {
  CercleControle A  = new CercleControle();
  A.créer();
  A.afficher();
  System.out.print("Entrer une valeur d'agrandissement :");
  int plus = Lire.i();
  A.agrandir(plus);
  System.out.println("Apres agrandissement : ");
  A.afficher();
 }
}
```

L'objet `A` est créé en mémoire grâce à l'opérateur `new`. La valeur du rayon est initialisée à `0`. À l'appel de la méthode `créer()`, les variables d'instance `x` et `y` sont saisies au clavier, comme suit :

```
Position en x :  5
Position en y :  5
```

Ensuite, si l'utilisateur saisit pour le rayon une valeur négative

```
Rayon         :  -3
```

ou supérieure à 600

```
Rayon         :  654
```

le programme demande de nouveau de saisir une valeur pour le rayon. L'application cesse cette répétition lorsque l'utilisateur entre une valeur comprise entre `0` et `600`, comme suit :

```
Rayon         :  200
Centre 5, 5
Rayon : 200
```

Après affichage des données du cercle A, le programme demande

```
Entrer une valeur d'agrandissement : 450
```

La valeur du rayon vaut 200 + 450, soit 650. Ce nouveau rayon étant supérieur à 600, la valeur du rayon est bloquée par le programme à 600. L'affichage des données fournit

```
Apres agrandissement :
Centre 5, 5
Rayon : 600
```

La notion de constante

D'une manière générale, en programmation objet, les variables d'instance ne sont que très rarement déclarées en `public`. Pour des raisons de sécurité, tout objet se doit de contrôler les transformations opérées par l'application sur lui-même. C'est pourquoi les données d'une classe sont le plus souvent déclarées en mode `private`.

Il existe des données, appelées **constantes** qui, parce qu'elles sont importantes, doivent être visibles par toutes les méthodes de l'application. Ces données sont déclarées en mode `public`. Du fait de leur invariabilité, l'application ne peut modifier leur contenu.

Pour notre exemple, la valeur 600, correspondant à la taille (largeur et hauteur) supposée de l'écran, peut être considérée comme une donnée constante de l'application.

Il suffit de déclarer les variables « constantes » à l'aide du mot-clé `final`. Ainsi, la taille de l'écran peut être définie de la façon suivante :

```
public final int TailleEcran = 600 ;
```

Notons que la taille de l'écran est une valeur indépendante de l'objet `Cercle`. Quelle que soit la forme à dessiner (carré, cercle, etc.), la taille de l'écran est toujours la même. C'est pourquoi il est logique de déclarer la variable `TailleEcran` comme constante de classe à l'aide du mot-clé `static`.

```
public final static int TailleEcran = 600 ;
```

De cette façon, la variable `TailleEcran` est accessible en consultation depuis toute l'application, mais elle ne peut en aucun cas être modifiée, étant déclarée `final`.

Les méthodes `créer()` et `agrandir()` s'écrivent alors de la façon suivante :

```
public void créer()    {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   do  {
     System.out.print(" Rayon            :  ");
     r = Lire.i();
   } while ( r < 0 || r > TailleEcran) ;
}

public void agrandir(int nr)  {
   if (r + nr < 0) r = 0;
```

```
    else if ( r + nr > TailleEcran) r = TailleEcran ;
    else r = r + nr;
  }
```

Des méthodes invisibles

Comme nous l'avons observé précédemment, les données d'une classe sont généralement déclarées en mode `private`. Les méthodes, quant à elles, sont le plus souvent déclarées `public`, car ce sont elles qui permettent l'accès aux données protégées. Dans certains cas particuliers, il peut arriver que certaines méthodes soient définies en mode `private` . Elles deviennent alors inaccessibles depuis les classes extérieures.

Ainsi, le contrôle systématique des données est toujours réalisé par l'objet lui-même, et non par l'application qui utilise les objets. Par conséquent, les méthodes qui ont pour charge de réaliser cette vérification peuvent être définies comme méthodes internes à la classe puisqu'elles ne sont jamais appelées par l'application.

Par exemple, le contrôle de la validité de la valeur du rayon n'est pas réalisée par l'application `FaireDesCercles` mais correspond à une opération interne à la classe `Cercle`. Ce contrôle est réalisé différemment suivant que le cercle est à créer ou à agrandir (*voir les méthodes* `créer()` *et* `agrandir()` *ci-dessus*).

- Soit le rayon n'est pas encore connu, et la vérification s'effectue dès la saisie de la valeur. C'est ce que réalise la méthode suivante :

```
private int rayonVérifié() {
 int tmp;
 do  {
    System.out.print(" Rayon            :  ");
     tmp = Lire.i();
  } while ( tmp < 0 || tmp > TailleEcran) ;
return tmp;
}
```

- Soit le rayon est déjà connu, auquel cas la vérification est réalisée à partir de la valeur passée en paramètre de la méthode :

```
private int rayonVérifié (int tmp) {
 if (tmp < 0) return 0;
   else if ( tmp > TailleEcran) return TailleEcran ;
   else return tmp;
}
```

Les méthodes `rayonVérifié()` sont appelées **méthodes d'implémentation** car elles sont déclarées en mode privé. Leur existence n'est connue d'aucune autre classe. Seules les méthodes de la classe `Cercle` peuvent les exploiter, et elles ne sont pas directement exécutables par l'application. Elle sont cependant très utiles à l'intérieur de la classe où elles sont définies (*voir les sections « Les constructeurs » et « L'héritage »*).

Remarquons, en outre, que nous venons de définir deux méthodes portant le nom `rayonVérifié()`. Le langage Java n'interdit pas la définition de méthodes portant le même nom. Dans cette situation, on dit que ces méthodes sont **surchargées** (*voir la section « La surcharge de constructeurs »*).

Les constructeurs

Grâce aux différents niveaux de protection et aux méthodes contrôlant l'accès aux données, il devient possible de construire des outils appropriés aux objets manipulés.

Cependant, la protection des données d'une classe passe aussi par la notion de constructeurs d'objets. En effet, les constructeurs sont utilisés pour initialiser correctement les données d'un objet au moment de la création de l'objet en mémoire.

Le constructeur par défaut

Le langage Java définit, pour chaque classe construite par le programmeur, un constructeur par défaut. Celui-ci initialise, lors de la création d'un objet, toutes les données de cet objet à `0` pour les entiers, à `0.0` pour les réels, à `'\0'` pour les caractères et à `null` pour les `String` ou autres types structurés.

Le constructeur par défaut est appelé par l'opérateur `new` lors de la réservation de l'espace mémoire. Ainsi, lorsque nous écrivons :

```
Cercle C = new Cercle();
```

nous utilisons le terme `Cercle()`, qui représente en réalité le constructeur par défaut (il ne possède pas de paramètre) de la classe `Cercle`.

Un constructeur est une méthode, puisqu'il y a des parenthèses `()` derrière son nom d'appel, qui porte le nom de la classe associée au type de l'objet déclaré.

Définir le constructeur d'une classe

L'utilisation du constructeur par défaut permet d'initialiser systématiquement les données d'une classe. L'initialisation proposée peut parfois ne pas être conforme aux valeurs demandées par le type.

Dans ce cas, le langage Java offre la possibilité de définir un constructeur propre à la classe de l'objet utilisé. Cette définition est réalisée en écrivant une méthode portant le même nom que sa classe. Les instructions qui la composent permettent d'initialiser les données de la classe, conformément aux valeurs demandées par le type choisi.

Par exemple, le constructeur de la classe `Cercle` peut s'écrire de la façon suivante :

```
public Cercle()    {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   r = rayonVérifié();
}
```

En observant la structure du constructeur `Cercle()`, nous constatons qu'un constructeur n'est pas typé. Aucun type de retour n'est placé dans son en-tête. Mais attention ! le fait d'écrire l'en-tête `public void Cercle()` ou encore `public int Cercle()` a pour résultat de créer une simple méthode, qui a pour nom `Cercle()` et qui n'est pas celle appelée par l'opérateur `new`. Il ne s'agit donc pas d'un constructeur.

Une fois correctement défini, le constructeur est appelé par l'opérateur `new`, comme pour le constructeur par défaut. L'instruction :

```
Cercle A = new Cercle();
```

fait appel au constructeur défini ci-dessus. Le programme exécuté demande, dès la création de l'objet `A`, de saisir les données le concernant, avec une vérification concernant la valeur du rayon grâce à la méthode `rayonVérifié()`. De cette façon, l'application est sûre d'exploiter des objets dont la valeur est valide dès leur initialisation.

Remarquons que :

- Lorsqu'un constructeur est défini par le programmeur, le constructeur proposé par défaut par le langage Java n'existe plus.
- La méthode `créer()` et le constructeur ainsi définis ont un rôle identique. La méthode `créer()` devient par conséquent inutile.

La surcharge de constructeurs

Le langage Java permet la définition de plusieurs constructeurs, ou méthodes, à l'intérieur d'une même classe, du fait que la construction des objets peut se réaliser de différentes façons. Lorsqu'il existe plusieurs constructeurs, on dit que le constructeur est **surchargé**.

Dans la classe `Cercle`, il est possible de définir deux constructeurs supplémentaires :

```
public Cercle(int centrex, int centrey)        {
  x = centrex ;
  y = centrey;
}
public Cercle(int centrex, int centrey, int rayon)        {
  this( centrex, centrey) ;
  r = rayonVérifié(rayon);
}
```

Pour déterminer quel constructeur doit être utilisé, l'interpréteur Java regarde, lors de son appel, la liste des paramètres définis dans chaque constructeur. La construction des trois objets `A`, `B` et `C` suivants fait appel aux trois constructeurs définis précédemment :

```
Cercle A = new Cercle();
Cercle B = new Cercle(10, 10);
Cercle c = new Cercle(10, 10, 30);
```

Lors de la déclaration de l'objet `A`, le constructeur appelé est celui qui ne possède pas de paramètre (le constructeur par défaut, défini à la section « Définir le constructeur d'une classe »), et les valeurs du centre et du rayon du cercle `A` sont celles saisies au clavier par l'utilisateur.

La création de l'objet `B` fait appel au constructeur qui possède deux paramètres de type entier. Les valeurs du centre et du rayon du cercle `B` sont donc celles passées en paramètre du constructeur, soit (`10`, `10`) pour (`B.x`, `B.y`). Aucune valeur n'étant précisée pour le rayon, `B.r` est automatiquement initialisé à `0`.

Le mot-clé `this`

La création de l'objet `C` est réalisée par le constructeur qui possède trois paramètres entiers. Ces paramètres permettent l'initialisation de toutes les données définies dans la classe `Cercle`.

Remarquons que, grâce à l'instruction `this(centrex, centrey)`, le constructeur possédant deux paramètres est appelé à l'intérieur du constructeur possédant trois paramètres.

Le mot-clé `this()` représente l'appel au second constructeur de la même classe possédant deux paramètres entiers, puisque `this()` est appelé avec deux paramètres entiers. Il permet l'utilisation du constructeur précédent pour initialiser les coordonnées du centre avant d'initialiser correctement la valeur du rayon grâce à la méthode `rayonVérifié(rayon)`, qui est elle-même surchargée. Comme pour les constructeurs, le compilateur choisit la méthode `rayonVérifié()`, définie avec un paramètre entier.

Pour finir, remarquons que le terme `this()` doit toujours être placé comme première instruction du constructeur qui l'utilise.

L'héritage

L'héritage est le dernier concept fondamental de la programmation objet étudiée dans ce chapitre. Ce concept permet la réutilisation des fonctionnalités d'une classe, tout en apportant certaines variations, spécifiques de l'objet héritant.

Avec l'héritage, les méthodes définies pour un ensemble de données sont réutilisables pour des variantes de cet ensemble. Par exemple, si nous supposons qu'une classe `Forme` définisse un ensemble de comportements propres à toute forme géométrique, alors :

- Ces comportements peuvent être réutilisés par la classe `Cercle`, qui est une forme géométrique particulière. Cette réutilisation est effectuée sans avoir à modifier les instructions de la classe `Forme`.
- Il est possible d'ajouter d'autres comportements spécifiques des objets `Cercle`. Ces nouveaux comportements sont valides uniquement pour la classe `Cercle` et non pour la classe `Forme`.

La relation « est un »

En pratique, pour déterminer si une classe `B` **hérite** d'une classe `A`, il suffit de savoir s'il existe une relation « **est un** » entre `B` et `A`. Si tel est le cas, la syntaxe de déclaration est la suivante :

```
class B extends A        {
// données et méthodes de la classe B
}
```

Dans ce cas, on dit que :

- `B` est une **sous-classe** de `A` ou encore une **classe dérivée** de `A`.
- `A` est une **super-classe** ou encore une **classe de base**.

Un cercle « est une » forme géométrique

En supposant que la classe `Forme` possède des caractéristiques communes à chaque type de forme géométrique (les coordonnées d'affichage à l'écran, la couleur, etc.), ainsi que des comportements communs (afficher, déplacer, etc.), la classe `Forme` s'écrit de la façon suivante :

```
public class Forme        {
 protected int x, y ;
 private couleur ;

 public  Forme() {            // Le constructeur de la classe Forme
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   System.out.print(" Couleur de la forme :  ");
   couleur = Lire.i();
 }

 public void afficher()  { //Affichage des données de la classe
  System.out.println(" Position en " + x + ", " + y);
  System.out.println(" Couleur : " + couleur);
 }

 public void déplacer(int nx, int ny) {    // Déplace les coordonnées de la
  x = nx;                                  // forme en (nx, ny) passées en
  y = ny;                                  // paramètre de la fonction
 }
} // Fin de la classe Forme
```

Sachant qu'un objet `Cercle` « est une » forme géométrique particulière, la classe `Cercle` hérite de la classe `Forme` en écrivant :

```
public class Cercle extends Forme     {
 private int r ;          //  rayon

  public Cercle()    {    // Le constructeur de la classe Cercle
   System.out.print(" Rayon            :  ");
   r = rayonVérifié();
}
private int rayonVérifié() {
// Voir la section Des méthodes invisibles
}
private int rayonVérifié (int tmp) {
// Voir la section Des méthodes invisibles
}

 public void afficher() { //Affichage des données de la classe
  super.afficher();
  System.out.println(" Rayon : " + r);
```

```
  }

 public double périmètre() {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
}

 public void agrandir(int nr) {              // Augmente la valeur courante du
  r = rayonVérifié(r + nr);                  // rayon avec la valeur passée en
 }                                           // paramètre
} // Fin de la classe Cercle
```

Un cercle est une forme géométrique (`Cercle extends Forme`) qui possède un rayon (`private int r`) et des comportements propres aux cercles, soit, par exemple, le calcul du périmètre (`périmètre()`) ou encore la modification de sa taille (`agrandir()`). Un cercle peut être déplacé, comme toute forme géométrique. Les méthodes de la classe `Forme` restent donc opérationnelles pour les objets `Cercle`.

En examinant de plus près les classes `Cercle` et `Forme`, nous remarquons que :

- La notion de constructeur existe aussi pour les classes dérivées (*voir la section « Le constructeur d'une classe héritée »*).
- Les données `x`, `y` sont déclarées `protected` (*voir la section « La protection des données héritées »*).
- La fonction `afficher()` existe sous deux formes différentes dans la classe `Forme` et la classe `Cercle`. Il s'agit là du concept de polymorphisme (*voir la section « Le polymorphisme »*).

Le constructeur d'une classe héritée

Les classes dérivées possèdent leurs propres constructeurs, qui sont appelés par l'opérateur `new`, comme dans :

```
Cercle A = new Cercle( );
```

Pour construire un objet dérivé, il est indispensable de construire d'abord l'objet associé à la classe mère. Pour construire un objet `Cercle`, nous devons définir ses coordonnées et sa couleur. Le constructeur de la classe `Cercle` doit appeler le constructeur de la classe `Forme`.

Par défaut, s'il n'y a pas d'appel explicite au constructeur de la classe supérieure, comme c'est le cas pour notre exemple, le compilateur recherche de lui-même le constructeur par défaut (sans paramètre) de la classe supérieure. En construisant l'objet `A`, l'interpréteur exécute aussi le constructeur par défaut de la classe `Forme`. L'ensemble des données du cercle (`x`, `y`, `couleur` et `r`) est alors correctement initialisé par saisie des valeurs au clavier.

Ce fonctionnement pose problème lorsqu'il n'existe pas de constructeur par défaut. Supposons que nous remplacions le constructeur de la classe `Forme` par :

```
public Forme(int nx, int ny) {        // Le nouveau constructeur de la
  x = nx ;                            // classe Forme
```

```
    y = ny ;
    couleur = 0;
  }
```

Dans cette situation, lors de la construction de l'objet `A`, le compilateur recherche le constructeur par défaut de la classe supérieure, soit `Forme()` sans paramètre. Ne le trouvant pas, il annonce une erreur du type `no constructor matching Forme() found in class Forme`.

Le mot-clé `super`

Pour éviter ce type d'erreur, la solution consiste à appeler directement le constructeur de la classe mère depuis le constructeur de la classe :

```
public Cercle(int xx, int yy)        {    // Le constructeur de la classe Cercle
    super(xx, yy);
    System.out.print(" Rayon            :  ");
    r = rayonVérifié();
}
```

De cette façon, comme le terme `super()`, qui représente le constructeur de la classe supérieure possèdant deux entiers en paramètres, l'interpréteur peut finalement construire l'objet `A` (`Cercle A = new Cercle(5, 5)`), par appel du constructeur de la classe `Forme` à l'intérieur du constructeur de la classe `Cercle`.

Remarquons que le terme `super` est obligatoirement la première instruction du constructeur de la classe dérivée. La liste des paramètres (deux `int`) permet de préciser au compilateur quel est le constructeur utilisé en cas de surcharge de constructeurs.

La protection des données héritées

En héritant de la classe `A`, la classe `B` hérite des données et méthodes de la classe `A`. Cela ne veut pas forcément dire que la classe `B` ait accès à toutes les données et méthodes de la classe `A`. En effet, héritage n'est pas synonyme d'accessibilité.

Lorsqu'une donnée de la classe supérieure est déclarée en mode `private`, la classe dérivée ne peut ni consulter ni modifier directement cette donnée héritée. L'accès ne peut se réaliser qu'au travers des méthodes de la classe supérieure.

Pour notre exemple, la donnée `couleur` étant déclarée `private` dans la classe `Forme`, le constructeur suivant génère l'erreur `variable couleur in class Forme not accessible from class Cercle`.

```
public Cercle(int xx, int yy)   {   // Le constructeur de la classe Cercle
    super(xx, yy);
    couleur = 20 ;
    r = 10;
}
```

Il est possible, grâce à la protection `protected`, d'autoriser l'accès en consultation et modification des données de la classe supérieure. Toutes les données de la classe `A` sont alors accessibles depuis la classe `B`, mais pas depuis une autre classe.

Dans notre exemple, si la donnée `couleur` est déclarée `protected` dans la classe `Forme`, alors le constructeur de la classe `Cercle` peut modifier sa valeur.

Le polymorphisme

La notion de polymorphisme découle directement de l'héritage. Par polymorphisme, il faut comprendre qu'une méthode peut se comporter différemment suivant l'objet sur lequel elle est appliquée.

Lorsqu'une même méthode est définie à la fois dans la classe mère et dans la classe fille, l'exécution de la forme (méthode) choisie est réalisée en fonction de l'objet associé à l'appel et non plus suivant le nombre et le type des paramètres, comme c'est le cas lors de la surcharge de méthodes à l'intérieur d'une même classe.

Pour notre exemple, la méthode `afficher()` est décrite dans la classe `Forme` et dans la classe `Cercle`. Cette double définition ne correspond pas à une véritable surcharge de fonctions. Ici, les deux méthodes `afficher()` sont définies sans aucun paramètre. Le choix de la méthode ne peut donc s'effectuer sur la différence des paramètres. Il est effectué par rapport à l'objet sur lequel la méthode est appliquée. Observons l'exécution du programme suivant :

```
public class FormerDesCercles            {
 public static void main(String [] arg)    {
  Cercle A  = new Cercle(5, 5);
  A.afficher();
  Forme F = new Forme (10, 10, 3);
  F.afficher();
 }
}
```

L'appel du constructeur de l'objet `A` nous demande de saisir la valeur du rayon :

```
Rayon            :  7
```

La méthode `afficher()` est appliquée à `A`. Puisque `A` est de type `Cercle`, l'affichage correspond à celui réalisé par la méthode définie dans la classe `Cercle`, soit :

```
Position en 5, 5
Couleur : 20
Rayon : 7
```

La forme `F` est ensuite créée puis affichée à l'aide la méthode `afficher()` de la classe `Forme`, `F` étant de type `Forme` :

```
Position en 10, 1
Couleur : 3
```

Remarquons que, lorsqu'une méthode héritée est définie une deuxième fois dans la classe dérivée, l'héritage est supprimé. Le fait d'écrire `A.afficher()` ne permet plus d'appeler directement la méthode `afficher()` de la classe `Forme`.

Pour appeler la méthode définie dans la classe supérieure, la solution consiste à utiliser le terme `super`, qui recherche la méthode à exécuter en remontant dans la hiérarchie.

Dans notre exemple, `super.afficher()` permet d'appeler la méthode `afficher()` de la classe `Forme`.

Grâce à cette technique, si la méthode d'affichage pour une `Forme` est transformée, cette transformation est automatiquement répercutée pour un `Cercle`.

Résumé

Lorsque l'interpréteur Java rencontre le mot-clé **`static`** devant une variable (**variable de classe**), il réserve un seul et unique emplacement mémoire pour cette variable. Si ce mot-clé est absent, l'interpréteur peut construire en mémoire la variable déclarée non `static` (**variable d'instance**) en plusieurs exemplaires. Cette présence ou cette absence du mot-clé `static` permet de différencier les variables des objets.

Les objets sont définis en mémoire par l'intermédiaire d'une **adresse** (**référence**). Lorsqu'un objet est passé en paramètre d'une fonction, la valeur passée au paramètre formel est l'adresse de l'objet. De cette façon, si la méthode transforme les données du paramètre formel, elle modifie aussi les données de l'objet effectivement passé en paramètre. Ainsi, tout objet passé en paramètre d'une méthode voit, en sortie de la méthode, ses données transformées par la méthode. Ce mode de transmission des données est appelé **passage** de paramètres **par référence**.

L'objectif principal de la programmation objet est d'écrire des programmes qui contrôlent par eux-mêmes le bien-fondé des opérations qui leur sont appliquées. Ce contrôle est réalisé grâce au principe d'**encapsulation** des données. Par ce terme, il faut comprendre que les données d'un objet sont protégées, de la même façon qu'un médicament est protégé par la fine capsule qui l'entoure. L'encapsulation passe par le **contrôle des données** et des comportements de l'objet à travers les niveaux de **protection**, l'**accès** contrôlé aux données et la notion de **constructeur** de classe.

Le langage Java propose trois niveaux de protection, `public`, `private` et `protected`. Lorsqu'une donnée est totalement protégée (`private`), elle ne peut être modifiée que par les méthodes de la classe où la donnée est définie.

On distingue les méthodes qui consultent la valeur d'une donnée sans pouvoir la modifier (**accesseur en consultation**) et celles qui modifient après contrôle et validation la valeur de la donnée (**accesseur en modification**).

Les constructeurs sont des méthodes particulières, déclarées uniquement `public`, qui portent le même nom que la classe où ils sont définis. Ils permettent le contrôle et la validation des données dès leur initialisation.

Par défaut, si aucun constructeur n'est défini dans une classe, le langage Java propose un constructeur par défaut, qui initialise toutes les données de la classe à `0` ou à `null`, si les données sont des objets. Si un constructeur est défini, le constructeur par défaut n'existe plus.

L'**héritage** permet la réutilisation des objets et de leur comportement, tout en apportant de légères variations. Il se traduit par le principe suivant : on dit qu'une classe `B` hérite d'une classe `A` (`B` étant une sous-classe de `A`) lorsqu'il est possible de mettre la relation « est un » entre `B` et `A`.

De cette façon, toutes les méthodes, ainsi que les données déclarées `public` ou `protected`, de la classe `A` sont applicables à la classe `B`. La syntaxe de déclaration d'une sous-classe est la suivante :

```
class B extends A         {
// données et méthodes de la classe B
}
```

Le projet « Gestion d'un compte bancaire »

Encapsuler les données d'un compte bancaire

La protection `privée` et l'accès aux données

a. Déclarez toutes les variables d'instance des types `Compte` et `LigneComptable` en mode `private`. Que se passe-t-il lors de la phase de compilation de l'application `Projet` ?

Pour remédier à cette situation, la solution est de construire des méthodes d'accès aux données de la classe `Compte` et `LigneComptable`. Ces méthodes ont pour objectif de fournir au programme appelant la valeur de la donnée recherchée. Par exemple, la fonction `quelTypeDeCompte()` suivante fournit en retour le type du compte recherché :

```
public String quelTypeDeCompte()     {
    return typeCpte;
}
```

b. Écrivez, suivant le même modèle, toutes les méthodes d'accès aux données `val_courante`, `taux`, `numéroCpte`, etc.

c. Modifiez l'application `Projet` et la classe `Compte` de façon à pouvoir accéder aux données `numéroCpte` de la classe `Compte` et aux valeurs de la classe `LigneComptable`.

Le contrôle des données

L'encapsulation des données permet le contrôle de la validité des données saisies pour un objet. Un compte bancaire ne peut être que de trois types : `Epargne`, `Courant` ou `Joint`. Il est donc nécessaire, au moment de la saisie du `type` du compte, de contrôler l'exactitude du type entré. La méthode `contrôleType()` suivante réalise ce contrôle :

```
private String contrôleType()    {
 char tmpc;
 String tmpS = "Courant";
 do {
  System.out.print("Type du compte [Types possibles : C(ourant), ");
  System.out.print("J(oint), E(pargne)] : ");
  tmpc = Lire.c();
 } while ( tmpc != 'C' && tmpc != 'J' && tmpc != 'E');
  switch (tmpc) {
   case 'C' : tmpS = "Courant";
              break;
```

```
    case 'J' : tmpS = "Joint";
               break;
    case 'E' : tmpS = "Epargne";
               break;
   }
   return tmpS;
  }
```

À la sortie de la fonction, nous sommes certains que le type retourné correspond aux types autorisés par le cahier des charges.

a. Dans la classe `Compte`, sachant que la valeur initiale ne peut être négative à la création d'un compte, écrivez la méthode `contrôleValinit()`.

b. Dans la classe `LigneComptable`, écrivez les méthodes `contrôleMotif()` et `contrôleMode()`, qui vérifient respectivement le motif (`Salaire`, `Loyer`, `Alimentation`, `Divers`) et le mode (`CB`, `Virement`, `Chèque`) de paiement pour une ligne comptable.

> ✔ Pour contrôler la validité de la date, voir la section *« Le projet… »* du chapitre 10, *« Collectionner un nombre indéterminé d'objets »*.

c. Modifiez les méthodes `créerCpte()` et `créerLigneComptable()` de façon que les données des classes `Compte()` et `LigneComptable()` soient valides.

Les constructeurs de classe

Les constructeurs `Compte()` et `LigneComptable()` s'inspirent pour une grande part des méthodes `créerCpte()` et `créerLigneComptable()`.

a. Remplacez directement `créerCpte()` par `Compte()`. Que se passe-t-il lors de l'exécution du programme ?

b. Déplacez l'appel au constructeur dans l'option `1`, de façon à construire l'objet au moment de sa création. Que se passe-t-il en phase de compilation ? Pourquoi ?

c. Utilisez la notion de surcharge de constructeur pour construire un objet `C` de deux façons :

- Les valeurs initiales du compte sont passées en paramètre.
- Les valeurs initiales sont saisies au clavier, comme le fait la méthode `créerCpte()`.

d. À l'aide de ces deux constructeurs, modifiez l'application `Projet` de façon à pouvoir l'exécuter correctement.

Comprendre l'héritage

Protection des données héritées

Sachant qu'un compte d'épargne est un compte bancaire ayant un taux de rémunération,

a. Écrivez la classe `CpteEpargne` en prenant soin de déclarer la nouvelle donnée en mode `private`.

b. Modifiez le type `Compte` de façon à supprimer tout ce qui fait appel au compte d'épargne (donnée et méthodes).

Un compte d'épargne modifie la valeur courante par le calcul des intérêts, en fonction du taux d'épargne. Il ne peut ni modifier son numéro, ni son type.

c. Quels modes de protection doit-on appliquer aux différentes données héritées de la classe `Compte` ?

Le contrôle des données d'un compte d'épargne

Sachant que le taux d'un compte d'épargne ne peut être négatif, écrivez la méthode `contrôleTaux()`.

Le constructeur d'une classe dérivée

En supposant que le constructeur de la classe `CpteEpargne` s'écrive de la façon suivante :

```
public CpteEpargne() {
  super("Epargne");
  taux = contrôleTaux();
}
```

a. Recherchez à quel constructeur de la classe `Compte` fait appel `CpteEpargne()`. Pourquoi ?

b. Modifiez ce constructeur de façon que la donnée `typeCpte` prenne la valeur `Epargne`.

Le polymorphisme

De la méthode `afficherCpte()` :

a. Dans la classe `CpteEpargne`, écrivez la méthode `afficherCpte()`, sachant qu'afficher les données d'un compte d'épargne revient à afficher les données d'un compte, suivi du taux d'épargne.

De l'objet `C`, déclaré de type `Compte` :

b. Dans l'application `Projet`, modifiez l'option `1`, de façon à demander à l'utilisateur s'il souhaite créer un compte simple ou un compte d'épargne. Selon la réponse, construisez l'objet `C` en appelant le constructeur approprié.