Anne Tasso

Le livre de Java premier langage

8e édition

 ISBN : 978-2-212-13538-1

EYROLLES

Avant-propos

Organisation de l'ouvrage

Ce livre est tout particulièrement destiné aux débutants qui souhaitent aborder l'apprentissage de la programmation en utilisant le langage Java comme premier langage.

Les concepts fondamentaux de la programmation y sont présentés de façon évolutive, grâce à un découpage de l'ouvrage en trois parties, chacune couvrant un aspect différent des outils et techniques de programmation.

Le chapitre introductif, « Naissance d'un programme », constitue le préalable nécessaire à la bonne compréhension des parties suivantes. Il introduit aux mécanismes de construction d'un algorithme, compte tenu du fonctionnement interne de l'ordinateur, et explique les notions de langage informatique, de compilation et d'exécution à travers un exemple de programme écrit en Java.

La première partie concerne l'étude des « Outils et techniques de base » :

- Le chapitre 1, « Stocker une information », aborde la notion de variables et de types. Il présente comment stocker une donnée en mémoire, calculer des expressions mathématiques ou échanger deux valeurs, et montre comment le type d'une variable peut influencer le résultat d'un calcul.
- Le chapitre 2, « Communiquer une information », explique comment transmettre des valeurs à l'ordinateur par l'intermédiaire du clavier et montre comment l'ordinateur fournit des résultats en affichant des messages à l'écran.
- Le chapitre 3, « Faire des choix », examine comment tester des valeurs et prendre des décisions en fonction du résultat. Il traite de la comparaison de valeurs ainsi que de l'arborescence de choix. Avec en exemple, la nouvelle structure de test `switch` de la version 7 de Java.
- Le chapitre 4, « Faire des répétitions », est consacré à l'étude des outils de répétition et d'itération. Il aborde les notions d'incrémentation et d'accumulation de valeurs (compter et faire la somme d'une collection de valeurs).

La deuxième partie, « Initiation à la programmation orientée objet », introduit les concepts fondamentaux indispensables à la programmation objet.

- Le chapitre 5, « De l'algorithme paramétré à l'écriture de fonctions », montre l'intérêt de l'emploi de fonctions dans la programmation. Il examine les différentes étapes de leur création.

- Le chapitre 6, « Fonctions, notions avancées », décrit très précisément comment manipuler les fonctions et leurs paramètres. Il définit les termes de variable locale et de classe, et explique le passage de paramètres par valeur.
- Le chapitre 7, « Les classes et les objets », explique à partir de l'étude de la classe `String`, ce que sont les classes et les objets dans le langage Java. Il montre ensuite comment définir de nouvelles classes et construire des objets propres à l'application développée. Avec en exemple, une nouvelle façon de comparer des chaînes de caractères grâce à la nouvelle structure de test `switch` de la version 7 de Java.
- Le chapitre 8, « Les principes du concept d'objet », développe plus particulièrement comment les objets se communiquent l'information, en expliquant notamment le principe du passage de paramètres par référence. Il décrit ensuite les principes fondateurs de la notion d'objet, c'est-à-dire l'encapsulation des données (protection et contrôle des données, constructeur de classe) ainsi que l'héritage entre classes et la notion d'interfaces.

La troisième partie, « Outils et techniques orientés objet », donne tous les détails sur l'organisation, le traitement et l'exploitation intelligente des objets.

- Le chapitre 9, « Collectionner un nombre fixe d'objets », concerne l'organisation des données sous la forme d'un tableau de taille fixe.
- Le chapitre 10, « Collectionner un nombre indéterminé d'objets », présente les différents outils qui permettent d'organiser dynamiquement en mémoire les ensembles de données de même nature. Il est également consacré aux différentes techniques d'archivage et à la façon d'accéder aux informations stockées sous forme de fichiers.
- Le chapitre 11, « Dessiner des objets », couvre une grande partie des outils graphiques proposés par le langage Java (bibliothèques AWT et Swing). Il analyse le concept événement-action.
- Le chapitre 12, « Créer une interface graphique », expose dans un premier temps le fonctionnement de base de l'environnement de programmation NetBeans. Puis, à travers trois exemples très pratiques, il montre comment concevoir des applications munies d'interfaces graphiques conviviales.
- Le chapitre 13, « Développer une application Android », décrit comment créer votre toute première application Android, tout en expliquant la structure de base nécessaire au déploiement de cette application. Il présente ensuite le développement d'une application plus élaborée ainsi que sa mise à disposition sur un serveur dédié.

Ce livre contient également en annexe :

- un guide d'installation détaillé des outils nécessaires au développement des applications Java (Java, NetBeans), sous Linux, Mac OS X et sous Windows 2000, NT, XP et Vista ;
- toutes les explications nécessaires pour construire votre environnement de développement d'applications Java ou Android, que ce soit en mode commande ou en utilisant la plateforme NetBeans ;

- un index, qui vous aidera à retrouver une information sur le thème que vous recherchez (les mots-clés du langage, les exemples, les principes de fonctionnement, les classes et leurs méthodes, etc.).

Chaque chapitre se termine sur une série d'exercices offrant au lecteur la possibilité de mettre en pratique les notions qui viennent d'être étudiées. Un projet est également proposé au fil des chapitres afin de développer une application de gestion d'un compte bancaire. La mise en œuvre de cette application constitue un fil rouge qui permettra au lecteur de combiner toutes les techniques de programmation étudiées au fur et à mesure de l'ouvrage, afin de construire une véritable application Java.

Extension Web Les codes sources des exemples, des exercices et du projet sont téléchargeables depuis l'extension Web *www.annetasso.fr/Java* en cliquant sur le lien Sources.

Table des matières

Partie II
Initiation à la programmation orientée objet

Remarques

Pour déterminer une relation `if-else`, observons qu'un « bloc `else` » se rapporte toujours au dernier « bloc `if` » rencontré, auquel un `else` n'a pas encore été attribué.

Les blocs `if` et `else` étant délimités par les accolades ouvrantes et fermantes, il est conseillé, pour éviter toute erreur, de bien relier chaque parenthèse ouvrante avec sa fermante.

L'instruction switch, ou comment faire des choix multiples

Lorsque le nombre de choix possible est plus grand que deux, l'utilisation de la structure `if-else` devient rapidement fastidieuse. Les imbrications des blocs demandent à être vérifiées avec précision, sous peine d'erreur de compilation ou d'exécution.

C'est pourquoi, le langage Java propose l'instruction `switch` (traduire par selon, ou suivant), qui permet de programmer des choix multiples selon une syntaxe plus claire.

Construction du switch

L'écriture de l'instruction `switch` obéit aux règles de syntaxe suivantes :

```
switch (valeur)
{
  case étiquette 1 :
      // Une ou plusieurs instructions
  break ;
  case étiquette 2 :
  case étiquette 3 :
      // Une ou plusieurs instructions
  break ;
  default :
      // Une ou plusieurs instructions
}
```

La variable `valeur` est évaluée. Suivant cette valeur, le programme recherche l'`étiquette` correspondant à la valeur obtenue et définie à partir des instructions `case` *`étiquette`*.

- Si le programme trouve une étiquette correspondant au contenu de la variable `valeur`, il exécute la ou les instructions qui suivent l'étiquette, jusqu'à rencontrer le mot-clé `break`.
- S'il n'existe pas d'étiquette correspondant à `valeur`, alors le programme exécute les instructions de l'étiquette `default`.

- Le type de la variable `valeur` ne peut être que `char` ou `int`, `byte`, `short` ou `long`. Il n'est donc pas possible de tester des valeurs réelles.
- Une étiquette peut contenir aucune, une ou plusieurs instructions.
- L'instruction `break` permet de sortir du bloc `switch`. S'il n'y a pas de `break` pour une étiquette donnée, le programme exécute les instructions de l'étiquette suivante.

Remarques Notes sur la version Java 7

À partir de la version 7 de Java, le test sur des chaînes de caractères est autorisé. Il devient possible de réaliser un `switch` en utilisant des mots comme étiquette. Par exemple :

```
String choix ;
Scanner lectureClavier = new Scanner(System.in);
System.out.println("Votre choix  (oui/non) ? : ") ;
choix = lectureClavier.nextLine();

switch (choix)
{
  case  "oui" :
        System.out.println("Vous avez saisi oui !") ;
  break ;
  case "non" :
       System.out.println("Vous avez saisi non !") ;
  break ;
  default :
       System.out.println("Vous avez saisi ni oui ni non !") ;
}
```

Calculer le nombre de jours d'un mois donné

Pour mettre en pratique les notions théoriques abordées à la section précédente, nous allons écrire un programme qui calcule et affiche le nombre de jours d'un mois donné.

Le nombre de jours dans un mois peut varier entre les valeurs `28`, `29`, `30` ou `31`, suivant le mois et l'année. Les mois de janvier, mars, mai, juillet, août, octobre et décembre sont des mois de `31` jours. Les mois d'avril, juin, septembre et novembre sont des mois de `30` jours. Seul le mois de février est particulier, puisque son nombre de jours est de `29` jours pour les années bissextiles et de `28` jours dans le cas contraire. Sachant cela, nous devons :

- Demander la saisie au clavier du numéro du mois ainsi que de l'année recherchée.

- Créer autant d'étiquettes qu'il y a de mois dans une année, c'est-à-dire `12`. Compte tenu du fonctionnement de la structure `switch`, chaque étiquette est une valeur entière correspondant au numéro du mois de l'année (`1` pour janvier, `2` pour février, etc.).
- Regrouper les étiquettes relatives aux mois à `31` jours et stocker cette dernière valeur dans une variable spécifique.
- Regrouper les étiquettes relatives aux mois à `30` jours et stocker cette dernière valeur dans une variable spécifique.
- Pour l'étiquette relative au mois de février, tester la valeur de l'année pour savoir si l'année concernée est bissextile ou non. Une année est bissextile tous les `quatre` ans, sauf lorsque le millésime est divisible par `100` et non pas par `400`. En d'autres termes, pour qu'une année soit bissextile, il suffit que l'année soit un nombre divisible par 4 et non divisible par `100` ou alors par `400`. Dans tous les autres cas, l'année n'est pas bissextile.

Compte tenu de toutes ces remarques, nous devons dans un premier temps déclarer trois variables entières, une pour représenter le mois, la deuxième l'année, et la troisième le nombre de jours par mois. Sachant que le mois et le nombre de jours par mois ne dépassent jamais la valeur `127`, nous pouvons les déclarer de type `byte`. Pour l'année, le type `short` suffit, puisque les valeurs de ce type peuvent aller jusqu'à `32767`.

Exemple : code source complet

```
import java.util.*;
public class JourParMois  // Le fichier s'appelle JourParMois.java
{
  public static void main (String [] parametre)
  {
   byte mois, nbjours = 0 ;
   short année ;
   Scanner lectureClavier = new Scanner(System.in);
   System.out.println("De quel mois s'agit-il ? : ") ;
   mois = lectureClavier.nextByte();
   System.out.println("De quelle annee ? : ") ;
   année = lectureClavier.nextShort();
   switch(mois)
   {
     case 1 : case 3 :    // Pour les mois à 31 jours
     case 5 : case 7 :
     case 8 : case 10 :
     case 12 :
            nbjours = 31 ;
            break ;
     case 4 : case 6 :    // Pour les mois à 30 jours
```

```
      case 9 : case 11 :
             nbjours  = 30 ;
             break ;
     case 2 :          // Pour le cas particulier du mois de février
          if (année % 4 == 0 && année % 100 != 0 || année %
                              400 == 0)
               nbjours  = 29 ;
          else  nbjours  = 28 ;
          break ;
      default :         // En cas d'erreur de frappe
           System.out.println("Impossible, ce mois n'existe pas ") ;
           System.exit(0) ;
    }
    System.out.print(" En " + annee + ", le mois n° " + mois) ;
    System.out.println(" a " + nbjours + " jours ") ;
  } // Fin du main()
} // Fin de la class JourParMois
```

Question Que se passe-t-il si l'utilisateur entre les valeurs suivantes :

```
De quel mois s'agit-il ? : 5
De quelle annee ? : 1999
```

Réponse Le programme affiche la réponse suivante :

```
En 1999 le mois n° 5 a 31 jours
```

Le programme recherche l'étiquette `5`. Il exécute les instructions qui suivent jusqu'à rencontrer un `break`. Pour l'étiquette `5`, le programme exécute les instructions des étiquettes `7`, `8`, `10` et `12` car ces étiquettes ne possèdent ni instruction, ni `break`. Seule l'étiquette `12` possède une instruction, qui affecte la valeur `31` à la variable `nbjours`. L'instruction `break` qui suit permet de sortir de la structure `switch`. Le programme exécute enfin l'instruction située immédiatement après le `switch`, c'est-à-dire l'affichage du message annonçant le résultat.

Question Que se passe-t-il si l'utilisateur entre les valeurs suivantes :

```
De quel mois s'agit-il ? : 2
De quelle annee ? : 2000
```

Réponse Le programme affiche la réponse suivante :

```
En 2000 le mois n° 2 a 29 jours
```

Ici, le programme va directement à l'étiquette `2`, qui est composée d'un test sur l'année pour savoir si l'année est bissextile. Une année est bissextile lorsque son millésime est divisible par `4`, à l'exception des années dont le millésime est divisible par `100` et non pas par `400`. La valeur `2 000` est divisible par `4`, `100` et `400` puisque le reste de la division entière (`%`) de `2000` par `4`, `100` ou `400` est nul. La variable `nbjours` prend donc la valeur `29`. Le programme sort ensuite du `switch` grâce à l'instruction `break` qui suit et exécute pour finir l'affichage du résultat.

Question Que se passe-t-il si l'utilisateur entre les valeurs suivantes :

```
De quel mois s'agit-il ? : 15
De quelle annee ? : 1999
```

Réponse Le programme affiche la réponse suivante :

```
Impossible, ce mois n'existe pas
```

L'étiquette `15` n'étant pas définie dans le bloc `switch`, le programme exécute les instructions qui composent l'étiquette `default`. Le programme affiche un message d'erreur et termine son exécution grâce à l'instruction `System.exit(0) ;`

Remarque Grâce à l'étiquette `default`, le programme connaît les instructions à exécuter dans le cas de choix « anormaux » (erreur de frappe, par exemple, ou valeur saisie n'entrant pas dans l'intervalle des valeurs possibles traitées par le programme). De cette façon, il devient possible de prévenir d'éventuelles erreurs pouvant causer l'arrêt brutal de l'exécution du programme.

Comment choisir entre if-else et switch ?

La structure `switch` ne permet de tester que des égalités de valeurs entières (`byte`, `short`, `int` ou `long`) ou de type caractère `char` (et `String` à partir de la version 7 de Java). Elle **ne peut** donc **pas** être utilisée pour :

- Tester des valeurs réelles (`float` ou `double`).
- Rechercher si la valeur est plus grande, plus petite ou différente d'une certaine étiquette.

Par contre, l'instruction `if-else` peut être employée dans tous les cas en testant tout type de variable, selon toute condition.

Remarques Si une condition parmi d'autres conditions envisagées a une plus grande probabilité d'être satisfaite, celle-ci doit être placée en premier test dans une structure `if-else`, de façon à éviter à l'ordinateur d'effectuer de trop nombreux tests inutiles.

Si toutes les conditions ont une probabilité voisine ou équivalente d'être réalisées, la structure `switch` est plus efficace. Elle ne demande qu'une seule évaluation, alors que dans les instructions `if-else` imbriquées, chaque condition doit être évaluée.

Remarque Dans ce chapitre, les termes « applications mobiles » recouvrent par extension les applications pour téléphones mobiles et tab lettes tactiles. Le mode de prog rammation de ces deux appareils est quasi identique , seul le système d'e xploitation impor te dans le choix du langage de programmation.

Les mobiles basés sur un système d'exploitation de type Android (HTC, Samsung, Sony) utilisent des langages de programmation tels que Java ou AS3-Air, alors que ceux basés sur un système d'exploitation de type iOS (iPhone, iPad) utilisent des langages de programmation tels que Objective C ou également AS3-Air.

Ainsi, le développement d'une application Android requiert l'utilisation d'un environnement de développement spécifique. Dans cet ouvrage, nous avons choisi de présenter l'IDE NetBeans et son plug-in adapté pour la création d'applications Android. Pour installer cet environnement, reportez-vous à l'annexe « Guide d'installations », section « Installation d'un environnement de développement pour Android ».

Il existe aussi d'autres plug-ins Android notamment pour Eclipse, IntelliJ ou encore l'environnement de développement AIDE (pour *Android Java IDE*) qui a la particularité de s'installer directement sur une tablette Android.

Bonjour le monde : votre première application mobile

L'objectif est de créer une toute première application Android afin de se familiariser avec les outils de développement, de compilation, d'exécution et comprendre la structure générale d'une telle application.

Extension Web Vous trouverez tous les codes de cette première application dans le réper toire `Sources/Exemple/Chapitre13/NetBeansProjects/BonjourLeMonde`.

Création d'un projet Android

La création et l'exécution d'une application Android s'effectuent dans le cadre d'un projet NetBeans au sein duquel sont regroupées toutes les ressources nécessaires à la bonne marche de l'application.

Pour créer un projet Android, une fois le plug-in installé et NetBeans lancé, vous devez sélectionner l'item Nouveau projet du menu Fichier.

Dans la boîte de dialogue qui apparaît (figure 13-1), choisissez la catégorie Android et comme type de projet, Android Project. Cliquez ensuite sur le bouton Suivant.

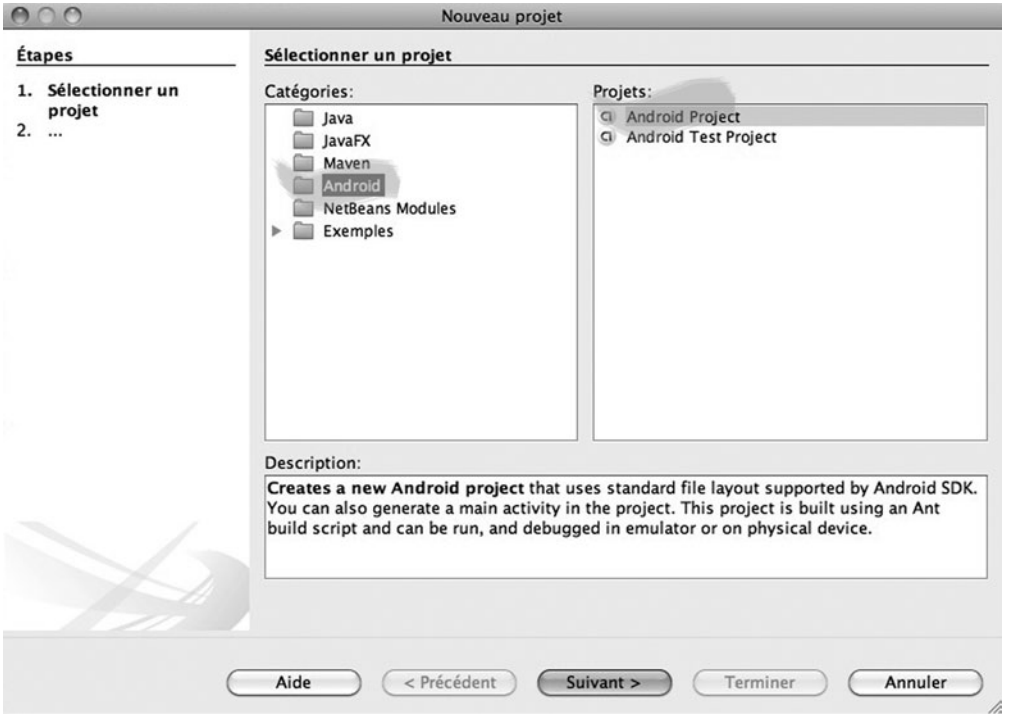


Figure 13-1 *La boîte de dialogue Nouveau projet*

Une nouvelle boîte de dialogue apparaît, nommée Nouveau Android Application (figure 13-2), dans laquelle vous devez :

1. Saisir le nom du projet (ici, `BonjourLeMonde`).
2. Si le dossier d'enregistrement du projet indiqué par défaut ne vous convient pas, spécifier un autre emplacement en cliquant sur le bouton Browse.
3. Définir le nom du package (ici, `android.PremierProjet`).
4. Nommer le fichier principal de l'application (ici, `Main`).
5. Sélectionner la version du mobile Android sur laquelle vous souhaitez exécuter la simulation (ici, `Android 2.2`).
6. Valider l'enregistrement du projet en cliquant sur le bouton Terminer.

Pour en savoir plus La notion de package est définie au chapitre 12, « Créer une interface graphique », section « Développer en mode graphique ».

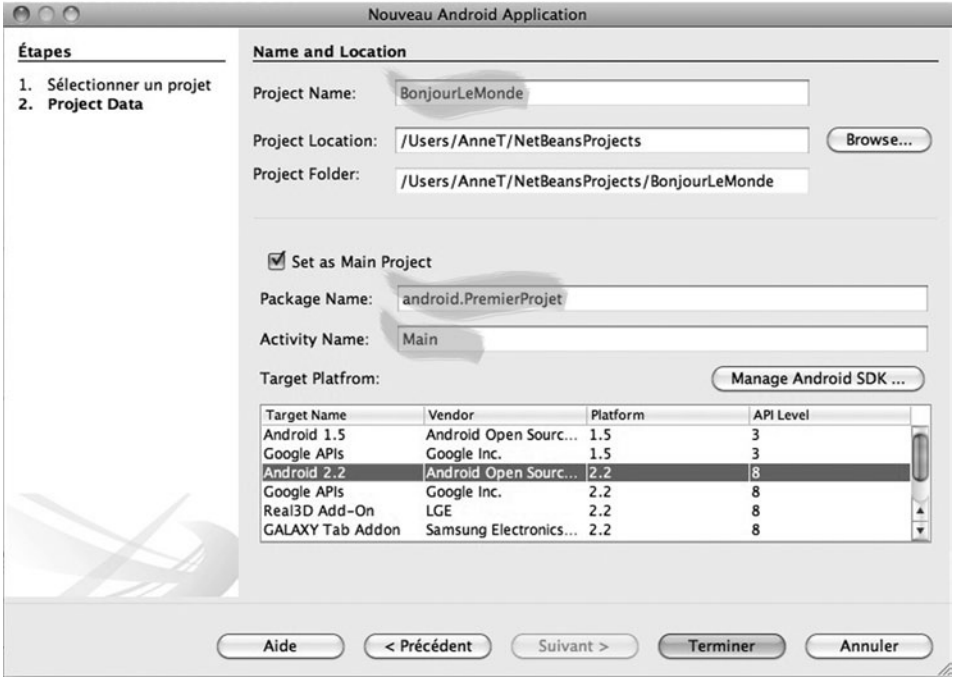


Figure 13-2 *La boîte de dialogue Nouveau Android Application*

Une fois le projet créé et enregistré, NetBeans affiche à l'écran un ensemble de panneaux vides (figure 13-3). Seul la panneau Projets indique que `BonjourLeMonde` a bien été créé.

Figure 13-3 *Le panneau Projets*

Ce projet présente une structure assez complexe. Vous pouvez également observer une erreur de syntaxe indiquée par un point d'exclamation blanc sur fond rouge sur une partie de l'arborescence. Avant de comprendre et de corriger cette erreur, examinons plus précisément le contenu de cette arborescence.

Arborescence du projet Android

Quelle que soit l'interface de développement utilisée, un projet de type Android possède une structure bien définie. Grâce à cette hiérarchie, les ressources et les codes sources sont facilement repérables, et le projet devient plus lisible puisque bien organisé.

L'arborescence automatiquement générée par NetBeans se présente sous la forme suivante (figure 13-3, repères ❶ à ❹) :

❶ Le répertoire `Source Packages` contient toutes les classes Java nécessaires au bon fonctionnement de l'application. La classe principale, créée en même temps que le projet, est visible en double cliquant sur la ressource `Main.java` située dans l'arborescence du projet. Nous examinerons ce fichier plus en détail à la section « Les classes Java ».

❷ Le répertoire `Generated Source Packages` est généré par NetBeans à chaque fois que vous lancez une compilation. Les fichiers, contenus dans ce répertoire, ne doivent pas être modifiés autrement que par NetBeans.

❸ Le répertoire `Resources` contient des sous-répertoires dans lesquels sont enregistrées toutes les ressources utiles à l'application comme des images (*drawable*) ou des fichiers descriptifs des composants d'affichage utilisés par l'application (*layout*). Il contient également des valeurs textuelles (*values*). Ces éléments seront détaillés à la section « Les fichiers descriptifs ».

❹ Le répertoire `Important Files`, comme son nom l'indique, contient des fichiers importants dont `AndroidManifet.xml` dans lequel sont définis les activités et services proposés par l'application. Nous examinerons son utilité dans la section « L'application Liste de courses » à la fin de ce chapitre.

Compilation et exécution de projet

Comme nous l'avons constaté précédemment, le projet présente dès sa création une erreur de syntaxe, qui sera corrigée à la première compilation/exécution. L'erreur affichée par NetBeans (figure 13-4) indique que le « package R n'existe pas ».

Sans aucune modification et après compilation, en cliquant sur le petit triangle vert situé au centre de la barre d'outils ou en appuyant sur la touche F6 de votre clavier, l'erreur disparaît. Que se passe-t-il ?

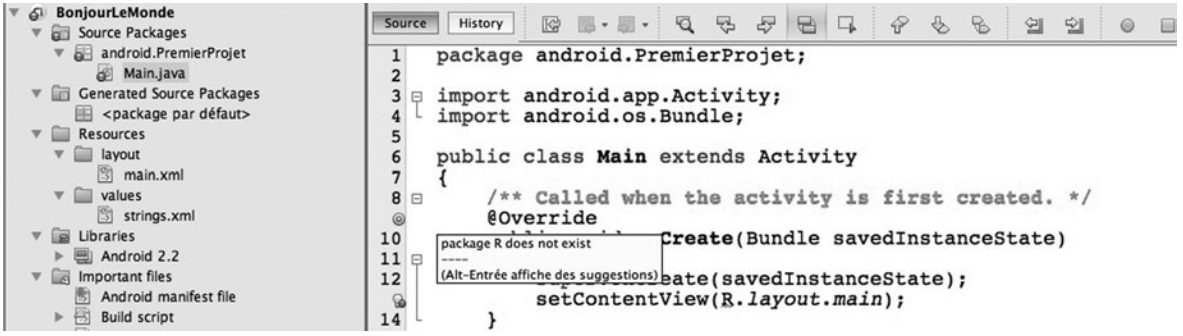


Figure 13-4 *Erreur de syntaxe à la création d'un projet Android.*

Si vous examinez plus attentivement l'arborescence du projet `BonjourLeMonde` (figure 13-5), vous remarquez qu'un nouveau package est apparu, nommé `android.PremierProjet`, dans lequel se trouvent deux fichiers, `BuildConfig.java` et `R.java`.

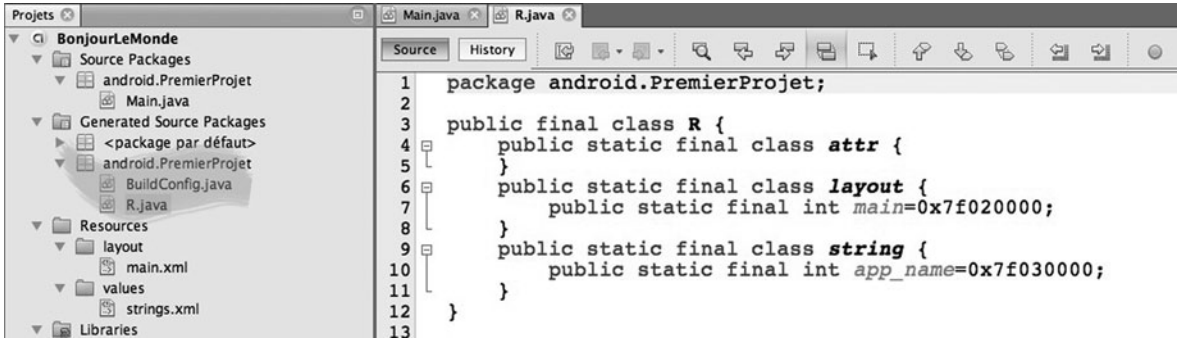


Figure 13-5 *Un nouveau package est généré à la compilation.*

En générant le fichier `R.java` qui définit les classes statiques `layout` et `string`, NetBeans corrige l'erreur détectée à la ligne 13 du fichier `Main.java`. La variable `R.layout.main` existe désormais. Nous examinerons comment se construit cette variable à partir des fichiers ressources dans la section suivante.

Une fois le projet compilé, NetBeans lance l'exécution de l'application en ouvrant le simulateur Android défini lors de la création du projet (voir l'étape 5 de la section « Création d'un projet Android »).

Pour voir l'application s'exécuter, il convient de déverrouiller le téléphone qui se présente en mode Pause au lancement de la simulation (figure 13-6, repère ❶). Une fois ouvert, le téléphone affiche l'application `Main` (figure 13-6, repère ❷).

Figure 13-6 *L'application Main s'affiche une fois le téléphone déverrouillé.*

Remarque Il n'est pas nécessaire de fermer le simulateur Android à chaque nouvelle exécution, car il met du temps à se lancer. Il charge l'application modifiée par vos soins, à chaque fois que vous lancez une compilation.

L'application s'intitule `Main` et affiche le texte `Hello World, Main` sur un écran noir. Ces valeurs ont été créées par défaut lors de la construction du projet. Elles sont modifiables grâce aux fichiers ressources, que nous allons à présent détailler.

Les fichiers ressources

Les fichiers ressources sont utilisés pour stocker les différents médias employés par l'application. Il peut s'agir de photos, de sons ou de vidéos. Mieux encore, il est possible de définir la façon dont vous souhaitez agencer les composants graphiques de votre application en décrivant leur organisation à l'aide d'une structure XML.

XML en quelques mots

L'utilisation d'un fichier au format XML permet de simplifier la façon d'agencer les composants au sein d'une application Android.

En effet, le format XML (*eXtensible Markup Language*) est un langage de description de données. Grâce à sa structure, il permet d'organiser les données en les nommant et en

les ordonnant selon une hiérarchie qui décrit l'agencement des composants au sein de l'application.

La syntaxe du langage XML est assez proche de celle du langage HTML, composée de balises et d'attributs dont le nom reflète le composant graphique utilisé par l'application. Ainsi, les lignes suivantes :

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="OK"

/>
```

ont pour résultat de créer un bouton, grâce à la balise prédéfinie `<Button>`, qui aura une forme particulière décrite par les attributs `android:layout_width` et `android:layout_height`. La valeur `fill_parent` indique au gestionnaire d'affichage que le bouton a la même taille que son parent soit, le plus souvent, celle de l'écran du téléphone. La valeur `wrap_content` indique, quant à elle, que le bouton s'adapte à son contenu.

Ici, l'attribut `layout_width` recevant la valeur `fill_parent`, la largeur du bouton correspond à celle de l'écran du téléphone (figure 13-7, repère ❶). L'attribut `layout_height` ayant pour valeur `wrap_content`, la hauteur du bouton s'adapte à son contenu, c'est-à-dire à la hauteur du texte `OK`, puisque la balise `android:text` prend la valeur `OK`.

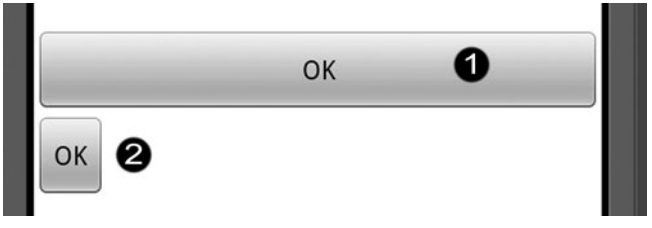


Figure 13-7 *L'attribut layout_width est utilisé pour modifier la largeur d'un bouton.*

En revanche, initialiser l'attribut `android:layout_width` à la valeur `wrap_content` comme suit :

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="OK"
/>
```

a pour conséquence de créer un bouton de largeur plus petite, qui sera ainsi adaptée à la taille de la chaîne de caractères `OK` (figure 13-7, repère ❷).

Les ressources layout

Les ressources relatives à l'agencement des composants graphiques utilisés par votre application sont stockées au sein de fichiers XML, dans le répertoire `layout`.

Remarque

Le nom d'un fichier ressource a pour extension `.xml`. Ce nom est ensuite utilisé par NetBeans pour générer le fichier `R.java` (voir section « Les classe Java » ci-après). Il ne doit comporter ni majuscules, ni espace, sous peine de créer des erreurs de syntaxe et donc de rendre impossible l'exécution de l'application.

Examinons plus attentivement le fichier `main.xml`, construit par défaut par NetBeans à la création du projet :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
    >
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Hello World, Main"
    />
</LinearLayout>
```

Grâce aux informations stockées dans ce fichier, nous observons que l'application construite par défaut utilise un gestionnaire d'affichage de type `LinearLayout` qui contient un composant de type `TextView`.

Un gestionnaire d'affichage (*layout* en anglais) peut être vu comme un conteneur de composants régit par des règles d'affichage qui lui sont spécifiques. Ainsi, lorsque nous utilisons un `LinearLayout`, nous demandons à l'application d'afficher les composants qu'il contient, les uns après les autres, dans l'ordre de définition des balises, soit verticalement, soit horizontalement.

La balise `LinearLayout` est composée ici de quatre attributs :

1. `xmlns:android` (traduire `xmlns` par *XML name space* ou « espace de nom XML ») permet de certifier que tous les noms de balises et leurs attributs utilisés dans ce fichier sont régis par les spécifications Android.
2. `android:orientation` informe que l'application place verticalement tous les composants contenus dans le layout puisque l'attribut est initialisé à `vertical`.

3. `android:layout_width` détermine la largeur du conteneur, ici `fill_parent`, soit la largeur de l'écran du téléphone.
4. `android:layout_height` détermine la hauteur du conteneur, ici `fill_parent`, soit la hauteur de l'écran du téléphone.

`LinearLayout` ne contient ici qu'un seul et unique composant. Il s'agit d'un composant `TextView`, utilisé pour afficher une zone de texte dont la largeur et la hauteur sont définies par les attributs `layout_width` et `layout_height`, respectivement. Le texte affiché est initialisé à la valeur transmise à l'attribut `android:text`, soit ici `Hello World, Main`. Vous pouvez dès à présent modifier le texte. Écrivez, par exemple, `Une toute première application android` (voir figure 13-8) pour voir votre première application afficher un texte plus original.

Remarque

Il existe d'autres types de layouts, comme `RelativeLayout`, qui propose de placer les éléments qu'il contient les uns par rapport aux autres. Ses attributs sont, par exemple, `android:layout_above` ou `android:layout_below`.

Les ressources values

Le répertoire `values` contient des fichiers XML qui décrivent les variables utilisées par l'application. Les variables de type `String` sont définies dans un fichier nommé `strings.xml`, et les tableaux dans un fichier nommé `array.xml`.

Le fichier `strings.xml`, créé par NetBeans, contient les lignes suivantes :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Main</string>
</resources>
```

Il existe par défaut une chaîne de caractères nommée `app_name` qui contient le nom de l'application. Il s'agit du nom affiché (`Main`) en titre de l'application. Pour modifier ce titre (voir figure 13-8), il suffit donc de changer la valeur de `app_name` comme suit :

```
<string name="app_name">BonjourLeMonde</string>
```

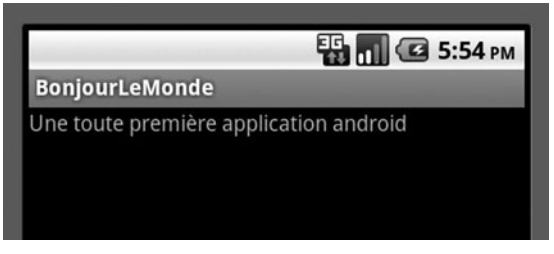


Figure 13-8 *Le texte et le titre de l'application sont modifiables grâce aux fichiers ressources.*

Il est possible de déclarer d'autres chaînes de caractères dans le fichier `strings.xml`. Pour cela, il suffit d'écrire par exemple :

```
<resources>
    <string name="app_name">BonjourLeMonde</string>
    <string name="uneCouleur">rouge</string>
</resources>
```

La variable ressource `uneCouleur` ainsi définie est de type `String` et contient la chaîne de caractères `rouge`.

Les ressources drawable

Les ressources stockées dans le répertoire `drawable` correspondent à des photos et des graphiques utilisés par votre application. Ce répertoire n'est pas créé par NetBeans lors de la mise en place de l'arborescence d'un projet de type Android.

Pour insérer le répertoire dans l'arborescence de votre projet, vous devez :

1. Cliquer droit sur le répertoire Resources du panneau Projets (figure 13-9, repère ❶) et sélectionner Nouveau>Dossier.
2. Dans la nouvelle boîte de dialogue qui apparaît, nommée Nouveau Dossier (figure 13-9, repère ❷), saisir le nom du dossier (ici `drawable`), puis cliquer sur Terminer.

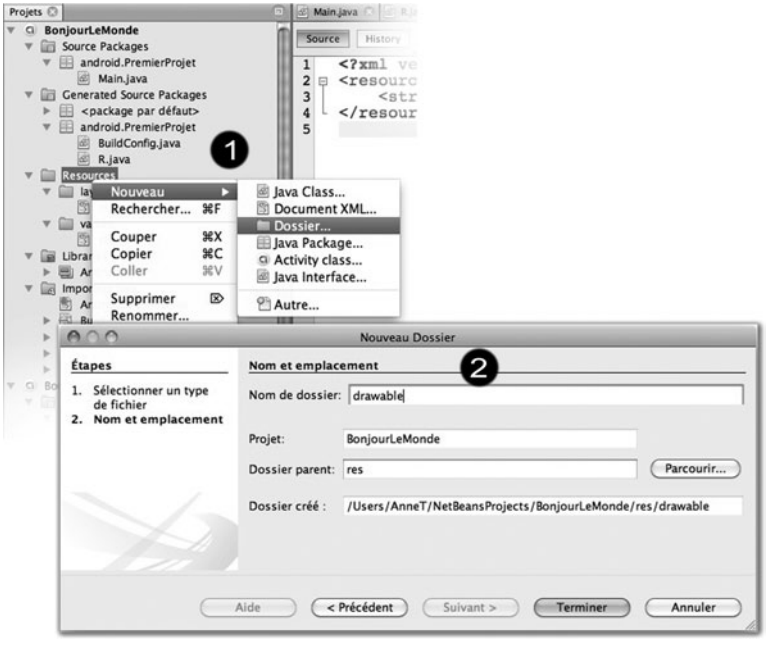
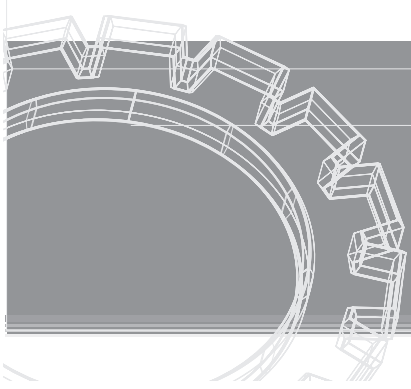


Figure 13-9 *Insérer un répertoire dans l'arborescence du projet BonjourLeMonde*

Chapitre 8

Les principes du concept 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.

Remarque

Lorsque l'interpréteur rencontre le mot-clé `static` devant une variable ou une méthode, il réserve un seul et unique emplacement mémoire pour y stocker la valeur ou le pseudo-code associé. 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 permet 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 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.

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

  public void créer()  {
   Scanner lectureClavier = new Scanner(System.in);
   System.out.print(" Position en x :  ");
   x = lectureClavier.nextInt();
   System.out.print(" Position en y :  ");
   y = lectureClavier.nextInt();
   System.out.print(" Rayon           :  ");
   r = lectureClavier.nextInt();
   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 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.

Pour en savoir plus Voir, au chapitre 6, « Fonctions, notions avancées », la section « Variable de 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 cercles : " + Cercle.nombre);

  Cercle B = new Cercle();
  B.créer();
  System.out.println("Nombre de cercles : " + 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 = lectureClavier.nextInt()`, …). 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).

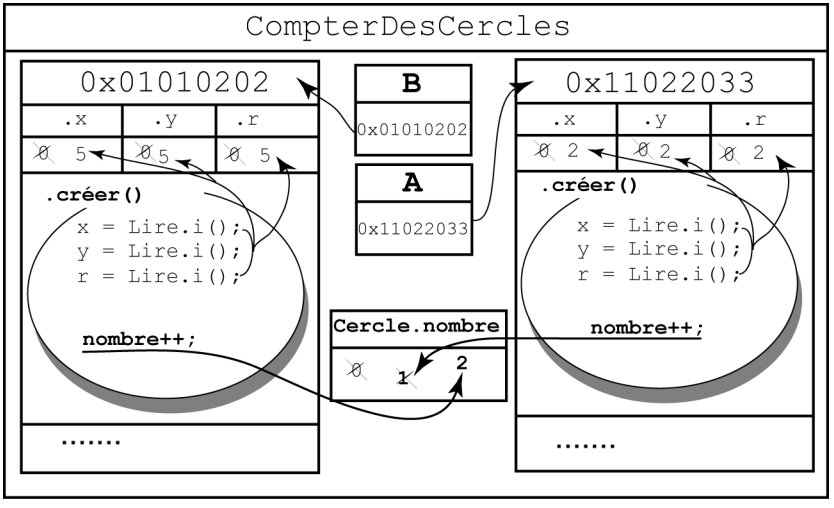


Figure 8-1 *La variable de classe Cercle.nombre est créée en mémoire, avec l'objet A. Grâce au mot-clé static, il y a, non pas réservation d'un nouvel espace mémoire (pour la variable nombre) lors de la création de l'objet B, mais préservation de l'espace mémoire ainsi que de la valeur stockée.*

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 permet 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, en l'appliquant à la coordonnée `x` puis à la coordonnée `y` des centres des deux cercles à échanger.

Pour en savoir plus Les mécanismes d'échange de valeurs sont définis au chapitre 1, « Stocker une information ».

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

Pour en savoir plus La classe `Cercle` est définie 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-code de l'objet `B` qui est interprété, comme illustré à la figure 8-2.

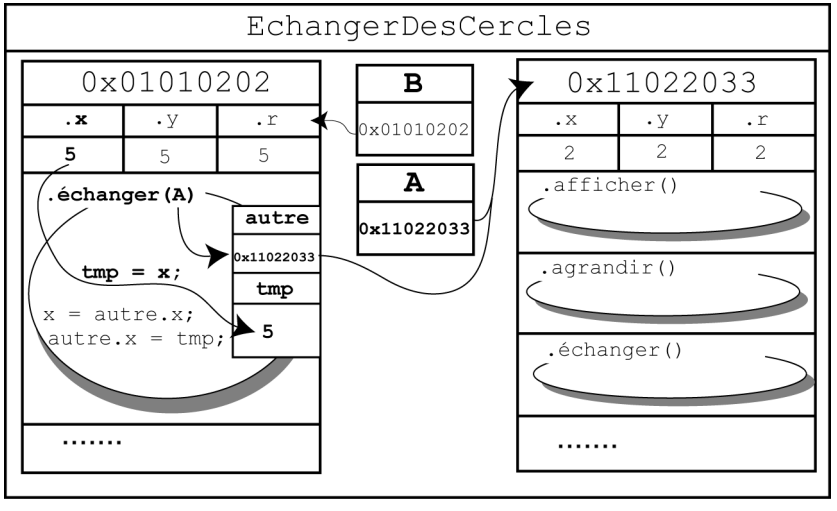


Figure 8-2 *L'instruction B.échanger(A) fait appel à la méthode échanger() de l'objet B. Les données x, y et r utilisées par cette méthode sont celles de l'objet B.*

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`.

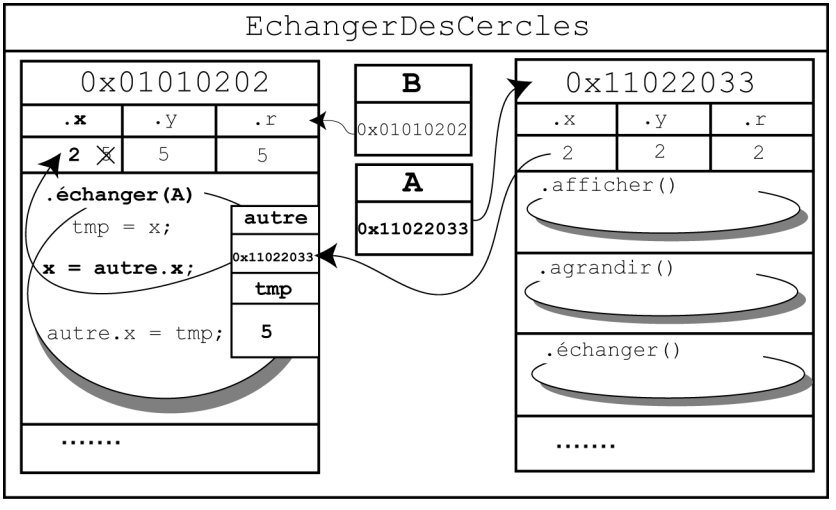


Figure 8-3 *L'objet autre est le paramètre formel de la méthode échanger(). En écrivant B.échanger(A), l'objet autre stocke la référence mémorisée en A. De cette façon, autre.x représente également A.x. La variable x de l'instance B prend la valeur de A.x grâce à l'instruction x = autre.x.*

- 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 83). La variable d'instance `A.x` contenant la valeur `2`, `x` (`B.x`) prend la valeur `2`.
- Pour finir l'échange sur les abscisses, la donnée `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`.

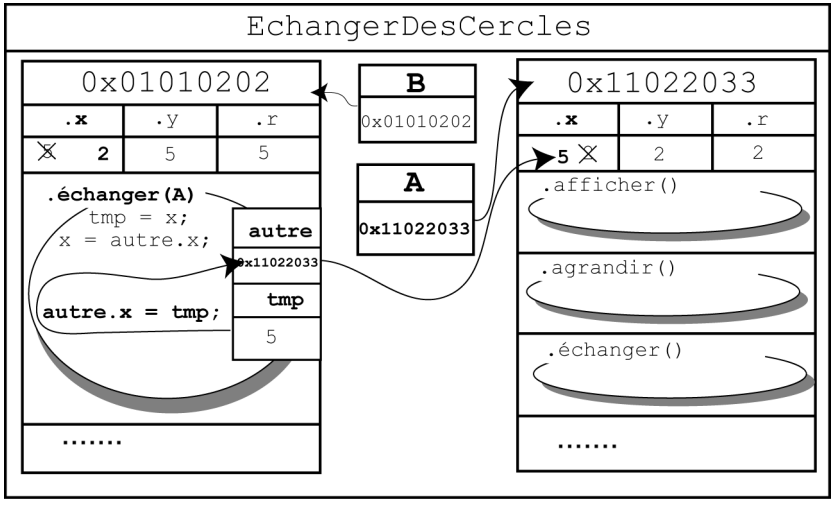


Figure 8-4 *autre et A définissent la même référence, ou adresse. C'est pourquoi le fait de modifier autre.x revient aussi à modifier A.x. Ainsi, l'instruction autre.x = tmp fait que A.x prend la valeur stockée dans tmp.*

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 :
Centre 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`.

Remarque

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**.

Remarque

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 de 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`.

```
import java.util.*;
public class FaireDesCerclesPrives
{
 public static void main(String [] arg)
 {
  Scanner lectureClavier = new Scanner(System.in);
  CerclePrive A  = new CerclePrive();
  A.afficher();
  System.out.println(" Entrez le rayon : ");
  A.r = lectureClavier.nextInt();
  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 = lectureClavier.next-Int();`, 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 r in class CerclePrive not accessible from class FaireDesCerclesPrives`.

Question

Que se passe-t-il si l'on place le terme `private` devant la méthode `afficher()` ?

Réponse

Lors de la compilation du fichier `FaireDesCerclesPrivés`, le message d'erreur `afficher() has private access in CerclePrive` s'affiche.

En effet, si la méthode `afficher()` est définie en `private`, elle n'est plus accessible depuis l'extérieur de la classe `CerclePrivé`. Il n'est donc pas possible de l'appeler depuis la fonction `main()` définie dans la classe `FaireDesCerclesPrivés`.

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.

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.

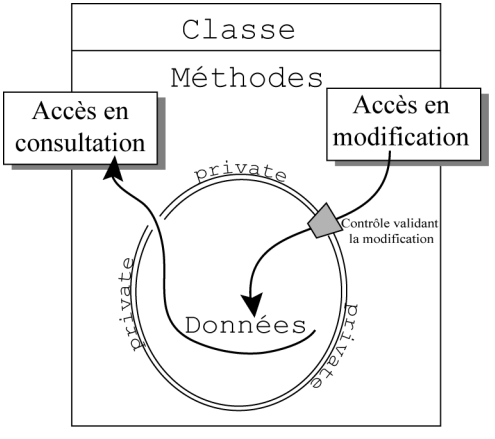


Figure 8-5 *Lorsque les données d'un objet sont protégées, l'objet possède ses propres méthodes, qui permettent soit de consulter la valeur réelle de ses données, soit de modifier les données. La validité de ces modifications est contrôlée par les méthodes définies dans la classe.*

Remarque

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 v aleur 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 peut 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 observé (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 :

```
import java.util.*;
public class CercleControle  {
  private int x, y, r ; // position du centre et rayon
  public void créer()    {
   Scanner lectureClavier = new Scanner(System.in);
   System.out.print(" Position en x :  ");
    x = lectureClavier.nextInt();
   System.out.print(" Position en y :  ");
    y = lectureClavier.nextInt();
   do  {
     System.out.print(" Rayon            :  ");
  r = lectureClavier.nextInt();   } 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 :

```
import java.util.*;
public class FaireDesCerclesControles {
 public static void main(String [] arg) {
  Scanner lectureClavier = new Scanner(System.in);
  CercleControle A  = new CercleControle();
  A.créer();
  A.afficher();
  System.out.print("Entrer une valeur d'agrandissement :");
  int plus = lectureClavier.nextInt();
  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 en 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
```

Convention de nommage

En programmation objet, les conventions stipulent que le nom des méthodes d'accès doit être donné en suivant une règle particulière :

- Les méthodes d'accès en lecture (consultation) doivent commencer par `get`.
- Les méthodes d'accès en écriture (modification) doivent commencer par `set`.
- Derrière les termes `set` ou `get`, suit obligatoirement le nom de la propriété dont la première lettre est en majuscule.

Par exemple la méthode qui fournit la valeur du rayon s'écrit, dans la classe `Cercle` :

```
// Méthode d'accès en lecture
public int getRayon(){
    return rayon;
  }
```

La méthode qui autorise la modification du périmètre s'écrit :

```
// Méthode d'accès en écriture
public void setRayon(int r){
    rayon = r;
  }
```

L'accès aux données d'un cercle dans l'application `FaireDesCerclesControles` s'écrit alors :

```
// Changer la valeur du rayon
A.setRayon(10);
// Afficher (consulter) la nouvelle valeur du rayon
System.out.println("Apres modification : " + A.getRayon());
```

Utiliser cette convention de nommage simplifie la lecture du code. En effet, en un seul coup d'œil, nous sommes en mesure de savoir quelle propriété de l'objet est consultée (`get`) ou modifiée (`set`), par simple lecture du nom de la propriété, dans le nom de la fonction.

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()  {
   Scanner lectureClavier = new Scanner(System.in);
   System.out.print(" Position en x :  ");
   x = lectureClavier.nextInt();
```

```
   System.out.print(" Position en y :  ");
   y = lectureClavier.nextInt();
   do  {
     System.out.print(" Rayon            :  ");
     r = lectureClavier.nextInt();
   } 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 = lectureClavier.nextInt();
  } 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;
}
```

Remarque

Les méthodes `rayonVérifié()` sont appelées **méthodes d'implémentation** ou encore méthodes « métier », 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 »).

Notons 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.

Remarque

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()  {
   Scanner lectureClavier = new Scanner(System.in);
   System.out.print(" Position en x :  ");
   x = lectureClavier.nextInt();
   System.out.print(" Position en y :  ");
   y = lectureClavier.nextInt();
   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.

Question

Que se passe-t-il si l'on écrit l'en-tête du constructeur comme suit :

```
public void Cercle()
```

ou encore :

```
public int Cercle()
```

Réponse

Le fait de placer un type (`int`, `void`, ...) dans l'en-tête du constructeur a pour conséquence de créer une méthode, qui a pour nom `Cercle()`. Il s'agit bien d'une méthode et non d'un constructeur. Elle n'est donc pas appelée par l'opérateur `new`.

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.

Remarques

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 constr ucteur 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 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`.

Signalons 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, notons 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 :

```
import java.util.*;
public class Forme    {
 protected int x, y ;
 private couleur ;

 public  Forme() {      // Le constructeur de la classe Forme
   Scanner lectureClavier = new Scanner(System.in);
   System.out.print(" Position en x :  ");
   x = lectureClavier.nextInt();
   System.out.print(" Position en y :  ");
   y = lectureClavier.nextInt();
   System.out.print(" Couleur de la forme :  ");
   couleur = lectureClavier.nextInt();
 }

 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
  x = nx;                    // de la 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 observons 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`.

Remarque Par défaut, s'il n'y a pas d'appel explicite au constructeur de la classe supérieure, comme c'est le cas dans notre e xemple, le compilateur recherche de lui-même le constr ucteur par déf aut (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.

Question Que se passe-t-il si nous remplaçons 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;
}
```

Réponse 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`.

Lorsqu'il n'existe pas de constructeur par défaut dans la classe supérieure, l'interpréteur ne peut plus l'exécuter au moment de la construction de l'objet de la classe dérivée. Il est nécessaire de faire appel à l'outil `super()`.

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 :

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

De cette façon, le terme `super()` représentant le constructeur de la classe supérieure possédant 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`.

Remarque

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.

Remarque

Lorsqu'une donnée de la classe supérieure est déclarée en mode `private`, la classe dérivée ne peut ni consulter ni modifier directement cette donnée hér itée. 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
   super(xx, yy);                   // Cercle
   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, 10
Couleur : 3
```

Remarque

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`.

Les interfaces

Nous l'avons vu à la section « Les objets contrôlent leur fonctionnement » de ce chapitre, les objets de la vie réelle sont manipulés par une interface appropriée et les objets informatiques proposent également une interface qui nous permet de communiquer avec eux.

Qu'est-ce qu'une interface ?

En pratique, une interface correspond par exemple à une fenêtre de contrôle ou un panneau précisant les informations en cours de traitement. L'interface de communication se doit d'être suffisamment générale pour être utilisable dans le plus grand nombre de cas, tout en proposant un modèle d'utilisation approprié à sa fonction.

Une interface correspond donc à une classe qui définit non pas un modèle d'objet (une sorte de moule) mais un ensemble de comportements possibles, sans que ces comportements soient réellement décrits.

Plus précisément, lorsqu'un utilisateur clique par exemple sur le bouton « Valider » d'une application, une action doit être menée. Cette action diffère selon l'application. Il peut s'agir de confirmer l'envoi d'un message électronique ou encore de supprimer un fichier.

La classe modélisant le bouton de validation doit donc proposer une méthode nommée par exemple `actionAmener()` (en anglais `actionPerformed()`) sans décrire explicitement le code de cette action. Seul l'utilisateur (c'est-à-dire le programmeur d'applications) est à même de décrire l'action à réaliser.

Remarque

Le traitement des événements et la description des actions associées sont étudiés plus précisément au chapitre 11 « Dessiner des objets », section « Exemple : associer un bouton à une action ».

Cette classe de modélisation des comportements correspond en pratique à une interface.

Syntaxe d'une interface

Une interface, dans le langage Java, définit les noms des méthodes associées aux comportements. Elle ressemble à une classe puisqu'elle s'écrit comme suit :

```
interface uneInterface {
   public type methode1() ;
   public type methode2() ;
   // d'autres en-têtes de méthodes
}
```

Les règles d'écriture d'une interface sont simples :

- une interface est définie au sein d'un fichier qui porte son nom suivi de l'extension `.java` ;
- le terme `interface` remplace le terme `class` ;
- les comportements proposés par l'interface sont définis à partir des en-têtes de méthodes (signature).

Principe de fonctionnements

Une fois l'interface définie, les méthodes sont concrètement décrites au sein des différentes classes qui implémentent l'interface. Pour cela, vous devez :

- placer le terme `implements` lors de la création de la classe ;
- décrire ce que réalise les méthodes, comme suit :

```
public class UneClasse implements uneInterface {
   public type methode1() {
   // Ici sont décrites les actions à mener pour cette méthode au sein de cette classe
   }
   public type methode2() {
   // Ici sont décrites les actions à mener pour cette méthode au sein de cette classe
   }
}
```

Une seconde classe peut également implémenter la même interface et dans ce cas, le code s'écrit :

```
public class UneAutreClasse implements uneInterface {
   public type methode1() {
   // Ici sont décrites les actions à mener pour cette méthode au sein de cette classe
   }
   public type methode2() {
   // Ici sont décrites les actions à mener pour cette méthode au sein de cette classe
   }
}
```

Les méthodes `methode1()` et `methode2()` ne se comportent pas de la même façon selon la classe dans laquelle elles sont définies. Elles contiennent des instructions différentes d'une classe à l'autre.

Les objets implémentant l'interface `uneInterface` sont ensuite créés dans l'application de la façon suivante :

```
UneClasse premier = new UneClasse();
UneAutreClasse second = new UneAutreClasse();
// appeler la methode1() de UneClasse
```

```
premier.methode1();
// appeler la methode1() de UneAutreClasse
second.methode1();
```

Remarque Une interface modélise des compor tements, et non des objets . Il est donc impossib le de créer une instance d'interface à l'aide de l'opérateur `new`.

Calculs géométriques

L'objectif de cet exemple est de construire une interface qui permette de calculer n'importe quels surface et périmètre d'une forme géométrique. Pour cela, nous allons utiliser les notions d'héritage et d'interface étudiées au cours des deux sections précédentes.

Cahier des charges

L'application principale crée autant de formes qu'elle le souhaite (cercle ou rectangle) en utilisant les classes `Forme`, `Cercle` et `Rectangle`.

Pour en savoir plus La classe `Rectangle` est réalisée en exercice, à la fin de ce chapitre.

La classe `Forme` implémente l'interface `CalculGeometrique` qui définit deux méthodes `surface()` et `perimetre()`.

Le calcul de la surface d'une forme au sein de la classe `Forme` n'est pas possible, car la forme n'est pas encore réellement définie. Les méthodes `surface()` et `perimetre()` au sein de cette classe retournent une valeur négative.

Les classes `Cercle` et `Rectangle` héritent toutes deux de la classe `Forme`, elles implémentent par conséquent l'interface `CalculGeometrique`. Les méthodes `surface()` et `perimetre()` au sein de ces différentes classes retournent la valeur du périmètre et de la surface calculée à partir de la formule correspondant à la forme géométrique associée à la classe.

Exemple : code source

L'interface `CalculGeometrique` définit les méthodes `surface()` et `perimetre()` comme suit :

```
interface CalculGeometrique {
     public double surface();
     public double perimetre();
}
```

La classe `Forme` implémente l'interface `CalculGeometrique` et décrit explicitement les méthodes `surface()` et `perimetre()`. Ces dernières retournent la valeur `-1`.

```
public class Forme implements CalculGeometrique {
   protected int x, y, couleur ;
   // Définition du constructeur et de la méthode afficher()
   // Description des méthodes  surface() et perimetre() pour la classe Forme
   public double surface() {
          return -1 ;
   }
   public double perimetre() {
          return -1 ;
   }
```

La classe `Cercle` hérite de la classe `Forme`. Par héritage, elle implémente l'interface `CalculGeometrique`. Les méthodes `perimetre()` et `surface()` associées à la classe `Cercle` retournent respectivement la valeur correspondant au calcul mathématique du périmètre d'un cercle et de sa surface.

```
public class Cercle extends Forme{
   private int r ;
   public double surface() (
     return Math.PI *r*r ;
   }
   public double perimetre() (
      return 2*Math.PI*r ;
   }
}
```

L'application finale crée des objets de type `Cercle` ou `Forme` comme suit :

```
Cercle A  = new Cercle(5, 5);
A.afficher();
if ( A.perimetre() >=0) {
  System.out.println("Le périmètre de A vaut " + A.perimetre());
} else {
  System.out.println("Calcul impossible");
}
Forme F = new Forme (10, 10);
F.afficher();
if ( F.perimetre() >=0) {
  System.out.println("Le périmètre de F vaut " + F.perimetre());
} else {
  System.out.println("Calcul impossible");
}
```

Le périmètre d'un cercle ou d'une forme est calculé en utilisant la méthode correspondant au type de l'objet utilisé.

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 cette 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 terme `implements` est utilisé pour créer des interfaces. Une interface définit tous les types de comportements d'un objet sans décrire explicitement le code.

Exercices

La protection des données

Les méthodes d'accès en écriture

Exercice 8.1 Reprendre l'application `Bibliotheque` et la classe `Livre` développées au cours de l'exercice 7.4 du chapitre précédent et, modifier les propriétés de la classe, de façon à les rendre privées. Que se passe-t-il lors de la compilation de l'application `Bibliotheque` ? Pourquoi ?

Exercice 8.2 Pour corriger l'erreur de compilation, vous devez mettre en place des méthodes d'accès en écriture afin de permettre la modification des propriétés depuis l'extérieur de la classe `Livre`.

a. En supposant que la méthode `setTitre()` est appelée depuis la fonction `main()` comme suit :

```
System.out.print("Entrez le titre : ");
livrePoche.setTitre(lectureClavier.next());
```

insérer à l'intérieur de la classe `Livre` la méthode `setTitre()` afin de pouvoir modifier la propriété `titre`.

b. En vous inspirant de la méthode `setTitre()`, créez les méthodes autorisant la modification des autres propriétés de la classe indépendamment les unes des autres.

c. Modifier l'application `Bibliotheque` en tenant compte des nouvelles méthodes d'accès en écriture.

d. Est-il nécessaire de créer une méthode `setCode()` ? Pourquoi ?

Les méthodes d'accès en lecture

Exercice 8.3 a. Pour faire en sorte que l'application `Bibliotheque` puisse afficher les propriétés de la classe `Livre` indépendamment les unes des autres, insérer à l'intérieur de la classe `Livre` les méthodes `getTitre()`, `getNomAuteur()`, `getPrenomAuteur()`, `getCategorie()`, `getIsbn()` et `getCode()`. Ces méthodes retournent au programme appelant la propriété indiquée par le nom de la méthode.

b. Modifier l'application `Bibliotheque` afin de n'afficher que le titre et le code du livre `livrePoche`.

Les méthodes invisibles (métier)

Exercice 8.4 Pour répondre à la question 8.2.d, renommez la méthode `calculerLeCode()` par `setCode()` et faites en sorte que cette méthode ne soit pas accessible par aucune autre classe que la classe `Livre`.

Les constructeurs

Exercice 8.5 a. Le constructeur par défaut de la classe `Livre` permet de saisir les données d'un livre . Écrire le constructeur en utilisant les méthodes d'accès en écriture réalisées en 8.2.

Remarque L'utilisation des méthodes d'accès en écriture au sein du constructeur `Livre` n'est pas réellement obligatoire. Mais plus généralement, cela peut être utile pour s'assurer de la validité des données (voir exercice 8.6, ci-après).

b. Sans la modifier, exécutez l'application `Bibliotheque`. Que se passe-t-il ? Pourquoi ? Transformez l'application afin d'éviter ce problème.

c. Surchargez le constructeur par défaut, en définissant un nouveau constructeur qui initialise les propriétés d'un livre à partir des valeurs qui lui sont fournis en paramètre.

d. Dans l'application `Bibliotheque`, créez un objet `unPolar` initialisé dès la création aux données suivantes : `"Le mystère de la chambre jaune"`, `"Leroux"`, `"Gaston"`, `"Policier"` et `"2253005495"`. Affichez le contenu de l'objet `unPolar`.

L'héritage

En examinant la figure 8.6, et en vous aidant des notions acquises au cours des exercices précédents et des exercices réalisés au chapitre 7, nous allons créer les classes `Cercle`, `Rectangle` et `Triangle` à partir de la classe `Forme`.

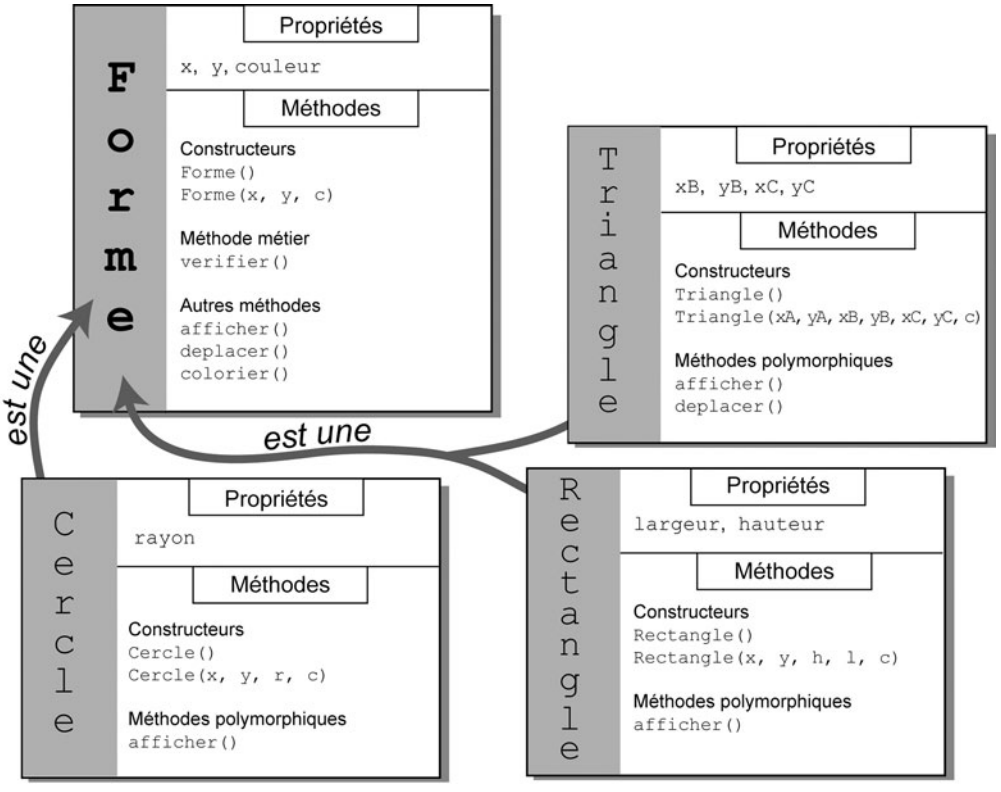


Figure 8-6 *Le cercle, le rectangle et le triangle sont des formes. Les classes qui les définissent héritent de la classe mère Forme.*

La classe Forme

Exercice 8.6 Sachant que toute forme géométrique est définie par :

- Une couleur.
- Une position en X et en Y définissant les coordonnées du point de référence pour placer la forme à l'écran.

Et que :

- La couleur varie entre `0` et `10`.
- La propriété `X` est comprise entre `0` et `800`.
- La propriété `Y` est comprise entre `0` et `600`.

a. Définir les propriétés de la classe `Forme` en mode `protected`.

b. Définir des constantes pour la largeur (`800`) et la hauteur (`600`) de la fenêtre d'affichage ainsi que pour le nombre de couleurs maximum proposé (`10`).

c. Reprendre la méthode `verifier()` de l'exercice 7.7 du chapitre précédent et définissez la comme une méthode métier (invisible).

La méthode vérifie la validité des valeurs pour toutes les propriétés de la classe (`couleur`, `x`…), modifiez la méthode de façon à passer en paramètre un message indiquant à quelle propriété sera attribuée la saisie. L'appel à la méthode pourra s'effectuer de la façon suivante :

```
couleur = verifier("couleur", 0, couleurMax);
```

ou encore

```
largeur = verifier("Largeur", 0, largeurEcran);
```

Surcharger la méthode `verifier`() en créant une méthode vérifiant une valeur passée en paramètre.

d. Écrire un constructeur :

- par défaut qui permet de saisir les données d'une forme. Les données saisies doivent être vérifiées en utilisant la première forme de la méthode `verifier`() ;
- muni des trois paramètres permettant d'initialiser directement, les propriétés de la classe `Forme`. Les données passées en paramètre doivent être vérifiées en utilisant la seconde forme de la méthode `verifier`().

e. Écrire la méthode `deplacer()` qui déplace une forme à partir des valeurs passées en paramètres. Par exemple si le point de référence de la forme est positionnée en `100`, `100`, la méthode `deplacer(10, 10)` a pour résultat de placer le point de référence de la forme en `110`, `110`. Les nouvelles coordonnées de la forme doivent être vérifiées.

f. Écrire la méthode `colorier()` qui change la couleur d'une forme en fonction de la valeur passée en paramètre. La valeur de la nouvelle couleur doit être vérifiée.

g. Écrire la méthode `afficher()` qui affiche les propriétés de la classe `Forme`.

La classe Rectangle

Exercice 8.7 Sachant que tout rectangle est une forme géométrique possédant une hauteur dont la valeur est comprise entre `0` et `600`, et une largeur dont la valeur est comprise entre `0` et `800` :

a. Définir la classe `Rectangle` à partir de la classe `Forme`.

b. Définir les propriétés de la classe `Rectangle` en mode privée.

c. Écrire un constructeur :

- par défaut qui permet de saisir la hauteur et la largeur d'un rectangle. Ces valeurs doivent être vérifiées ;
- muni de cinq paramètres permettant d'initialiser directement l'ensemble des propriétés `x`, `y`, `couleur`, `largeur` et `hauteur` de la classe `Rectangle`. Ce constructeur fait appel au constructeur avec paramètre, de la classe `Forme`. Les données passées en paramètres doivent être vérifiées.

d. Écrire la méthode `afficher()` qui affiche les propriétés de la classe `Rectangle` ainsi que celles de la classe `Forme`.

e. Écrire les méthodes `perimetre()` et `surface()` qui calculent le périmètre et la surface d'un rectangle.

La classe Triangle

Exercice 8.8 Sachant que tout triangle est une forme géométrique possédant trois sommets dont les valeurs en `X` sont comprises entre `0` et `800` et en en `Y` sont comprises entre `0` et `600` :

a. Définir la classe `Triangle` à partir de la classe `Forme`.

b. Définir les propriétés de la classe `Triangle` en mode privée. Les coordonnées `X` et `Y` de la classe `Forme` – point de référence du triangle – correspondent aux coordonnées du premier sommet.

c. Écrire un constructeur :

- par défaut qui permet de saisir des deux sommets restants du triangle. Ces valeurs doivent être vérifiées.
- muni de sept paramètres permettant d'initialiser directement l'ensemble des propriétés `x`, `y`, `couleur`, `x1`, `y1`, `x2` et `y2` de la classe `Triangle`. Ce constructeur fait appel au constructeur avec paramètre de la classe `Forme`. Les données passées en paramètres doivent être vérifiées.

d. Écrire la méthode `afficher()` qui affiche les propriétés de la classe.

L'application FaireDesFormesGeometriques

Exercice 8.9

a. Écrire une application qui permet la création d'un cercle, d'un rectangle, et d'un triangle.

b. Vérifier que les valeurs des propriétés de chaque classe ne peuvent être saisies en dehors des limites imposées.

c. Afficher les valeurs de chacune des formes.

d. Déplacer toutes les formes de `10` pixels en `X` et `20` en `Y`. Que se passe-t-il pour le triangle ? Pourquoi ? Comment faire pour que le triangle se déplace correctement ?

e. Dans la classe `Triangle`, écrire la méthode `deplacer()` afin de déplacer tous les sommets du triangle à partir des valeurs passées en paramètre. La méthode fait appel à la méthode `deplacer()` de la classe supérieure afin de déplacer le premier sommet du triangle. Elle vérifie également que les nouveaux sommets ne sortent pas de la fenêtre.

f. Afficher les périmètres et les surfaces de tous les rectangles et les cercles créés au cours de l'application.

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";
 Scanner lectureClavier = new Scanner(System.in);
 do {
  System.out.print("Type du compte [Types possibles : C(ourant),
                   J(oint), E(pargne)] : ");
  tmpc = lectureClavier.next().charAt(0);
 } 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 en savoir plus Pour contrôler la validité de la date, voir la section « Le projet : Gestion d'un compte bancaire » du chapitre 10, « Collectionner un nombre indéterminé d'objets ».

c. Modifiez les méthodes `créerCpte()` et `créerLigneComptable()` de façon à ce 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 paramètres.

   – 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 à ce 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, suivies 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.