Programmation Objet

1ère année DUT Informatique

AUTEUR : B. Girau

UNIVERSITÉ NANCY 2
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE
2 ter boulevard Charlemagne
CS 5227
54052 • NANCY cedex

Tél : 03.83.91.31.31
Fax : 03.83.28.13.33
http://www.iuta.univ-nancy2.fr

Table des matières

9 Compléments sur les classes

9.1 Introduction à Javadoc

Le langage Java dispose d'un mécanisme automatisant la documentation technique de ses sources, appelé *javadoc*. Lors de la définition d'une classe, il suffit d'inclure des commentaires commençant par `/**` pour que javadoc en déduise qu'il s'agit de commentaires qu'il doit prendre en compte. Il attribue ces commentaires à la déclaration qui suit immédiatement. De plus, certaines balises (`@param @return` ...) permettent de commenter précisément le rôle de certains éléments de la déclaration qui suit. A partir de ces commentaires, javadoc édite des pages `html` avec le même aspect que l'API officielle. Attention : seules les déclarations d'accès `public` ou `protected` sont documentées par javadoc.

Exemple : la prétraitement par javadoc de la classe suivante

```
1  /**
2   * Cette classe représente les rectangles en coordonnées entières (pixels)
3   * @author Bernard Girau
4   */
5  public class UnRectangle {
6      /**
7       * le coin inférieur gauche du rectangle
8       */
9      private UnPoint coin;
10
11     /**
12      * la largeur (axe des abscisses) du rectangle
13      */
14     private int largeur;
15
16     /**
17      * la hauteur (axe des ordonnées) du rectangle
18      */
19     private int hauteur;
20
21     /**
22      * crée un rectangle à l'origine de taille unitaire
23      */
24     public UnRectangle() {
25         coin=new UnPoint();
26         largeur=1;
27         hauteur=1;
28     }
29
30     /**
31      * calcule le périmètre du rectangle
32      * @return le périmètre
33      */
34     public int perimetre() {
35         return 2*(hauteur+largeur);
36     }
37     /**
38      * translate le rectangle
39      * @param dx nombre de pixels de translation en abscisse
40      * @param dy nombre de pixels de translation en ordonnée
41      */
```

```
42      public void translation(int dx,int dy) {
43          coin.translation(dx,dy);
44      }
45  }
```

produit la page suivante

Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Class UnRectangle

```
java.lang.Object
  └UnRectangle
```

public class **UnRectangle**
extends java.lang.Object

Cette classe représente les rectangles en coordonnées entières (pixels)

Constructor Summary

`UnRectangle()`	crée un rectangle à l'origine de taille unitaire

Method Summary

int	`perimetre()` calcule le périmètre du rectangle
void	`translation(int dx, int dy)` translate le rectangle

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

UnRectangle

```
public UnRectangle()
```

crée un rectangle à l'origine de taille unitaire

Method Detail

perimetre

```
public int perimetre()
```

calcule le périmètre du rectangle

Returns:
le périmètre

translation

```
public void translation(int dx,
                        int dy)
```

translate le rectangle

Parameters:
`dx` - nombre de pixels de translation en abscisse
`dy` - nombre de pixels de translation en ordonnée

Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Pour générer la documentation *javadoc* d'un fichier de nom `NomClasse.java`, il faut lancer la commande suivante :

```
javadoc NomClasse.java
```

La commande `javadoc` dispose d'un grand nombre d'options. Par exemple, pour indiquer l'auteur et la version, on écrira :

```
javadoc -author -version NomClasse.java
```

Si on veut générer la documentation pour un ensemble de classes présentes dans un même répertoire, on peut lancer la commande :

```
javadoc *.java
```

Le fichier `index.html` donne alors accès à la documentation de toutes les classes.

9.2 Affichage

Tout objet instancié peut être affiché via `System.out.println`. Pour adapter l'affichage aux propriétés de l'objet, il suffit de définir la méthode `toString` (en-tête imposé, voir exemple), qui retourne une chaîne de caractère représentative de l'objet.

```
1   class UneClasse {
2     private int val;
3     public UneClasse(int v) {
4       val=v;
5     }
```

```
6     public String toString() {
7       return ("je suis un objet UneClasse, et ma valeur est "+val);
8     }
9   }
10  class EssaiUneClasse {
11    public static void main(String args[]) {
12      UneClasse uc=new UneClasse(7*2);
13      System.out.println(uc);
14      // affiche "je suis un objet UneClasse, et ma valeur est 14"
15    }
16  }
```

10 Propriétés de classe

Les propriétés indiquées par chaque classe (attribut ou méthode) sont en fait réparties entre propriétés de classe et propriétés d'instance. Une propriété d'instance est une propriété dont la valeur/comportement est spécifique à chaque objet créé (instancié) de la classe. Une propriété de classe est en revanche identique pour toutes les instances de la classe. Elle existe (et est accessible) même en l'absence d'instance de cette classe.

10.1 Différences entre propriétés d'instance et de classe (catégorie `static`)

Jusqu'à présent, seuls les attributs et méthodes d'instance ont été utilisés.

- Les attributs d'instance sont propres à chaque objet : dès qu'un objet est créé, une case mémoire est attribuée à chacun de ses attributs d'instance.
  Exemple : si deux objets de classe `UnRectangle` sont créés, il y aura deux cases mémoire `largeur`, chacune attribuée à un seul objet `UnRectangle`.
- Les méthodes d'instance ont un comportement propre à chaque objet : leur comportement dépend de l'état de l'objet qui exécute la méthode, i.e. de la valeur de ses attributs d'instance.
  Exemple : si deux objets de classe `UnRectangle` sont créés, chacun dispose d'un attribut `largeur` et d'un attribut `hauteur`, les valeurs de ces attributs pouvant différer d'un objet à l'autre, le résultat de l'appel à la méthode `perimetre` va donc fournir deux résultats différents suivant l'objet `UnRectangle` qui l'exécute.
- Les attributs de classe ne sont liés à aucun objet. Ils désignent un seul emplacement mémoire par classe. On peut donc les initialiser dès leur déclaration. La réservation de l'emplacement mémoire (et l'initialisation éventuelle) est effectuée une et une seule fois au lancement du programme qui utilise cette classe.
- Les méthodes de classe ne sont liées à aucun objet. Elles peuvent être appelées directement, sans qu'un objet n'ait à être créé. C'est le cas de la méthode principale `main`.
- Les attributs et méthodes de classe sont indiquées par la catégorie `static` lors de leur déclaration.
- Il est possible de fixer une valeur définitive à un attribut de classe, grâce à la catégorie `final`. Il s'agit alors d'une *constante*.

```
1   public class ConstantesPhysiques {
2       public final static double g=9.80665;
3   }
```

- Pour accéder à un attribut de classe `public` depuis une autre classe, on utilise le nom de la classe et non pas une référence sur un objet de cette classe. Idem pour les appels de méthodes.

```
1   public class TestStatic {
2       public static int val_int=10;
3       public static void afficheValInt() {
4           System.out.println(val_int);
5       }
6   }
7   public class ProgrammeTest {
```

```
8       public static void main(String args[]) {
9           int v=TestStatic.val_int;
10          TestStatic.afficheValInt();
11      }
12  }
```

❑ Les méthodes peuvent utiliser les attributs comme des variables, néanmoins *seules les méthodes d'instance d'une classe peuvent utiliser des attributs d'instance de cette classe*.
Exemple : la méthode `perimetre` de la classe `UnRectangle` ne peut pas être déclarée `static`.

❑ Les méthodes peuvent faire appel à d'autres méthodes, néanmoins *seules les méthodes d'instance d'une classe peuvent appeler des méthodes d'instance de cette classe*.
Exemple : le programme suivant ne compile pas

```
1   public class CaMarchePas {
2     private int valeur;
3     public void afficheValeur() {
4       System.out.println(valeur); // ok car valeur et afficheValeur
5                                   // sont des propriétés d'instance
6     }
7     public static void main(String[] args) {
8       valeur=4; // interdit car valeur n'est pas un attribut de classe
9       afficheValeur(); // interdit car afficheValeur n'est pas une méthode de classe
10    }
11  }
```

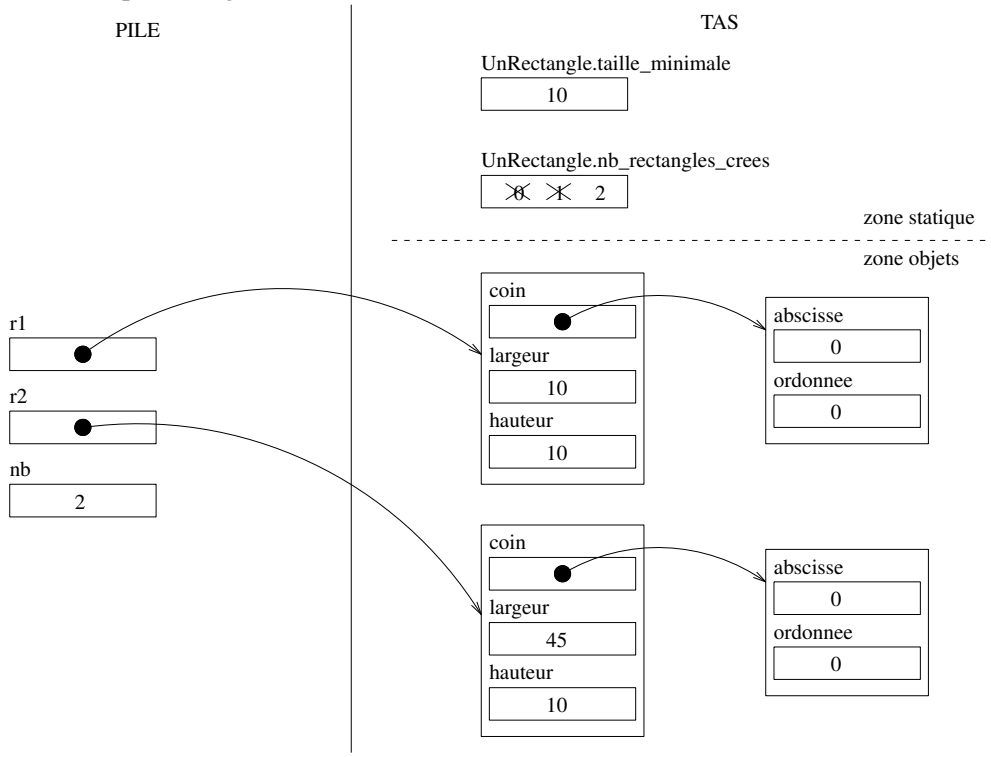
`valeur` est un attribut d'instance, donc il ne désigne un emplacement mémoire que lorsqu'un objet de classe `CaMarchePas` est créé. De plus, il y a un exemplaire de cet attribut pour chaque objet.
Or la méthode `main` est une méthode de classe, qu'on peut donc appeler n'importe quand et qui ne dépend d'aucun objet. Il lui est interdit d'utiliser `valeur`, puisqu'en l'absence de toute indication d'objet, cet attribut n'existe pas.

❑ Exemple d'utilisation :

```
1   public class UnRectangle {
2
3       private UnPoint coin;
4       private int largeur,hauteur;
5       private static int nb_rectangles_crees=0;
6       public final static int taille_minimale=10;
7
8       public UnRectangle() {
9           nb_rectangles_crees=nb_rectangles_crees+1;
10          coin=new UnPoint();
11          largeur=taille_minimale;
12          hauteur=taille_minimale;
13      }
14
15      public UnRectangle(int l,int h) {
16          nb_rectangles_crees=nb_rectangles_crees+1;
17          coin=new UnPoint();
18          if (l<1) largeur=taille_minimale; else largeur=l;
19          if (h<1) hauteur=taille_minimale; else hauteur=h;
20      }
21
22      public static int getNbRectanglesCrees() {
23          return nb_rectangles_crees;
24      }
25  }
26  public class TestRectangles {
```

```
27        public static void main(String[] args) {
28            UnRectangle r1=new UnRectangle();
29            UnRectangle r2=new UnRectangle(45,-15);
30            int nb=UnRectangle.getNbRectanglesCrees();
31        }
32    }
```

Une représentation de l'état de la mémoire après la dernière instruction peut être la suivante (on notera que le tas comporte désormais une zone réservée aux attributs de classe, accessible non pas via une référence, mais simplement grâce au nom de la classe).



10.2 Exemple : méthodes mathématiques prédéfinies

Java met à disposition des programmeurs une classe qui réunit des outils de calcul mathématique. Ces outils existent en dehors de toute instance de cette classe `Math`.

- Les attributs correspondent à des constantes mathématiques (`final`) :
  - `Math.E // 2.71828...`
  - `Math.PI // 3.14159...`

  Il est impossible de modifier ces attributs (i.e. de leur affecter une nouvelle valeur).
- Les méthodes correspondent notamment aux fonctions mathématiques élémentaires, cf tableau 1.

  Remarques :

  `ceil` : plus petit entier supérieur ou égal au paramètre - valeur retournée de type `double` afin de pouvoir gérer les grands nombres.

  `floor` : plus petit entier inférieur ou égal au paramètre.

  `rint` : plus proche entier.

  `random` : pas de paramètre, résultat entre 0 et 1 (type `double`).

10.3 Cas particulier : attributs de classe de type objet

Un attribut de classe d'une classe A est directement accessible, sans qu'il soit nécessaire de construire d'abord un objet de classe A. Mais cet attribut peut lui-même être une référence sur une instance créée, de classe A ou d'une autre classe.

Nom	Paramètre(s)	Résultat	Interprétation
`sin`	double	double	sinus
`cos`	double	double	cosinus
`tan`	double	double	tangente
`asin`	double	double	arcsinus
`acos`	double	double	arccosinus
`atan`	double	double	arctangente
`exp`	double	double	exponentielle
`log`	double	double	logarithme (ln)
`sqrt`	double	double	racine carrée
`pow`	double x,double y	double	x^y
`ceil`	double	double	arrondi supérieur
`floor`	double	double	arrondi inférieur
`rint`	double	double	arrondi
`abs`	int	int	valeur absolue
`abs`	double	double	valeur absolue
`random`		double	résultat aléatoire

TABLE 1 – Méthodes `public static` de la classe `Math`

Il est même possible qu'un attribut de classe de type objet soit déclaré constant : il est alors impossible de modifier la référence qu'il contient dès son initialisation. Mais attention, il faut alors s'assurer que cet objet appartient à une classe d'objets non modifiables (comme `String`, et non pas `StringBuffer` par exemple) si on veut vraiment désigner une constante.

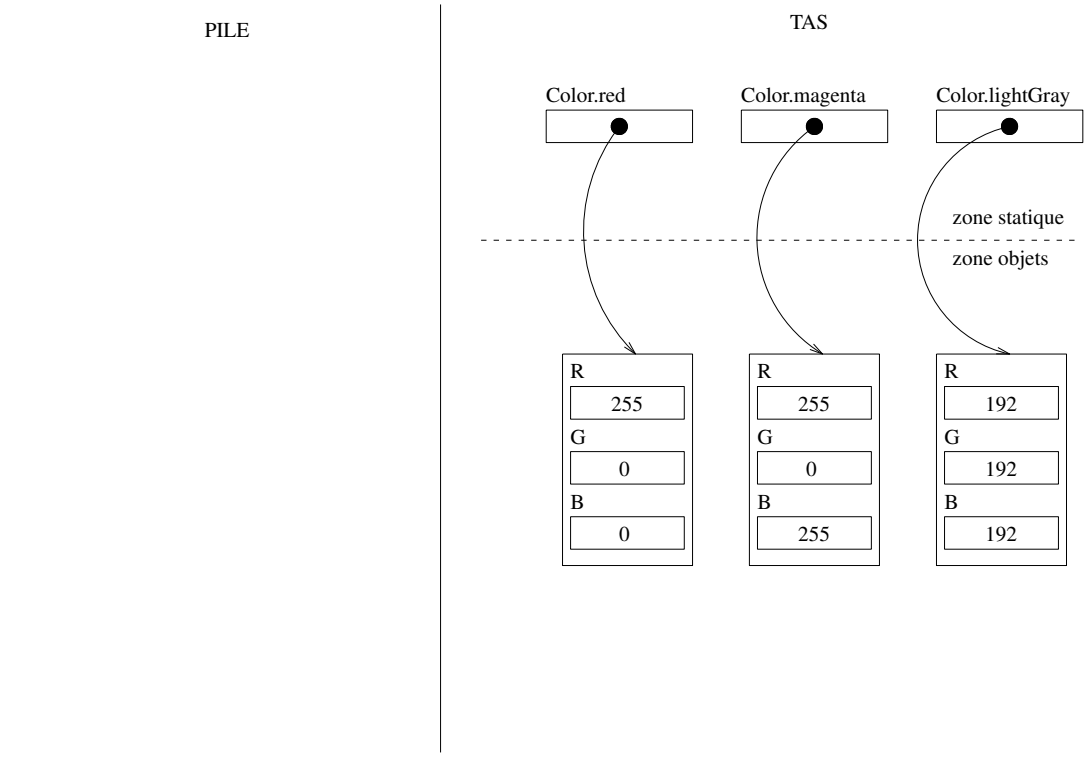
Un exemple de ce genre de constantes objets est souvent utilisé : les couleurs. En effet la classe `Color` contient (très approximativement) les déclarations suivantes :

```
1  /**
2   * classe modélisant les couleurs
3   */
4  public class Color {
5      private int R,G,B;
6      /**
7       * couleur rouge
8       */
9      public final static Color red=new Color(255,0,0);
10     /**
11      * couleur magenta
12      */
13     public final static Color magenta=new Color(255,0,255);
14     /**
15      * couleur gris clair
16      */
17     public final static Color lightgray=new Color(192,192,192);
18     // ...
19     /**
20      * constructeur de couleurs (système RGB)
21      */
22     public Color(int r,int g,int b) {
23         R=r;
24         G=g;
25         B=b;
```

```
26          }
27          // ...
28      }
```

La classe `Color` ne définit aucune méthode permettant de modifier une couleur après sa création.

Lorsqu'un programme utilise la classe `Color`, la zone statique de la mémoire est initialisée de la façon suivante :



11 Héritage

11.1 Principe et déclaration

Si plusieurs classes partagent une *base* commune (i.e. des propriétés communes), on peut regrouper ces propriétés dans une seule classe. Il suffit alors de *dériver* de cette unique classe pour disposer automatiquement des propriétés qu'elle définit. On dit alors que les objets de toute classe dérivée héritent les propriétés de la classe dont elle dérive.

Par exemple, un cercle et une ellipse sont des figures définies à partir d'un centre. La translation revient au même (déplacement du centre). On peut donc créer une classe `FigureCentree` qui définit un centre et une méthode de translation et dont `Cercle` et `Ellipse` vont dériver.

Ce mécanisme d'héritage permet notamment :

- des économies d'écriture : les propriétés communes ne sont explicitées qu'une seule fois, on se contente ensuite dans les classes dérivées d'effectuer de simples ajouts et modifications,
- le sous-typage : cette notion correspond approximativement à l'idée que la relation "dérive de" implique la relation "est un",
- la généricité : cette notion correspond approximativement à l'idée que certaines manipulations impliquant des objets de classes distinctes dérivées d'une même classe n'impliquent que des propriétés communes à toutes ces classes (par exemple la création de tableaux de cercles ou d'ellipses simplement en vue du calcul du barycentre n'a pas à différencier cercles et ellipses, on créera donc des tableaux de `FigureCentree` qu'on pourra indifféremment remplir avec des objets de classe `Cercle` ou `Ellipse` dont on calculera le barycentre).

La syntaxe de déclaration de l'héritage est la suivante :

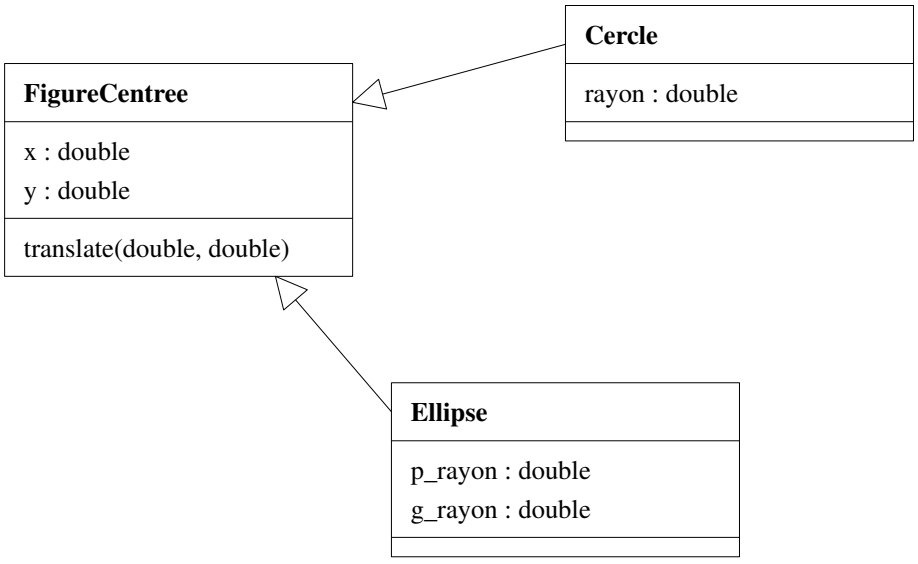
```
public class NomClasseDérivée extends NomClasseBase {
   ...
}
```

Exemple :

```
1  public class FigureCentree {
2    private double x,y;
3    public void translate(double dx,double dy) {
4      x+=dx;
5      y+=dy;
6    }
7  }
8  public class Cercle extends FigureCentree {
9    private double rayon;
10 }
11 public class Ellipse extends FigureCentree {
12   private double p_rayon,g_rayon;
13 }
14 public class EssaiFigureCentree {
15   public static void main(String args[]) {
16     Cercle c=new Cercle();
17     c.translate(1.0,2.0);
18     Ellipse e=new Ellipse();
19     e.translate(7.0,2.3);
20   }
21 }
```

Attention : seul l'héritage simple est autorisé en Java. Il est impossible de dériver une classe de plusieurs autres classes.

11.2 Représentation UML



11.3 Sous-typage

Au niveau de la mémoire, une instance de la classe dérivée *contient* une instance de la classe de base (i.e. contient les propriétés qui définissent la classe de base).

Il ne faut donc pas confondre l'héritage avec l'utilisation d'attributs objets. Dans l'exemple suivant

```
1   public class EllipseMalProgrammee {
2     private FigureCentree base;
3     private double p_rayon,g_rayon;
4   }
```

Un objet de classe `EllipseMalProgrammee` ne fait que contenir une référence à un objet de classe `FigureCentree`. Donc pour translater un tel objet ellipse, on ne peut pas directement lui demander d'exécuter la méthode `translate`, qu'il ne contient pas. Il est nécessaire de définir une méthode `translate` spécifique à la classe `EllipseMalProgrammee`, et ayant l'aspect suivant :

```
1   public class EllipseMalProgrammee {
2     private FigureCentree base;
3     public void translate(double dx,double dy) {
4       base.translate(dx,dy);
5     }
6   }
```

Lors d'une dérivation de classe, un objet de la classe dérivée contient toutes les propriétés de la classe de base, et on peut lui rajouter des attributs ou des méthodes supplémentaires. Par conséquent, en prenant en compte les définitions suivantes :

- Un type objet est la définition d'un ensemble de méthodes.
- *Un type objet B est un sous-type d'un type objet A si l'ensemble des méthodes décrit par B contient celui décrit par A*

on aboutit à la conclusion suivante : si une classe B est dérivée d'une classe A, alors B est un sous-type de A.

La gestion des types objets par `java` obéit aux règles suivantes :

- Le sous-typage permet la promotion de types. Une variable dont le type est une classe A peut contenir une référence à n'importe quel objet d'une classe dérivée de A (i.e. sous-type du type de la variable).

```
1   class A {}
2   class B extends A {}
3   class Test {
4       public static void main(String[] args) {
5           A varA=new B();
6       }
7   }
```

- A la compilation, l'existence des attributs et méthodes utilisées est vérifiée vis à vis du type de la variable de référence indiquée.
- Il est possible d'indiquer à `java` une conversion forcée en sens inverse (i.e. vers un sous-type), de façon à faire appel à des méthodes rajoutées dans la classe dérivée.

```
1   class A {}
2   class B extends A {
3       void m() {}
4   }
5   class Test {
6       public static void main(String[] args) {
7           A varA=new B();
8           B varB=(B)varA;
9           (B)varA.m();
```

```
10      }
11  }
```

❑ En héritage, l'appel de méthode objet provoque l'utilisation du mécanisme de liaison dynamique ou liaison tardive. Cela signifie que la méthode à appeler est déterminée au lancement du programme, et non à la compilation. Le comportement de l'objet dépend de son type réel, plutôt que du type de la variable faisant référence à cet objet. Autrement dit, Java détermine dynamiquement la méthode à appeler au moment de l'exécution.

❑ L'opérateur `instanceof` permet de s'assurer qu'un objet est bien d'un type voulu. Si une variable `var` de classe A contient une référence à un objet de classe B dérivée de A, alors les expressions `(var instanceof A)` et `(var instanceof B)` ont pour valeur `true`. En revanche, si `var` contient une référence à un objet de classe A, alors `(var instanceof B)` a pour valeur `false`.

```
1   class A {}
2   class B extends A {
3       void m() {}
4   }
5   class Test {
6       public static void main(String[] args) {
7           A varA=new B();
8           if (varA instanceof B) {
9               ((B)varA).m();
10          }
11      }
12  }
```

❑ Exemple : déterminez les erreurs à la compilation et à l'exécution que produirait le programme suivant

```
1   public class FigureCentree {
2       private double x,y;
3       public void translate(double dx,double dy) {
4           x=x+dx;
5           y=y+dy;
6       }
7   }
8
9   public class Cercle extends FigureCentree {
10      private double rayon;
11      public void setRayon(double r) {
12          rayon=r;
13      }
14  }
15
16  public class TestHeritage {
17    public static void main(String[] args) {
18      FigureCentree a;
19      Cercle b;
20      b=new Cercle();
21      a=b;
22      b.translate(1,1.5);
23      b.setRayon(2.3);
24      a.translate(3.2,-6.7);
25      a.setRayon(9.8);
26      b=a;
27      ((Cercle)a).setRayon(3.1);
28      a=new FigureCentree();
29      ((Cercle)a).setRayon(1.8);
30      if (a instanceof Cercle) {
```

```
31          ((Cercle)a).setRayon(1.8);
32        }
33      }
34    }
```

11.4 Héritage et méthodes d'instance redéfinies

Lorsqu'une classe est dérivée, il est possible de redéfinir des méthodes existantes dans la classe de base, dans le cas où ces méthodes, communes aux deux classes, aient des comportements différents suivant la classe de l'objet qui les exécute.

L'en-tête de la méthode doit être identique, on se contente de redéfinir le corps de la méthode. Lors de l'exécution, la méthode qui est effectivement exécutée dépend alors du type véritable de l'objet, et non du type de la variable dans laquelle la référence à l'objet est stockée.

```
1   public class Base {
2     public String nom() {
3       return "Base";
4     }
5     public void affiche() {
6       System.out.println(nom());
7     }
8   }
9   public class Derivee extends Base {
10    public String nom() {
11      return "Dérivée";
12      /* cette méthode est donc redéfinie, donc si un objet de classe Base
13         ou Dérivée exécute la méthode affiche() et fait ainsi appel à
14         nom(), la méthode véritablement exécutée dépendra du type
15         de l'objet, bien que l'appel soit fait dans une méthode affiche()
16         qui elle n'est pas redéfinie */
17    }
18  }
19  public class TestDerivee {
20    public static void main(String[] args) {
21      Base b=new Base();
22      Derivee d=new Derivee();
23      b.affiche(); // affiche ``Base'' car b contient une référence
24                   // à un objet de classe Base
25      d.affiche(); // affiche ``Dérivée'' car d contient une référence
26                   // à un objet de classe Dérivée
27      b=d;
28      b.affiche(); // affiche ``Dérivée'' car b contient une référence
29                   // à un objet de classe Dérivée, bien que b soit de type Base
30    }
31  }
```

Il est possible d'interdire la redéfinition d'une méthode grâce à la catégorie `final` :

```
public final TypeRetour nomMéthode (paramètres) {
   ...
}
```

Cette catégorie peut également permettre d'interdire la dérivation à partir d'une classe :

```
public final class nom_classe {
   ...
}
```

Les classes `String` et `StringBuffer` sont des exemples de classes `final`. Il est donc impossible de créer une classe d'objets qui héritent les propriétés des chaînes de caractères.

11.5 Héritage, constructeurs, et accès privé

Si une classe A définit des attributs d'accès privé, les méthodes définies (ou redéfinies) dans une classe B dérivant de A ne peuvent avoir accès à ces attributs. Il faut donc passer par les méthodes d'accès et de modification définies dans la classe A.

```
1  public class UnRectangle {
2    private double longueur,largeur;
3    public double getLongueur() {
4      return longueur;
5    }
6    public double getLargeur() {
7      return largeur;
8    }
9  }
10 public class Carre extends UnRectangle {
11   // un carré est un rectangle dont la largeur est égale à la longueur
12   public double getCote() {
13     // on ne peut pas écrire ici
14     //       return longueur;
15     // car longueur est un attribut qui existe dans la classe Carre par héritage,
16     // mais n'est pas accessible,
17     // d'où l'instruction suivante
18     return getLongueur();
19   }
20 }
```

Cette règle pose problème pour initialiser les attributs hérités à la construction d'un objet. Mais on peut accéder aux méthodes constructeurs de la classe parente grâce à la variable cachée `super`.

```
1  public class UnRectangle {
2    private double longueur,largeur;
3    public UnRectangle(double lo,double la) {
4      longueur=lo;
5      largeur=la;
6    }
7  }
8
9  public class Carre extends UnRectangle {
10   public Carre(double cote) {
11     super(cote,cote);  // exécute le corps du constructeur UnRectangle(double,double)
12   }
13 }
```

Attention :

- l'appel à `super(...)` doit être la première instruction du constructeur de la classe dérivée,
- si le programmeur ne fait pas d'appel explicite à un constructeur `super(...)` alors `java` rajoute implicitement comme première instruction un appel au constructeur vide de la classe parente `super()` (il faut alors s'assurer de l'existence de ce constructeur).

D'autre part la variable cachée `super` (non modifiable, comme `this`) permet de faire appel à une méthode de la classe parente lorsque celle-ci est redéfinie.

```
1  public class UnRectangle {
2    private double longueur,largeur;
3    public void affiche() {
4      System.out.print("Rectangle");
5    }
6  }
7  public class Carre extends UnRectangle {
8    public void affiche() {
9      super.affiche();
10     System.out.print(" spécial (Carré)");
11     /* si un objet de classe Carre exécute la méthode affiche(),
12        le texte affiché sera ‘‘Rectangle spécial (Carré)’’ */
13   }
14 }
```

11.6 Classe `Object`

Le mécanisme d'héritage est implicitement présent dans tout programme Java, car toute classe hérite d'une même classe `Object` qui décrit l'ensemble des propriétés communes à tous les objets.

Une de ces propriétés est la possibilité d'être affiché. En fait, lors d'un appel à `System.out.println`, `java` affiche la chaîne de caractères obtenue par la méthode d'instance `toString` déjà mentionnée. Cette méthode est définie dans la classe `Object`, se contentant de retourner une chaîne de caractères représentant la référence de l'objet. Lors de la programmation d'une classe, il est alors possible (comme indiqué précédemment) de redéfinir cette méthode. Dans ce cas, lors d'un affichage, la méthode `toString` véritablement exécutée sera la méthode redéfinie. C'est donc ce mécanisme de redéfinition qui fait que l'affichage s'adapte automatiquement lors de l'exécution à la nature exacte des objets affichés.

```
1  public class UnPoint {
2     private int abscisse,ordonnee;
3     public String toString() {
4         return super.toString()
5                +" : point de coordonnées ("+abscisse+","+ordonnee+")";
6     }
7  }
8  public class TestToString {
9     public static void main(String[] args) {
10        System.out.println(new UnPoint());
11        // affiche ‘‘UnPoint@23bcc1 : point de coordonnées (0,0)’’
12    }
13 }
```

La classe `Object` contient d'autres méthodes qui peuvent être utilisées ou redéfinies dans n'importe quelle classe.

L'existence de cette classe parente de toutes les classes permet de manipuler des variables génériques,

déclarées de classe `Object` mais pouvant donc contenir des références à des objets de n'importe quelle classe. On peut ainsi par exemple créer des tableaux génériques.

```
1  Object[] tab=new Object[3];
2  tab[0]="bonjour";
3  tab[1]=new StringBuffer();
4  tab[2]=new Object();
```

12 Modèles abstraits

12.1 Classes abstraites

Lorsque plusieurs classes ont une méthode en commun, il est préférable d'indiquer cette méthode dans une classe de base dont les classes dérivent. Néanmoins il peut arriver que le comportement de cette méthode ne puisse être défini dès la classe de base. La méthode est alors déclarée de façon abstraite (catégorie `abstract`) : on annonce juste son existence commune, pas sa réalisation.

`public abstract` *TypeRetour nomMéthode* `(`*paramètres*`);`

Si une classe contient au moins une méthode abstraite, alors cette classe est elle-même abstraite. **Il est impossible d'instancier une classe abstraite**.

```
public abstract class nom_classe {
   ...
}
```

Toute classe non abstraite dérivée d'une classe abstraite doit implémenter `toutes` les méthodes abstraites héritées (on enlève alors le mot-clé `abstract`).

Cela n'empêche pas d'appeler la méthode abstraite par l'intermédiaire d'une variable de référence dont le type est la classe abstraite. Car de toute façon l'objet référencé dispose d'une programmation de cette méthode (la classe véritable de l'objet ne peut pas être abstraite, et le choix de la méthode véritablement exécutée se fait de façon dynamique, donc en tenant compte de la nature exacte de l'objet).

Exemple :

```
1  /**
2   * classe abstraite réunissant l'ensemble des propriétés des figures centrées
3   */
4  public abstract class FigureCentree {
5    private double x,y;
6    /**
7     * constructeur indiquant le centre de la figure
8     * @param c_x abscisse du centre
9     * @param c_y ordonnée du centre
10    */
11   public FigureCentree(double c_x,double c_y) {
12     x=c_x;
13     y=c_y;
14   }
15
16   /**
17    * méthode indiquant l'abscisse du centre
18    * @return abscisse du centre de la figure
19    */
```

```
20    public double getAbscisse() {
21      return x;
22    }
23    /**
24     * méthode indiquant l'ordonnée du centre
25     * @return ordonnée du centre de la figure
26     */
27    public double getOrdonnee() {
28      return y;
29    }
30    /**
31     * méthode de translation commune à toutes les figures centrées
32     * @param dx longueur de translation en abscisse
33     * @param dy longueur de translation en ordonnée
34     */
35    public void translate(double dx,double dy) {
36      x+=dx;
37      y+=dy;
38    }
39    /**
40     * méthode abstraite de dilatation commune à toutes les figures centrées
41     * @param coeff coefficient de dilatation
42     */
43    public abstract void dilate(double coeff);
44  }
45
46  /**
47   * classe représentant la notion géométrique de cercle
48   */
49  class Cercle extends FigureCentree {
50    private double rayon;
51    /**
52     * constructeur indiquant le centre du cercle et son rayon
53     * @param c_x abscisse du centre
54     * @param c_y ordonnée du centre
55     * @param r rayon
56     */
57    public Cercle(double c_x,double c_y,double r) {
58      super(c_x,c_y);
59      rayon=r;
60    }
61
62    /**
63     * méthode de dilatation propre aux cercles,
64     * définit la méthode abstraite de la classe FigureCentree
65     * @param coeff coefficient de dilatation
66     */
67    public void dilate(double coeff) {
68      rayon*=coeff;
69    }
70  }
71
72  /**
73   * classe représentant la notion géométrique d'ellipse
```

```
74    */
75   class Ellipse extends FigureCentree {
76     private double p_rayon,g_rayon;
77     /**
78      * constructeur indiquant le centre du cercle et ses rayons
79      * @param c_x abscisse du centre
80      * @param c_y ordonnée du centre
81      * @param p_r petit rayon
82      * @param g_r grand rayon
83      */
84     public Ellipse(double c_x,double c_y,double p_r,double g_r) {
85       super(c_x,c_y);
86       p_rayon=p_r;
87       g_rayon=g_r;
88     }
89     /**
90      * méthode de dilatation propre aux ellipses,
91      * définit la méthode abstraite de la classe FigureCentree
92      * @param coeff coefficient de dilatation
93      */
94     public void dilate(double coeff) {
95       p_rayon*=coeff;
96       g_rayon*=coeff;
97     }
98   }
99
100  public class EssaiAbstract {
101    public static void main(String args[]) {
102      FigureCentree[] tab={new Cercle(1.0,1.2,2.4),new Ellipse(2.3,1.8,1.7,3.8)};
103      for (int i=0;i<tab.length;i++) {
104        tab[i].translate(1.5,0.5);
105        tab[i].dilate(4.9);
106      }
107    }
108  }
```

12.2 Interfaces

Lorsqu'une classe ne contient aucun attribut et que toutes ses méthodes sont des méthodes d'instance abstraites, cette classe peut être déclarée comme une interface : c'est une énumération de fonctionnalités offertes. Le mot-clé `abstract` devient inutile, puisque toutes les méthodes sont abstraites.

- Déclaration :

```
public interface NomInterface {
   public TypeRetour nomMéthode1 (paramètres);
   public TypeRetour nomMéthode2 (paramètres);
   ...
}
```

- Implantation : une classe peut "implanter" une interface en précisant le corps des différentes méthodes de l'interface.

```
public class NomClasse implements NomInterface {
   ...
}
```

❑ Une telle classe n'est instanciable que si toutes les méthodes de l'interface sont implantées, sinon la classe reste abstraite.

```
1   public interface UneInterface {
2     public void m0();
3     public void m1();
4   }
5
6   public abstract class UneClasseEncoreAbstraite implements UneInterface {
7     public void m0() {
8       ...
9     }
10  }
11
12  public class UneClasseInstanciable implements UneInterface {
13    public void m0() {
14      ...
15    }
16    public void m1() {
17      ...
18    }
19  }
```

❑ Une classe peut simultanément dériver et implanter (la dérivation est indiquée avant l'implantation). Elle dispose alors par héritage de toutes les méthodes de la classe parente et de l'interface implantée (dont elle doit préciser le contenu pour être instanciable).

```
1   public interface Dessinable {
2     public void dessine();
3   }
4
5   public class Cercle extends FigureCentree implements Dessinable {
6     private double rayon;
7     public Cercle(double c_x,double c_y,double r) {
8       super(c_x,c_y);
9       rayon=r;
10    }
11
12    public void dilate(double coeff) {
13      rayon*=coeff;
14    }
15    public void dessine() {
16      // appels de méthodes graphiques
17    }
18  }
19
20  public class Ellipse extends FigureCentree implements Dessinable {
21    ...
22  }
23
24  public class EssaiInterface {
25    public static void main(String args[]) {
26      Dessinable[] tab={new Cercle(1.0,1.2,2.4),new Ellipse(2.3,1.8,1.7,3.8)};
27      for (int i=0;i<tab.length;i++) {
28        tab[i].dessine();
29      }
30    }
```

```
31   }
```

- Les implantations multiples sont autorisées (contrairement à l'héritage multiple). En effet l'héritage multiple impliquerait des mécanismes complexes à gérer par rapport aux appels de méthodes héritées de différentes classes. En revanche, ça ne pose aucun problème dans le cas des interfaces, puisque les méthodes véritablement exécutées ne peuvent être que définies au moment des implantations.

```
public class NomClasse implements NomInterface1,NomInterface2,... {
   ...
}
```

- Une interface peut dériver d'autres interfaces. Le mot-clé `extends` est utilisé pour la dérivation d'interfaces, cette dérivation pouvant être multiple. Une interface dérivée contient alors toutes les méthodes des interfaces parentes. Une classe qui implante l'interface dérivée doit donc implanter toutes les méthodes de cette interface et de ses interfaces parentes.

```
1   public interface Coloriable {
2     public void colorieRouge();
3     public void colorieBleu();
4   }
5
6   public interface Dessinable {
7     public void dessine();
8   }
9
10  public interface DessinColoriable extends Coloriable,Dessinable {
11  }
12
13  public class Cercle extends FigureCentree implements DessinColoriable {
14    public void dessine() {
15      ...
16    }
17
18    public void colorieRouge() {
19      ...
20    }
21    public void colorieBleu() {
22      ...
23    }
24  }
```

- Il est possible de déclarer des attributs dans des interfaces, mais ce sont alors des constantes (i.e. `final static` sous-entendu).

```
1   public interface A {
2     int var = 0;  // initialisation obligatoire
3   }
```

Les principaux apports des interfaces sont de permettre un pseudo héritage multiple (pseudo car on hérite des méthodes sans contenu), et de généraliser l'emploi du polymorphisme : un objet répond à un message (méthode) défini dans l'interface (type objet) par une méthode définie dans la classe spécifique de l'objet.

```
1   public interface ObjetParlant {
2     public void jeParle();
3   }
4   public abstract class ObjetPonctuant {
5     public abstract String maPonctuation();
6     public void jePonctue() {
7       System.out.println(maPonctuation());
8     }
```

```
 9  }
10  public class Chuchoteur extends ObjetPonctuant implements ObjetParlant {
11    public String maPonctuation() {
12      return ".";
13    }
14    public void jeParle() {
15      System.out.print("bonjour");
16    }
17  }
18  public class Crieur extends ObjetPonctuant implements ObjetParlant {
19    public String maPonctuation() {
20      return "!!!";
21    }
22    public void jeParle() {
23      System.out.print("BONJOUR");
24    }
25  }
26  public class FaitParler {
27    static void blabla(ObjetParlant[] tab) {
28      for (int i=0;i<tab.length;i++) {
29        tab[i].jeParle();
30        if (tab[i] instanceof ObjetPonctuant)
31            tab[i].jePonctue();
32      }
33    }
34
35  public static void main(String[] args) {
36      ObjetParlant[] tab=new ObjetParlant[2];
37      tab[0]=new Chuchoteur();
38      tab[1]=new Crieur();
39      blabla(tab);  // ce qui affiche ‘‘bonjour.’’ puis ‘‘BONJOUR!!!’’
40    }
41  }
```

12.3 Représentation UML