

25.04.2001 :

Mise en forme du code html (titres-hx[verdana], paragraphes-p[Georgia], code-blockquote).

6 : Réutiliser les classes

Une des caractéristiques les plus excitantes de Java est la réutilisation du code. Mais pour être vraiment révolutionnaire, il faut faire plus que copier du code et le changer.

C'est l'approche utilisée dans les langages procéduraux comme C, et ça n'a pas très bien fonctionné. Comme tout en Java, la solution réside dans les classes. On réutilise du code en créant de nouvelles classes, mais au lieu de les créer depuis zéro, on utilise les classes que quelqu'un a construit et testé.

L'astuce est d'utiliser les classes sans détériorer le code existant. Dans ce chapitre nous verrons deux manières de faire. La première est plutôt directe : On crée simplement des objets de nos classes existantes à l'intérieur de la nouvelle classe. Ça s'appelle *la composition*, parce que la nouvelle classe se compose d'objets de classes existantes. On réutilise simplement les fonctionnalités du code et non sa forme.

La seconde approche est plus subtile. On crée une nouvelle classe comme un *type* d'une classe existante. On prend littéralement la forme d'une classe existante et on lui ajoute du code sans modifier la classe existante. Cette magie s'appelle *l'héritage*, et le compilateur fait le plus gros du travail. L'héritage est une des pierres angulaires de la programmation par objets, et a bien d'autres implications qui seront explorées au chapitre 7.

Il s'avère que beaucoup de la syntaxe et du comportement sont identiques pour la composition et l'héritage (cela se comprend parce qu'ils sont tous deux des moyens de construire des nouveaux types à partir de types existants). Dans ce chapitre, nous apprendrons ces mécanismes de réutilisation de code.

Syntaxe de composition

Jusqu'à maintenant, la composition a été utilisée assez fréquemment. On utilise simplement des références sur des objets dans de nouvelles classes. Par exemple, supposons que l'on souhaite un objet qui contient plusieurs objets de type **String**, quelques types primitifs et un objet d'une autre classe. Pour les objets, on met des références à l'intérieur de notre nouvelle classe, mais on définit directement les types primitifs:

```
// ! c06:SprinklerSystem.java
// La composition pour réutiliser du code.

class WaterSource {
    private String s;

    WaterSource() {
        System.out.println("WaterSource()");
        s = new String("Constructed");
    }

    public String toString() { return s; }
```

```
}

public class SprinklerSystem {

    private String valve1, valve2, valve3, valve4;

    WaterSource source;

    int i;

    float f;

    void print() {

        System.out.println("valve1 = " + valve1);

        System.out.println("valve2 = " + valve2);

        System.out.println("valve3 = " + valve3);

        System.out.println("valve4 = " + valve4);

        System.out.println("i = " + i);

        System.out.println("f = " + f);

        System.out.println("source = " + source);

    }

    public static void main(String[] args) {

        SprinklerSystem x = new SprinklerSystem();

        x.print();

    }

} ///:~
```

Une des méthodes définies dans **WaterSource** est spéciale : **toString()**. Vous apprendrez plus tard que chaque type non primitif a une méthode **toString()**, et elle est appelée dans des situations spéciales lorsque le compilateur attend une **String** alors qu'il ne trouve qu'un objet. Donc dans une expression:

```
System.out.println("source = " + source);
```

le compilateur voit que vous essayez d'ajouter un objet **String** ("source = ") à un **WaterSource**. Ceci n'a pas de sens parce qu'on peut seulement ajouter une **String** à une autre **String**, donc il se dit qu'il va convertir **source** en une **String** en appelant **toString()** ! Après avoir fait cela il combine les deux **Strings** et passe la **String** résultante à **System.out.println()**. Dès qu'on veut permettre ce comportement avec une classe qu'on crée, il suffit simplement de définir une méthode **toString()**.

Au premier regard, on pourrait supposer — Java étant sûr et prudent comme il l'est — que le compilateur construirait automatiquement des objets pour chaque référence dans le code ci-dessus ; par exemple, en appelant le constructeur par défaut pour **WaterSource** pour initialiser **source**. Le résultat de l'instruction d'impression affiché est en fait :

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null

i = 0
f = 0.0

source = null
```

Les types primitifs qui sont des champs d'une classe sont automatiquement initialisés à zéro, comme précisé dans le chapitre 2. Mais les références objet sont initialisées à **null**, et si on essaye d'appeler des méthodes pour l'un d'entre eux, on obtient une exception. En fait il est bon (et utile) qu'on puisse les afficher sans lancer d'exception.

On comprend bien que le compilateur ne crée pas un objet par défaut pour chaque référence parce que cela induirait souvent une surcharge inutile. Si on veut initialiser les références, on peut faire :

1. Au moment où les objets sont définis. Cela signifie qu'ils seront toujours initialisés avant que le constructeur ne soit appelé ;
2. Dans le constructeur pour la classe ;
3. Juste avant d'utiliser l'objet, ce qui est souvent appelé *initialisation paresseuse*.

Cela peut réduire la surcharge dans les situations où l'objet n'a pas besoin d'être créé à chaque fois.

Les trois approches sont montrées ici:

```
// ! c06:Bath.java
// Initialisation dans le constructeur avec composition.

class Soap {
    private String s;

    Soap() {
        System.out.println("Soap()");
        s = new String("Constructed");
    }

    public String toString() { return s; }
}

public class Bath {
    private String

    // Initialisation au moment de la définition:
```

```
s1 = new String("Happy"),
s2 = "Happy",
s3, s4;
Soap castille;
int i;
float toy;
Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
}
void print() {
    // Initialisation différée:
    if(s4 == null)
        s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
}
public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
}
} ///:~
```

Notez que dans le constructeur de **Bath** une instruction est exécutée avant que toute initialisation ait lieu. Quand on n'initialise pas au moment de la définition, il n'est pas encore garanti qu'on exécutera une initialisation avant qu'on envoie un message à un objet — sauf l'inévitable exception à l'exécution.

Ici la sortie pour le programme est :

```
Inside Bath()

Soap()

s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy

i = 47

toy = 3.14

castille = Constructed
```

Quand **print()** est appelé, il remplit **s4** donc tout les champs sont proprement initialisés au moment où ils sont utilisés.

La syntaxe de l'héritage

L'héritage est une partie primordiale de Java (et des langages de programmation par objets en général). Il s'avère qu'on utilise toujours l'héritage quand on veut créer une classe, parce qu'à moins d'hériter explicitement d'une autre classe, on hérite implicitement de la classe racine standard **Object**.

La syntaxe de composition est évidente, mais pour réaliser l'héritage il y a une forme distinctement différente. Quand on hérite, on dit « Cette nouvelle classe est comme l'ancienne classe ». On stipule ceci dans le code en donnant le nom de la classe comme d'habitude, mais avant l'accolade ouvrante du corps de la classe, on met le mot clé **extends** suivi par le nom de *la classe de base*. Quand on fait cela, on récupère automatiquement toutes les données membres et méthodes de la classe de base. Voici un exemple:

```
// ! c06:Detergent.java

// Syntaxe d'héritage & propriétés.

class Cleanser {

    private String s = new String("Cleanser");

    public void append(String a) { s += a; }

    public void dilute() { append(" dilute()"); }

    public void apply() { append(" apply()"); }

    public void scrub() { append(" scrub()"); }

    public void print() { System.out.println(s); }

    public static void main(String[] args) {

        Cleanser x = new Cleanser();
```

```

        x.dilute(); x.apply(); x.scrub();

        x.print();
    }
}

public class Detergent extends Cleanser {

    // Change une méthode:
    public void scrub() {

        append(" Detergent.scrub()");

        super.scrub(); // Appel de la version de la classe de base
    }

    // Ajoute une méthode à l'interface:
    public void foam() { append(" foam()"); }

    // Test de la nouvelle classe:
    public static void main(String[] args) {

        Detergent x = new Detergent();

        x.dilute();

        x.apply();

        x.scrub();

        x.foam();

        x.print();

        System.out.println("Testing base class:");

        Cleanser.main(args);
    }
} ///:~

```

Ceci montre un certain nombre de caractéristiques. Premièrement, dans **Cleanser** la méthode **append()**, les **Strings** sont concaténées dans **s** en utilisant l'opérateur **+=**, qui est l'un des opérateurs (avec « + ») que les créateurs de Java « ont surchargé » pour travailler avec les **Strings**.

Deuxièmement, tant **Cleanser** que **Detergent** contiennent une méthode **main()**. On peut créer une **main()** pour chacune de nos classes, et il est souvent recommandé de coder de cette manière afin de garder le code de test dans la classe. Même si on a beaucoup de classes dans un programme, seulement la méthode **main()** pour une classe invoquée sur la ligne de commande sera appelée. Aussi longtemps que **main()** est **public**, il importe peu que la classe dont elle fait partie soit **public** ou non. Donc dans ce cas, quand on écrit **java Detergent**, **Detergent.main()** sera appelée. Mais on peut également écrire **java Cleanser** pour invoquer **Cleanser.main()**, même si **Cleanser** n'est pas une classe **public**. Cette technique de mettre une **main()** dans chaque classe permet de tester facilement chaque classe. Et on n'a pas besoin d'enlever la méthode **main()** quand on a finit de tester ; on peut la laisser pour tester plus tard.

Ici, on peut voir que **Detergent.main()** appelle **Cleanser.main()** explicitement, en passant les même arguments depuis la ligne de commande (quoiqu'il en soit, on peut passer n'importe quel tableau de **String**).

Il est important que toutes les méthodes de **Cleanser** soient **public**. Il faut se souvenir que si on néglige tout modifieur d'accès, par défaut l'accès sera « friendly », lequel permet d'accéder seulement aux membres du même package. Donc, *au sein d'un même package*, n'importe qui peut utiliser ces méthodes s'il n'y a pas de spécificateur d'accès. **Detergent** n'aurait aucun problème, par exemple. Quoiqu'il en soit, si une classe d'un autre package devait hériter de **Cleanser** il pourrait accéder seulement aux membres **public**. Donc pour planifier l'héritage, en règle générale mettre tous les champs **private** et toutes les méthodes **public** (les membres **protected** permettent également d'accéder depuis une classe dérivée ; nous verrons cela plus tard). Bien sûr, dans des cas particuliers on devra faire des ajustements, mais cela est une règle utile.

Notez que **Cleanser** contient un ensemble de méthodes dans son interface : **append()**, **dilute()**, **apply()**, **scrub()**, et **print()**. Parce que **Detergent** est *dérivé de Cleanser* (à l'aide du mot-clé **extends**) il récupère automatiquement toutes les méthodes de son interface, même si elles ne sont pas toutes définies explicitement dans **Detergent**. On peut penser à l'héritage *comme à une réutilisation de l'interface* (l'implémentation vient également avec elle, mais ceci n'est pas le point principal).

Comme vu dans **scrub()**, il est possible de prendre une méthode qui a été définie dans la classe de base et la modifier. Dans ce cas, on pourrait vouloir appeler la méthode de la classe de base dans la nouvelle version. Mais à l'intérieur de **scrub()** on ne peut pas simplement appeler **scrub()**, car cela produirait un appel récursif, ce qui n'est pas ce que l'on veut. Pour résoudre ce problème, Java a le mot-clé **super** qui réfère à la super classe de la classe courante. Donc l'expression **super.scrub()** appelle la version de la classe de base de la méthode **scrub()**.

Quand on hérite, on n'est pas tenu de n'utiliser que les méthodes de la classe de base. On peut également ajouter de nouvelles méthodes à la classe dérivée exactement de la manière dont on met une méthode dans une classe : il suffit de la définir. La méthode **foam()** en est un exemple.

Dans **Detergent.main()** on peut voir que pour un objet **Detergent** on peut appeler toutes les méthodes disponible dans **Cleanser** aussi bien que dans **Detergent** (e.g., **foam()**).

Initialiser la classe de base

Depuis qu'il y a deux classes concernées - la classe de base et la classe dérivée - au lieu d'une seule, il peut être un peu troublant d'essayer d'imaginer l'objet résultant produit par la classe dérivée. De l'extérieur, il semble que la nouvelle classe a la même interface que la classe de base et peut-être quelques méthodes et champs additionnels. Mais l'héritage ne se contente pas simplement de copier l'interface de la classe de base. Quand on crée un objet de la classe dérivée, il contient en lui un *sous-objet* de la classe de base. Ce sous-objet est le même que si on crée un objet de la classe de base elle-même. C'est simplement que, depuis l'extérieur, le sous-objet de la classe de base est enrobé au sein de l'objet de la classe dérivée.

Bien sûr, il est essentiel que le sous-objet de la classe de base soit correctement initialisé et il y a un seul moyen de garantir cela: exécuter l'initialisation dans le constructeur, en appelant la constructeur de la classe de base, lequel a tous les connaissances et les privilèges appropriés pour exécuter l'initialisation de la classe de base. Java insère automatiquement les appels au constructeur de la classe de base au sein du constructeur de la classe dérivée. L'exemple suivant montre comment cela fonctionne avec 3 niveaux d'héritage :

```
// ! c06:Cartoon.java
```

```
// Appels de constructeur durant l'initialisation

class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} ///:~
```

La sortie de ce programme montre les appels automatiques:

```
Art constructor
Drawing constructor
Cartoon constructor
```

On peut voir que la construction commence par la classe la plus haute dans la hiérarchie, donc la classe de base est initialisée avant que les constructeurs de la classe dérivée puisse y accéder.

Même si on ne crée pas de constructeur pour **Cartoon()**, le compilateur fournira un constructeur.

Constructeurs avec paramètres

L'exemple ci-dessus a des constructeurs par défaut ; ils n'ont pas de paramètres. C'est facile pour le compilateur d'appeler ceux-ci parce qu'il n'y a pas de questions à se poser au sujet des arguments à passer. Si notre classe n'a

pas de paramètres par défaut, ou si on veut appeler le constructeur d'une classe de base avec paramètre, on doit explicitement écrire les appels au constructeur de la classe de base en utilisant le mot clé **super** ainsi que la liste de paramètres appropriée : **super** et la liste de paramètres appropriée:

```
// ! c06:Chess.java
// Héritage, constructeurs et paramètres.

class Game {
    Game(int i) {
        System.out.println("Game constructor");
    }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        System.out.println("BoardGame constructor");
    }
}

public class Chess extends BoardGame {
    Chess() {
        super(11);
        System.out.println("Chess constructor");
    }

    public static void main(String[] args) {
        Chess x = new Chess();
    }
} ///:~
```

Si on n'appelle pas le constructeur de la classe de base dans **BoardGame()**, le compilateur va se plaindre qu'il ne peut pas trouver le constructeur de la forme **Game()**. De plus, l'appel du constructeur de la classe de base *doit* être la première chose que l'on fait dans le constructeur de la classe dérivée. Le compilateur va le rappeler si on se trompe.

Attraper les exceptions du constructeur de base.

Comme nous venons de le préciser, le compilateur nous force à placer l'appel du constructeur de la classe de

base en premier dans le constructeur de la classe dérivée. Cela signifie que rien ne peut être placé avant lui. Comme vous le verrez dans le chapitre 10, cela empêche également le constructeur de la classe dérivée d'attraper une exception qui provient de la classe de base. Ceci peut être un inconvénient de temps en temps.

Combiner composition et héritage.

Il est très classique d'utiliser ensemble la composition et l'héritage. L'exemple suivant montre la création d'une classe plus complexe, utilisant à la fois l'héritage et la composition, avec la nécessaire initialisation des constructeurs:

```
// ! c06:PlaceSetting.java
// Mélanger composition & héritage.

class Plate {
    Plate(int i) {
        System.out.println("Plate constructor");
    }
}

class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        System.out.println(
            "DinnerPlate constructor");
    }
}

class Utensil {
    Utensil(int i) {
        System.out.println("Utensil constructor");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        System.out.println("Spoon constructor");
    }
}
```

```
    }  
}  
  
class Fork extends Utensil {  
    Fork(int i) {  
        super(i);  
        System.out.println("Fork constructor");  
    }  
}  
  
class Knife extends Utensil {  
    Knife(int i) {  
        super(i);  
        System.out.println("Knife constructor");  
    }  
}  
  
// Une manière culturelle de faire quelque chose:  
class Custom {  
    Custom(int i) {  
        System.out.println("Custom constructor");  
    }  
}  
  
public class PlaceSetting extends Custom {  
    Spoon sp;  
    Fork frk;  
    Knife kn;  
    DinnerPlate pl;  
    PlaceSetting(int i) {  
        super(i + 1);  
        sp = new Spoon(i + 2);  
        frk = new Fork(i + 3);  
        kn = new Knife(i + 4);  
    }  
}
```

```

        pl = new DinnerPlate(i + 5);

        System.out.println(

            "PlaceSetting constructor");
    }

    public static void main(String[] args) {

        PlaceSetting x = new PlaceSetting(9);

    }

} ///:~

```

Tant que le compilateur nous force à initialiser les classes de base, et requiert que nous le fassions directement au début du constructeur, il ne vérifie pas que nous initialisons les objets membres, donc nous devons nous souvenir de faire attention à cela.

Garantir un nettoyage propre

Java ne possède pas le concept C++ de *destructeur*, une méthode qui est automatiquement appelée quand un objet est détruit. La raison est probablement qu'en Java la pratique est simplement d'oublier les objets plutôt que les détruire, laissant le ramasse-miette réclamer la mémoire selon les besoins.

Souvent cela convient, mais il existe des cas où votre classe pourrait, durant son existence, exécuter des tâches nécessitant un nettoyage. Comme mentionné dans le chapitre 4, on ne peut pas savoir quand le ramasse-miettes sera exécuté, ou s'il sera appelé. Donc si on veut nettoyer quelque chose pour une classe, on doit écrire une méthode particulière, et être sûr que l'utilisateur sait qu'il doit appeler cette méthode. Par dessus tout, comme décrit dans le chapitre 10 (« Gestion d'erreurs avec les exceptions ») - on doit se protéger contre une exception en mettant un tel nettoyage dans une clause **finally**.

Considérons un exemple d'un système de conception assisté par ordinateur qui dessine des images sur l'écran:

```

// ! c06:CADSystem.java

// Assurer un nettoyage propre.

import java.util.*;

class Shape {

    Shape(int i) {

        System.out.println("Shape constructor");

    }

    void cleanup() {

        System.out.println("Shape cleanup");

    }

}

```

```
class Circle extends Shape {  
    Circle(int i) {  
        super(i);  
        System.out.println("Drawing a Circle");  
    }  
    void cleanup() {  
        System.out.println("Erasing a Circle");  
        super.cleanup();  
    }  
}
```

```
class Triangle extends Shape {  
    Triangle(int i) {  
        super(i);  
        System.out.println("Drawing a Triangle");  
    }  
    void cleanup() {  
        System.out.println("Erasing a Triangle");  
        super.cleanup();  
    }  
}
```

```
class Line extends Shape {  
    private int start, end;  
    Line(int start, int end) {  
        super(start);  
        this.start = start;  
        this.end = end;  
        System.out.println("Drawing a Line : " +  
            start + ", " + end);  
    }  
    void cleanup() {  
        System.out.println("Erasing a Line : " +
```

```
        start + ", " + end);

    super.cleanup();
}

}

public class CADSystem extends Shape {

    private Circle c;
    private Triangle t;
    private Line[] lines = new Line[10];

    CADSystem(int i) {
        super(i + 1);

        for(int j = 0; j < 10; j++)
            lines[j] = new Line(j, j*j);

        c = new Circle(1);
        t = new Triangle(1);

        System.out.println("Combined constructor");
    }

    void cleanup() {
        System.out.println("CADSystem.cleanup()");

        // L'ordre de nettoyage est l'inverse
        // de l'ordre d'initialisation

        t.cleanup();
        c.cleanup();

        for(int i = lines.length - 1; i >= 0; i--)
            lines[i].cleanup();

        super.cleanup();
    }

    public static void main(String[] args) {
        CADSystem x = new CADSystem(47);

        try {
            // Code et gestion des exceptions...
        } finally {
            x.cleanup();
        }
    }
}
```

```

    }
} ///:~

```

Tout dans ce système est une sorte de **Shape** (lequel est une sorte d'**Object** puisqu'il hérite implicitement de la classe racine). Chaque classe redéfinit la méthode **cleanup()** de **Shape** en plus d'appeler la méthode de la classe de base en utilisant **super**. Les classes **Shape** spécifiques —**Circle**, **Triangle** et **Line**— ont toutes des constructeurs qui « dessinent », bien que n'importe quelle méthode appelée durant la durée de vie d'un objet pourrait être responsable de faire quelque chose qui nécessite un nettoyage. Chaque classe possède sa propre méthode **cleanup()** pour restaurer les choses de la manière dont elles étaient avant que l'objet n'existe.

main(), on peut noter deux nouveaux mot-clés qui ne seront pas officiellement introduits avant le chapitre 10 : **try** et **finally**. Le mot-clé **try** indique que le bloc qui suit (délimité par les accolades) est une *région gardée*, ce qui signifie qu'elle a un traitement particulier. Un de ces traitements particuliers est que le code dans la clause **finally** suivant la région gardée est *toujours* exécuté, quelle que soit la manière dont le bloc **try** termine. Avec la gestion des exceptions, il est possible de quitter le bloc **try** de manière non ordinaire. Ici la clause **finally** dit de toujours appeler **cleanup()** pour x, quoiqu'il arrive. Ces mot-clés seront expliqués plus en profondeur dans le chapitre 10.

Notez que dans votre méthode **cleanup** vous devez faire attention à l'ordre d'appel pour les méthodes **cleanup** de la classe de base et celle des objet-membres au cas où un des sous-objets dépend des autres. En général, vous devriez suivre la même forme qui est imposée pour le compilateur C++ sur ses destructeurs : premièrement exécuter tout le nettoyage spécifique à votre classe, dans l'ordre inverse de la création. En général, cela nécessite que les éléments de la classe de base soient encore viable. Ensuite appeler la méthode **cleanup** de la classe de base, comme démontré ici.

Il y a beaucoup de cas pour lesquels le problème de nettoyage n'est pas un problème ; on laisse le ramasse-miettes faire le travail. Mais quand on doit le faire explicitement, diligence et attention sont requis.

L'ordre du ramasse-miettes

Il n'y a pas grand chose sur quoi on puisse se fier quand il s'agit de ramasse-miettes. Le ramasse-miette peut ne jamais être appelé. S'il l'est, il peut réclamer les objets dans n'importe quel ordre. Il est préférable de ne pas se fier au ramasse-miette pour autre chose que libérer la mémoire. Si on veut que le nettoyage ait lieu, faites vos propres méthodes de nettoyage et ne vous fiez pas à **finalize()**. Comme mentionné dans le chapitre 4, Java peut être forcé d'appeler tous les finalizers.

Cacher les noms

Seuls les programmeurs C++ pourraient être surpris par le masquage de noms, étant donné que le fonctionnement est différent dans ce langage. Si une classe de base Java a un nom de méthode qui est surchargé plusieurs fois, redéfinir ce nom de méthode dans une sous-classe ne cachera aucune des versions de la classe de base. Donc la surcharge fonctionne sans savoir si la méthode était définie à ce niveau ou dans une classe de base:

```

// ! c06:Hide.java

// Surchage le nom d'une méthode de la classe de base

// dans une classe dérivée ne cache pas

// les versions de la classe de base.

```

```
class Homer {  
    char doh(char c) {  
        System.out.println("doh(char)");  
        return 'd';  
    }  
    float doh(float f) {  
        System.out.println("doh(float)");  
        return 1.0f;  
    }  
}  
  
class Milhouse {}  
  
class Bart extends Homer {  
    void doh(Milhouse m) {}  
}  
  
class Hide {  
    public static void main(String[] args) {  
        Bart b = new Bart();  
        b.doh(1); // doh(float) utilisé  
        b.doh('x');  
        b.doh(1.0f);  
        b.doh(new Milhouse());  
    }  
} ///:~
```

Comme nous le verrons dans le prochain chapitre, il est beaucoup plus courant de surcharger les méthodes de même nom et utilisant exactement la même signature et le type retour que dans la classe de base. Sinon cela peut être source de confusion. C'est pourquoi le C++ ne le permet pas, pour empêcher de faire ce qui est probablement une erreur.

Choisir la composition à la place de l'héritage

La composition et l'héritage permettent tous les deux de placer des sous-objets à l'intérieur de votre nouvelle classe. Vous devriez vous demander quelle est la différence entre les deux et quand choisir l'une plutôt que

l'autre.

La composition est généralement utilisée quand on a besoin des caractéristiques d'une classe existante dans une nouvelle classe, mais pas son interface. On inclut un objet donc on peut l'utiliser pour implémenter une fonctionnalité dans la nouvelle classe, mais l'utilisateur de la nouvelle classe voit l'interface qu'on a défini et non celle de l'objet inclus. Pour ce faire, il suffit d'inclure des objets **private** de classes existantes dans la nouvelle classe.

Parfois il est sensé de permettre à l'utilisateur d'une classe d'accéder directement à la composition de notre nouvelle classe ; pour ce faire on déclare les objets membres **public**. Les objets membres utilisent l'implémentation en se cachant les uns des autres, ce qui est une bonne chose. Quand l'utilisateur sait qu'on assemble un ensemble de parties, cela rend l'interface plus facile à comprendre. Un objet **car** (voiture en anglais) est un bon exemple:

```
// ! c06:Car.java  
  
// Composition avec des objets publics.
```

```
class Engine {  
    public void start() {}  
    public void rev() {}  
    public void stop() {}  
}
```

```
class Wheel {  
    public void inflate(int psi) {}  
}
```

```
class Window {  
    public void rollup() {}  
    public void rolldown() {}  
}
```

```
class Door {  
    public Window window = new Window();  
    public void open() {}  
    public void close() {}  
}
```

```

public class Car {

    public Engine engine = new Engine();

    public Wheel[] wheel = new Wheel[4];

    public Door left = new Door(),
           right = new Door(); // 2-door

    public Car() {

        for(int i = 0; i < 4; i++)

            wheel[i] = new Wheel();

    }

    public static void main(String[] args) {

        Car car = new Car();

        car.left.window.rollup();

        car.wheel[0].inflate(72);

    }

} ///:~

```

Du fait que la composition de la voiture fait partie de l'analyse du problème (et non pas simplement de la conception sous-jacente), rendre les membre **publics** aide le programmeur client à comprendre comment utiliser la classe et nécessite moins de complexité de code pour le créateur de la classe. Quoiqu'il en soit, gardez à l'esprit que c'est un cas spécial et qu'en général on devrait définir les champs **privés**.

Quand on hérite, on prend la classe existante et on en fait une version spéciale. En général, cela signifie qu'on prend une classe d'usage général et on l'adapte à un cas particulier. Avec un peu de bon sens, vous verrez que ça n'a pas de sens de composer une voiture en utilisant un objet véhicule. Une voiture ne contient pas un véhicule, *c'est* un véhicule. La relation *est-un* s'exprime avec l'héritage et la relation *a-un* s'exprime avec la composition.

protected

Maintenant que nous avons introduit l'héritage, le mot clé **protected** prend finalement un sens. Dans un monde idéal, les membres **private** devraient toujours être des membres **private** purs et durs, mais dans les projets réels il arrive souvent qu'on veuille cacher quelque chose au monde au sens large et qu'on veuille permettre l'accès pour les membres des classes dérivées. Le mot clé **protected** est un moyen pragmatique de faire. Il dit « Ceci est **private** en ce qui concerne la classe utilisatrice, mais c'est disponible pour quiconque hérite de cette classe ou appartient au même package ». C'est pourquoi **protected** en Java est automatiquement « friendly ».

La meilleure approche est de laisser les membres de données **private**. Vous devriez toujours préserver le droit de changer l'implémentation sous-jacente. Ensuite vous pouvez permettre l'accès contrôlé pour les héritiers de votre classe à travers les méthodes **protected** :

```

// ! c06:Orc.java

// Le mot clé protected .

import java.util.*;

```

```

class Villain {
    private int i;

    protected int read() { return i; }

    protected void set(int ii) { i = ii; }

    public Villain(int ii) { i = ii; }

    public int value(int m) { return m*i; }
}

public class Orc extends Villain {
    private int j;

    public Orc(int jj) { super(jj); j = jj; }

    public void change(int x) { set(x); }
} ///:~

```

On peut voir que **change()** a accès à **set()** parce qu'il est **protected**.

Développement incrémental

Un des avantages de l'héritage est qu'il supporte le *développement incrémental* en permettant d'ajouter du code sans créer de bogues dans le code existant. Ceci permet également d'isoler les nouveaux bogues dans le nouveau code. En héritant d'une classe existante et fonctionnelle et en ajoutant des données membres et des méthodes et en redéfinissant des méthodes existantes, on laisse le code existant - que quelqu'un d'autre peut encore utiliser - inchangé et non bogue. Si un bogue survient, on sait alors qu'il est dans le nouveau code, lequel est beaucoup plus rapide et facile à lire que si on avait modifié le code existant.

Il est plutôt surprenant de voir que les classes sont séparées proprement. Nous n'avons même pas besoin du code source des méthodes afin de pouvoir les utiliser. Au pire, on importe simplement un package. Ceci est vrai à la fois pour l'héritage et la composition.

Il est important de réaliser que le développement d'un programme est un processus incrémental, exactement comme l'apprentissage humain. On peut analyser autant que l'on veut, mais on ne connaîtra pas toutes les réponses au démarrage d'un projet. Vous aurez beaucoup plus de succès et de retour immédiat si vous commencez par faire « grandir » votre projet comme un organisme organique et évolutif plutôt que de le construire comme un gratte-ciel en verre.

Bien que l'héritage pour l'expérimentation puisse être une technique utile, à un moment donné les choses se stabilisent et vous devez jeter un regard neuf à votre hiérarchie de classes pour la réduire en une structure plus logique. Souvenons nous qu'en dessous de tout, l'héritage est utilisé pour exprimer une relation qui dit : « Cette nouvelle classe est *du type de* l'ancienne classe ». Votre programme ne devrait pas être concerné par la manipulation de bits ici ou là, mais par la création et la manipulation d'objets de différents types afin d'exprimer un modèle dans les termes propres à l'espace du problème.

Transtypage ascendant

L'aspect le plus important de l'héritage n'est pas qu'il fournisse des méthodes pour les nouvelles classes. C'est la relation exprimée entre la nouvelle classe et la classe de base. Cette relation peut être résumée en disant : « La nouvelle classe est *un type de* la classe existante ».

Cette description n'est pas simplement une manière amusante d'expliquer l'héritage - c'est supporté directement par le langage. Comme exemple, considérons une classe de base appelée **Instrument** qui représente les instruments de musique et une classe dérivée appelée **Wind**. Puisque l'héritage signifie que toutes les méthodes de la classe de base sont également disponibles pour la classe dérivée, n'importe quel message envoyé à la classe de base peut également être envoyé à la classe dérivée. Si la classe **Instrument** a une méthode **play()**, les instruments **Wind** également. Cela signifie qu'il est exact de dire qu'un objet **Wind** est également un type de **Instrument**. L'exemple suivant montre comment le compilateur implémente cette notion :

```
// ! c06:Wind.java

// Héritage & transtypage ascendant.

import java.util.*;

class Instrument {

    public void play() {}

    static void tune(Instrument i) {

        // ...

        i.play();

    }

}

// Les objets Wind sont des instruments
// parce qu'ils ont la même interface:

class Wind extends Instrument {

    public static void main(String[] args) {

        Wind flute = new Wind();

        Instrument.tune(flute); // Transtypage ascendant

    }

} ///:~
```

Le point intéressant de cet exemple est la méthode **tune()**, qui accepte une référence **Instrument**. Cependant, dans **Wind.main()** la méthode **tune()** est appelée en lui donnant une référence **Wind**. Étant donné que Java est strict au sujet de la vérification de type, il semble étrange qu'une méthode qui accepte un type acceptera littéralement un autre type jusqu'à ce qu'on réalise qu'un objet **Wind** est également un objet **Instrument**, et il n'y a pas de méthode que **tune()** pourrait appeler pour un **Instrument** qui ne serait également dans **Wind**. À

l'intérieur de `tune()`, le code fonctionne pour **Instrument** et tout ce qui dérive de **Instrument**, et l'acte de convertir une référence **Wind** en une référence **Instrument** est appelé *transtypage ascendant*.

Pourquoi le transtypage ascendant ?

La raison de ce terme est historique et basée sur la manière dont les diagrammes d'héritage ont été traditionnellement dessinés : avec la racine au sommet de la page, et grandissant vers le bas. Bien sûr vous pouvez dessiner vos diagrammes de la manière que vous trouvez le plus pratique. Le diagramme d'héritage pour **Wind.java** est :



Transtyper depuis une classe dérivée vers la classe de base nous déplace **vers le haut** dans le diagramme, on fait donc communément référence à un *transtypage ascendant*. Le transtypage ascendant est toujours sans danger parce qu'on va d'un type plus spécifique vers un type plus général. La classe dérivée est un sur-ensemble de la classe de base. Elle peut contenir plus de méthodes que la classe de base, mais elle contient *au moins* les méthodes de la classe de base. La seule chose qui puisse arriver à une classe pendant le transtypage ascendant est de perdre des méthodes et non en gagner. C'est pourquoi le compilateur permet le transtypage ascendant sans transtypage explicite ou une notation spéciale.

On peut également faire l'inverse du transtypage ascendant, appelé *transtypage descendant*, mais cela génère un dilemme qui est le sujet du chapitre 12.

Composition à la place de l'héritage revisité

En programmation orienté objet, la manière la plus probable pour créer et utiliser du code est simplement de mettre des méthodes et des données ensemble dans une classe puis d'utiliser les objets de cette classe. On utilisera également les classes existantes pour construire les nouvelles classes avec la composition. Moins fréquemment on utilisera l'héritage. Donc bien qu'on insiste beaucoup sur l'héritage en apprenant la programmation orientée objet, cela ne signifie pas qu'on doive l'utiliser partout où l'on peut. Au contraire, on devrait l'utiliser avec parcimonie, seulement quand il est clair que l'héritage est utile. Un des moyens les plus clairs pour déterminer si on doit utiliser la composition ou l'héritage est de se demander si on aura jamais besoin de faire un transtypage ascendant de la nouvelle classe vers la classe de base. Si on doit faire un transtypage ascendant, alors l'héritage est nécessaire, mais si on n'a pas besoin de faire un transtypage ascendant, alors il faut regarder avec attention pour savoir si on a besoin de l'héritage. Le prochain chapitre (polymorphisme) fournit une des plus excitantes raisons pour le transtypage ascendant, mais si vous vous rappelez de vous demander « Ai-je besoin de transtypage ascendant ? », vous aurez un bon outil pour décider entre composition et héritage.

Le mot clé final

Le mot clé Java **final** a des sens légèrement différents suivant le contexte, mais en général il signifie « Cela ne peut pas changer ». Vous pourriez vouloir empêcher les changements pour deux raisons : conception ou efficacité. Parce que ces deux raisons sont quelque peu différentes, il est possible de mal utiliser le mot clé **final**.

Les sections suivantes parlent des trois endroits où le mot clé **final** peut être utilisé : données, méthodes et classes.

Données finales

Beaucoup de langages de programmation ont un moyen de dire au compilateur que cette donnée est constante. Une constante est utile pour deux raisons:

1. Elle peut être une *constante lors de la compilation* qui ne changera jamais ;
2. Elle peut être une valeur initialisée à l'exécution qu'on ne veut pas changer.

Dans le cas d'une constante à la compilation, le compilateur inclut « en dur » la valeur de la constante pour tous les calculs où elle intervient ; dans ce cas, le calcul peut être effectué à la compilation, éliminant ainsi un surcoût à l'exécution. En Java, ces sortes de constantes doivent être des primitives et sont exprimées en utilisant le mot-clé **final**. Une valeur doit être donnée au moment de la définition d'une telle constante.

Un champ qui est à la fois **static** et **final** a un emplacement de stockage fixe qui ne peut pas être changé.

Quand on utilise **final** avec des objets références plutôt qu'avec des types primitifs la signification devient un peu confuse. Avec un type primitif, **final** fait de la *valeur* une constante, mais avec un objet référence, **final** fait de la « référence » une constante. Une fois la référence liée à un objet, elle ne peut jamais changer pour pointer vers un autre objet. Quoiqu'il en soit, l'objet lui même peut être modifié ; Java ne fournit pas de moyen de rendre un objet arbitraire une constante. On peut quoiqu'il en soit écrire notre classe de manière que les objets paraissent constants. Cette restriction inclut les tableaux, qui sont également des objets.

Voici un exemple qui montre les champs **final**:

```
// ! c06:FinalData.java

// L'effet de final sur les champs.

class Value {
    int i = 1;
}

public class FinalData {

    // Peut être des constantes à la compilation

    final int i1 = 9;

    static final int VAL_TWO = 99;

    // Constantes publiques typiques:

    public static final int VAL_THREE = 39;

    // Ne peuvent pas être des constantes à la compilation:

    final int i4 = (int)(Math.random()*20);

    static final int i5 = (int)(Math.random()*20);

    Value v1 = new Value();

    final Value v2 = new Value();
```

```

static final Value v3 = new Value();

// Tableaux:
final int[] a = { 1, 2, 3, 4, 5, 6 };

public void print(String id) {
    System.out.println(
        id + " : " + "i4 = " + i4 +
        ", i5 = " + i5);
}

public static void main(String[] args) {
    FinalData fd1 = new FinalData();

    // ! fd1.i1++; // Erreur : on ne peut pas changer la valeur
    fd1.v2.i++; // L'objet n'est pas une constante!
    fd1.v1 = new Value(); // OK -- non final
    for(int i = 0; i < fd1.a.length; i++)
        fd1.a[i]++; // L'objet n'est pas une constante!

    // ! fd1.v2 = new Value(); // Erreur : Ne peut pas
    // ! fd1.v3 = new Value(); // changer la référence
    // ! fd1.a = new int[3];

    fd1.print("fd1");

    System.out.println("Creating new FinalData");

    FinalData fd2 = new FinalData();

    fd1.print("fd1");
    fd2.print("fd2");
}

} ///:~

```

Etant donné que **i1** et **VAL_TWO** sont des primitives **final** ayant une valeur à la compilation, elles peuvent être toutes les deux utilisées comme constantes à la compilation et ne sont pas vraiment différentes. **VAL_THREE** nous montre la manière la plus typique de définir ces constantes : **public** afin qu'elles puissent être utilisées en dehors du package, **static** pour souligner qu'il ne peut y en avoir qu'une seulement, et **final** pour dire que c'est une constante. Notez que les primitives **final static** avec des valeurs initiales constantes (ce sont des constantes à la compilation) sont nommées avec des lettres capitales par convention, avec des mots séparés par des underscores. Ce sont comme des constantes C, d'où cette convention est originaire. Notons également que **i5** ne peut pas être connu à la compilation, donc elle n'est pas en lettres capitales.

Le fait que quelque chose soit **final** ne signifie pas que sa valeur est connue à la compilation. Ceci est montré

par l'initialisation de **i4** et **i5** à l'exécution en utilisant des nombres générés aléatoirement. La portion de cet exemple montre également la différence entre mettre une valeur **final static** ou non **static**. Cette différence n'est visible que quand les valeurs sont initialisées à l'exécution, tandis que les valeurs à la compilation sont traitées de même par le compilateur. Et vraisemblablement optimisées à la compilation. La différence est montrée par la sortie d'une exécution:

```
fd1 : i4 = 15, i5 = 9
Creating new FinalData
fd1 : i4 = 15, i5 = 9
fd2 : i4 = 10, i5 = 9
```

Notez que les valeurs de **i4** pour **fd1** et **fd2** sont uniques, mais la valeur de **i5** n'est pas changée en créant un second objet **FinalData**. C'est parce qu'elle est **static** et initialisée une fois pour toutes lors du chargement et non à chaque fois qu'un nouvel objet est créé.

Les variables **v1** jusqu'à **v4** montre le sens de références **final**. Comme on peut le voir dans **main()**, le fait que **v2** soit **final** ne signifie pas qu'on ne peut pas changer sa valeur. Quoiqu'il en soit, on ne peut pas réaffecter un nouvel objet à **v2**, précisément parce qu'il est **final**. C'est ce que **final** signifie pour une référence. On peut également voir que ce sens reste vrai pour un tableau, qui est une autre sorte de référence. Il n'y a aucun moyen de savoir comment rendre les références du tableau elle-mêmes **final**. Mettre les références **final** semble moins utile que mettre les primitives **final**.

Finals sans initialisation

Java permet la création de *finals sans initialisation*, qui sont des champs déclarés **final**, mais n'ont pas de valeur d'initialisation. Dans tous les cas, un final sans initialisation *doit* être initialisé avant d'être utilisé, et le compilateur doit s'en assurer. Quoiqu'il en soit, les finals sans initialisation fournissent bien plus de flexibilité dans l'usage du mot-clé **final** depuis que, par exemple, un champ **final** à l'intérieur d'une classe peut maintenant être différent pour chaque objet tout en gardant son caractère immuable. Voici un exemple:

```
// ! c06:BlankFinal.java

// Les membres des données final sans initialisation

class Poppet { }

class BlankFinal {

    final int i = 0; // Final initialisé

    final int j; // Final sans initialisation

    final Poppet p; // Référence final sans initialisation

    // Les finals doivent être initialisés

    // dans le constructeur:

    BlankFinal() {
```



```

    j = 1; // Initialise le final sans valeur initiale
    p = new Poppet();
}
BlankFinal(int x) {
    j = x; // Initialise le final sans valeur initiale
    p = new Poppet();
}
public static void main(String[] args) {
    BlankFinal bf = new BlankFinal();
}
} ///:~

```

Vous êtes forcés d'initialiser un **final** soit avec une expression au moment de la définition, soit dans chaque constructeur. De cette manière il est garanti que le champ **final** sera toujours initialisé avant son utilisation.

Arguments final

Java permet de définir les arguments **final** en les déclarant comme tels dans la liste des arguments. Cela signifie qu'à l'intérieur de la méthode on ne peut pas changer ce vers quoi pointe l'argument:

```

// ! c06:FinalArguments.java
// Utilisation de « final » dans les arguments d'une méthode.

class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        // ! g = new Gizmo(); // Illégal -- g est final
    }
    void without(Gizmo g) {
        g = new Gizmo(); // OK -- g n'est pas final
        g.spin();
    }
}

// void f(final int i) { i++; } // Ne peut pas changer
// On peut seulement lire depuis une primitive final:

```

```

int g(final int i) { return i + 1; }

public static void main(String[] args) {

    FinalArguments bf = new FinalArguments();

    bf.without(null);

    bf.with(null);

}

} ///:~

```

A noter qu'on peut encore affecter une référence **null** à un argument qui est final sans que le compilateur ne l'empêche, comme on pourrait le faire pour un argument non-**final**.

Les méthodes **f()** et **g()** montre ce qui arrive quand les arguments primitifs sont **final**: on peut lire l'argument, mais on ne peut pas le changer.

Méthodes final

Les méthodes **final** ont deux raisons d'être. La première est de mettre un « verrou » sur la méthode pour empêcher toute sous-classe de la redéfinir. Ceci est fait pour des raisons de conception quand on veut être sûr que le comportement d'une méthode est préservé durant l'héritage et ne peut pas être redéfini.

La deuxième raison est l'efficacité. Si on met une méthode **final**, on permet au compilateur de convertir tout appel à cette méthode en un appel *incorporé*. Quand le compilateur voit un appel à une méthode **final**, il peut à sa discrétion éviter l'approche normale d'insérer du code pour exécuter l'appel de la méthode (mettre les arguments sur la pile, sauter au code de la méthode et l'exécuter, revenir au code courant et nettoyer les arguments de la pile, s'occuper de la valeur de retour) et à la place remplacer l'appel de méthode avec une copie du code de cette méthode dans le corps de la méthode courante. Ceci élimine le surcoût de l'appel de méthode. Bien entendu, si une méthode est importante, votre code commencera alors à grossir et vous ne verrez plus le gain de performance dû au code « incorporé », parce que toute amélioration sera cachée par le temps passé à l'intérieur de la méthode. Ceci implique que le compilateur Java est capable de détecter ces situations et de choisir sagement si oui ou non il faut « incorporer » une méthode **final**. Quoiqu'il en soit, il est mieux de ne pas faire confiance à ce que peut faire le compilateur et de mettre une méthode **final** seulement si elle est plutôt petite ou si on veut explicitement empêcher la surcharge.

final et private

Toutes les méthodes **private** sont implicitement **final**. Parce qu'on ne peut pas accéder à une méthode **private**, on ne peut pas la surcharger (même si le compilateur ne donne pas de messages d'erreur si on essaye de la redéfinir, on ne redéfinit pas la méthode, on a simplement créé une nouvelle méthode). On peut ajouter le mot-clé **final** à une méthode **private**, mais ça n'apporte rien de plus.

Ce problème peut rendre les choses un peu confuses, parce que si on essaye de surcharger une méthode **private** qui est implicitement **final** ça semble fonctionner:

```

// ! c06:FinalOverridingIllusion.java

// C'est seulement une impression qu'on peut

// surcharger une méthode private ou private final.

```

```
class WithFinals {  
    // Identique à « private » tout seul:  
    private final void f() {  
        System.out.println("WithFinals.f()");  
    }  
    // Également automatiquement « final »:  
    private void g() {  
        System.out.println("WithFinals.g()");  
    }  
}  
  
class OverridingPrivate extends WithFinals {  
    private final void f() {  
        System.out.println("OverridingPrivate.f()");  
    }  
    private void g() {  
        System.out.println("OverridingPrivate.g()");  
    }  
}  
  
class OverridingPrivate2  
    extends OverridingPrivate {  
    public final void f() {  
        System.out.println("OverridingPrivate2.f()");  
    }  
    public void g() {  
        System.out.println("OverridingPrivate2.g()");  
    }  
}  
  
public class FinalOverridingIllusion {  
    public static void main(String[] args) {  
        OverridingPrivate2 op2 =
```

```

        new OverridingPrivate2();

    op2.f();

    op2.g();

    // On peut faire un transtypage ascendant:

    OverridingPrivate op = op2;

    // Mais on ne peut pas appeler les méthodes:

    // ! op.f();

    // ! op.g();

    // Idem ici:

    WithFinals wf = op2;

    // ! wf.f();

    // ! wf.g();

}

} ///:~

```

« Surcharger » peut seulement arriver si quelque chose fait partie de l'interface de la classe de base. On doit être capable de faire un transtypage ascendant vers la classe de base et d'appeler la même méthode. Ce point deviendra clair dans le prochain chapitre. Si une méthode est **private**, elle ne fait pas partie de l'interface de la classe de base. C'est simplement du code qui est caché à l'intérieur de la classe, et il arrive simplement qu'elle a ce nom, mais si on définit une méthode **public**, **protected** ou « amies » dans la classe dérivée, il n'y a aucune connexion avec la méthode de même nom dans la classe de base. Étant donné qu'une méthode **private** est inatteignable et effectivement invisible, elle ne sert à rien d'autre qu'à l'organisation du code dans la classe dans laquelle elle est définie.

Classes final

Quand on dit qu'une classe entière est **final** (en faisant précéder sa définition par le mot-clé **final**) on stipule qu'on ne veut pas hériter de cette classe ou permettre à qui que ce soit de le faire. En d'autres mots, soit la conception de cette classe est telle qu'on n'aura jamais besoin de la modifier, soit pour des raisons de sûreté ou de sécurité on ne veut pas qu'elle soit sous-classée. Ou alors, on peut avoir affaire à un problème d'efficacité, et on veut s'assurer que toute activité impliquant des objets de cette classe sera aussi efficace que possible.

```

// ! c06:Jurassic.java

// Mettre une classe entière final.

class SmallBrain {}

final class Dinosaur {

    int i = 7;

```

```

    int j = 1;

    SmallBrain x = new SmallBrain();

    void f() {}
}

// ! class Further extends Dinosaur {}
// erreur : Ne peut pas étendre la classe final 'Dinosaur'

public class Jurassic {

    public static void main(String[] args) {

        Dinosaur n = new Dinosaur();

        n.f();

        n.i = 40;

        n.j++;

    }

} ///:~

```

Notons que les données membres peuvent ou non être **final**, comme on le choisit. Les mêmes règles s'appliquent à **final** pour les données membres sans tenir compte du fait que la classe est ou non **final**. Définir une classe comme **final** empêche simplement l'héritage - rien de plus. Quoiqu'il en soit, parce que cela empêche l'héritage, toutes les méthodes d'une classe **final** sont implicitement **final**, étant donné qu'il n'y a aucun moyen de les surcharger. Donc le compilateur a les mêmes options d'efficacité que si on définissait explicitement une méthode **final**.

On peut ajouter le modificateur **final** à une méthode dans une classe **final**, mais ça ne rajoute aucune signification.

Attention finale

Il peut paraître raisonnable de déclarer une méthode **final** alors qu'on conçoit une classe. On peut décider que l'efficacité est très importante quand on utilise la classe et que personne ne pourrait vouloir surcharger les méthodes de toute manière. Cela est parfois vrai.

Mais il faut être prudent avec ces hypothèses. En général, il est difficile d'anticiper comment une classe va être réutilisée, surtout une classe générique. Si on définit une méthode comme **final**, on devrait empêcher la possibilité de réutiliser cette classe par héritage dans les projets d'autres programmeurs simplement parce qu'on ne pourrait pas l'imaginer être utilisée de cette manière.

La bibliothèque standard de Java en est un bon exemple. En particulier, la classe **Vector** en Java 1.0/1.1 était communément utilisée et pourrait avoir encore été plus utile si, au nom de l'efficacité, toutes les méthodes n'avaient pas été **final**. Ça paraît facilement concevable que l'on puisse vouloir hériter et surcharger une telle classe fondamentale, mais les concepteurs d'une manière ou d'une autre ont décidé que ce n'était pas approprié. C'est ironique pour deux raisons. Premièrement, **Stack** hérite de **Vector**, ce qui dit qu'une **Stack** est un **Vector**,

ce qui n'est pas vraiment vrai d'un point de vue logique. Deuxièmement, beaucoup des méthodes importantes de **Vector**, telles que **addElement()** et **elementAt()** sont **synchronized**. Comme nous verrons au chapitre 14, ceci implique un surcoût significatif au niveau des performances qui rend caduque tout gain fourni par **final**. Ceci donne de la crédibilité à la théorie que les programmeurs sont constamment mauvais pour deviner où les optimisations devraient être faites. Il est vraiment dommage qu'une conception aussi maladroite ait fait son chemin dans la bibliothèque standard que nous utilisons tous. Heureusement, la bibliothèque de collections Java 2 remplace **Vector** avec **ArrayList**, laquelle se comporte bien mieux. Malheureusement, il y a encore beaucoup de code nouveau écrit qui utilise encore l'ancienne bibliothèques de collections.

Il est intéressant de noter également que **Hashtable**, une autre classe importante de la bibliothèque standard, n'a pas une seule méthode **final**. Comme mentionné à plusieurs endroits dans ce livre, il est plutôt évident que certaines classes ont été conçues par des personnes totalement différentes. Vous verrez que les noms de méthodes dans **Hashtable** sont beaucoup plus court comparés à ceux de **Vector**, une autre preuve. C'est précisément ce genre de choses qui ne devrait pas être évident aux utilisateurs d'une bibliothèque de classes. Quand plusieurs choses sont inconsistantes, cela fait simplement plus de travail pour l'utilisateur. Encore un autre grief à la valeur de la conception et de la qualité du code. À noter que la bibliothèque de collection de Java 2 remplace **Hashtable** avec **HashMap**.

Initialisation et chargement de classes

Dans des langages plus traditionnels, les programmes sont chargés tout d'un coup au moment du démarrage. Ceci est suivi par l'initialisation et ensuite le programme commence. Le processus d'initialisation dans ces langages doit être contrôlé avec beaucoup d'attention afin que l'ordre d'initialisation des **statics** ne pose pas de problème. C++, par exemple, a des problèmes si une **static** attend qu'une autre **static** soit valide avant que la seconde ne soit initialisée.

Java n'a pas ce problème parce qu'il a une autre approche du chargement. Parce que tout en Java est un objet, beaucoup d'actions deviennent plus facile, et ceci en est un exemple. Comme vous l'apprendrez plus complètement dans le prochain chapitre, le code compilé de chaque classe existe dans son propre fichier séparé. Ce fichier n'est pas chargé tant que ce n'est pas nécessaire. En général, on peut dire que « le code d'une classe est chargé au moment de la première utilisation ». C'est souvent au moment où le premier objet de cette classe est construit, mais le chargement se produit également lorsqu'on accède à un champ **static** ou une méthode **static**.

Le point de première utilisation est également là où l'initialisation des **statics** a lieu. Tous les objets **static** et le bloc de code **static** sera initialisé dans l'ordre textuel (l'ordre dans lequel ils sont définis dans la définition de la classe) au moment du chargement. Les **statics**, bien sûr, ne sont initialisés qu'une seule fois.

Initialisation avec héritage

Il est utile de regarder l'ensemble du processus d'initialisation, incluant l'héritage pour obtenir une compréhension globale de ce qui se passe. Considérons le code suivant:

```
// ! c06:Beetle.java

// Le processus complet d'initialisation.

class Insect {

    int i = 9;

    int j;
```

```
Insect() {  
    prt("i = " + i + ", j = " + j);  
    j = 39;  
}  
  
static int x1 =  
    prt("static Insect.x1 initialisé");  
  
static int prt(String s) {  
    System.out.println(s);  
    return 47;  
}  
}  
  
public class Beetle extends Insect {  
    int k = prt("Beetle.k initialisé");  
    Beetle() {  
        prt("k = " + k);  
        prt("j = " + j);  
    }  
    static int x2 =  
        prt("static Beetle.x2 initialisé");  
    public static void main(String[] args) {  
        prt("Constructeur Beetle");  
        Beetle b = new Beetle();  
    }  
} ///:~
```

La sortie de ce programme est:

```
static Insect.x1 initialisé  
static Beetle.x2 initialisé  
Constructeur Beetle  
i = 9, j = 0  
Beetle.k initialisé  
k = 47
```

j = 39

La première chose qui se passe quand on exécute **Beetle** en Java est qu'on essaye d'accéder à **Beetle.main()** (une méthode **static**), donc le chargeur cherche et trouve le code compilé pour la classe **Beetle** (en général dans le fichier appelé **Beetle.class**). Dans le processus de son chargement, le chargeur remarque qu'elle a une classe de base (c'est ce que le mot-clé **extends** veut dire), laquelle est alors chargée. Ceci se produit qu'on construise ou non un objet de la classe de base. Essayez de commenter la création de l'objet pour vous le prouver.

Si la classe de base a une classe de base, cette seconde classe de base sera à son tour chargée, etc. Ensuite, l'initialisation **static** dans la classe de base racine (dans ce cas, **Insect**) est effectuée, ensuite la prochaine classe dérivée, etc. C'est important parce que l'initialisation static de la classe dérivée pourrait dépendre de l'initialisation correcte d'un membre de la classe de base.

À ce point, les classes nécessaires ont été chargées, donc l'objet peut être créé. Premièrement, toutes les primitives dans l'objet sont initialisées à leurs valeurs par défaut et les références objets sont initialisées à **null** - ceci se produit en une seule passe en mettant la mémoire dans l'objet au zéro binaire. Ensuite le constructeur de la classe de base est appelé. Dans ce cas, l'appel est automatique, mais on peut également spécifier le constructeur de la classe de base (comme la première opération dans le constructeur **Beetle()** en utilisant **super**. La constructeur de la classe de base suit le même processus dans le même ordre que le constructeur de la classe dérivée. Lorsque le constructeur de la classe de base a terminé, les variables d'instance sont initialisées dans l'ordre textuel. Finalement le reste du corps du constructeur est exécuté.

Résumé

L'héritage et la composition permettent tous les deux de créer de nouveaux types depuis des types existants. Typiquement, quoiqu'il en soit, on utilise la composition pour réutiliser des types existants comme partie de l'implémentation sous-jacente du nouveau type, et l'héritage quand on veut réutiliser l'interface. Étant donné que la classe dérivée possède l'interface de la classe de base, on peut faire un **transtypage ascendant** vers la classe de base, lequel est critique pour le polymorphisme, comme vous le verrez dans le prochain chapitre.

En dépit de l'importance particulièrement forte de l'héritage dans la programmation orienté objet, quand on commence une conception on devrait généralement préférer la composition durant la première passe et utiliser l'héritage seulement quand c'est clairement nécessaire. La composition tend à être plus flexible. De plus, par le biais de l'héritage, vous pouvez changer le type exact de vos objets, et donc, le comportement, de ces objets membres à l'exécution. Par conséquent, on peut changer le comportement d'objets composés à l'exécution.

Bien que la réutilisation du code à travers la composition et l'héritage soit utile pour un développement rapide, on voudra généralement concevoir à nouveau la hiérarchie de classes avant de permettre aux autres programmeurs d'en devenir dépendant. Votre but est une hiérarchie dans laquelle chaque classe a un usage spécifique et n'est ni trop grosse (englobant tellement de fonctionnalités qu'elle en est difficile à manier pour être réutilisée) ni trop ennuyeusement petite (on ne peut pas l'utiliser par elle-même ou sans ajouter de nouvelles fonctionnalités).

Exercices

Les solutions aux exercices sélectionnés peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût depuis www.BruceEckel.com.

1. Créer deux classes, **A** et **B**, avec des constructeurs par défaut (liste d'arguments vide) qui s'annoncent eux-même. Faire hériter une nouvelle classe **C** de **A**, et créer une classe membre **B** à l'intérieur de **C**. Ne pas créer un constructeur pour **C**. Créer un objet d'une classe **C** et observer les résultats.

2. Modifier l'exercice 1 afin que **A** et **B** aient des constructeurs avec arguments au lieu de constructeurs par défaut. Écrire un constructeur pour **C** et effectuer toutes les initialisations à l'intérieur du constructeur de **C**.
3. Créer une simple classe. À l'intérieur d'une seconde classe, définir un champ pour un objet de la première classe. Utiliser l'initialisation paresseuse pour instancier cet objet.
4. Hériter une nouvelle classe de la classe **Detergent**. Redéfinir **scrub()** et ajouter une nouvelle méthode appelée **sterilize()**.
5. Prendre le fichier **Cartoon.java** et enlever le commentaire autour du constructeur de la classe **Cartoon**. Expliquer ce qui arrive.
6. Prendre le fichier **Chess.java** et enlever le commentaire autour du constructeur de la classe **Chess**. Expliquer ce qui se passe.
7. Prouver que des constructeurs par défaut sont créés pour vous par le compilateur.
8. Prouver que les constructeurs de la classe de base sont (a) toujours appelés et (b) appelés avant les constructeurs des classes dérivées.
9. Créer une classe de base avec seulement un constructeur qui ne soit pas un constructeur par défaut, et une classe dérivée avec à la fois un constructeur par défaut et un deuxième constructeur. Dans les constructeurs de la classe dérivée, appeler le constructeur de la classe de base.
10. Créer une classe appelée **Root** qui contient une instance de chaque classe (que vous aurez également créé) appelées **Component1**, **Component2**, et **Component3**. Dériver une classe **Stem** de **Root** qui contienne également une instance de chaque « component ». Toutes les classes devraient avoir un constructeur par défaut qui affiche un message au sujet de cette classe.
11. Modifier l'exercice 10 de manière à ce que chaque classe ait des constructeurs qui ne soient pas des constructeurs par défaut.
12. Ajouter une hiérarchie propre de méthodes **cleanup()** à toutes les classes dans l'exercice 11.
13. Créer une classe avec une méthode surchargée trois fois. Hériter une nouvelle classe, ajouter une nouvelle surcharge de la méthode et montrer que les quatre méthodes sont disponibles dans la classe dérivée.
14. Dans **Car.java** ajouter une méthode **service()** à **Engine** et appeler cette méthode dans **main()**.
15. Créer une classe à l'intérieur d'un package. Cette classe doit contenir une méthode **protected**. À l'extérieur du package, essayer d'appeler la méthode **protected** et expliquer les résultats. Maintenant hériter de cette classe et appeler la méthode **protected** depuis l'intérieur d'une méthode de la classe dérivée.
16. Créer une classe appelée **Amphibian**. De celle-ci, hériter une classe appelée **Frog**. Mettre les méthodes appropriées dans la classe de base. Dans **main()**, créer une **Frog** et faire un transtypage ascendant **Amphibian**, et démontrer que toutes les méthodes fonctionnent encore.
17. Modifier l'exercice 16 de manière que **Frog** redéfinisse les définitions de méthodes de la classe de base (fournir de nouvelles définitions utilisant les mêmes signatures des méthodes). Noter ce qui se passe dans **main()**.
18. Créer une classe avec un champ **static final** et un champ **final** et démontrer la différence entre les deux.
19. Créer une classe avec une référence **final** sans initialisation vers un objet. Exécuter l'initialisation de cette **final** sans initialisation à l'intérieur d'une méthode (pas un constructeur) juste avant de l'utiliser. Démontrer la garantie que le **final** doit être initialisé avant d'être utilisé, et ne peut pas être changé une fois initialisé.
20. Créer une classe avec une méthode **final**. Hériter de cette classe et tenter de redéfinir cette méthode.
21. Créer une classe **final** et tenter d'en hériter.
22. Prouver que le chargement d'une classe n'a lieu qu'une fois. Prouver que le chargement peut être causé soit par la création de la première instance de cette classe, soit par l'accès à un membre **static**.
23. Dans **Beetle.java**, hériter un type spécifique de coccinelle de la classe **Beetle**, suivant le même format des classes existantes. Regarder et expliquer le flux de sortie du programme.