

Traducteur : Jérôme DANNOVILLE

21.06.01 - Version 1.1 :

- Correction du code html,
guillemets "texte" -> « texte »,
texte: -> texte :,
texte? -> texte ? [Armel].

14.06.01 - Version 1.0 :

- Relecture en cours [JeromeDanno].

Texte original :

- Thinking in Java, 2nd edition, Revision 10
© 2000 by Bruce Eckel

7: Polymorphisme

Le polymorphisme est la troisième caractéristique essentielle d'un langage de programmation orienté objet, après l'abstraction et l'héritage.

Le polymorphisme fournit une autre dimension séparant la partie interface de l'implémentation qui permet de découpler le *quoi* du *comment*. Le polymorphisme améliore l'organisation du code et sa lisibilité de même qu'il permet la création de programmes *extensible* qui peuvent évoluer non seulement pendant la création initiale du projet mais également quand des fonctions nouvelles sont désirées.

L'encapsulation crée de nouveaux types de données en combinant les caractéristiques et les comportements. Cacher la mise en &oeil;uvre permet de séparer l'interface de l'implémentation en mettant les détails privés [**private**]. Cette sorte d'organisation mécanique est bien comprise par ceux viennent de la programmation procédurale. Mais le polymorphisme s'occupe de découpler au niveau des *types*. Dans le chapitre précédent, nous avons vu comment l'héritage permet le traitement d'un objet comme son propre type *ou* son type de base. Cette capacité est critique car elle permet à beaucoup de types (dérivé d'un même type de base) d'être traités comme s'ils n'étaient qu'un type, et permet à un seul morceau de code de traiter sans distinction tous ces types différents. L'appel de méthode polymorphe permet à un type d'exprimer sa distinction par rapport à un autre, un type semblable, tant qu'ils dérivent tous les deux d'un même type de base. Cette distinction est exprimée à travers des différences de comportement des méthodes que vous pouvez appeler par la classe de base.

Dans ce chapitre, vous allez comprendre le polymorphisme [également appelé en anglais *dynamic binding* ou *late binding* ou encore *run-time binding*] en commençant par les notions de base, avec des exemples simples qui évacuent tout ce qui ne concerne pas le comportement polymorphe du programme.

Upcasting

Dans le chapitre 6 nous avons vu qu'un objet peut être manipulé avec son propre type ou bien comme un objet de son type de base. Prendre la référence d'un objet et l'utiliser comme une référence sur le type de base est appelé *upcasting* (*transtypage* en français), en raison du mode de représentation des arbres d'héritages avec la classe de base en haut.

On avait vu le problème repris ci-dessous apparaître:

```
//: c07:music:Music.java
// Héritage & upcasting.
class Note {
    private int value;

    private Note(int val) { value = val; }

    public static final Note
        MIDDLE_C = new Note(0),
        C_SHARP   = new Note(1),
        B_FLAT    = new Note(2);
} // Etc.

class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}

// Les objets Wind sont des instruments
// car ils ont la même interface:
class Wind extends Instrument {
    // Redéfinition de la méthode de l'interface:
    public void play(Note n) {
        System.out.println("Wind.play()");
    }
}

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }

    public static void main(String[] args) {
        Wind flute = new Wind();
```

```

    tune(flute); // Upcasting
}
} ///:~

```

La méthode **Music.tune()** accepte une référence sur un **Instrument**, mais également sur tout ce qui dérive de **Instrument**. Dans le **main()**, ceci se matérialise par une référence sur un objet **Wind** qui est passée à **tune()**, sans qu'un changement de type (un cast) soit nécessaire. Ceci est correct; l'interface dans **Instrument** doit exister dans **Wind**, car **Wind** hérite de la classe **Instrument**. Utiliser l'upcast de **Wind** vers **Instrument** peut « rétrécir » cette interface, mais au pire elle est réduite à l'interface complète **dInstrument**.

Pourquoi utiliser l'upcast?

Ce programme pourrait vous sembler étrange. Pourquoi donc *oublier* intentionnellement le type d'un objet? C'est ce qui arrive quand on fait un upcast, et il semble beaucoup plus naturel que **tune()** prenne tout simplement une référence sur **Wind** comme argument. Ceci introduit un point essentiel: en faisant ça, il faudrait écrire une nouvelle méthode **tune()** pour chaque type de **Instrument** du système. Supposons que l'on suive ce raisonnement et que l'on ajoute les instruments à cordes [**Stringed**] et les cuivres [**Brass**]:

```

//: c07:music2:Music2.java
// Surcharger plutôt que d'utiliser l'upcast.

```

```

class Note {
    private int value;

    private Note(int val) { value = val; }

    public static final Note
        MIDDLE_C = new Note(0),
        C_SHARP = new Note(1),
        B_FLAT = new Note(2);
} // Etc.

class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}

class Wind extends Instrument {
    public void play(Note n) {

```

```
        System.out.println("Wind.play()");
    }
}

class Stringed extends Instrument {
    public void play(Note n) {
        System.out.println("Stringed.play()");
    }
}

class Brass extends Instrument {
    public void play(Note n) {
        System.out.println("Brass.play()");
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // Pas d' upcast
        tune(violin);
        tune(frenchHorn);
    }
}
```

```
} ///:~
```

Ceci fonctionne, mais avec un inconvénient majeur: il vous faut écrire des classes spécifique à chaque ajout d'une classe **Instrument**. Ceci implique davantage de programmation dans un premier temps, mais également beaucoup de travail à venir si l'on désire ajouter une nouvelle méthode comme **tune()** ou un nouveau type d'**Instrument**. Sans parler du compilateur qui est incapable de signaler l'oubli de surcharge de l'une de vos méthodes qui fait que toute cette construction utilisant les types devient assez compliquée.

Ne serait-il pas plus commode d'écrire une seule méthode qui prenne la classe de base en argument plutôt que toutes les classes dérivées spécifiques? Ou encore, ne serait-il pas agréable d'oublier qu'il y a des classes dérivées et d'écrire votre code en ne s'adressant qu'à la classe de base?

C'est exactement ce que le polymorphisme vous permet de faire. Souvent, ceux qui viennent de la programmation procédurale sont déroutés par le mode de fonctionnement du polymorphisme.

The twist

L'ennui avec **Music.java** peut être visualisé en exécutant le programme. L'output est **Wind.play()**. C'est bien sûr le résultat attendu, mais il n'est pas évident de comprendre le fonctionnement.. Examinons la méthode **tune()**:

```
public static void tune(Instrument i) {  
  
    // ...  
  
    i.play(Note.MIDDLE_C);  
  
}
```

Elle prend une référence sur un **Instrument** en argument. Comment le compilateur peut-il donc deviner que cette référence sur un **Instrument** pointe dans le cas présent sur un objet **Wind** et non pas un objet **Brass** ou un objet **Stringed**? Hé bien il ne peut pas. Mieux vaut examiner le mécanisme d'*association [binding]* pour bien comprendre la question soulevée.

Liaison de l'appel de méthode

Raccorder un appel de méthode avec le corps de cette méthode est appelé *association*. Quand cette association est réalisée avant l'exécution du programme (par le compilateur et l'éditeur de lien, s'il y en a un), c'est de l'*association prédéfinie*. Vous ne devriez pas avoir déjà entendu ce terme auparavant car avec les langages procéduraux, c'est imposé. Les compilateurs C n'ont qu'une sorte d'appel de méthode, l'association prédéfinie.

Ce qui déroute dans le programme ci-dessus tourne autour de l'association prédéfinie car le compilateur ne peut pas connaître la bonne méthode à appeler lorsqu'il ne dispose que d'une référence sur **Instrument**.

La solution s'appelle l'*association tardive*, ce qui signifie que l'association est effectuée à l'exécution en se basant sur le type de l'objet. L'association tardive est également appelée *association dynamique [dynamic binding ou run-time binding]*. Quand un langage implémente l'association dynamique, un mécanisme doit être prévu pour déterminer le type de l'objet lors de l'exécution et pour appeler ainsi la méthode appropriée. Ce qui veut dire que le compilateur ne connaît toujours pas le type de l'objet, mais le mécanisme d'appel de méthode trouve et effectue l'appel vers le bon corps de méthode. Les mécanismes d'association tardive varient selon les

langages, mais vous pouvez deviner que des informations relatives au type doivent être implantées dans les objets.

Toutes les associations de méthode en Java utilisent l'association tardive à moins que l'on ait déclaré une méthode **final**. Cela signifie que d'habitude vous n'avez pas à vous préoccuper du déclenchement de l'association tardive, cela arrive automatiquement.

Pourquoi déclarer une méthode avec **final**? On a vu dans le chapitre précédant que cela empêche quelqu'un de redéfinir cette méthode. Peut-être plus important, cela « coupe » effectivement l'association dynamique, ou plutôt cela indique au compilateur que l'association dynamique n'est pas nécessaire. Le compilateur génère du code légèrement plus efficace pour les appels de méthodes spécifiés **final**. Cependant, dans la plupart des cas cela ne changera pas la performance globale de votre programme; mieux vaut utiliser **final** à la suite d'une décision de conception, et non pas comme tentative d'amélioration des performances.

Produire le bon comportement

Quand vous savez qu'en Java toute association de méthode se fait de manière polymorphe par l'association tardive, vous pouvez écrire votre code en vous adressant à la classe de base en sachant que tous les cas des classes dérivées fonctionneront correctement avec le même code. Dit autrement, vous « envoyez un message à un objet et laissez l'objet trouver le comportement adéquat. »

L'exemple classique utilisée en POO est celui de la forme [shape]. Cet exemple est généralement utilisé car il est facile à visualiser, mais peut malheureusement sous-entendre que la POO est cantonnée au domaine graphique, ce qui n'est bien sûr pas le cas.

Dans cet exemple il y a une classe de base appelée **Shape** et plusieurs types dérivés: **Circle**, **Square**, **Triangle**, etc. Cet exemple marche très bien car il est naturel de dire qu'un cercle est « une sorte de forme. » Le diagramme d'héritage montre les relations :



l'upcast pourrait se produire dans une instruction aussi simple que :

```
Shape s = new Circle();
```

On crée un objet **Circle** et la nouvelle référence est assignée à un **Shape**, ce qui semblerait être une erreur (assigner un type à un autre), mais qui est valide ici car un Cercle [**Circle**] *est* par héritage une sorte de forme [**Shape**]. Le compilateur vérifie la légalité de cette instruction et n'émet pas de message d'erreur.

Supposons que vous appeliez une des méthode de la classe de base qui a été redéfinie dans les classes dérivées :

```
s.draw();
```

De nouveau, vous pourriez vous attendre à ce que la méthode **draw()** de **Shape** soit appelée parce que c'est après tout une référence sur **Shape**. Alors comment le compilateur peut-il faire une autre liaison? Et malgré tout

le bon **Circle.draw()** est appelé grâce à la liaison tardive (polymorphisme).

L'exemple suivant le montre de manière légèrement différente :

```
//: c07:Shapes.java
// Polymorphisme en Java.

class Shape {
    void draw() {}
    void erase() {}
}

class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}

class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
```

```

        System.out.println("Triangle.erase()");
    }
}

public class Shapes {
    public static Shape randShape() {
        switch((int)(Math.random() * 3)) {
            default:

                case 0: return new Circle();
                case 1: return new Square();
                case 2: return new Triangle();
        }
    }

    public static void main(String[] args) {
        Shape[] s = new Shape[9];

        // Remplissage du tableau avec des formes [shapes]:
        for(int i = 0; i < s.length; i++)
            s[i] = randShape();

        // Appel polymorphe des méthodes:
        for(int i = 0; i < s.length; i++)
            s[i].draw();
    }
} ///:~

```

La classe de base **Shape** établit l'interface commune pour tout ce qui hérite de **Shape** — C'est à dire, toutes les formes (shapes en anglais) peuvent être dessinées [draw] et effacées [erase]. Les classes dérivées redéfinissent ces méthodes pour fournir un comportement unique pour chaque type de forme spécifique.

La classe principale **Shapes** contient une méthode statique **randShape()** qui rend une référence sur un objet sélectionné de manière aléatoire à chaque appel. Remarquez que la généralisation se produit sur chaque instruction **return**, qui prend une référence sur un cercle [circle], un carré [square], ou un triangle et la retourne comme le type de retour de la méthode, en l'occurrence **Shape**. Ainsi à chaque appel de cette méthode vous ne pouvez pas voir quel type spécifique vous obtenez, puisque vous récupérez toujours une simple référence sur **Shape**.

Le **main()** a un tableau de références sur **Shape** remplies par des appels à **randShape()**. Tout ce que l'on sait dans la première boucle c'est que l'on a des objets formes [**Shapes**], mais on ne sait rien de plus (pareil pour le compilateur). Cependant, quand vous parcourez ce tableau en appelant **draw()** pour chaque référence dans la

seconde boucle, le bon comportement correspondant au type spécifique se produit comme par magie, comme vous pouvez le constater sur l'output de l'exemple :

```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

Comme toutes les formes sont choisies aléatoirement à chaque fois, vous obtiendrez bien sûr des résultats différents. L'intérêt de choisir les formes aléatoirement est d'illustrer le fait que le compilateur ne peut avoir aucune connaissance spéciale lui permettant de générer les appels corrects au moment de la compilation. Tous les appels à **draw()** sont réalisés par liaison dynamique.

Extensibilité

Revenons maintenant à l'exemple sur l'instrument de musique. En raison du polymorphisme, vous pouvez ajouter autant de nouveaux types que vous voulez dans le système sans changer la méthode **tune()**. Dans un programme orienté objet bien conçu, la plupart ou même toutes vos méthodes suivront le modèle de **tune()** et communiqueront seulement avec l'interface de la classe de base. Un tel programme est *extensible* parce que vous pouvez ajouter de nouvelles fonctionnalités en héritant de nouveaux types de données de la classe de base commune. Les méthodes qui utilisent l'interface de la classe de base n'auront pas besoin d'être retouchées pour intégrer de nouvelles classes.

Regardez ce qui se produit dans l'exemple de l'instrument si vous ajoutez des méthodes dans la classe de base et un certain nombre de nouvelles classes. Voici le schéma :



Toutes ces nouvelles classes fonctionnent correctement avec la vieille méthode **tune()**, sans modification. Même si **tune()** est dans un fichier séparé et que de nouvelles méthodes sont ajoutées à l'interface de **Instrument**, **tune()** fonctionne correctement sans recompilation. Voici l'implémentation du diagramme présenté ci-dessus :

```
//: c07:music3:Music3.java
// Un programme extensible.
```

```
import java.util.*;

class Instrument {
    public void play() {
        System.out.println("Instrument.play()");
    }
    public String what() {
        return "Instrument";
    }
    public void adjust() {}
}

class Wind extends Instrument {
    public void play() {
        System.out.println("Wind.play()");
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play() {
        System.out.println("Percussion.play()");
    }
    public String what() { return "Percussion"; }
    public void adjust() {}
}

class Stringed extends Instrument {
    public void play() {
        System.out.println("Stringed.play()");
    }
    public String what() { return "Stringed"; }
```

```
    public void adjust() {}
}

class Brass extends Wind {
    public void play() {
        System.out.println("Brass.play()");
    }
    public void adjust() {
        System.out.println("Brass.adjust()");
    }
}

class Woodwind extends Wind {
    public void play() {
        System.out.println("Woodwind.play()");
    }
    public String what() { return "Woodwind"; }
}

public class Music3 {
    // Indépendants des types, ainsi les nouveaux types
    // ajoutés au système marchent toujours bien:
    static void tune(Instrument i) {
        // ...
        i.play();
    }
    static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        Instrument[] orchestra = new Instrument[5];
        int i = 0;
        // Upcasting pendant l'ajout au tableau:
    }
}
```

```

    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();

    tuneAll(orchestra);
}
} ///:~

```

Les nouvelles méthodes sont **what()**, qui renvoie une référence sur une **String** décrivant la classe, et **adjust()**, qui fournit un moyen d'ajuster chaque instrument.

Dans le **main()**, quand on met quelque chose dans le tableau d' **Instrument**, on upcast automatiquement en **Instrument**.

Vous pouvez constater que la méthode **tune()** ignore fort heureusement tous les changements qui sont intervenus autour d'elle, et pourtant cela marche correctement. C'est exactement ce que le polymorphisme est censé fournir. Vos modifications ne peuvent abîmer les parties du programme qui ne devraient pas être affectées. Dit autrement, le polymorphisme est une des techniques majeures permettant au programmeur de « séparer les choses qui changent des choses qui restent les mêmes. »

Redéfinition et Surcharge

Regardons sous un angle différent le premier exemple de ce chapitre. Dans le programme suivant, l'interface de la méthode **play()** est changée dans le but de la redéfinir, ce qui signifie que vous n'avez pas *redéfinie* la méthode, mais plutôt *surchargée*. Le compilateur vous permet de surcharger des méthodes, il ne proteste donc pas. Mais le comportement n'est probablement pas celui que vous vouliez. Voici l'exemple :

```

///: c07:WindError.java
// Changement accidentel de l'interface.

class NoteX {
    public static final int
        MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX {
    public void play(int NoteX) {
        System.out.println("InstrumentX.play()");
    }
}

```

```

class WindX extends InstrumentX {

    // OUPS! L'interface de la méthode change:

    public void play(NoteX n) {

        System.out.println("WindX.play(NoteX n)");

    }

}

public class WindError {

    public static void tune(InstrumentX i) {

        // ...

        i.play(NoteX.MIDDLE_C);

    }

    public static void main(String[] args) {

        WindX flute = new WindX();

        tune(flute); // Ce n'est pas le comportement souhaité!

    }

} ///:~

```

Il y a un autre aspect déroutant ici. Dans **InstrumentX**, la méthode **play()** a pour argument un **int** identifié par **NoteX**. Bien que **NoteX** soit un nom de classe, il peut également être utilisé comme identificateur sans erreur. Mais dans **WindX**, **play()** prend une référence de **NoteX** qui a pour identificateur **n** (bien que vous puissiez même écrire **play(NoteX NoteX)** sans erreur). En fait, il s'avère que le programmeur a désiré redéfinir **play()** mais s'est trompé de type. Du coup le compilateur a supposé qu'une surcharge était souhaitée et non pas une redéfinition. Remarquez que si vous respectez la convention standard de nommage Java, l'identificateur d'argument serait **noteX** ('n' minuscule), ce qui le distinguerait du nom de la classe.

Dans **tune**, le message **play()** est envoyé à l'**InstrumentX i**, avec comme argument un de membres de **NoteX** (**MIDDLE_C**). Puisque **NoteX** contient des définitions d'**int**, ceci signifie que c'est la version avec **int** de la méthode **play()**, dorénavant surchargée, qui est appelée. Comme elle *n'a pas* été redéfinie, c'est donc la méthode de la classe de base qui est utilisée.

L'output est le suivant :

```
InstrumentX.play()
```

Ceci n'est pas un appel polymorphe de méthode. Dès que vous comprenez ce qui se passe, vous pouvez corriger le problème assez facilement, mais imaginez la difficulté pour trouver l'anomalie si elle est enterrée dans un gros programme.

Classes et méthodes abstraites

Dans tous ces exemples sur l'instrument de musique, les méthodes de la classe de base **Instrument** étaient toujours factices. Si jamais ces méthodes sont appelées, c'est que vous avez fait quelque chose de travers. C'est parce que le rôle de la classe **Instrument** est de créer une interface *commune* pour toutes les classes dérivées d'elle.

La seule raison d'avoir cette interface commune est qu'elle peut être exprimée différemment pour chaque sous-type différent. Elle établit une forme de base, ainsi vous pouvez dire ce qui est commun avec toutes les classes dérivées. Une autre manière d'exprimer cette factorisation du code est d'appeler **Instrument** une classe de base abstraite (ou simplement une classe abstraite). Vous créez une classe abstraite quand vous voulez manipuler un ensemble de classes à travers cette interface commune. Toutes les méthodes des classes dérivées qui correspondent à la signature de la déclaration de classe de base seront appelées employant le mécanisme de liaison dynamique. (Cependant, comme on l'a vu dans la dernière section, si le nom de la méthode est le même comme la classe de base mais les arguments sont différents, vous avez une surcharge, ce qui n'est pas probablement que vous voulez.)

Si vous avez une classe abstraite comme **Instrument**, les objets de cette classe n'ont pratiquement aucune signification. Le rôle d'**Instrument** est uniquement d'exprimer une interface et non pas une implémentation particulière, ainsi la création d'un objet **Instrument** n'a pas de sens et vous voudrez probablement dissuader l'utilisateur de le faire. Une implémentation possible est d'afficher un message d'erreur dans toutes les méthodes d'**Instrument**, mais cela retarde le diagnostic à l'exécution et exige un code fiable et exhaustif. Il est toujours préférable de traiter les problèmes au moment de la compilation.

Java fournit un mécanisme qui implémente cette fonctionnalité: c'est la méthode abstraite [38]. C'est une méthode qui est incomplète; elle a seulement une déclaration et aucun corps de méthode. Voici la syntaxe pour une déclaration de méthode abstraite [abstract] :

```
abstract void f();
```

Une classe contenant des méthodes abstraites est appelée une *classe abstraite*. Si une classe contient une ou plusieurs méthodes abstraites, la classe doit être qualifiée comme **abstract**. (Autrement, le compilateur signale une erreur.)

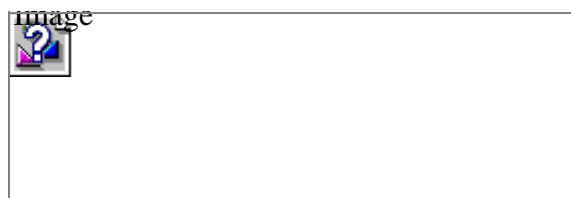
Si une classe abstraite est incomplète, comment doit réagir le compilateur si quelqu'un essaye de créer un objet de cette classe? Il ne peut pas créer sans risque un objet d'une classe abstraite, donc vous obtenez un message d'erreur du compilateur. C'est ainsi que le compilateur assure la pureté de la classe abstraite et ainsi vous n'avez plus à vous soucier d'un usage impropre de la classe.

Si vous héritez d'une classe abstraite et que vous voulez fabriquer des objets du nouveau type, vous devez fournir des définitions de méthode correspondant à toutes les méthodes abstraites de la classe de base. Si vous ne le faites pas (cela peut être votre choix), alors la classe dérivée est aussi abstraite et le compilateur vous forcera à qualifier cette classe avec le mot clé **abstract**.

Il est possible de créer une classe abstraite sans qu'elle contienne des méthodes abstraites. C'est utile quand vous avez une classe pour laquelle avoir des méthodes abstraites n'a pas de sens et que vous voulez empêcher la création d'instance de cette classe.

La classe **Instrument** peut facilement être changée en une classe abstraite. Seules certaines des méthodes seront abstraites, puisque créer une classe abstraite ne vous oblige pas à avoir que des méthodes abstraites. Voici à

quoi cela ressemble :



Voici l'exemple de l'orchestre modifié en utilisant des classes et des méthodes abstraites :

```
//: c07:music4:Music4.java
// Classes et méthodes abstraites.
import java.util.*;

abstract class Instrument {
    int i; // Alloué à chaque fois
    public abstract void play();
    public String what() {
        return "Instrument";
    }
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play() {
        System.out.println("Wind.play()");
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play() {
        System.out.println("Percussion.play()");
    }
    public String what() { return "Percussion"; }
```

```
    public void adjust() {}
}

class Stringed extends Instrument {
    public void play() {
        System.out.println("Stringed.play()");
    }
    public String what() { return "Stringed"; }
    public void adjust() {}
}

class Brass extends Wind {
    public void play() {
        System.out.println("Brass.play()");
    }
    public void adjust() {
        System.out.println("Brass.adjust()");
    }
}

class Woodwind extends Wind {
    public void play() {
        System.out.println("Woodwind.play()");
    }
    public String what() { return "Woodwind"; }
}

public class Music4 {
    // Ne se préoccupe pas des types: des nouveaux
    // ajoutés au système marcheront très bien:
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}
```



```

static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
        tune(e[i]);
}

public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];

    int i = 0;

    // Upcast lors de l'ajout au tableau:

    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();

    tuneAll(orchestra);
}
} ///:~

```

Vous pouvez voir qu'il n'y a vraiment aucun changement excepté dans la classe de base.

Il est utile de créer des classes et des méthodes abstraites parce qu'elles forment l'abstraction d'une classe explicite et indique autant à utilisateur qu'au compilateur comment elles doivent être utilisées.

Constructeurs et polymorphisme

Comme d'habitude, les constructeurs se comportent différemment des autres sortes de méthodes. C'est encore vrai pour le polymorphisme. Quoique les constructeurs ne soient pas polymorphes (bien que vous puissiez avoir un genre de "constructeur virtuel", comme vous le verrez dans le chapitre 12), il est important de comprendre comment les constructeurs se comportent dans des hiérarchies complexes combiné avec le polymorphisme. Cette compréhension vous aidera à éviter de désagréables plats de nouilles.

Ordre d'appel des constructeurs

L'ordre d'appel des constructeurs a été brièvement discuté dans le chapitre 4 et également dans le chapitre 6, mais c'était avant l'introduction du polymorphisme.

Un constructeur de la classe de base est toujours appelé dans le constructeur d'une classe dérivée, en remontant la hiérarchie d'héritage de sorte qu'un constructeur pour chaque classe de base est appelé. Ceci semble normal car le travail du constructeur est précisément de construire correctement l'objet. Une classe dérivée a seulement accès à ses propres membres, et pas à ceux de la classe de base (dont les membres sont en général **private**). Seul le constructeur de la classe de base a la connaissance et l'accès appropriés pour initialiser ses propres éléments. Par conséquent, il est essentiel que tous les constructeurs soient appelés, sinon l'objet ne serait pas entièrement construit. C'est pourquoi le compilateur impose un appel de constructeur pour chaque partie d'une classe

dérivée. Il appellera silencieusement le constructeur par défaut si vous n'appellez pas explicitement un constructeur de la classe de base dans le corps du constructeur de la classe dérivée. S'il n'y a aucun constructeur de défaut, le compilateur le réclamera (dans le cas où une classe n'a aucun constructeur, le compilateur générera automatiquement un constructeur par défaut).

Prenons un exemple qui montre les effets de la composition, de l'héritage, et du polymorphisme sur l'ordre de construction :

```
//: c07:Sandwich.java
// Ordre d'appel des constructeurs.

class Meal {
    Meal() { System.out.println("Meal()"); }
}

class Bread {
    Bread() { System.out.println("Bread()"); }
}

class Cheese {
    Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
    Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() {
        System.out.println("PortableLunch()");
    }
}
```

```

class Sandwich extends PortableLunch {
    Bread b = new Bread();
    Cheese c = new Cheese();
    Lettuce l = new Lettuce();
    Sandwich() {
        System.out.println("Sandwich()");
    }
    public static void main(String[] args) {
        new Sandwich();
    }
} ///:~

```

Cet exemple utilise une classe complexe et d'autres classes, chaque classe a un constructeur qui s'annonce lui-même. La classe importante est **Sandwich**, qui est au troisième niveau d'héritage (quatre, si vous comptez l'héritage implicite de **Object**) et qui a trois objets membres. Quand un objet **Sandwich** est créé dans le **main()**, l'output est :

```

Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()

```

Ceci signifie que l'ordre d'appel des constructeurs pour un objet complexe est le suivant :

1. Le constructeur de la classe de base est appelé. Cette étape est répétée récursivement jusqu'à ce que la racine de la hiérarchie soit construite d'abord, suivie par la classe dérivée suivante, etc, jusqu'à atteindre la classe la plus dérivée.
2. Les initialiseurs des membres sont appelés dans l'ordre de déclaration
3. Le corps du constructeur de la classe dérivée est appelée.

L'ordre d'appel des constructeurs est important. Quand vous héritez, vous savez tout au sujet de la classe de base et pouvez accéder à tous les membres **public** et **protected** de la classe de base. Ceci signifie que vous devez pouvoir présumer que tous les membres de la classe de base sont valides quand vous êtes dans la classe dérivée. Dans une méthode normale, la construction a déjà eu lieu, ainsi tous les membres de toutes les parties de l'objet ont été construits. Dans le constructeur, cependant, vous devez pouvoir supposer que tous les membres que vous utilisez ont été construits. La seule manière de le garantir est d'appeler d'abord le constructeur de la classe de base. Ainsi, quand êtes dans le constructeur de la classe dérivée, tous les membres que vous pouvez accéder dans la classe de base ont été initialisés. Savoir que tous les membres sont valides à l'intérieur du constructeur est également la raison pour laquelle, autant que possible, vous devriez initialiser tous les objets membres (c'est

à dire les objets mis dans la classe par composition) à leur point de définition dans la classe (par exemple, **b**, **c**, et **l** dans l'exemple ci-dessus). Si vous suivez cette recommandation, vous contribuerez à vous assurer que tous les membres de la classe de base *et* les objets membres de l'objet actuel aient été initialisés. Malheureusement, cela ne couvre pas tous les cas comme vous allez le voir dans le paragraphe suivant.

La méthode `finalize()` et l'héritage

Quand vous utilisez la composition pour créer une nouvelle classe, vous ne vous préoccupez pas de l'achèvement des objets membres de cette classe. Chaque membre est un objet indépendant et traité par le garbage collector indépendamment du fait qu'il soit un membre de votre classe. Avec l'héritage, cependant, vous devez redéfinir **`finalize()`** dans la classe dérivée si un nettoyage spécial doit être effectué pendant la phase de garbage collection. Quand vous redéfinissez **`finalize()`** dans une classe fille, il est important de ne pas oublier d'appeler la version de **`finalize()`** de la classe de base, sinon l'achèvement de la classe de base ne se produira pas. L'exemple suivant le prouve :

```
//: c07:Frog.java
// Test de la méthode finalize avec l'héritage.
```

```
class DoBaseFinalization {
    public static boolean flag = false;
}

class Characteristic {
    String s;

    Characteristic(String c) {
        s = c;

        System.out.println(
            "Creating Characteristic " + s);
    }

    protected void finalize() {
        System.out.println(
            "finalizing Characteristic " + s);
    }
}

class LivingCreature {
    Characteristic p =
        new Characteristic("is alive");
```

```
LivingCreature() {  
    System.out.println("LivingCreature()");  
}  
  
protected void finalize() {  
    System.out.println(  
        "LivingCreature finalize");  
  
    // Appel de la version de la classe de base, à la fin!  
    if(DoBaseFinalization.flag)  
  
        try {  
            super.finalize();  
        } catch(Throwable t) {}  
    }  
}
```

```
class Animal extends LivingCreature {  
    Characteristic p =  
        new Characteristic("has heart");  
    Animal() {  
        System.out.println("Animal()");  
    }  
  
    protected void finalize() {  
        System.out.println("Animal finalize");  
        if(DoBaseFinalization.flag)  
  
            try {  
                super.finalize();  
            } catch(Throwable t) {}  
        }  
    }  
}
```

```
class Amphibian extends Animal {  
    Characteristic p =
```

```
        new Characteristic("can live in water");
    Amphibian() {
        System.out.println("Amphibian()");
    }
    protected void finalize() {
        System.out.println("Amphibian finalize");
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
}

public class Frog extends Amphibian {
    Frog() {
        System.out.println("Frog()");
    }
    protected void finalize() {
        System.out.println("Frog finalize");
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
    public static void main(String[] args) {
        if(args.length != 0 &&
            args[0].equals("finalize"))
            DoBaseFinalization.flag = true;
        else
            System.out.println("not finalizing bases");
        new Frog(); // Devient instantanément récupérable par le garbage collector
    }
}
```

```

    System.out.println("bye!");

    // Force l'appel des finalisations:

    System.gc();

}

} ///:~

```

La classe **DoBaseFinalization** a simplement un indicateur qui contrôle l'appel de **super.finalize()** pour toutes les classes de la hiérarchie. Cet indicateur est positionné par un argument de l'appel en ligne, vous pouvez ainsi visualiser le comportement selon l'invocation ou non de la finalisation dans la classe de base.

Chaque classe dans la hiérarchie contient également un objet de la classe **Characteristic**. Vous constaterez que les objets de **Characteristic** sont toujours finalisés indépendamment de l'appel conditionné des finaliseurs de la classe de base.

Chaque méthode **finalize()** redéfinie doit au moins avoir accès aux membres **protected** puisque la méthode **finalize()** de la classe **Object** est **protected** et le que compilateur ne vous permettra pas de réduire l'accès pendant l'héritage (« Friendly » est moins accessible que **protected**).

Dans **Frog.main()**, l'indicateur **DoBaseFinalization** est configuré et un seul objet **Frog** est créé. Rappelez-vous que la phase de garbage collection, et en particulier la finalisation, ne peut pas avoir lieu pour un objet particulier, ainsi pour la forcer, l'appel à **System.gc()** déclenche le garbage collector, et ainsi la finalisation. Sans finalisation de la classe de base, l'output est le suivant :

```

not finalizing bases

Creating Characteristic is alive

LivingCreature()

Creating Characteristic has heart

Animal()

Creating Characteristic can live in water

Amphibian()

Frog()

bye!

Frog finalize

finalizing Characteristic is alive

finalizing Characteristic has heart

finalizing Characteristic can live in water

```

Vous pouvez constater qu'aucune finalisation n'est appelée pour les classes de base de **Frog** (les objets membres, eux, sont achevés, comme on s'y attendait). Mais si vous ajoutez l'argument « finalize » sur la ligne de commande, on obtient ce qui suit :

```

Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water

```

Bien que l'ordre de finalisation des objets membres soit le même que l'ordre de création, l'ordre de finalisation des objets est techniquement non spécifié. Cependant, vous avez le contrôle sur cet ordre pour les classes de base. Le meilleur ordre à suivre est celui qui est montré ici, et qui est l'ordre inverse de l'initialisation. Selon le modèle qui est utilisé pour des destructeurs en C++, vous devez d'abord exécuter la finalisation des classes dérivées, puis la finalisation de la classe de base. La raison est que la finalisation des classes dérivées pourrait appeler des méthodes de la classe de base qui exigent que les composants de la classe de base soient toujours vivants, donc vous ne devez pas les détruire prématurément.

Comportement des méthodes polymorphes dans les constructeurs

La hiérarchie d'appel des constructeurs pose un dilemme intéressant. Qu'arrive t-il si à l'intérieur d'un constructeur vous appelez une méthode dynamiquement attachée de l'objet en cours de construction? À l'intérieur d'une méthode ordinaire vous pouvez imaginer ce qui arriverait: l'appel dynamiquement attaché est résolu à l'exécution parce que l'objet ne peut pas savoir s'il appartient à la classe dans laquelle se trouve la méthode ou bien dans une classe dérivée. Par cohérence, vous pourriez penser que c'est ce qui doit arriver dans les constructeurs.

Ce n'est pas ce qui se passe. Si vous appelez une méthode dynamiquement attachée à l'intérieur d'un constructeur, c'est la définition redéfinie de cette méthode est appelée. Cependant, *l'effet* peut être plutôt surprenant et peut cacher des bugs difficiles à trouver.

Le travail du constructeur est conceptuellement d'amener l'objet à l'existence (qui est à peine un prouesse ordinaire). À l'intérieur de n'importe quel constructeur, l'objet entier pourrait être seulement partiellement formé - vous pouvez savoir seulement que les objets de la classe de base ont été initialisés, mais vous ne pouvez pas connaître les classes filles qui hérite de vous. Cependant, un appel de méthode dynamiquement attaché, atteint « extérieurement » la hiérarchie d'héritage. Il appelle une méthode dans une classe dérivée. Si vous faites ça à

l'intérieur d'un constructeur, vous appelez une méthode qui pourrait manipuler des membres non encore initialisés - une recette très sûre pour le désastre.

Vous pouvez voir le problème dans l'exemple suivant :

```
//: c07:PolyConstructors.java
// Constructeurs et polymorphisme ne conduisent
// pas ce à quoi que vous pourriez vous attendre.
abstract class Glyph {
    abstract void draw();
    Glyph() {
        System.out.println("Glyph() before draw()");
        draw();
        System.out.println("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
    int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        System.out.println(
            "RoundGlyph.RoundGlyph(), radius = "
            + radius);
    }
    void draw() {
        System.out.println(
            "RoundGlyph.draw(), radius = " + radius);
    }
}

public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
}
```

```
} ///:~
```

Dans **Glyph**, la méthode dessiner [**draw()**] est abstraite; elle a donc été conçue pour être redéfinie. En effet, vous êtes forcés de la redéfinir dans **RoundGlyph**. Mais le constructeur de **Glyph** appelle cette méthode et l'appel aboutit à **RoundGlyph.draw()**, ce qui semble être l'intention. Mais regardez l'output :

```
Glyph() before draw()

RoundGlyph.draw(), radius = 0

Glyph() after draw()

RoundGlyph.RoundGlyph(), radius = 5
```

Quand le constructeur de **Glyph** appelle **draw()**, le rayon [**radius**] n'a même pas encore la valeur initiale de 1, il vaut zéro. Le résultat serait probablement réduit à l'affichage d'un point ou même à rien du tout, avec vous, fixant un écran désespérément vide essayant de comprendre pourquoi le programme ne marche pas.

L'ordre de l'initialisation décrit dans la section précédente n'est pas complètement exhaustif, et c'est la clé qui va résoudre le mystère. La procédure d'initialisation est la suivante :

1. La zone allouée à l'objet est initialisée à zéro binaire avant tout.
2. Les constructeurs des classes de base sont appelés comme décrit précédemment. Puis, la méthode **draw()** redéfinie est appelée (et oui, *avant* l'appel du constructeur de **RoundGlyph**), et utilise **radius** qui vaut zéro à cause de la première étape.
3. Les initialiseurs des membres sont appelés dans l'ordre de déclaration.
4. Le corps du constructeur de la classe dérivée est appelé

Le bon côté est que tout est au moins initialisé au zéro (selon la signification de zéro pour un type de donnée particulier) et non laissé avec n'importe quelles valeurs. Cela inclut les références d'objet qui sont incorporés à l'intérieur d'une classe par composition, et qui passent à **null**. Ainsi si vous oubliez d'initialiser une référence vous obtiendrez une exception à l'exécution. Tout le reste est à zéro, qui est habituellement une valeur que l'on repère en examinant l'output.

D'autre part, vous devez être assez horrifiés du résultat de ce programme. Vous avez fait une chose parfaitement logique et pourtant le comportement est mystérieusement faux, sans aucune manifestation du compilateur (C++ a un comportement plus correct dans la même situation). Les bugs dans ce goût là peuvent facilement rester cachés et nécessiter pas mal de temps d'investigation.

Il en résulte la recommandation suivante pour les constructeurs: « Faire le minimum pour mettre l'objet dans un bon état et si possible, ne pas appeler de méthodes. » Les seules méthodes qui sont appelables en toute sécurité à l'intérieur d'un constructeur sont celles qui sont finales dans la classe de base (même chose pour les méthodes privées, qui sont automatiquement finales.). Celles-ci ne peuvent être redéfinies et ne réservent donc pas de surprise.

Concevoir avec l'héritage

Après avoir vu le polymorphisme, c'est un instrument tellement astucieux qu'on dirait que tout doit être hérité. Ceci peut alourdir votre conception; en fait si vous faites le choix d'utiliser l'héritage d'entrée lorsque vous créez une nouvelle classe à partir d'une classe existante, cela peut devenir inutilement compliqué.

Une meilleure approche est de choisir d'abord la composition, quand il ne vous semble pas évident de choisir entre les deux. La composition n'oblige pas à concevoir une hiérarchie d'héritage, mais elle est également plus flexible car il est alors possible de choisir dynamiquement un type (et son comportement), alors que l'héritage requiert un type exact déterminé au moment de la compilation. L'exemple suivant l'illustre :

```
//: c07:Transmogrify.java
// Changer dynamiquement le comportement
// d'un objet par la composition.

abstract class Actor {
    abstract void act();
}

class HappyActor extends Actor {
    public void act() {
        System.out.println("HappyActor");
    }
}

class SadActor extends Actor {
    public void act() {
        System.out.println("SadActor");
    }
}

class Stage {
    Actor a = new HappyActor();

    void change() { a = new SadActor(); }

    void go() { a.act(); }
}

public class Transmogrify {
    public static void main(String[] args) {
        Stage s = new Stage();

        s.go(); // Imprime "HappyActor"

        s.change();
    }
}
```

```

    s.go(); // Imprime "SadActor"
}
} ///:~

```

Un objet **Stage** contient une référence vers un **Actor**, qui est initialisé par un objet **HappyActor**. Cela signifie que **go()** produit un comportement particulier. Mais puisqu'une référence peut être reliée à un objet différent à l'exécution, une référence à un objet **SadActor** peut être substituée dans **a** et alors le comportement produit par **go()** change. Ainsi vous gagnez en flexibilité dynamique à l'exécution (également appelé le *State Pattern*. Voir *Thinking in Patterns with Java*, téléchargeable sur www.BruceEckel.com). Par contre, vous ne pouvez pas décider d'hériter différemment à l'exécution; cela doit être complètement déterminé à la compilation.

Voici une recommandation générale: « Utilisez l'héritage pour exprimer les différences de comportement, et les champs pour exprimer les variations d'état. » Dans l'exemple ci-dessus, les deux sont utilisés: deux classes différentes héritent pour exprimer la différence dans la méthode **act()**, et **Stage** utilise la composition pour permettre à son état d'être changé. Dans ce cas, ce changement d'état provoque un changement de comportement.

Héritage pur contre extension

Lorsque l'on étudie l'héritage, il semblerait que la façon la plus propre de créer une hiérarchie d'héritage est de suivre l'approche « pure. » A savoir que seules les méthodes qui ont été établies dans la classe de base ou l'**interface** sont surchargeables dans la classe dérivée, comme le montre ce diagramme :



Ceci peut se nommer une relation « est-un » pure car l'interface d'une classe établie ce qu'elle est. L'héritage garantit que toute classe dérivée aura l'interface de la classe de base et rien de moins. Si vous suivez le diagramme ci-dessus, les classes dérivées auront également *pas plus* que l'interface de la classe de base.

Ceci peut être considéré comme une *substitution pure*, car les objets de classe dérivée peuvent être parfaitement substitués par la classe de base, et vous n'avez jamais besoin de connaître d'information supplémentaire sur les sous-classes quand vous les utilisez :



Cela étant, la classe de base peut recevoir tout message que vous pouvez envoyer à la classe dérivée car les deux ont exactement la même interface. Tout ce que vous avez besoin de faire est d'utiliser l'upcast à partir de la classe dérivée et de ne jamais regarder en arrière pour voir quel type exact d'objet vous manipulez.

En la considérant de cette manière, une relation pure « est-un » semble la seule façon sensée de pratiquer, et toute autre conception dénote une réflexion embrouillée et est par définition hachée. Ceci aussi est un piège. Dès que vous commencez à penser de cette manière, vous allez tourner en rond et découvrir qu'étendre

l'interface (ce que, malencontreusement, le mot clé **extends** semble encourager) est la solution parfaite à un problème particulier. Ceci pourrait être qualifié de relation « est-comme-un » car la classe dérivée est *comme* la classe de base, elle a la même interface fondamentale mais elle a d'autres éléments qui nécessitent d'implémenter des méthodes additionnelles :



Mais si cette approche est aussi utile et sensée (selon la situation) elle a un inconvénient. La partie étendue de l'interface de la classe dérivée n'est pas accessible à partir de la classe de base, donc une fois que vous avez utilisé l'upcast vous ne pouvez pas invoquer les nouvelles méthodes :



Si vous n'upcastez pas dans ce cas, cela ne va pas vous incommoder, mais vous serez souvent dans une situation où vous aurez besoin de retrouver le type exact de l'objet afin de pouvoir accéder aux méthodes étendues de ce type. La section suivante montre comment cela se passe.

Downcasting et identification du type à l'exécution

Puisque vous avez perdu l'information du type spécifique par un *upcast* (en remontant la hiérarchie d'héritage), il est logique de retrouver le type en redescendant la hiérarchie d'héritage par un *downcast*. Cependant, vous savez qu'un upcast est toujours sûr; la classe de base ne pouvant pas avoir une interface plus grande que la classe dérivée, ainsi tout message que vous envoyez par l'interface de la classe de base a la garantie d'être accepté. Mais avec un downcast, vous ne savez pas vraiment qu'une forme (par exemple) est en réalité un cercle. Cela pourrait plutôt être un triangle ou un carré ou quelque chose d'un autre type.



Pour résoudre ce problème il doit y avoir un moyen de garantir qu'un downcast est correct, ainsi vous n'allez pas effectuer un cast accidentel vers le mauvais type et ensuite envoyer un message que l'objet ne pourrait accepter. Ce serait assez imprudent.

Dans certains langages (comme C++) vous devez effectuer une opération spéciale afin d'avoir un cast ascendant sûr, mais en Java *tout cast* est vérifié! Donc même si il semble que vous faites juste un cast explicite ordinaire, lors de l'exécution ce cast est vérifié pour assurer qu'en fait il s'agit bien du type auquel vous vous attendez. Si il ne l'est pas, vous récupérez une **ClassCastException**. Cette action de vérifier les types au moment de l'exécution est appelé *run-time type identification* (RTTI). L'exemple suivant montre le comportement de la RTTI :

```
//: c07:RTTI.java
// Downcasting & Run-time Type
// Identification (RTTI).
import java.util.*;

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

public class RTTI {
    public static void main(String[] args) {
        Useful[] x = {
            new Useful(),
            new MoreUseful()
        };
        x[0].f();
        x[1].g();
        // Compilation: méthode non trouvée dans Useful:
        //! x[1].u();
        ((MoreUseful)x[1]).u(); // Downcast/RTTI
        ((MoreUseful)x[0]).u(); // Exception envoyée
    }
} ///:~
```

Comme dans le diagramme, **MoreUseful** étend l'interface de **Useful**. Mais puisque il a hérité, on peut faire un

upcast vers un **Useful**. Vous pouvez voir ceci se produire dans l'initialisation du tableau **x** dans **main()**. Comme les deux objets du tableau sont de la classe **Useful**, vous pouvez envoyer les méthodes **f()** et **g()** aux deux, et si vous essayez d'invoquer **u()**, qui existe seulement dans **MoreUseful**, vous aurez un message d'erreur à la compilation.

Si vous voulez accéder à l'interface étendue d'un objet **MoreUseful**, vous pouvez essayer un downcast. Si c'est le type correct, cela fonctionnera. Autrement, vous allez recevoir une **ClassCastException**. Vous n'avez pas besoin d'écrire un code spécial pour cette exception, car elle indique une erreur du programmeur qui pourrait arriver n'importe où dans un programme.

La RTTI est plus riche qu'un simple cast. Par exemple, il y a une façon de connaître le type que vous manipulez *avant* d'essayer de le downcaster. Tout le Chapitre 12 est consacré à l'étude de différents aspects du « run-time type identification » Java.

Résumé

Polymorphisme signifie « différentes formes. » Dans la programmation orientée objet, vous avez la même physionomie (l'interface commune dans la classe de base) et différentes formes qui utilisent cette physionomie: les différentes versions des méthodes dynamiquement attachées.

Vous avez vu dans ce chapitre qu'il est impossible de comprendre, ou même créer, un exemple de polymorphisme sans utiliser l'abstraction et l'héritage. Le polymorphisme est une notion qui ne peut pas être présenté séparément (comme on peut le faire par exemple avec un **switch**), mais qui fonctionne plutôt en conjonction avec le schéma global #big picture# des relations entre classes. Les gens sont souvent troublés par d'autres dispositifs non-orientés objet de Java, comme la surcharge de méthode, qui sont parfois présentés comme étant orientés objet. Ne soyez pas dupe: si ce n'est pas de la liaison tardive, ce n'est pas du polymorphisme.

Pour utiliser le polymorphisme, et par conséquent les techniques orientées objet, pertinemment dans vos programmes vous devez élargir votre vision de la programmation pour y inclure non seulement les membres et les messages d'une classe individuelle, mais également ce qui est partagé entre les classes et leurs rapports entre elles. Bien que ceci exige un effort significatif, ça vaut vraiment le coup car il en résulte un développement plus rapide, un code mieux organisé, des programmes extensibles et une maintenance plus facile.

Exercices

1. Ajouter une nouvelle méthode à la classe de base de **Shapes.java** qui affiche un message, mais sans la redéfinir dans les classes dérivées. Expliquer ce qui se passe. Maintenant la redéfinir dans une des classes dérivées mais pas dans les autres, et voir ce qui se passe. Finalement, la redéfinir dans toutes les classes dérivées.
2. Ajouter un nouveau type de **Shape** à **Shapes.java** et vérifier dans **main()** que le polymorphisme fonctionne pour votre nouveau type comme il le fait pour les anciens types.
3. Changer **Music3.java** pour que **what()** devienne une méthode **toString()** de la classe racine **Object**. Essayer d'afficher les objets **Instrument** en utilisant **System.out.println()** (sans aucun cast).
4. Ajouter un nouveau type d'**Instrument** à **Music3.java** et vérifier que le polymorphisme fonctionne pour votre nouveau type.
5. Modifier **Music3.java** pour qu'il crée de manière aléatoire des objets **Instrument** de la même façon que **Shapes.java** le fait.
6. Créer une hiérarchie d'héritage de **Rongeur**: **Souris**, **Gerbille**, **Hamster**, etc. Dans la classe de base, fournir des méthodes qui sont communes à tous les **Rongeurs**, et les redéfinir dans les classes dérivées pour exécuter des comportements différents dépendant du type spécifique du **Rongeur**. Créer un tableau

de **Rongeur**, le remplir avec différent types spécifiques de **Rongeurs**, et appeler vos méthodes de la classe de base pour voir ce qui arrive.

7. Modifier l'Exercice 6 pour que **Rongeur** soit une classe **abstract**. Rendre les méthodes de **Rongeur** abstraites dès que possible.
8. Créer une classe comme étant **abstract** sans inclure aucune méthode **abstract**, et vérifier que vous ne pouvez créer aucune instance de cette classe.
9. Ajouter la classe **Pickle** à **Sandwich.java**.
10. Modifier l'Exercice 6 afin qu'il démontre l'ordre des initialisations des classes de base et des classes dérivées. Maintenant ajouter des objets membres à la fois aux classes de base et dérivées, et montrer dans quel ordre leurs initialisations se produisent durant la construction.
11. Créer une hiérarchie d'héritage à 3 niveaux. Chaque classe dans la hiérarchie devra avoir une méthode **finalize()**, et devra invoquer correctement la version de la classe de base de **finalize()**. Démontrer que votre hiérarchie fonctionne de manière appropriée.
12. Créer une classe de base avec deux méthodes. Dans la première méthode, appeler la seconde méthode. Faire hériter une classe et redéfinir la seconde méthode. Créer un objet de la classe dérivée, upcaster le vers le type de base, et appeler la première méthode. Expliquer ce qui se passe.
13. Créer une classe de base avec une méthode **abstract print()** qui est redéfinie dans une classe dérivée. La version redéfinie de la méthode affiche la valeur d'une variable **int** définie dans la classe dérivée. Au point de définition de cette variable, lui donner une valeur non nulle. Dans le constructeur de la classe de base, appeler cette méthode. Dans **main()**, créer un objet du type dérivé, et ensuite appeler sa méthode **print()**. Expliquer les résultats.
14. Suivant l'exemple de **Transmogrify.java**, créer une classe **Starship** contenant une référence **AlertStatus** qui peut indiquer trois états différents. Inclure des méthodes pour changer les états.
15. Créer une classe **abstract** sans méthodes. Dériver une classe et ajouter une méthode. Créer une méthode **static** qui prend une référence vers la classe de base, effectue un downcast vers la classe dérivée, et appelle la méthode. Dans **main()**, démontrer que cela fonctionne. Maintenant mettre la déclaration **abstract** pour la méthode dans la classe de base, éliminant ainsi le besoin du downcast.

[38] Pour les programmeurs C++, ceci est analogue aux *fonctions virtuelles pures* du C++.