

Chapitre 5: Améliorer les Classes

Présentation pour

Java Software Solutions Foundations of Program Design Deuxième Edition

par John Lewis et William Loftus

Java Software Solutions est publié par Addison-Wesley

Presentation slides are copyright 2000 by John Lewis and William Loftus. All rights reserved.
Instructors using the textbook may use and modify these slides for pedagogical purposes.

Améliorer les Classes

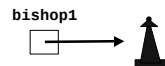
- Nous pouvons maintenant explorer des aspects variés des classes et objets plus en détails
- Le Chapitre 5 se concentre sur:
 - référence à un objet et alias
 - passage d'objets en paramètres
 - le modificateur `static`
 - Classes imbriquées
 - interfaces et polymorphisme
 - événements et auditeurs (*listeners*)
 - animation

2

Références

- Rappel du Chapitre 2, une référence à un objet contient l'adresse mémoire d'un objet
- Plutôt qu'utiliser une adresse arbitraire, nous illustrons une référence graphiquement comme un "pointeur" sur un objet

```
ChessPiece bishop1 = new ChessPiece();
```



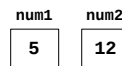
3

Affectation Revisitée

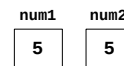
- Une affectation prend une copie de la valeur et la stocke dans une variable
- Pour les types de bases :

```
num2 = num1;
```

Avant



Après



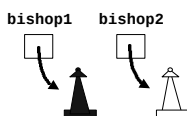
4

Affectation de Référence

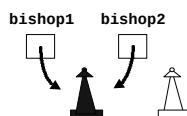
- Pour une référence à un objet, l'affectation copie l'emplacement en mémoire :

```
bishop2 = bishop1;
```

Avant



Après



5

Alias

- Deux références ou plus qui réfère au même objet sont des *alias* l'une de l'autre
- Un objet (et ses données) peut être accédé en utilisant des variables différentes
- Les alias sont utiles, mais doivent être gérés avec précautions
- Un changement à l'état d'un objet (ses variables) à travers une référence le change pour tous ses alias

6

Ramasse-miettes

- Lorsqu'un objet n'a plus de référence valide vers lui, il ne peut plus être accédé par le programme
- Il devient alors inutile, et est donc considéré comme une *miette*
- Java effectue périodiquement le *ramassage automatique des miettes*, retournant la mémoire d'un objet au système pour usage futur
- Avec d'autres langages, le ramassage des miettes est la responsabilité du programmeur

7

Passage d'Objets aux Méthodes

- Les paramètres des méthodes de Java sont *passés par valeurs*
- Ça implique qu'une copie du paramètre actuel (la valeur passée) est stockée dans le paramètre formel (dans l'entête de la méthode)
- Le passage de paramètres est essentiellement une affectation
- Lorsque un objet est passé à une méthode, le paramètre actuel et le paramètre formel deviennent des *alias*

Passage d'Objets aux Méthodes

- Les changements à un paramètre dans une méthode peuvent avoir des effets permanents ou non (à l'extérieur de la méthode)
- Voir [ParameterPassing.java](#) (page 226)
- Voir [ParameterTester.java](#) (page 228)
- Voir [Num.java](#) (page 230)
- Noter la différence entre changer une référence et changer l'objet sur lequel pointe cette référence

Le modificateur static

- Au Chapitre 2 nous avons discuté de méthodes statiques (aussi appelées méthodes de classe) qui peuvent être appelées avec le nom de la classe, plutôt qu'avec le nom d'un objet de la classe
- Par exemple, les méthodes de la classe `Math` sont statiques
- Pour rendre une méthode statique, nous appliquons le modificateur `static` à la définition de la méthode
- Le modificateur `static` peut aussi être appliqué à des variables
- Cela associe la variable ou la méthode avec la classe plutôt qu'avec un objet

10

Méthodes Statiques

```
class Helper
```

```
    public static int triple (int num)
    {
        int result;
        result = num * 3;
        return result;
    }
```

Parce qu'elle est statique, la méthode peut être appelée ainsi :

```
value = Helper.triple (5);
```

11

Méthodes Statiques

- L'ordre des modificateurs peut être interchangé, mais par convention les modificateurs de visibilité sont placés en premiers
- Rappel : la méthode `main` est statique; elle est appelée par le système sans créer un objet
- Les méthodes statiques ne peuvent référencer des variables d'instance, car les variables d'instance n'existent que lorsqu'un objet existe
- Par contre, elles peuvent référencer des variables statiques ou des variables locales

12

Variables Statiques

- Les variables statiques sont quelques fois appelées *variables de classes*
 - Normalement, chaque objet a sa propre espace de données
 - Si une variable est déclarée `static`, il n'existe qu'une seule copie de cette variable
- ```
private static float price;
```
- L'espace memoire d'une variable `static` est créée aussitôt que la classe dans laquelle elle est declarée est chargée

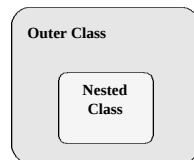
13

## Variables Statiques

- Tous les objets de la classe partagent l'accès à la variable `static`
- Lorsqu'un objet change la valeur d'une variable `static`, elle change pour tous les autres
- Les méthodes et variables `static` vont souvent ensemble
- Voir [CountInstances.java](#) (page 233)
- Voir [MyClass.java](#) (page 234)

## Classes Imbriquées

- En plus contenir des données et des méthodes, une classe peut aussi contenir d'autres classes
- Une classe déclarée à l'intérieur d'une autre classe est appelée une *classe imbriquée*



## Classes Imbriquées

- Une classe imbriquée a accès aux variables et méthodes de la classe externe, même s'ils sont déclarés privés
- Dans certaines situations, ça rend l'implémentation de ces classes plus faciles car elle peuvent facilement partager des informations
- De plus, la classe imbriquée peut être protégée d'un usage externe par la classe externe
- C'est une relation spéciale et devrait être utilisée avec précaution

## Classes Imbriquées

- Une classe imbriquée produit un fichier de bytecode séparé
- Si une classe imbriquée nommée `Inside` est déclarée dans une classe externe appelée `Outside`, deux fichiers bytecode seront produits:

```
Outside.class
Outside$Inside.class
```

- Une classe imbriquée peut être déclarée `static`, dans ce cas elle ne peut faire référence aux variables ou méthodes d'une instance
- Une classe imbriquée non statique est nommée *classe interne*

## Interfaces

- Une *interface* Java est une collection de méthodes abstraites et de constantes
- Une *méthode abstraite* est une entête de méthode sans corps de méthode
- Une méthode abstraite peut être déclarée en utilisant le modificateur `abstract`, mais comme toutes les méthodes d'une interface sont abstraites, ce n'est pas nécessaire
- Une interface est utilisée pour définir formellement un ensemble de méthodes qu'une classe doit implémenter

## Interfaces

interface est un mot réservé

```
public interface Doable
{
 public void doThis();
 public int doThat();
 public void doThis2 (float value, char ch);
 public boolean doTheOther (int num);
}
```

aucune méthode d'une interface n'a de définition (corps)

Chaque entête de méthode est suivi d'un point-virgule

## Interfaces

- Une interface ne peut être instanciée
- Les méthodes d'une interface ont une visibilité publique par défaut
- Une classe implémente formellement une interface en
  - l'affirmant dans l'entête de la classe
  - fournissant des implémentations pour chaque méthode abstraite de l'interface
- Si une classe affirme qu'elle implémente une interface, elle doit définir tous les méthodes de l'interface sinon le compilateur produira des erreurs.

## Interfaces

```
public class CanDo implements Doable
{
 public void doThis ()
 {
 // whatever
 }

 public void doThat ()
 {
 // whatever
 }

 // etc.
}
```

implements est un mot réservé

Chaque méthode listée dans Doable est définie

## Interfaces

- Une classe implémentant une interface peut aussi implémenter d'autres méthodes
- Voir [Speaker.java](#) (page 236)
- Voir [Philosopher.java](#) (page 237)
- Voir [Dog.java](#) (page 238)
- Une classe peut implémenter de multiples interfaces
- Les interfaces sont énumérées après le mot-clé `implements`, séparées par des virgules
- La classe doit implémenter toutes les méthodes d'une interfaces énumérée dans l'entête

## Polymorphisme via Interfaces

- Un nom d'interface peut servir de type pour une variable de référence à un objet

```
Doable obj;
```

- La référence `obj` peut être utilisée pour pointer n'importe quel objet de n'importe quelle classe qui implémente l'interface `Doable`
- La version de `doThis` qui est invoquée à la ligne suivante dépend du type d'objet auquel `obj` fait référence:

```
obj.doThis();
```

## Polymorphisme via Interfaces

- Cette référence est *polymorphique*, ce qui peut être défini comme "prenant plusieurs formes"
- Cette ligne de code pourrait exécuter des méthodes différentes à des temps différents si `obj` pointe vers un autre objet
- Voir [Talking.java](#) (page 240)
- À noter, les références polymorphiques doivent être résolues à l'exécution; on appelle ça *dynamic binding*
- L'utilisation prudente de références polymorphiques peut conduire à la conception élégante et robuste de logiciels

## Interfaces

- La *librairie standard de classes de Java* contient plusieurs interfaces qui sont utiles en certaines situations
- L'interface **Comparable** contient une méthode abstraite nommée `compareTo`, qui est utilisée pour comparer deux objets
- La classe **String** implémente **Comparable** qui nous donne la possibilité de placer des **String** en ordre alphabétique
- L'interface **Iterator** contient des méthodes permettant à l'utilisateur de se déplacer facilement dans une collection d'objets

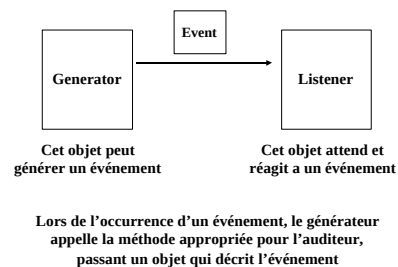
## Événements

- Un **event** est un objet qui représente des activités auxquels nous pourrions vouloir réagir
- Par exemple, nous pourrions vouloir que notre programme accomplisse certaines actions lors des occurrences suivantes:
  - La souris bouge
  - Un bouton de souris est activé
  - La souris est trainée (drag)
  - Un bouton est cliqué
  - Une clé du clavier est enfoncée
  - un timer expire
- Souvent, les événements correspondent à des actions de l'utilisateur, mais pas toujours

## Événements

- La *librairie standard de classes de Java* contient plusieurs classes représentant des événements typique
- Certains objets, tel un applet ou un bouton graphique, génèrent (fire) un événement lors de leurs occurrences
- D'autres objets, nommés *listeners*, réagissent aux événements
- Nous pouvons écrire des objets auditeurs (*listener*) pour faire tout ce que nous voulons lors de l'occurrence d'un événement

## Événements et auditeurs



## Interfaces Auditeurs (*Listener*)

- Nous pouvons créer un objet *listener* en écrivant une classe qui implémente une interface *listener*
- La *librairie standard de classes de Java* contient plusieurs interfaces correspondant à des catégories d'événements
- Par exemple, l'interface **MouseListener** contient des méthodes correspondant aux événements de la souris
- Après la création d'un auditeur, nous ajoutons l'auditeur à la composante qui pourrait générer l'événement afin de créer la relation formelle entre le générateur et l'auditeur

## Événements de Souris

- Voici des *événements de souris*:
  - *mouse pressed* - le bouton est appuyé
  - *mouse released* - le bouton est relâché
  - *mouse clicked* - le bouton est appuyé et relâché
  - *mouse entered* - le pointeur de la souris passe au dessus d'une composante particulière
  - *mouse exited* - le pointeur quitte une composante particulière
- Tout programme peut écouter pour certains, aucun, ou tous ces événements
- Voir [Dots.java](#) (page 246)
- Voir [DotsMouseListener.java](#) (page 248)

## Événements Déplacements de Souris

- Voici des événements de déplacements de la souris :
  - *mouse moved* – la souris se déplace
  - *mouse dragged* – la souris bouge pendant que le bouton est gardé enfoncé
- Il existe une interface **MouseMotionListener**
- Une classe peut servir de générateur et d'auditeur
- Une classe peut servir d'auditeur pour plusieurs types d'événements
- Voir [RubberLines.java](#) (page 249)

## Événements de Touches

- Voici des événements de touches:
  - *key pressed* – une touche du clavier est enfoncée
  - *key released* – une touche du clavier est relâchée
  - *key typed* – une touche du clavier est enfoncée et relâchée
- L'interface **KeyListener** s'occupe des événements de touches
- Les classes *listener* sont souvent implémentées comme classes internes, imbriquées dans les composantes qu'elles écoutent
- Voir [Direction.java](#) (page 253)

## Animations

- Une animation est une série de photos ou d'images qui changent constamment afin de créer l'illusion de mouvement
- Nous pouvons créer des animations avec Java en changeant une image quelques peu avec le temps
- La vitesse d'une animation Java est habituellement contrôlée par un objet **Timer**
- La classe **Timer** est définie dans le package `javax.swing`

## Animations

- Un objet **Timer** génère un **ActionEvent** à chaque `n` millisecondes (où `n` est fixé par le créateur d'objet)
- L'interface **ActionListener** contient une méthode `actionPerformed`
- Lorsque le timer expire (générant un **ActionEvent**) l'animation peut être mise à jour
- Voir [Rebound.java](#) (page 258)