

Chapitre 7: Héritage

Présentation pour

Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

Java Software Solutions is published by 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.

Héritage

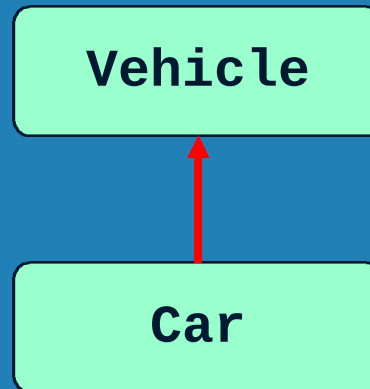
- ★ **L'héritage est une autre technique orientée-objet qui améliore la conception de logiciels et permet une meilleur réutilisation**
- ★ **Le chapitre 7 se concentre sur :**
 - Dériver de nouvelles classes
 - Créer une hiérarchie de classes
 - Le modificateur `protected`
 - polymorphisme via l'héritage
 - L'utilisation de l'héritage dans les interfaces usagers graphiques

Héritage

- ★ L'héritage permet au concepteur de logiciels de dériver une nouvelle classe à partir d'une classe existante
- ★ La classe existante est la *classe parent*, ou *superclasse*, ou encore la *classe de base*
- ★ La classe dérivée est appelée la *classe enfant* ou *sous-classe*.
- ★ Comme le nom l'implique, la classe enfant hérite des caractéristiques du parent
- ★ C'est à dire, la classe enfant hérite des méthodes et des données définies pour la classe parent

Héritage

- ★ Une relation d'héritage est souvent illustrée dans un *diagramme de classes*, à l'aide d'une flèche pointant vers la classe parent



L'héritage crée une *relation is-a*, ce qui veut dire que l'enfant *est une* version plus spécialisée du parent

Dérivation de Sous-classes

- ★ En Java, nous utilisons le mot réservé **extends** pour établir une relation d'héritage

```
class Car extends Vehicle
{
    // class contents
}
```

- ★ Voir [Words.java](#) (page 324)
- ★ Voir [Book.java](#) (page 325)
- ★ Voir [Dictionary.java](#) (page 326)

Contrôle de l'héritage

- ✱ Les modificateurs de visibilité déterminent quels membres d'une classe sont hérités et lesquels ne le sont pas
- ✱ Les variables et méthodes déclarées de visibilité `public` sont héritées, alors que celles contrairement à ceux dont la visibilité est `private`
- ✱ Par contre, des variables `public` vont à l'encontre de notre but d'encapsuler les données
- ✱ Il existe un troisième modificateur de visibilité qui est utile en situations d'héritage : `protected`

Le Modificateur `protected`

- ✱ Le modificateur de visibilité `protected` permet à un membre d'une classe de base d'être hérité par la classe enfant
- ✱ La visibilité `protected` offre plus d'encapsulation que `public`
- ✱ Par contre, la visibilité `protected` est moins encapsulé que la visibilité `private`
- ✱ Les détails de chaque modificateur sont donnés à l'Annexe F

La Référence super

- ★ Les constructeurs ne sont pas hérités, même s'ils ont une visibilité public
- ★ Par contre, il est souvent souhaitable de pouvoir utiliser le constructeur du parent pour initialiser la partie de l'objet héritée du parent
- ★ La référence **super** peut être utilisée pour faire référence à la classe parent. Celle-ci est souvent utilisée pour invoquer le constructeur du parent
- ★ Voir Words2.java (page 328)
- ★ Voir Book2.java (page 329)
- ★ Voir Dictionary2.java (page 330)

Héritage Simple vs. Multiple

- ✱ Java supporte l'*héritage simple*, donc une classe dérivée n'a qu'une seule classe parent
- ✱ L'*héritage Multiple* permet qu'une classe soit dérivée de deux classes ou plus, héritant des membres de tous ses parents
- ✱ Les collisions, telles le même nom de variable dans deux parents, doivent être résolues
- ✱ La plupart du temps, l'utilisation des interfaces nous permet les mêmes avantages que l'héritage multiple mais sans l'*overhead*

Redéfinition de Méthodes

- ★ Une classe enfant peut redéfinir une méthode héritée pour favoriser la sienne
- ★ C'est à dire, un enfant peut redéfinir une méthode qu'il hérite de son parent
- ★ La nouvelle méthode doit conserver la même signature que la méthode du parent, tout en ayant du code différent
- ★ Le type de l'objet qui exécute la méthode détermine quelle version de la méthode est invoquée

Redéfinition de Méthodes

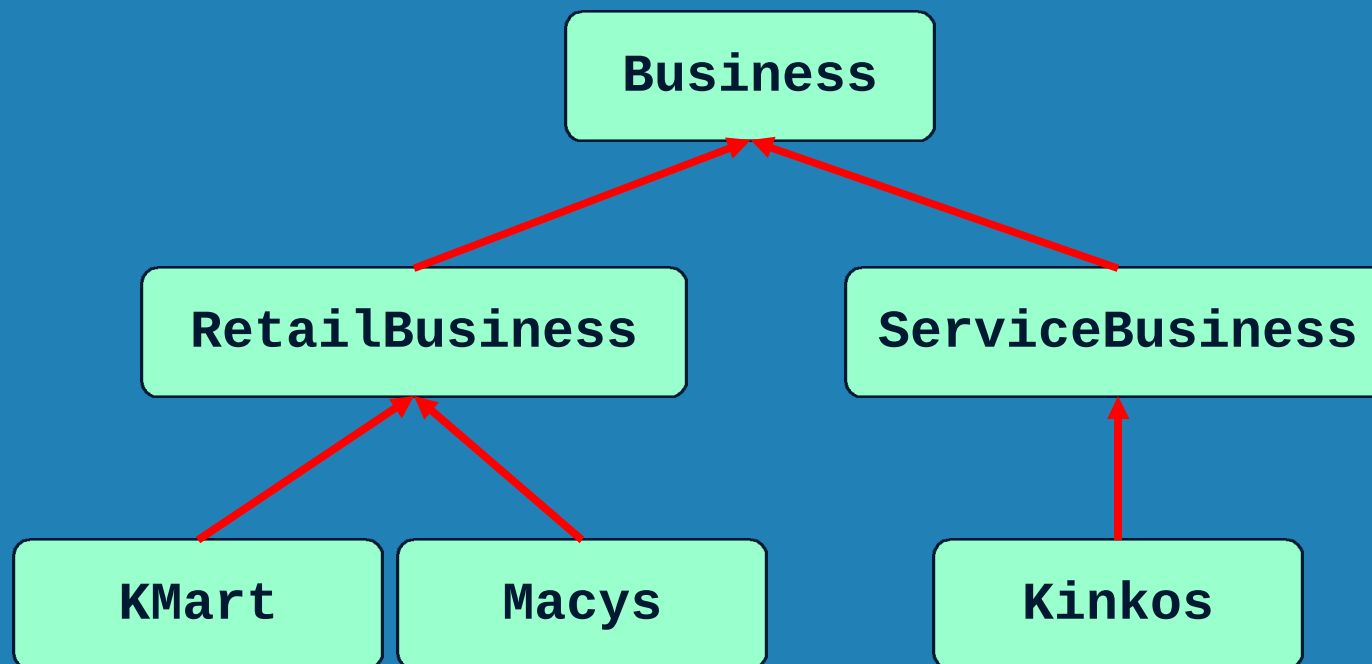
- ✱ Voir Messages.java (page 332)
- ✱ Voir Thought.java (page 333)
- ✱ Voir Advice.java (page 334)
- ✱ À noter, une méthode du parent peut être invoquée explicitement en utilisant la référence `super`
- ✱ si une méthode est déclarée avec le modificateur `final`, elle ne peut être redéfinie
- ✱ Le concept de redéfinition peut s'appliquer aux données (*shadowing variables*), mais généralement ce n'est pas utile

Surcharge vs. Surdéfinition

- ✱ Ne vous mélangez entre les concepts de surcharge et de redéfinition
- ✱ La surcharge traite de plusieurs méthodes d'une même classe ayant le même nom mais des signatures différentes
- ✱ La redéfinition traite de deux méthodes, une dans la classe parent et une dans la classe enfant, avec la même signature
- ✱ La surcharge permet de définir une opération similaire de façon différente pour des données différentes
- ✱ La redéfinition permet de définir une opération similaire de façon différente pour des objets différents

Hiérarchie de Classes

- ★ Une classe qui est l'enfant d'un parent peut être le parent d'un autre enfant, formant ainsi une *hiérarchie de classes*



Hiérarchie de Classes

- ★ Deux enfants du même parent sont appelés *frères*
- ★ Une bonne conception de classes va placer toutes les particularités communes aussi haut que possible dans la hiérarchie
- ★ Un membre hérité sera continuellement passé vers le bas de la hiérarchie
- ★ La hiérarchie de classes doit souvent être étendue et modifiée pour s'adapter à des besoins changeant
- ★ Il n'existe pas de hiérarchie de classes unique qui soit appropriée pour toutes les situations

La Classe Object

- ★ Une classe nommée **Object** est définie dans le package **java.lang** de la librairie standard de classes de Java
- ★ Toutes les classes sont dérivées de la classe **Object**
- ★ Si une classe n'est pas définie explicitement comme étant l'enfant d'une classe existante, elle est présumée être un enfant de la classe **Object**
- ★ La classe **Object** est donc la racine de toutes hiérarchies de classes

La Classe Object

- ✱ La classe **Object** contient quelques méthodes utiles, qui sont héritées par toutes les classes
- ✱ Par exemple, la méthode **toString** est définie dans la classe **Object**
- ✱ Chaque fois que nous avons définie **toString**, nous la redéfinissons
- ✱ La méthode **toString** dans la classe **Object** est définie pour retourner une chaîne de caractères contenant le nom de la classe de l'objet, un **@** et une valeur en hexa

La Classe Object

- ✱ C'est pourquoi la méthode `println` peut être appelée `toString` pour tous les objets qui lui sont passés – tous les objets sont assurés d'avoir une méthode `toString` via l'héritage
- ✱ Voir [Academia.java](#) (page 339)
- ✱ Voir [Student.java](#) (page 340)
- ✱ Voir [GradStudent.java](#) (page 341)
- ✱ La méthode `equals` de la classe `Object` détermine si deux références sont des alias
- ✱ Vous pouvez choisir de redéfinir `equals` pour vérifier l'égalité d'une autre façon

Classes Abstraites

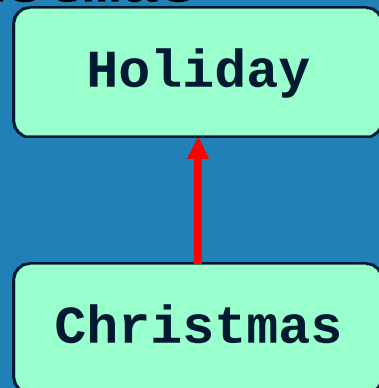
- ✱ Une classe abstraite représente un concept générique dans la hiérarchie de classes
- ✱ Une classe abstraite ne peut être instanciée
- ✱ Nous utilisons le modificateur `abstract` sur le prototype de la classe pour déclarer qu'elle est abstraite
- ✱ Une classe abstraite contient souvent des méthodes abstraites (comme les interfaces), par contre ce n'est pas nécessaire

Classes Abstraites

- ✱ L'enfant d'une classe abstraite doit redéfinir les méthodes abstraites de son parent, sinon elle sera aussi considérée abstraite
- ✱ Une méthode abstraite ne peut pas être définie comme `final` (car elle doit être redéfinie) ou `static` (car elle ne possède pas encore de définition)
- ✱ L'utilisation d'une classe abstraite est une décision de conception; elle aide à établir des éléments communs dans une classe trop générale pour être instanciée

Références et Héritage

- ★ Un référence sur un objet peut référer à un objet de sa classe, ou à un objet de toute classe qui lui est lié par héritage
- ★ Par exemple, si la classe `Holiday` est utilisée pour dériver une classe enfant `Christmas`, alors une référence sur `Holiday` pourrait être utilisée pour pointer sur un objet `Christmas`



```
Holiday day;  
day = new Christmas();
```

Références et Héritage

- ★ Affecter un objet descendant à une référence ancêtre est considéré une conversion non-dégradante, et peut se faire avec une simple affectation
- ★ Affecter un objet ancêtre à une référence descendante est aussi possible, par contre, c'est considéré une conversion dégradante et doit absolument être faite avec un *cast*
- ★ La conversion non-dégradante est la plus utile

Polymorphisme via Héritage

- ★ Dans le Chapitre 5, nous avons vu qu'une interface peut être utilisée pour créer une *référence polymorphe*
- ★ Souvenez-vous qu'une référence polymorphe peut faire référence à différents types d'objets à différent temps
- ★ L'héritage peut aussi servir de base au polymorphisme
- ★ Une référence sur un objet peut faire référence à un objet à un moment donné, elle peut ensuite changer pour faire référence à un autre objet (lié par l'héritage) à un autre moment

Polymorphisme via Héritage

- ✳ Supposons que la classe `Holiday` a une méthode nommée `celebrate`, et que la classe `Christmas` la redéfinie
- ✳ Considérons l'invocation suivante:

`day.celebrate();`

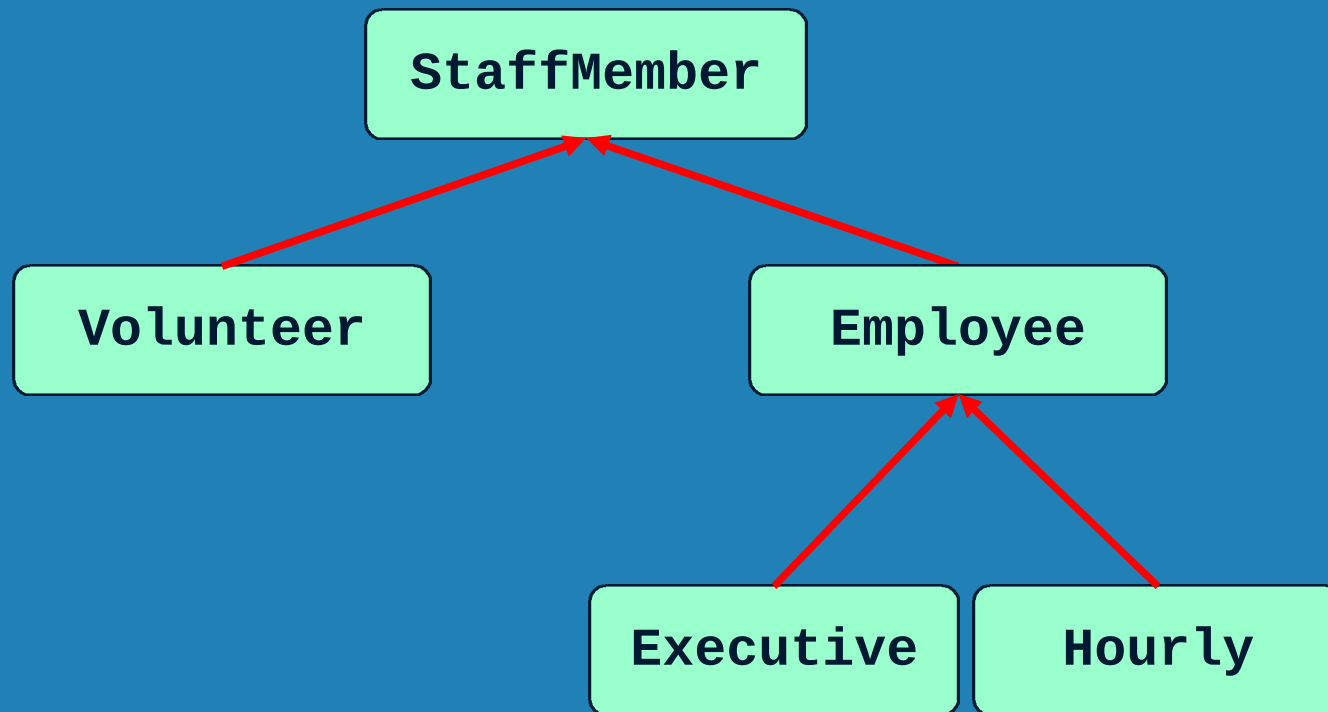
- ✳ Si `day` fait référence à un objet `Holiday`, il invoque la version `Holiday` de `celebrate`; S'il fait référence à un objet `Christmas`, il invoque la version `Christmas`

Polymorphisme via Héritage

- ✱ C'est le type de l'objet qui est référencé, et non le type de la référence, qui détermine quelle méthode sera invoquée
- ✱ Notez que, si une invocation est dans une boucle, la même ligne de code pourrait exécuter des méthodes différentes à des temps différents
- ✱ Les références polymorphes sont donc résolues à l'exécution, et non à la compilation

Polymorphisme via Héritage

- ★ Considérons la hiérarchie de classes suivante :



Polymorphisme via Héritage

- ✱ Considérons maintenant la tâche de payer les employés
- ✱ Voir Firm.java (page 345)
- ✱ Voir Staff.java (page 346)
- ✱ Voir StaffMember.java (page 348)
- ✱ Voir Volunteer.java (page 349)
- ✱ Voir Employee.java (page 351)
- ✱ Voir Executive.java (page 352)
- ✱ Voir Hourly.java (page 353)

Accès Indirect

- ✱ Un membre hérité peut être référencé directement par son nom dans la classe enfant, comme s'il était déclaré dans celle-ci
- ✱ Même si une méthode ou une variable n'est pas héritée par un enfant, elle peut être accédée indirectement à travers une méthode du parent
- ✱ Voir FoodAnalysis.java (page 355)
- ✱ Voir FoodItem.java (page 356)
- ✱ Voir Pizza.java (page 357)

Hiérarchies d'Interface

- ✱ L'héritage peut s'appliquer aux interfaces autant qu'aux classes
- ✱ Une interface peut être utilisée comme parent d'une autre
- ✱ L'interface enfant hérite de toutes les méthodes abstraites du parent
- ✱ Une classe qui implémente l'interface enfant doit définir toutes les méthodes des interfaces parent et enfant
- ✱ Notez que la hiérarchie de classes et la hiérarchie des interfaces sont distinctes (elles ne se chevauchent pas)

Applets et Héritage

- ★ Une applet est une excellente exemple d'héritage
- ★ Rappelez-vous qu'en définissant une applet, nous dérivons (*extend*) la classe `Applet`
- ★ La classe `Applet` peut déjà gérer les détails de sa création et de son exécution, incluant l'interaction avec le fureteur web
- ★ Nos classes applet n'ont qu'à gérer les aspects qui touchent spécifiquement à notre applet particulière

Dériver les Classes Event Adapter

- ★ Au chapitre 5, nous avons discuté de la création de classes d'auditeurs (*listener*) en implémentant une interface particulière (ex. `MouseListener`)
- ★ Un auditeur peut aussi être créé en dérivant une classe *adapter* spéciale de la librairie de classes de Java
- ★ Chaque interface *listener* a une classe *adapter* correspondante (par ex. la classe `MouseAdapter`)
- ★ Chaque classe *adapter* implémente le *listener* correspondant et fournit des définitions de méthodes vides

Dériver les Classes Event Adapter

- ★ Lorsque vous dérivez une classe *listener* à partir d'une classe *adapter*, vous redéfinissez toutes les méthodes d'événements intéressante (comme la méthode `mouseClicked`)
- ★ Notez que ceci élimine le besoin de créer des définitions vide pour des événements non utilisés
- ★ Voir [OffCenter.java](#) (page 360)

Composantes du GUI

- ★ Une *composante du GUI* est un objet représentant une entité visuelle dans une interface usager graphique (comme un bouton ou un slider)
- ★ Les composantes peuvent générer des événements auxquels les objets *listener* peuvent réagir
- ★ Par exemple, une applet est une composante qui peut générer des événements de souris
- ★ Une applet est aussi une sorte spéciale de composante, appelée un *container*, dans lequel d'autres composantes peuvent être placées

Composantes du GUI

- ✱ Voir Fahrenheit.java (page 363)
- ✱ Les composantes sont organisées en hiérarchie de classes héritées de manière à pouvoir partager des caractéristiques
- ✱ Lorsque nous définissons certaines méthodes, comme la méthode `paint` d'une applet, nous redéfinissons une méthode définie dans la classe `Component`, qui est héritée par la classe `Applet`
- ✱ Voir Doodle.java (page 367)
- ✱ Voir DoodleCanvas.java (page 369)