

Programmation orientée objet en JAVA

CH3 (Héritage)

Héritage et Polymorphisme

Classes et Sous-classes

Ou

Extension de Classes

Histoire de l'héritage

- Dans les années soixante, deux programmeurs ont créés un programme pour simuler le trafic
 - Ils ont utilisé des objets pour leurs véhicules
 - Cars
 - Trucks
 - Bikes
 - Ils ont remarqué que tous les véhicules faisaient les mêmes choses
 - Turn left (Tourner à gauche)
 - Turn right (Tourner à droite)
 - Brake (Freiner)
 - Go (aller)

Plan #1 - Les Objets Bike, Car and Truck

- Créer une classe pour chaque véhicule
 - Bike
 - Car
 - Truck

Bike

TurnLeft()

TurnRight()

Brake()

Go()

Car

TurnLeft()

TurnRight()

Brake()

Go()

Truck

TurnLeft()

TurnRight()

Brake()

Go()

Plan #1 - Avantages

- C'est rapide et facile à comprendre

Bike

TurnLeft()

TurnRight()

Brake()

Go()

Car

urnLeft()

TurnRight()

Brake()

Go()

Truck

TurnLeft()

TurnRight()

Brake()

Go()

Plan #1 - DesAvantages

- Le Code est répété dans chaque objet
 - Modification du code dans Brake () nécessite 3 changements dans 3 objets différents
- Les noms de méthodes peuvent être modifiés.
 - Après un certain temps, les objets ne sont pas similaires

Bike

TurnLeft() -> Left()

TurnRight() -> Right()

Brake()

Go()

Car

TurnLeft()

TurnRight()

Brake()

Go() -> Move()

Trucks

TurnLeft()

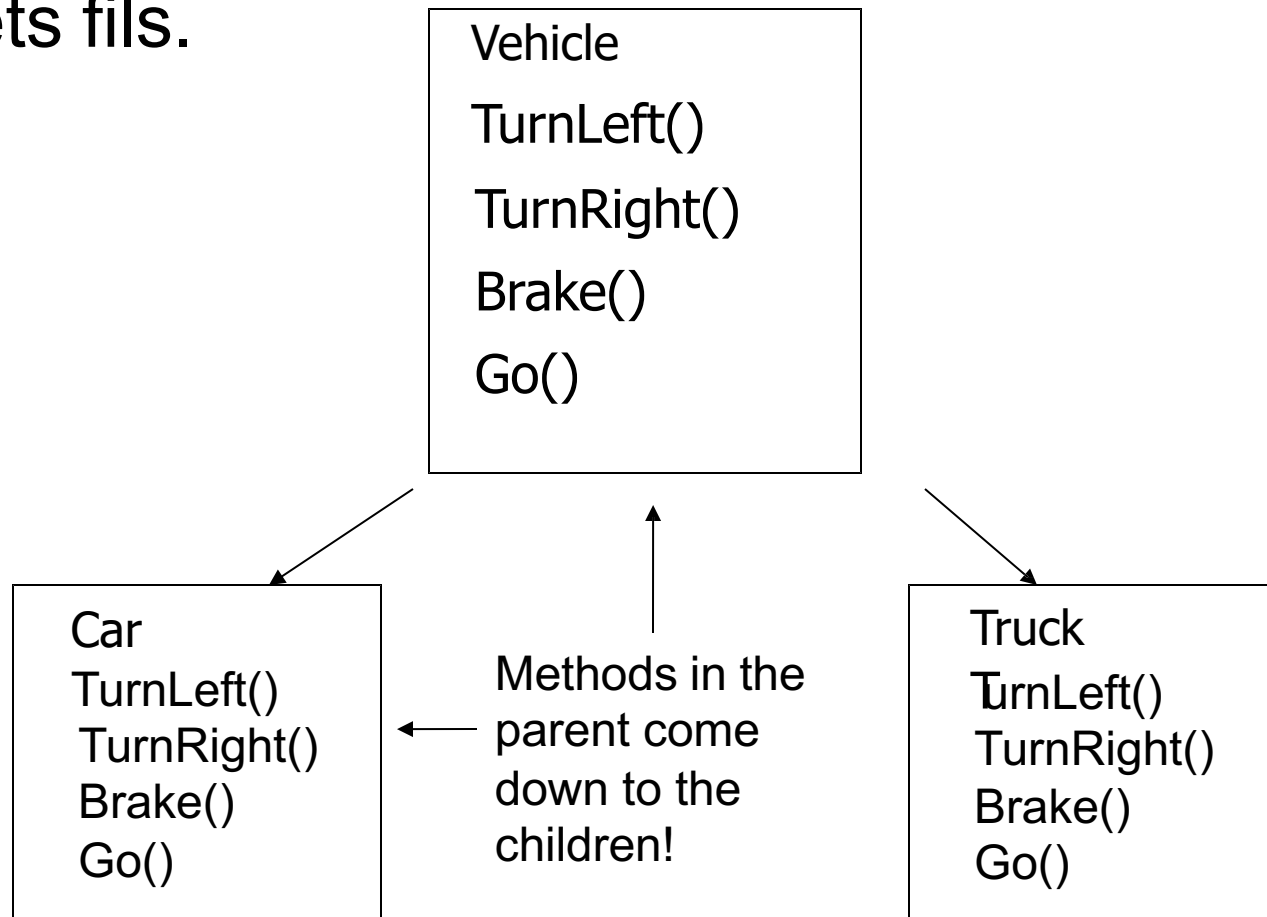
TurnRight()

Brake()

Go() -> Start()

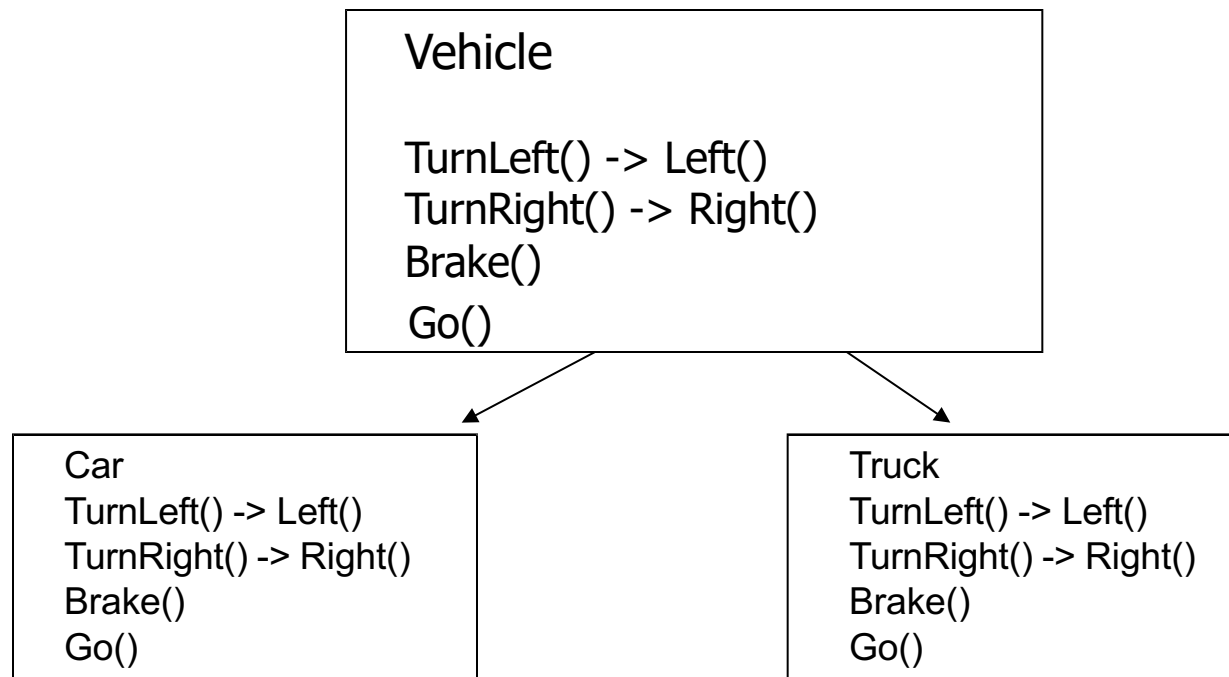
Plan #2 - Héritage

- Faire un objet avec des méthodes en commun.
- Le code de l'objet parent est utilisé dans les objets fils.



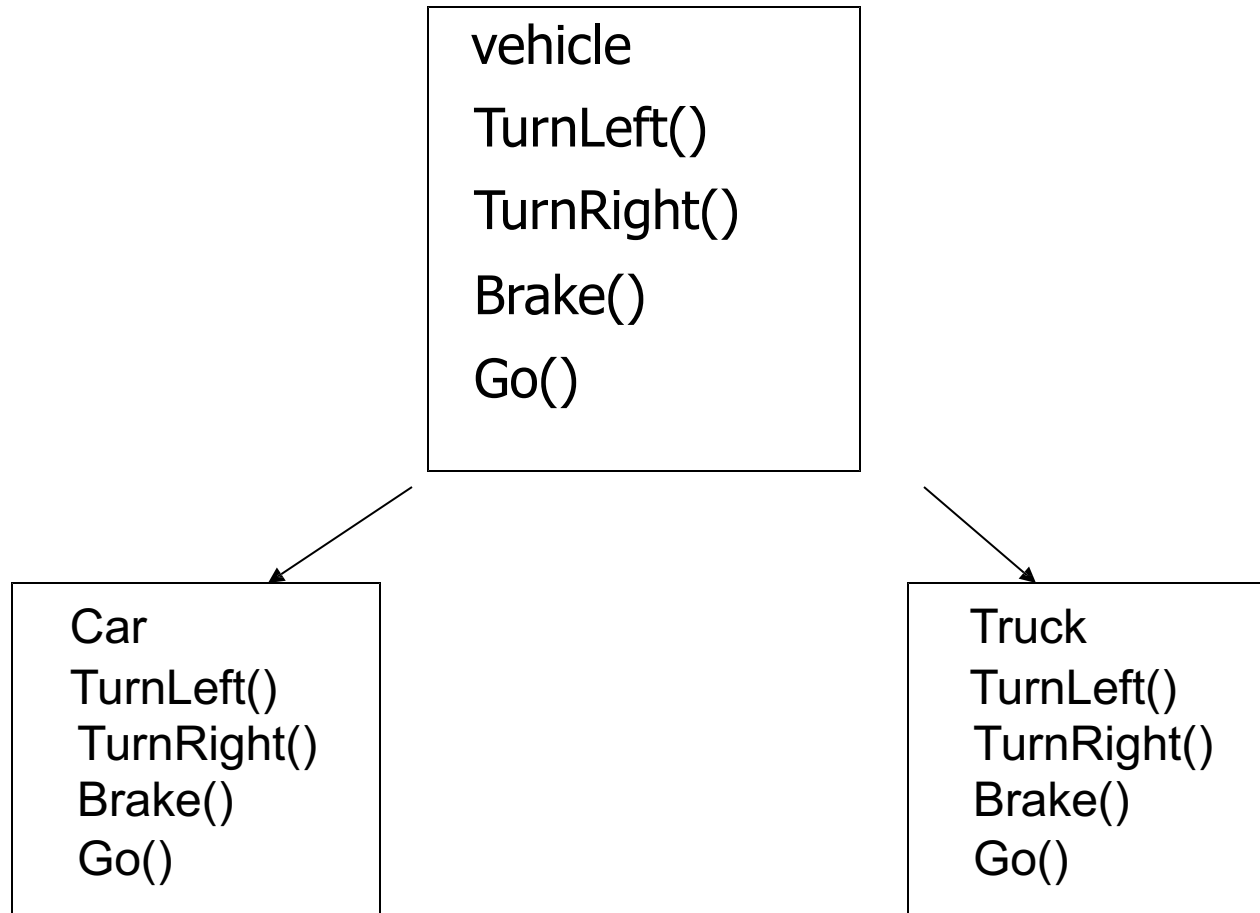
Plan #2 – Avantages

- Un changement dans le code de la méthode dans le parent change automatiquement les classes des enfants
 - Le code de la méthode est cohérent et facile à maintenir
- Un changement dans le nom de la méthode dans le parent change automatiquement les fils.
 - Les noms de la méthode est cohérent et facile à maintenir
- Nous pouvons changer une classe que quelqu'un d'autre a créé
 - Il est difficile d'écrire votre propre classe de bouton. Mais nous pouvons ajouter des changements à la classe de bouton en utilisant l'héritage



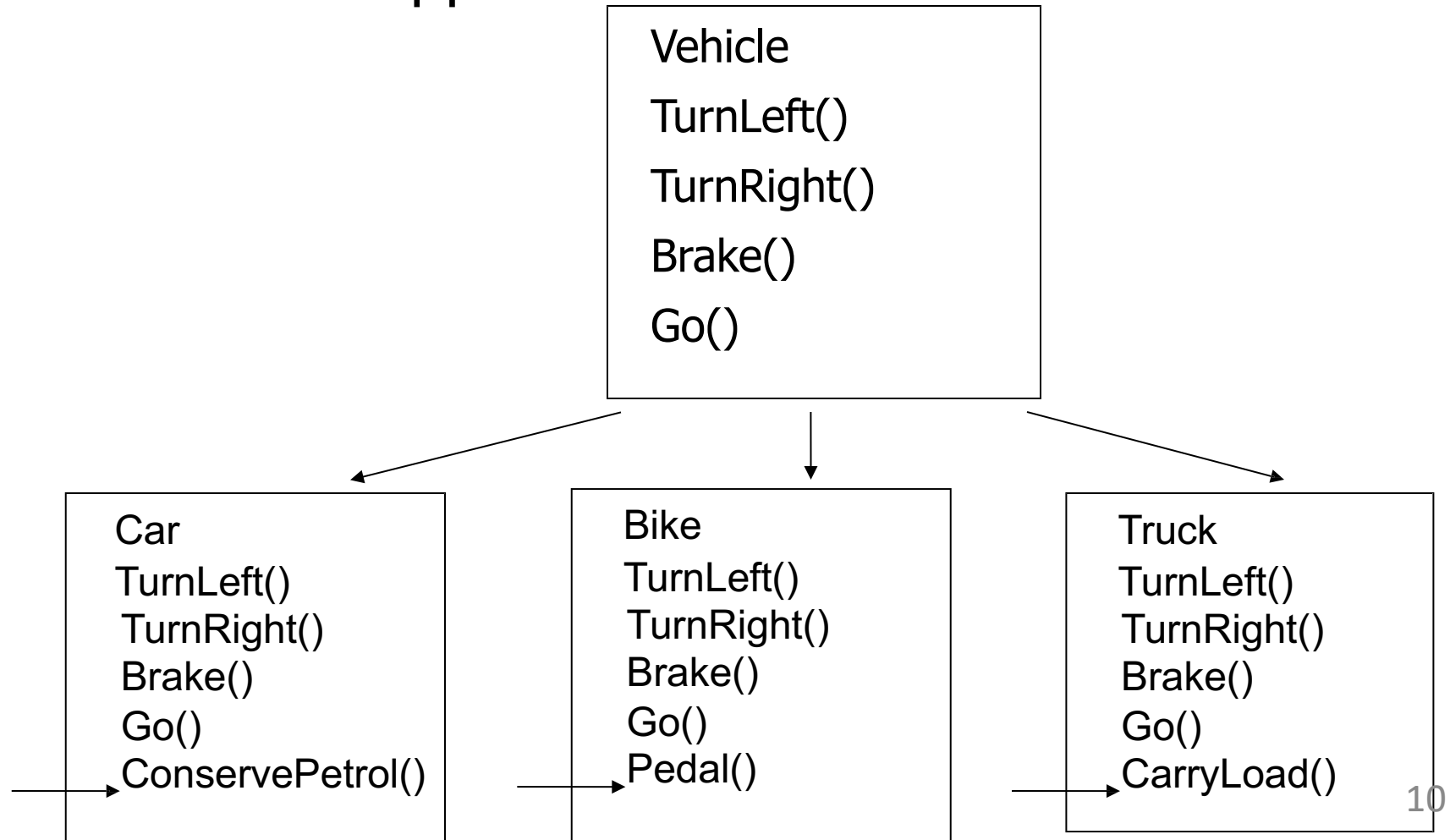
Plan #2 – DesAvantages

- L'héritage exige du code spécial
- L'héritage exige plus de compréhension



Différences dans l'héritage

- Chaque objet enfant peut avoir différents membres supplémentaires.



La Réutilisation

- Réutilisation -- la construction de nouveaux composants en utilisant des composants existants-- est encore un autre aspect important de paradigme OO..
- Il est toujours bon / "productif" si nous sommes en mesure de réutiliser quelque chose qui existe déjà plutôt que de créer la même chose une fois de plus.
- En utilisant des composants logiciels existants pour en créer de nouveaux, nous tirons profit de tout l'effort qui est investi dans la conception, la mise en œuvre, et l'essai du logiciel existant
- *La réutilisation du logiciel* est au cœur de l'héritage
- Ceci est réalisé en créant de nouvelles classes, par la réutilisation des propriétés de classes existantes.

L'héritage (*Réutilisation*)

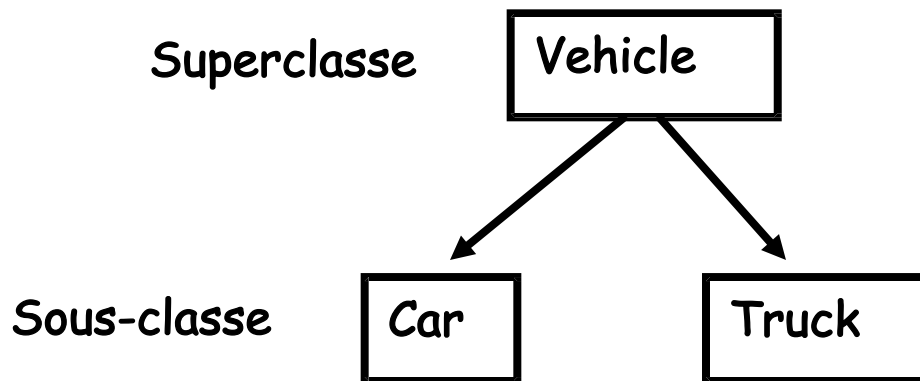
- Outre la composition, l'héritage est une autre technique orienté objet fondamental utilisé pour organiser et créer des classes réutilisables. C'est une forme de réutilisation de logiciels
- *L'héritage* permet à un développeur logiciel de dériver une nouvelle classe à partir d'une existante

Introduction à l'héritage

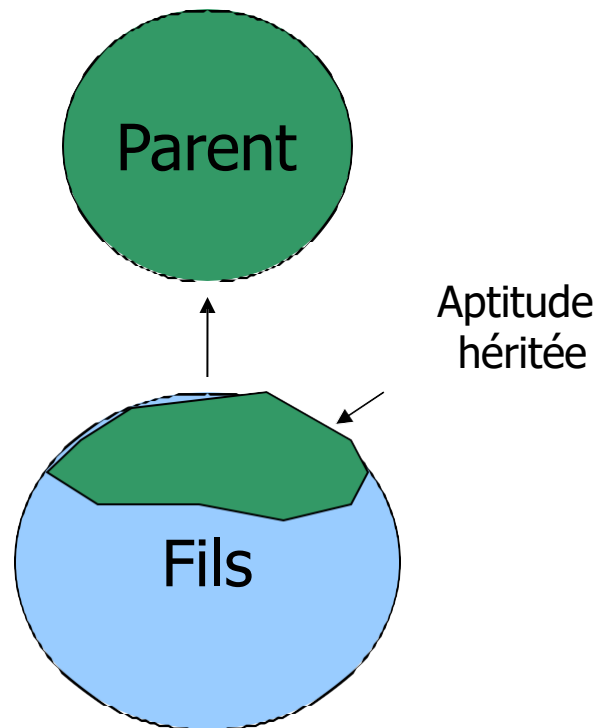
- L'héritage est une construction orientée objet qui favorise la réutilisation du code en permettant de créer une nouvelle classe en utilisant des classes existantes comme un "point de départ"..
 - On declare simplement quelle classe qu'on souhaite utiliser comme une base pour hériter de ses membres (champs et méthodes).
 - Vous pouvez ajouter ou modifier ses membres une fois qu'on en a hérité
- Terminologie
 - Superclasse (classe parent, classe de base)
 - La classe d'origine dont les membres sont hérités
 - Typiquement, c'est la classe plus générale
 - Sous-class (classe fils, classe dérivée)
 - La classe qui hérite les membres de la super-classe
 - Dans cette classe, nous pouvons spécialiser la classe générale par la modification et l'ajout de fonctionnalités.

Introduction à l'héritage

- Quand une classe **ré-utilise les capacités** définies dans une autre classe.
- La nouvelle **sous-classe** gagne toutes les méthodes et les attributs de la **superclasse**



Introduction à l'héritage



Gains de l'héritage

- Les classes partagent souvent des fonctionnalités (capacités/aptitudes)
- Nous voulons éviter de re-coder ces fonctionnalités (capacités/aptitudes)
- La réutilisation de ces fonctionnalités serait préférable pour:
 - Améliorer la maintenabilité
 - Réduire les coûts
 - Améliorer la modélisation du "monde réel"

Bénéfices de l'héritage

- Avec l'héritage, un objet peut hériter le comportement d'un autre objet, réutilisant ainsi son code.
- **Epargner l'effort** de "réinventer la roue"
- Permet de construire sur le code existant, **spécialisé** sans avoir à copier, réécrire, etc.
- Pour créer la sous-classe, nous avons besoin de programmer **uniquement les différences** entre la superclasse et la sous-classe qui en hérite.
- Permet de la **flexibilité** dans les définitions de classe.

Héritage

- Pour adapter une classe dérivée, le programmeur peut ajouter de nouvelles variables ou méthodes, ou peut modifier celles hérités
- Créer une nouvelle classe d'une classe existante
 - Absorber les données et les comportements de la classe existante
 - Améliorer avec de nouvelles capacités ou des capacités modifiées
- Utilisée pour éliminer du code redondant
- Exemple
 - Classe Cercle hérite de la classe Forme
 - Cercle *extends* Forme

Définir une Sous-classe

```
class NomSousClasse extends NomSuperClasse  
{  
  //déclaration des champs;  
  //déclaration des méthodes;  
}
```

```
public class Chat extends Animal { ...
```

```
public class Chien extends Animal { ...
```

Héritage

Une classe *dérivée* étend une classe de *base*. Elle hérite de toutes ses méthodes (comportements) et ses attributs (données) et elle peut avoir des comportements et des attributs supplémentaires qui lui sont propres.

Classe de Base

class A
Attributs de la class de Base
Méthodes de la class de Base

Classe Dérivée

class B extends A
attributs hérités de base Attributs supplémentaires
méthodes hérités de base Méthodes supplémentaires

Définition des Méthodes dans la Classe fils

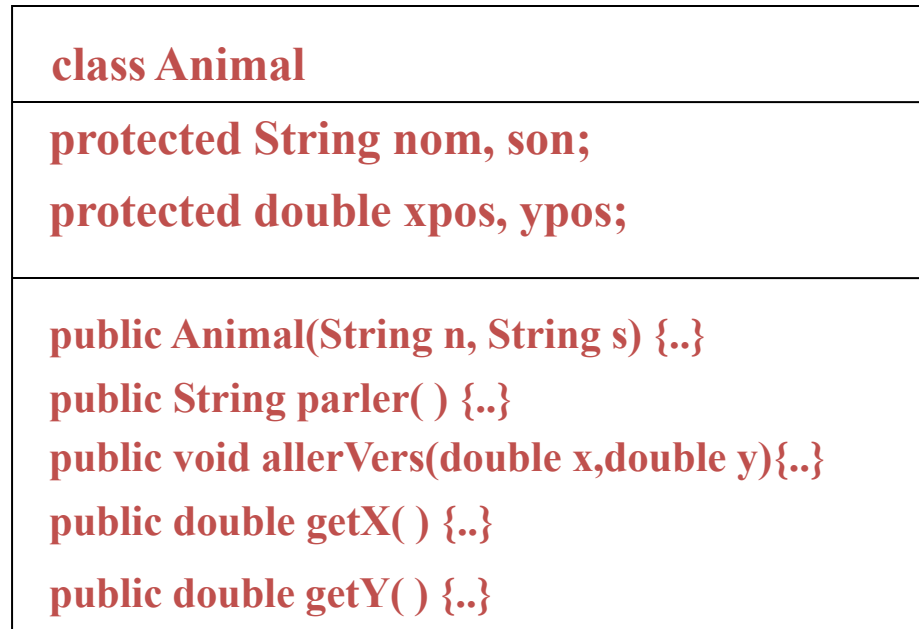
- Une classe fils peut (*passer outre/primer sur*) **remplacer** la définition d'une méthode héritée en faveur d'une autre méthode propre à elle.
 - càd, un fils peut redéfinir une méthode qu'il hérite de son parent
 - la nouvelle méthode doit avoir la même signature que la méthode du parent, mais peut avoir un code différent dans le corps
- En Java, toutes les méthodes sauf les constructeurs redefinissent les méthodes de leur classe ancêtre par **remplacement**. E.g.:
 - la classe *Animal* a une méthode *manger()*
 - la classe *Oiseau* a une méthode *manger()* et *Oiseau extends Animal*
 - la variable *b* est de la classe *Oiseau*, i.e. *Oiseau o = ...*
 - *o.manger()* simplement invoque la méthode *manger()* de la class *Oiseau*

Héritage

Classe de base

Les classes dérivées
ont leurs propres
constructeurs

Les classes dérivées
peuvent ajouter leurs
propres
comportements
uniques

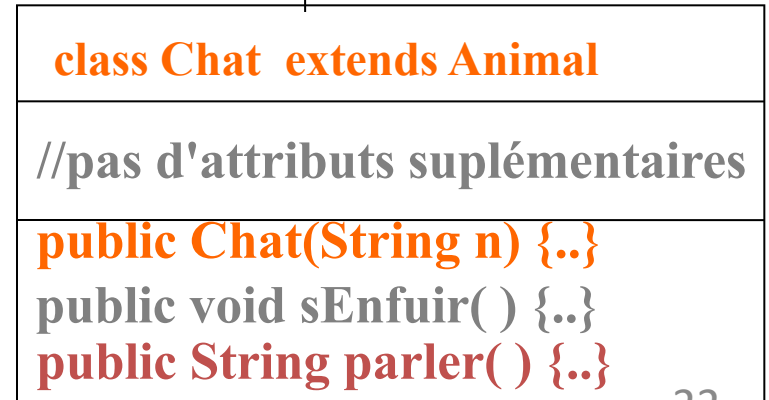
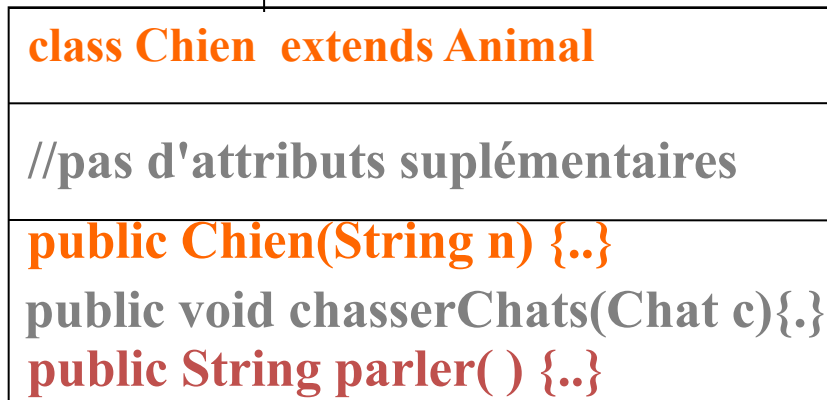


← nom de la classe

← attributs
(données)

← méthodes
(comportement)

Classes Dérivées



Redéfinition de méthodes

- Souvent, une sous-classe (fils) veut changer le comportement d'une classe (parent) sans changer son interface. Dans ce cas, la sous-classe peut redéfinir les méthodes héritées qu'elle veut changer en sa faveur.
- Pour remplacer (redéfinir) une méthode, vous devez re-déclarer (définir / redéfinir) la méthode dans la sous-classe (le fils). La nouvelle méthode doit avoir la même signature exacte que la méthode de la superclasse (méthode des parents) que vous remplacez, mais peut avoir un corps différent (implémentation).

- Dans Animal:

```
public String parler { return null; }  
//On ne sait pas comment un animal communique en general.
```

- Dans Chat:

```
public String parler { return "Meow"; }
```

- Dans Chien:

```
public String parler { return "Bow-Wow"; }
```

Redéfinition de méthodes

- Dans Vehicle:

```
public boolean aReservoir { return false; }  
//Bicyclette et calèche n'ont en pas.
```

- Dans Car:

```
public boolean aReservoir{ return true; }
```

Ici nous disons que la méthode `aReservoir` de `Car` remplace `aReservoir` de `vehicle`: elle se comporte différemment

- La classe / type de l'objet utilisé pour exécuter une méthode redéfinie détermine quelle version de la méthode est invoquée
- Si on a `Vehicle v` et `Car c`, alors

```
v.aReservoir() returns false
```

```
c.aReservoir() returns true
```


Redéfinition de méthodes

```
Chat c = new Chat();  
c.allerVers(5,3);
```

Java va d'abord chercher `allerVers(double x, double y)` dans la classe `Chat`. Il ne la trouvera pas, il cherchera alors dans la superclasse. Là il la trouvera, la méthode est alors invoquée.

Maintenant considérons:

```
c.parler();
```

`parler()` existe dans les *deux* classes `Chat` et dans sa superclasse, `Animal`. Notez aussi qu'ils ont la même signature exacte. Toutefois, parce que Java commence à regarder dans la classe courante, la version de `parler()` qui est invoquée, sera celle de la classe `Chat`. **Dans ce cas, la version de la méthode qui était dans la classe dérivée (`Chat`) a remplacé (*Overrode / Override*) la version de la méthode qui était dans la classe parent.**

Redéfinition de méthodes

Overriding methods

- Lorsqu' on remplace (redéfinit/Override) une méthode:
 - On doit avoir la même signature exacte
 - Sinon, on *surcharge* juste la méthode, et les deux versions de la méthode sont disponibles

Overriding (Redéfinition)

- Si une méthode est déclarée avec le modificateur `final`, elle ne peut pas être remplacé
- Seules les méthodes non-privés peuvent être redéfinies parce que les méthodes privées ne sont pas visibles dans la sous-classe.

Overriding methods

- Remarque: On ne peut pas remplacer les méthodes *static*.
- Remarque: On ne peut pas remplacer des champs de données.
- Dans les deux cas ci-dessus, lorsqu'on tente de les remplacer (redéfinir), On est entrain de vraiment les cacher.

Overriding methods

Redéfinition de méthodes

- Lorsqu'on remplace (redéfinit) une méthode, on ne peut pas la rendre plus privée
 - Dans cet exemple, **Animal** définit une méthode
 - *Chaque* sous-classe **Animal** doit hériter de cette méthode, y compris les sous-classes de sous-classes
 - Rendre une méthode plus privée irait à l'encontre l'héritage

Overriding methods

Redéfinition de méthodes

- Une sous-classe peut remplacer (redéfinir) les méthodes de la superclasse
 - Les objets de type sous-classe utiliseront la nouvelle méthode
 - Les objets de type superclasse vont utiliser l'original

Override/Redéfinir une méthode permet à la sous-classe:

- d'étendre la méthode dans la super-classe et
- modifier le comportement dans la super classe de sorte que ça convienne au contexte de la sous-classe.

Example of Overriding

- Here we override to change method `getIdentifier()`

```
public class Person {  
    private String name;  
    private String noCIN;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getIdentifier() {  
        return noCIN;  
    }  
}
```

```
public class Student extends Person {  
    private String studentId;  
  
    public String getIdentifier() { // override  
        return studentId; // use student id instead of noCIN  
    }  
}
```

La Généralisation

Héritage

L'héritage exprime une association *Est-un* entre deux (ou plusieurs) classes. Un objet de la classe dérivée hérite de tous les attributs et les comportements de la classe de base et peut avoir des fonctionnalités supplémentaires {attributs et/ou comportements} qui lui sont propres. Ceci est ce qui est véhiculée par le mot-clé *extends*.

Une classe dérivée ne devrait pas hériter d'une classe de base pour obtenir certains, mais pas tous, de ses caractéristiques. Une telle utilisation de l'héritage est possible en Java (et se fait dans la pratique trop souvent), mais il est une utilisation incorrecte de l'héritage et doit être évitée!!

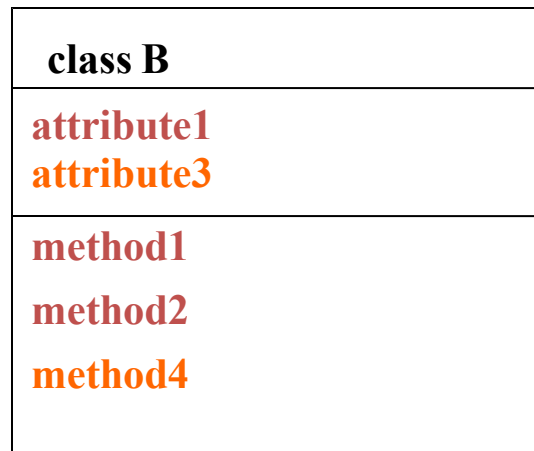
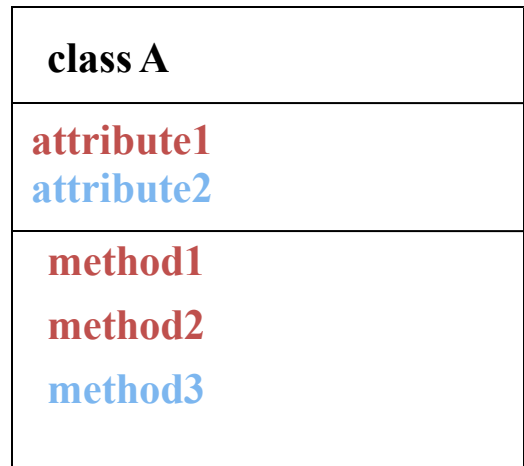
Au lieu d'hériter pour extraire une partie, mais non la totalité, des caractéristiques d'une classe parent, plutôt extraire (généraliser) les caractéristiques communes des deux classes et les placer dans une troisième classe à partir de laquelle les deux autres héritent.

La Généralisation ne concerne que les classes qu'on construit soi-même. On ne dispose pas de la capacité de généraliser les classes de la bibliothèque standard!

Héritage

Généralisation Les Classes A and B extend C avec des caractéristiques particulières à chacune

Considerons deux classes A et B qui ont certaines caractéristiques communes



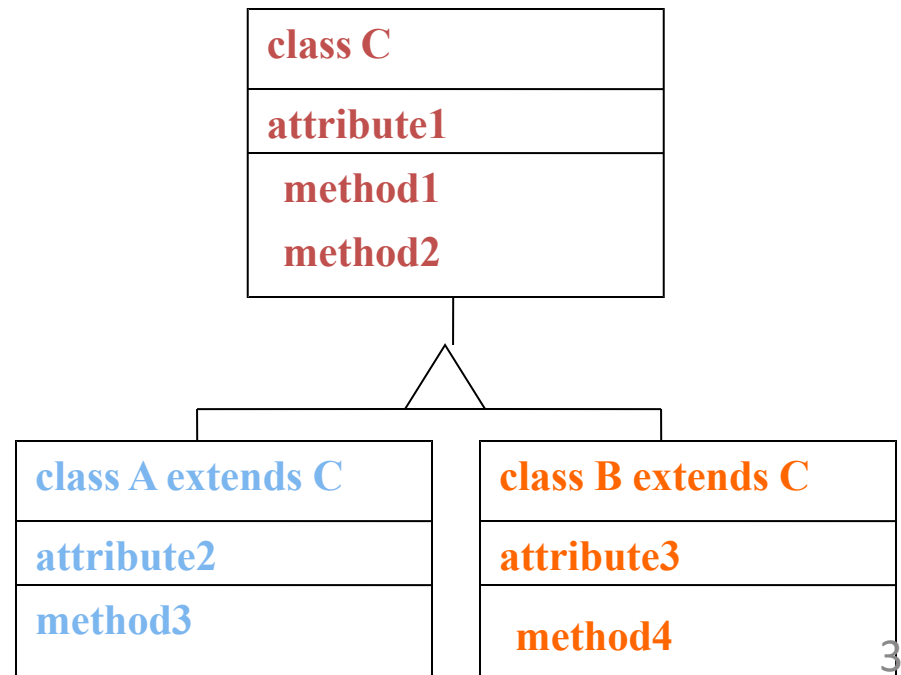
Caractéristiques commune à A et B



Caractéristiques appartenant seulement à A



Caractéristiques appartenant seulement à B



Le mot-clé `super`

- Le mot-clé ***super*** fait référence à la superclasse de la classe dans laquelle `super` apparaît. Le mot-clé `super` peut être utilisé dans une sous-classe pour désigner les membres (variables et méthodes définies dans la classe parent) de la superclasse de la sous-classe. Il est utilisé de deux façons:
 - Pour appeler un constructeur de la superclasse. Par exemple (note: le mot-clé *new* n'est pas utilisé):
 - `super () ;`
 - `super (params) ;`
 - Pour appeler une méthode de superclasse (il est seulement nécessaire de le faire lorsqu'on remplace --redéfinit-- la méthode).
 - Par exemple:
 - `super.superclassName () ;`
 - `super.superclassName (params) ;`

Method Overriding

- Si une sous-classe a une méthode d'une super-classe (même signature), cette méthode remplace (redéfinit) la méthode de la superclasse:

```
public class A { ...  
    public int M (float f, String s) { bodyA }  
}
```

```
public class B extends A { ...  
    public int M (float f, String s) { bodyB }  
}
```

- Si nous appelons **M** sur une instance de **B** (ou sous-classe de **B**), bodyB s'exécute
- Dans **B** on peut accéder **bodyA** avec: super.M(...)
- La méthode **M** de la sous-classe doit avoir le *même type de retour* que la méthode **M** de la superclasse *si elle a un type de retour primitif*. Si c'est une référence, il peut être une sous-classe de la classe du type de retour de la méthode **M** de la superclasse (covariance).

Note

- Une méthode d'instance ne peut être redéfinie que si elle est accessible. Ainsi, ***une méthode privée ne peut pas être remplacé***, parce qu'elle n'est pas accessible en dehors de sa propre classe.
- ***Si une méthode définie dans une sous-classe est privé dans sa superclasse, les deux méthodes sont complètement indépendantes.***

Héritage

Constructeurs pour Classes Derivées

Chaînage de constructeurs

- Supposons que la classe Animal a les constructeurs
Animal(), Animal(int poids), Animal(int poids, int dureeDeVie)
- Supposons que la classe Oiseau qui hérite de Animal a les constructeurs
Oiseau(), Oiseau(int poids), Oiseau(int poids, int dureeDeVie)
- Disons que nous créons un objet Oiseau, e.g. Oiseau o = new Oiseau(5)
- Ceci invoquera **en premier** le constructeur de la classe Animal (la superclasse de Oiseau) et **puis** le constructeur de la classe Oiseau
- Ceci est appelé **constructor chaining (chaînage de constructeurs)**:
Si la classe C0 extends C1 et C1 extends C2 et ... Cn-1 extends Cn = Object
puis lors de la création d'une instance de l'objet C0 le constructeur de Cn est
invoqué **en premier**, puis les constructeurs de Cn-1, ..., C2, C1, et finalement
le constructeur de C
 - Les constructeurs (dans chaque cas) sont choisis par leur signature, par exemple,
(), (Int), etc ...
 - Si aucun constructeur avec la signature correspondante n'est trouvé dans l'une des classe
Ci pour i > 0 alors le constructeur par défaut est exécuté pour cette classe

La Référence `super`

- Les constructeurs sont pas hérités, même si ils ont une visibilité *public*
- Pourtant, nous avons souvent besoin d'utiliser le constructeur de la classe parent à mettre en place la "partie parent" de l'objet
- Un constructeur de la classe fils est responsable de l'appel du constructeur de la classe parent. Il est appelé implicitement ou peut être invoquée explicitement en utilisant le mot-clé *super*.
- La référence `super` peut être utilisée pour faire référence à la classe parent, et est souvent utilisée pour invoquer le constructeur de la classe parent.

NOTE

- Invoquer le nom d'un constructeur de la superclasse dans une sous-classe provoque une erreur de syntaxe
- Il peut seulement être appelé à partir des constructeurs de les sous-classe, en utilisant le mot-clé super.

Appel de `super()` ou `super(params)`

- Java exige que la déclaration qui utilise le mot-clé *super* soit la première qui apparait dans le constructeur.

par conséquent

- La première ligne du constructeur du fils doit utiliser la référence `super` pour appeler le constructeur de la classe parent

Superclass

```
public class Person{
    protected String name;

    public Person() {
        name = "no_name_yet";
    }

    public Person(String
                    initialName) {
        this.name = initialName;
    }

    public String getName() {
        return name;
    }

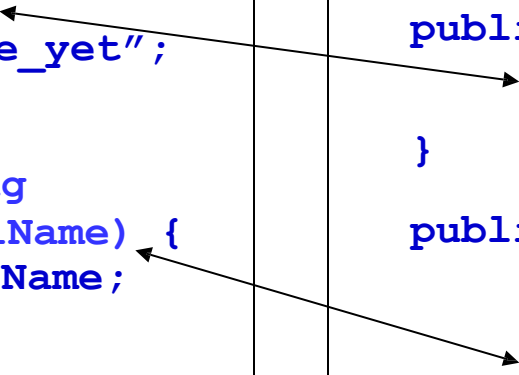
    public void setName(String
                        newName) {
        name = newName;
    }
}
```

Subclass

```
public class Student extends
    Person {
    private int studentNumber;

    public Student() {
        super(); // superclass
        studentNumber = 0;
    }

    public Student(String
                    initialName,
                    int initialStudentNumber) {
        super(initialName);
        studentNumber =
            initialStudentNumber;
    }
}
```



Inheritance and constructors

- *Si le mot-clé **super** n'est pas explicitement utilisé, le constructeur de la sous-classe appelle automatiquement le constructeur sans arguments de sa superclasse avant d'exécuter tout code dans son propre constructeur (Le compilateur insère **"super ()"** comme la première instruction dans le constructeur).*
 - Fonctionne uniquement, si la superclasse possède un constructeur sans paramètres.
- Si on ne souhaite pas utiliser le constructeur sans arguments du parent, on doit explicitement invoquer (avec le mot-clé **super**) un des autres constructeurs de la superclasse.
 - Si on fait cela, l'appel au constructeur de la superclasse doit être la première ligne de code dans le constructeur de la sous-classe c-à-d:

```
public class Fils extends Parent
{
    public Fils()
    {
        // un appel à Parent()
        super(); // la 1ère instruction
        // ou super(params);
        // plus n'importe quel code dont on besoin pour le Fils
    }
}
```

Si la superclasse n'a pas de constructeur sans arguments (n'a que des constructeurs avec paramètres), la première ligne de tout constructeur de la sous-classe DOIT explicitement invoquer un autre constructeur de la superclasse. Si cela n'est pas fait alors c'est une erreur.

Shape, Circle, Rectangle

- Considerons une superclasse: Shape
- Deux sous-classes: Cercle, Rectangle
- Storer le code et les data qui sont communes à toutes les sous-classes dans la superclasse, **Shape**:
 - color
 - getColor()
 - setColor()
- Cercle et Rectangle **héritent** toutes les variables d'instances et les méthodes qu'a la classe Shape.
- Cercle and Rectangle sont permises de définir de **nouvelles variables d'instances et de nouvelles méthodes** qui sont spécifiques à eux.

Cercle:

- center, radius
- getArea(), getPerimeter()

Shape class

```
public class Shape {  
    private String color;  
    public Shape() {  
        color = "red";  
    }  
    public String getColor() { return color;}  
    public String setColor( String newColor)  
    {  
        color = newColor;  
    }  
    public String toString() {  
        return "[" + color + "];"  
    }  
}
```

```
public class Circle extends Shape {  
    public static final double PI = 3.14159;  
    private Point center;  
    private double radius;  
  
    public Circle() {  
        super(); // calls the constructor of the superclass  
        center = new Point(0,0,0);  
        radius = 1.0;  
    }  
    public double getArea() {  
        return PI * radius * radius;  
    }  
    public double getPerimeter() {  
        return 2 * PI * radius;  
    }  
    public String toString() {  
        return super.toString() + "[" + center + "," + radius + "];"  
    }  
}
```

Important

Les Constructeurs d'une sous-classe peuvent et (devraient) initialiser seulement (directement) les champs de données de la sous-classe.

Qu'en est-il des champs de données d'une superclasse?

les initialiser en invoquant un constructeur de la superclasse avec les paramètres appropriés

L'initialisation des champs de la classe mère (parent) doit être effectuée en appelant le constructeur de la classe mère.. Les Champs de données privées appartenant à une superclasse doivent être initialisés en invoquant le constructeur de la superclasse avec les paramètres appropriés

`super(...);`

Si le constructeur de la sous-classe saute l'appel à la superclasse ... Java appelle automatiquement le constructeur par défaut (sans paramètres) qui initialise cette partie de l'objet héritée de la superclasse avant que la sous-classe commence à initialiser sa part de l'objet .

Point: S'assurer que les champs de données de la superclasse sont initialisés avant que la sous-classe commence à initialiser sa part de l'objet

Exemple

Le constructeur 2-arg de la classe
Circle:

```
public Circle(double radius, String color) {  
    setColor(color);  
    this.radius = radius;  
}
```

Peut être remplacé par:

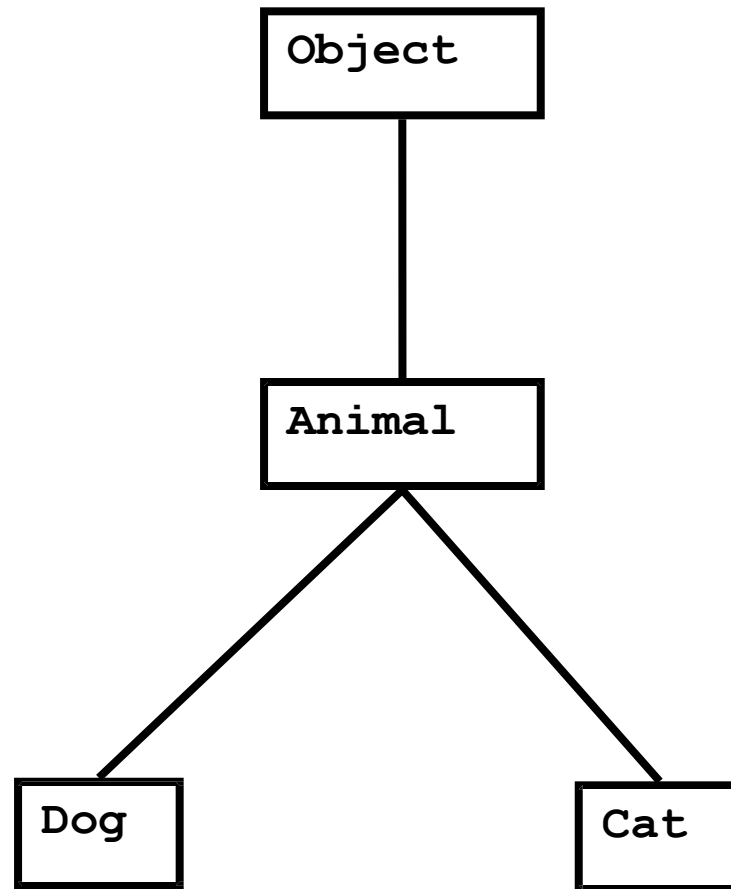
```
public Circle(double radius, String color) {  
    super(color);  
  
    this.radius = radius;  
}
```

super & this

- **super** signifie “regarder dans la superclasse”
- Constructeur: `super();`
- Méthode: `super.m();`
- champ: `super.x;`

- **this** signifie “regarder dans cette classe”
- Constructeur: `this();`
- Méthode: `this.m();`
- champ: `this.x;`

Inheritance Changes the Rules



Création d'un objet

Processus de création de l'objet

```
Chien c = new Chien ();
```

1. Créer la référence c
2. Commencer à créer Chien en entrant le constructeur de Chien et faisant appel au parent.
3. Commencer à créer Animal en entrant le constructeur de Animal et de faire appel à la classe mère (Parent)..
4. Créer la partie de Object
5. Créer la partie de Animal
6. Créer la partie de Chien

Step by Step

Chien c

Ref: Chien
c

Step by Step

```
Chien c = new Chien();
```

Ref: Chien
c

```
public Chien()  
{  
  
}
```

Step by Step

```
Chien c = new Chien();
```

Ref: Chien
c

```
public Chien()  
{  
  
}
```

```
public Animal()  
{  
  
}
```

Step by Step

```
Chien c = new Chien();
```



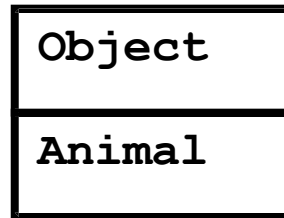
```
public Chien()  
{  
  
}
```

```
public Animal()  
{  
  
}
```

```
public Object()  
{  
  
}
```


Step by Step

```
Chien c = new Chien();
```

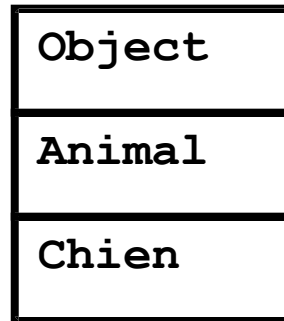


```
public Chien()  
{  
  
}
```

```
public Animal()  
{  
  
}
```

Step by Step

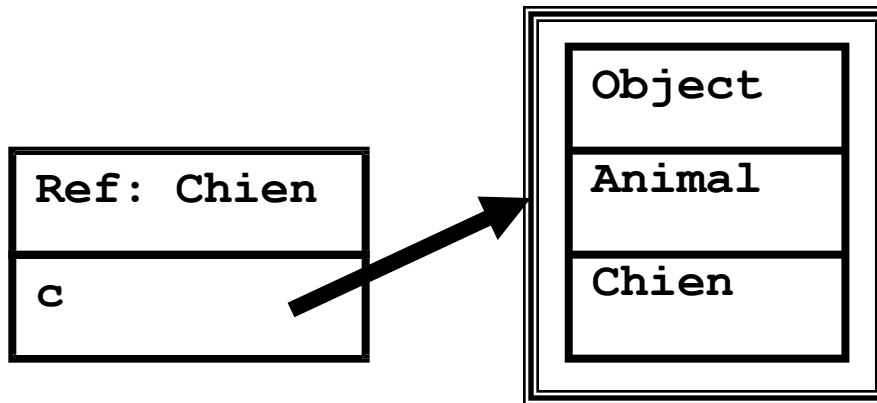
```
Chien c = new Chien();
```



```
public Chien()  
{  
  
}
```

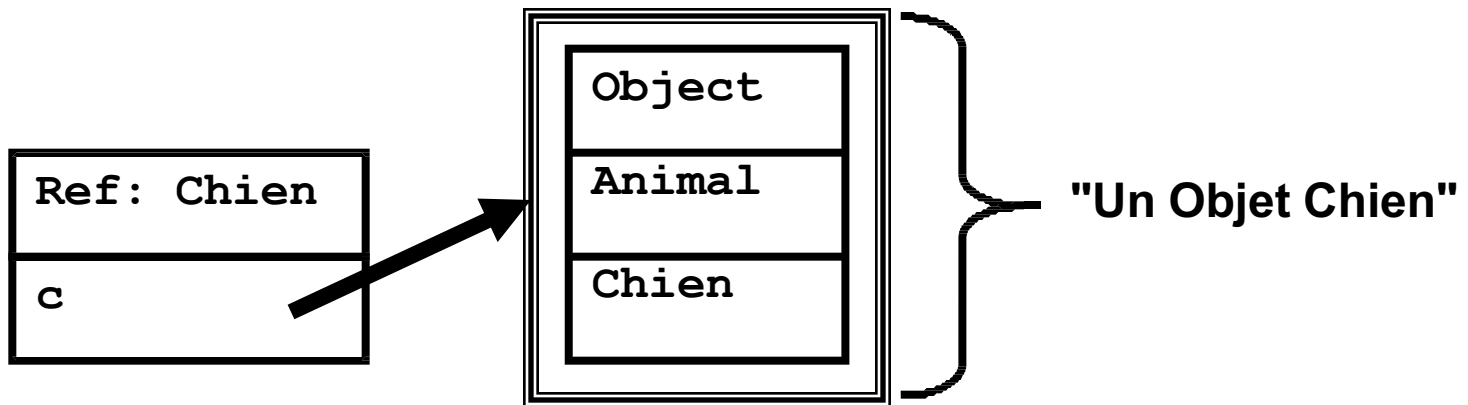
Step by Step

```
Chien c = new Chien();
```



Step by Step

```
Chien c = new Chien();
```



Inheritance

Modificateurs de Visibilité

Contrôle de l'héritage

- Les modificateurs de visibilité affectent la manière dont les membres de la classe peuvent être utilisés dans une classe fils (quels membres de la classe sont accessibles et lesquels ne le sont pas)
- Membres (variables et méthodes) déclarés avec une visibilité privée ne peuvent être référencés par leur nom dans une classe fils
- Ils peuvent être référencés (accessibles) dans la classe fils si ils sont déclarés avec une visibilité publique (hérités) - mais les variables publiques violent le principe de l'encapsulation
- Problème: Comment faire pour que les variables de classe / instance ne soient visibles que pour ses sous-classes?
- Solution: Java fournit un troisième modificateur de visibilité qui aide dans de telles situations d'héritage: protected

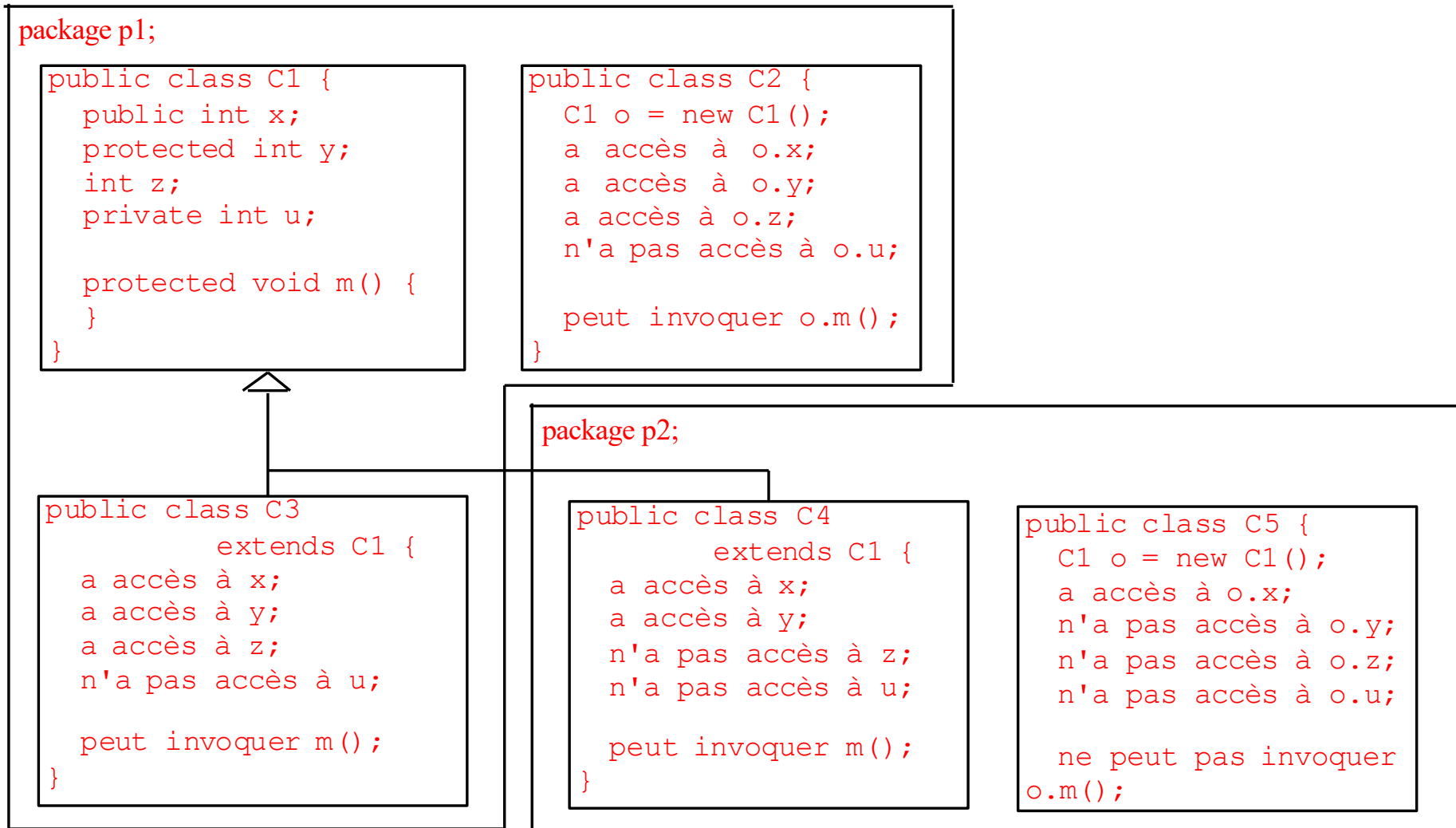
Protected Visibility for Superclass Data

- Les membres private ne sont pas accessibles aux sous-classes!
- Les membres protected sont accessibles aux sous-classes!
(Techniquement, accessible au niveau package)
(*non accessibles en dehors du package, sauf aux sous-classes*)
- Les sous-classes souvent écrits par d'autres, et les sous-classes devraient éviter de s'appuyer sur les détails de la superclasse
- **Alors** ... en général, private est mieux

Visibilité et Encapsulation

- L'encapsulation fournit une isolation contre le changement
- Plus de visibilité signifie moins d'encapsulation
- ***Donc:*** utiliser la visibilité minimale possible pour faire le travail!!

Visibilité et héritage



Visibilité et héritage

Visibility	Default	Public	Protected	Private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in same package	Yes	Yes	Yes	No
Subclass outside the same package	No	Yes	Yes	No
Non-subclass outside the same package	No	Yes	No	No
iq.opengenus.org				

Extend, don't modify

- Le principe *Open-Closed* : les entités logicielles (classes, modules, méthodes, etc.) devraient être ouverte pour l'extension mais fermées pour la modification
 - On doit concevoir les classes qui peuvent être étendues
 - On doit *jamais* avoir à modifier une classe afin de l'étendre?
- Parfois, on pourrait vouloir *interdire* à certaines méthodes cruciales d'être *redéfinies*
 - On peut marquer une méthode en tant que **final** (interdit la redéfinition)

La classe Object

La classe "Object"

- Toutes les classes Java sont des descendants de la classe *Object*.
- *Object* est parent par défaut des classes qui ne précisent pas leur ancêtres
- *Object* offre quelques services génériques dont la plupart sont à refaire, ... **à redéfinir**
(attention au fonctionnement par défaut)

La classe "Object"

```
public final Class getClass();
```

- Renvoie un objet de type Class qui permet de connaître le nom d'une classe

```
public boolean equals(Object o);
```

- Compare les champs un par un (pas les pointeurs mémoire comme ==)

```
protected Object clone();
```

- Allocation de mémoire pour une nouvelle instance d'un objet et recopie de son contenu

```
public String toString();
```

- Renvoie une chaîne décrivant la valeur de l'objet

La méthode equals()

- Les opérateurs d'égalité == et !=, lorsqu'ils sont utilisés avec des objets, ne produisent pas l'effet auquel on peut s'attendre. Au lieu de vérifier si un objet a les mêmes valeurs d'attributs que celles d'un autre objet, ces opérateurs déterminent plutôt si les deux objets sont le même objet, c'est-à-dire si les deux références contiennent la même adresse mémoire. Donc pour comparer des instances d'une classe et obtenir des résultats vraiment exploitables, vous devez implémenter des méthodes spéciales d'égalité dans votre classe.

La méthode equals()

- Égalité entre objet et non entre référence

```
Point p1 = new Point(2,1);
Point p2 = new Point(2,1);
if (p1==p2){...} // Ici c'est faux
if (p1.equals(p2)){...} // OK si equals est bien redéfini
p1 = p2 // Co-référence
if (p1==p2){...} // Vrai désormais
```

Très utile pour les String

- If (myString == "Hello") {...} Toujours Faux !!!
- If (myString.equals("Hello")) {...} OK

- Réflexive, symétrique, transitive et consistant
- Pour les classes que vous créez, vous êtes responsable.

Les fonctions hashCode() et toString()

.hashCode() renvoie un chiffre différent pour chaque instance différente

- .Si `o1.equals(o2)` alors
`o1.hashCode()==o2.hashCode()`
- .Le contraire n'est pas assuré mais préférable (@mémoire)
- .Très utile pour les *HashTable*.

.toString() conversion en *String* d'un objet :

```
String s = '' mon objet visible '' + monObjet;
```

```
String s = '' mon objet visible '' + monObjet.toString();
```

Copie d'un objet et égalité entre deux objets

- Pour avoir un duplicata (copie) d'un objet existant, il s'agit d'offrir un constructeur dont le seul paramètre est une référence à un objet de la même classe à laquelle appartient le constructeur. On appelle alors ce constructeur le *constructeur de recopie*.
- Voici un exemple qui illustre le constructeur copie et implémentation d'une méthode d'égalité :

```
// Fichier Point.java
// Définition de la classe Point dont les objets représentent des points dans
// le plan cartésien
```

```
public class Point
{
    private int x; // abscisse du point
    private int y; // ordonnée du point
```

```
    public Point(int abs, int ord)
    {
        x = abs;
        y = ord;
    }
```

```
    public Point(Point p) // le constructeur copie
    {
        x = p.x;
        y = p.y;
    }
```

```
    public int getX()
    { return x; }
```

```
    public int getY()
    { return y; }
```

```
//Définir une méthode compareTo pour l'objet en entier : retourne true si les attributs des objets ont les mêmes valeurs
```

```
    public boolean equals(Point p)
    {
        •          return (x == p.x && y == p.y) ;
        •      }
```

```
    public String toString()
    {
        return "[" + x + ", " + y + "]";
    }
}
```

- **// Fichier TestPoint.java**

- **point1** et **point2** sont des références à deux objets différents dont les attributs ont les mêmes valeurs

- **public class TestPoint**

- **{**

- **public static void main(String args[])**

- **{**

- **Point point1 = new Point(2, 3);**

- **Point point2 = new Point(point1);**

- **System.out.println("point1 = " + point1);**

- **System.out.println("point2 = " + point2);**

- **//equals** compare les valeurs des attributs

- **if (point2.equals(point1))**

- **System.out.print("\npoint2 égale point1");**

- **else**

- **System.out.print("\npoint2 n'égale pas point1");**

-

- **//==** compare les références (adresses)

- **if (point2 == point1)**

- **System.out.println(", mais point2 == point1");**

- **else**

- **System.out.println(", mais point2 != point1");**

- **}**

- **}**

-

- **Résultats de l'exécution :**

- point 1 = [2, 3]

- point 2 = [2, 3]

- point2 égale point1, mais point2 != point1

- L'implémentation locale de la méthode **equals()** garantit que **point1.equals(point2)** ne sera pas faux à moins que les deux objets Point représentent réellement deux points différents.
- Sans cette implémentation locale, l'expression **point1.equals(point2)** aurait invoqué la méthode **equals()** de la classe mère Object, laquelle aurait retourné la valeur **false** si les objets Point étaient distincts mais égaux (s'attend à recevoir un Object comme argument).
- La méthode **equals()** que nous avons implémentée ne vient pas cependant redéfinir la méthode **equals()** de la classe Object car leurs signatures sont différentes. En effet, celle de la classe Object nécessite un paramètre de la classe Object.
- ***Comment redéfinir cette méthode correctement*** .d'interrogation

Support de présentation

<http://www.cc.gatech.edu/~bleahy/>

<https://people.cs.umass.edu/~moss/187/lectures/>

<http://www.iro.umontreal.ca/~dift1170/A08/pages/classesObjets.ppt>

POO en JAVA

Héritage et Polymorphisme