L'héritage

P.O.O.

LMI 2 – Semestre 4 – Option Info

Année 2008-09

Plan du cours

L'héritage : un tour d'horizon

Accessibilité des membres

La redéfinition de méthodes

Les constructeurs

Graphe de la relation de typage

Plan du cours

L'héritage : un tour d'horizon

Accessibilité des membres

La redéfinition de méthodes

Les constructeurs

Graphe de la relation de typage

Les concepts de la programmation objet

- l'encapsulation : intégration des données et des traitements dans une même entité ; protection du code ;
- la réutilisation : accent mis sur la généricité du code, sur sa réutilisabilité dans des applications diverses ;

Les concepts de la programmation objet

- l'encapsulation : intégration des données et des traitements dans une même entité ; protection du code ;
- la réutilisation : accent mis sur la généricité du code, sur sa réutilisabilité dans des applications diverses ;
- l'héritage : définition d'une classe comme une *prolongation* d'une autre
- le polymorphisme : ce sont les objets destinataires des messages qui déterminent le traitement effectué.

L'héritage par l'exemple

```
public class Personne{
  private String nom;
  private String prenom;
  public Personne(String nom, String prenom){
    this.nom = nom;
    this.prenom = prenom;
  }
  public String getNom(){return nom;}

  public String getPrenom(){return prenom;}

  public String toString(){return prenom+nom;}
}//Personne
```

Exemple

Un étudiant, un travailleur (salarié ou indépendant), un retraité sont tous trois des personnes, avec un nom et un prénom. Mais ils ont chacun des particularités.

Un étudiant :

- le nom de sa formation
- le nom de son université
- le montant de la bourse qu'il reçoit

Un retraité :

- le montant de sa retraite

Exemple 1

Un travailleur :

- son métier
- le nom de son entreprise

Un salarié :

- son salaire

Un indépendant :

- le nombre de personnes qu'il embauche ;
- son chiffre d'affaires

Exemple 1

```
public class Etudiant extends Personne{
  private String universite;
  private String formation;
  // + toutes les methodes specifiques
}

public class Retraite extends Personne{
  private float montantRetraite;
  // + toutes les methodes specifiques
}

public class Travailleur extends Personne{
  private String metier;
  private String entreprise;
  // + toutes les methodes specifiques
```

Exemple 1

```
public class Salarie extends Travailleur{
  private float salaire;
  // + toutes les methodes specifiques
}

public class Independant extends Travailleur{
  private float chiffreAffaires;
  private int nbreSalaries;
  // + toutes les methodes specifiques
}
```

Héritage

Hériter, c'est étendre une classe

- ajouter des informations spécifiques
- spécialiser une classe à partir d'une autre

L'arbre de la relation d'héritage

- La relation d'héritage forme un arbre sur les classes.
- Toute classe qui n'hérite pas explicitement d'une autre classe (la classe `Personne`...) hérite de la classe `Object`.
- La classe `Object` est la racine de cet arbre.

L'arbre d'héritage

Soit une classe B qui hérite d'une classe A.

On dira que A est la classe mère, ou super-classe, et B la classe fille, ou sous-classe.

- Toute instance de B est une instance de A.
- Toute instance de B possède tous les membres de A plus les membres définis dans B.
- Au niveau de la classe, tous les membres statiques de A sont des membres statiques de B (et B possède en plus les membres statiques définis dans B).
- On peut redéfinir dans B les méthodes de A.

Plan du cours

L'héritage : un tour d'horizon

Accessibilité des membres

La redéfinition de méthodes

Les constructeurs

Graphe de la relation de typage

L'accès aux membres d'une classe

- Toute instance d'`Etudiant` est une instance de `Personne`.

L'accès aux membres d'une classe

- Toute instance d'`Etudiant` est une instance de `Personne`.
- Donc toute instance d'`Etudiant` a un `nom` et un `prenom`.

L'accès aux membres d'une classe

- Toute instance d'`Etudiant` est une instance de `Personne`.
- Donc toute instance d'`Etudiant` a un `nom` et un `prenom`.
- Pourtant, une instance d'`Etudiant` ne peut pas accéder directement à son `nom` et son `prenom`.

L'accès aux membres d'une classe

```
public class Etudiant extends Personne {
  public changeTonNom(String surnom){
    this.nom = surnom;  }
}
```

Erreur à la compilation ! !

L'accès aux membres d'une classe

Rappel : On ne peut accéder à un attribut `private` que depuis l'intérieur de la classe où il est défini.

L'accès aux membres d'une classe

Rappel : On ne peut accéder à un attribut `private` que depuis l'intérieur de la classe où il est défini.

Par contre

```
public class Etudiant extends Personne {
  public String cEstQuoiTonNom(){
    return this.getNom();
  }
}
```

L'accès aux membres d'une classe

Conclusion :

Une instance d'une sous-classe ne peut pas accéder directement aux membres privés de ses super-classes.

L'accès ne peut se faire que via des méthodes `public` ou `protected` (ou `package` suivant la localisation de la sous-classe).

L'attribut d'accès `protected`

L'attribut d'accès `protected` permet :

- aux classes du même package
- aux sous-classes (qu'elles soient dans le même package ou dans un autre)

d'accéder aux membres d'une classe.

`protected` est un niveau de visibilité entre `package` et `public`.

Plan du cours

L'héritage : un tour d'horizon

Accessibilité des membres

La redéfinition de méthodes

Les constructeurs

Graphe de la relation de typage

La redéfinition de méthodes

On peut définir dans une classe une méthode avec exactement la même signature que dans une de ses super-classes. C'est ce qu'on appelle la redéfinition de méthodes.
Exemples :

-
- traditionnellement, les méthodes `String toString()` et `boolean equals(Object o)` de la classe `Object` sont redéfinies.

La classe Object

La classe `Object` contient :

- le constructeur `Object()`
- les méthodes :
  - boolean equals(Object obj)
  - String toString()
  - Class getClass()
  - int hashCode()

La redéfinition de méthodes

- permet d'adapter le comportement aux spécificités de la sous-classe
- n'augmente pas les possibilités de la classe par rapport à ses super-classes
- mais spécialise son comportement

Exemple

```
public class Complexe implements IComplexe {
  private float im;
  private float re;
  ...
  public float im(){ return im; }
  ...
  public IComplexe add(IComplexe c){
    im += c.im();
    re += c.re();
    return this;
  }
  ...
}
```

Exemple

```
public class ComplexeNM extends Complexe {
  ...
  public IComplexe add(IComplexe c){
    return new ComplexeNM(re()+c.re(),
                          im()+c.im());
  }
}
```

La redéfinition de méthodes

On peut faire appel au code de sa super-classe par le mot-clé `super`. Exemple :

```
public class Etudiant extends Personne {

  public String toString(){
    return (super.toString()+" inscrit en "
            +formation+" a "+universite);
  }
}
```

La redéfinition de méthodes

Impératif : On doit respecter la signature de la méthode qu'on redéfinit !

- ses paramètres (nombre, type, ordre)
- son type de retour
- son attribut d'accessibilité : on peut élargir son accès.

La redéfinition de méthodes

Impératif : On doit respecter la signature de la méthode qu'on redéfinit !

- ses paramètres (nombre, type, ordre)
- son type de retour
- son attribut d'accessibilité : on peut élargir son accès.

Une méthode `package` peut être redéfinie en une méthode `public`.

Une méthode `public` ne peut pas devenir `private` dans une sous-classe.

La redéfinition de méthodes

```
public class A {
  public void toto(){}
}

public class B extends A {
  private void toto(){} // impossible !!
  // erreur a la compilation !!
}
```

Soit `b` une instance de `B`, comme toute instance de `B` est une instance de `A`, on doit toujours pouvoir faire `b.toto()` !

Plan du cours

L'héritage : un tour d'horizon

Accessibilité des membres

La redéfinition de méthodes

Les constructeurs

Graphe de la relation de typage

Les constructeurs

- Toute instance d'`Etudiant` est une instance de `Personne`.
- Toute instance de `Personne` est une instance d'`Object`.
- Lorsqu'on construit une instance d'`Etudiant`, cette instance se construit d'abord comme une instance de `Personne`, qui se construit elle-même d'abord comme une instance d'`Object`.

Tout constructeur d'une classe doit d'abord faire appel à un constructeur de sa super-classe.

Exemple

```
public class Personne {
  public Personne(String nom, String prenom){
    this.nom = nom;
    this.prenom = prenom;
  }
}

public class Etudiant
             extends Personne {
  private String formation;
  private String universite;
  public Etudiant(String nom, String prenom,
                  String formation, String universite){
    super(nom,prenom);
    this.formation = formation;
    .universite = universite;
```

Les constructeurs

- L'appel au constructeur de la super-classe doit être la première instruction dans un constructeur ;
- Il est obligatoirement explicite si la super-classe ne possède pas de constructeur sans paramètre.
- Il est implicite si la super-classe possède un constructeur sans paramètre.
  C'est le cas général pour une classe qui hérite directement de `Object`.

Plan du cours

L'héritage : un tour d'horizon

Accessibilité des membres

La redéfinition de méthodes

Les constructeurs

Graphe de la relation de typage

Conversions de type

supertype d'une classe A =
l'ensemble des classes "au-dessus" de `A` dans l'arborescence de la relation d'héritage + l'ensemble des interfaces que `A` implémente.

Toute instance de `A` est une instance de tous les éléments de son supertype.

Pour les types objets comme pour les types primitifs, la conversion de type est possible si on respecte la règle d'élargissement du domaine.

Conversions de type

Un objet o peut être affecté à une variable d'un type `A` si `A` est le type de o ou fait partie du sur-type de o.

Toute instance d'`Etudiant` est une instance de `Personne`, mais la réciproque est fausse.

Par contre, dans la suite du code, seuls les méthodes et attributs accessibles de `A` pourront être invoqués sur o.

Exemple

```
Personne etud =
    new Etudiant("Durand", "Paul",
                 "Licence Informatique",
                 "Universite d'Artois"); // OK
System.out.println(etud.getFormation());
        // refuse !! erreur a la compilation!!
        // getFormation() n'est pas
        // une methode de Personne
Etudiant p =
    new Personne("Dupont","Jacques");
        // !! refuse
        // !! erreur a la compilation
```

Exemple

IComplexe est une interface implémentée par Complexe, et ComplexeNM.

```
IComplexe c1 = new Complexe(3,2);
ComplexeNM c2 = new ComplexeNM();
Complexe c3 = new ComplexeNM();
ComplexeNM c4 = new Complexe(1,2); //NON -- Refusé !
IComplexe c5 = c3;
t3 = c2;
```

Le polymorphisme

Le polymorphisme :

à l'exécution, Java choisit la méthode à interpréter en fonction du type de l'objet à qui est envoyé le message.

Exemple

```
// quelque part
public static void afficheToi(Personne p){
  System.out.println(p);
}
// et plus loin...
Personne p1 =
       new Personne("Dupont","Jacques");
Personne p2 =
       new Etudiant("Durand", "Paul",
                    "Licence Informatique",
                    "Universite d'Artois");
afficheToi(p1);
afficheToi(p2);
```

Exemple

provoquera l'affichage suivant :

```
Jacques Dupont
Paul Durand inscrit en Licence Informatique
            a Universite d'Artois
```

Héritage multiple ?

Et si on veut créer la classe des étudiants-salariés ?

```
public class EtudiantSalarie extends Etudiant, Salarie {
// !! Refuse !! Erreur a la compilation !!
}
```

Il n'y a pas d'héritage multiple en Java.
Une classe ne peut hériter que d'une seule classe.

Héritage multiple ?

L'héritage multiple pose le problème du conflit de code : que faire lorsqu'on hérite de deux classes qui proposent deux implémentations différentes d'une même méthode ?

Java refuse donc l'héritage multiple.

Les interfaces permettent de lever cette limitation, car une classe peut implémenter plusieurs interfaces.

Privilégier l'héritage sémantique

Question

1. Un `Rectangle` est-il un `Carré` défini avec un `côté` supplémentaire ?

   ou bien

2. Un `Carré` est-il un `Rectangle` dont les deux côtés sont de la même longueur ?

Privilégier l'héritage sémantique

Question

1. Un `Rectangle` est-il un `Carré` défini avec un `côté` supplémentaire ?

   ou bien

2. Un `Carré` est-il un `Rectangle` dont les deux côtés sont de la même longueur ?

Réponse : Un `Carré` est un `Rectangle` !

Privilégier l'héritage sémantique

```
class Rectangle{
  int longueur;
  int largeur;
  Rectangle(int longueur, int largeur){
    this.longueur = longueur;
    this.largeur = largeur;
  }
  int perimetre(){
    return 2*(longueur+largeur);
  }
  int aire(){
    return longueur * largeur;
  }
}
```

Privilégier l'héritage sémantique

```
class Carre extends Rectangle{

  Carre(int longueur){
    super(longueur, longueur);
  }

}
```