Formation Orienté Objet et Java de base

Atelier 4 : L'héritage

Encadré par M.BOULCHAHOUB

Objectif de l'atelier

L'objectif de cet atelier est de comprendre et d'implémenter l'héritage entre les classes Java

.

Vous allez apprendre à:

- *La notion de l'héritage entre les classes Java*
- *Implémenter l'héritage entre les classes Java*
- *Comprendre la conversion implicite*
- *Comprendre la conversion explicite*

L'héritage

Supposons que dans votre projet, vous faites la gestion des étudiants.

Créer un projet Java avec le nom TPHeritage

Créer un package ma.gov.fst.heritage

Créer deux classes Etudiant et Professeur. Un étudiant est identifié par son nom, prénom et son âge. Un professeur est identifié par son nom, son prénom et son salaire.

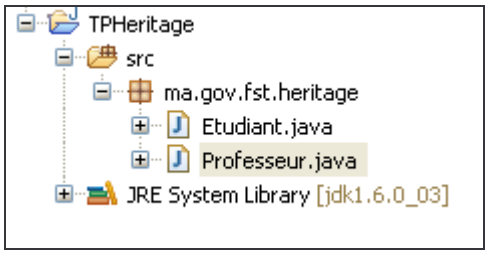
Les méthodes de la classe Etudiant sont parler() et marcher()
Les méthodes de la classe Professeur sont parler() et marcher()

Vous aurez dans votre projet :

Formation Orienté Objet et Java de base

Atelier 4 : L'héritage

Encadré par M.BOULCHAHOUB



Pour la classe Etudiant. Java :

```
package ma.gov.fst.heritage;

public class Etudiant {

	private String nom;
	private String prenom;
	private String age;
	public String getAge() {
		return age;
	}

	public void setAge(String age) {
		this.age = age;
	}
	public String getNom() {
		return nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public String getPrenom() {
		return prenom;
	}
	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}

	public void parler(){
		System.out.println("Je parle");
	}

	public void marcher(){
		System.out.println("Je marche");
	}

}
```

Pour la classe Professeur. Java :

```
package ma.gov.fst.heritage;
```

```
public class Professeur {

	private String nom;
	private String prenom;
	private String salaire;

	public String getSalaire() {
		return salaire;
	}
	public void setSalaire(String salaire) {
		this.salaire = salaire;
	}
	public String getNom() {
		return nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public String getPrenom() {
		return prenom;
	}
	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}

	public void parler(){
		System.out.println("Je parle");
	}

	public void marcher(){
		System.out.println("Je marche");
	}


}
```

Nous constatons que les attributs nom, prenom sont redondants (se répètent) au niveau de la classe Etudiant et Professeur.

Nous constatons également que les méthodes parler () et marcher () sont redondants (se répètent) au niveau de la classe Etudiant et Professeur.

La notion d'héritage permet de résoudre le problème de la redondance au niveau des classes en créant une classe mère pour les attributs et les méthodes qui se répètent.

Créant alors une classe Personne, classe mère d'Etudiant et Professeur.

Le lien d'héritage s'implémente en java à travers le mot réservé **extends** *Personne sera alors la classe mère. Etudiant et Professeur seront les classes filles.*

Vous aurez :

Pour la classe Personne :

```
package ma.gov.fst.heritage;

public class Personne {

	private String nom;
	private String prenom;
	public String getNom() {
		return nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public String getPrenom() {
		return prenom;
	}
	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}

	public void parler(){
		System.out.println("Je parle");
	}

	public void marcher(){
		System.out.println("Je marche");
	}


}
```

Pour la classe Etudiant :

```
package ma.gov.fst.heritage;

public class Etudiant extends Personne {

	private String age;

	public String getAge() {
		return age;
	}
```

```
	public void setAge(String age) {
		this.age = age;
	}

}
```

Pour la classe Professeur :

```
package ma.gov.fst.heritage;

public class Professeur extends Personne{

	private String salaire;

	public String getSalaire() {
		return salaire;
	}

	public void setSalaire(String salaire) {
		this.salaire = salaire;
	}

}
```

Les attributs et les méthodes qui se répètent doivent figurer au niveau de la classe mère.

Appel d'une méthode à partir d'une classe fille

Créer maintenant une classe Test avec la méthode main :
Instancier un objet de la classe Etudiant. Puis appeler la méthode marcher()

```
package ma.gov.fst.heritage;

public class Test {

	public static void main(String[] args) {
		Etudiant e =new Etudiant();
		e.marcher();
	}

}
```

Faire l'exécution de la classe Test.java et constater que l'objet de la classe Etudiant accède à la méthode marcher () par héritage. (Cette méthode n'est plus définie au niveau de la classe Etudiant)

La redéfinition

Vous pouvez changer le corps de la méthode marcher au niveau des classes filles Professeur et Etudiant. ***C'est la notion de la redéfinition des méthodes.***

Ci-dessous la redéfinition de la méthode marcher () au niveau de la classe Etudiant.

```
package ma.gov.fst.heritage;

public class Etudiant extends Personne{

	private String age;

	public String getAge() {
		return age;
	}

	public void setAge(String age) {
		this.age = age;
	}

	public void marcher(){
		System.out.println("Je marche comme un etudiant");
	}

}
```

Refaire l'exécution de la classe Test.java et constater la différence au niveau de l'exécution.

Lorsque vous faites la redéfinition d'une méthode :

Garder la même signature que celle déclarée au niveau de la classe mère (Le même modificateur d'accès ou un modificateur supérieur, le même type de retour, le même nom de la méthode…)

La conversion

Maintenant dans la méthode main de la classe Test, créer un objet professeur et un objet Personne.

```
package ma.gov.fst.heritage;
```

```
public class Test {

	public static void main(String[] args) {
		Etudiant e =new Etudiant();
		Professeur p = new Professeur();
		Personne pr = new Personne();
	}

}
```

Affecter l'objet e à l'objet pr.

```
public static void main(String[] args) {
		Etudiant e =new Etudiant();
		Professeur p = new Professeur();
		Personne pr = new Personne();

		pr=e;
}
```

*Dans ce cas aucune erreur de compilation n'est signalée par le compilateur. On parle d'***une conversion implicite (cast implicite)**.

Affecter maintenant l'objet pr à e. Vous aurez une erreur de compilation

```
public static void main(String[] args) {
		Etudiant e =new Etudiant();
		Professeur p = new Professeur();
		Personne pr = new Personne();

		pr=e;

		e=pr;
}
```

Il faut faire une conversion explicite pour corriger l'erreur de compilation. Vous aurez alors :

```
public static void main(String[] args) {
		Etudiant e =new Etudiant();
		Professeur p = new Professeur();
		Personne pr = new Personne();

		pr=e;

		e=(Etudiant)pr;
}
```

Exercice 1 : Articles à vendre

Une société vend des articles de papeterie. Vous vous limiterez aux articles suivants :

- stylos décrits par une référence, un nom ou descriptif (par exemple "stylo noir B2"), une marque, un prix unitaire et une couleur,
- ramettes de papier décrites par une référence, un nom, une marque, un prix unitaire et le grammage du papier (par exemple, 80 g/m²).

De plus cette société peut vendre ces articles par lots. Vous supposerez que les lots sont composés d'un certain nombre d'un même type d'articles, par exemple un lot de 10 stylos noirs B2 de la marque WaterTruc. Un lot a une référence et une marque (celle de l'article). Le nom du lot est déterminé par le nombre d'articles dont il est composé ; par exemple "Lot de 10 stylo noir B2" (vous êtes dispensé de mettre les noms au pluriel !). Le prix du lot est déterminé par le prix unitaire de l'article multiplié par le nombre d'articles, auquel est enlevé un certain pourcentage qui dépend du lot. Par exemple, si un lot de 10 stylos à 100 F a un pourcentage de réduction de 20 %, le prix du lot sera de 10 x 100 x (100 - 20) / 100 = 800 F. Le prix d'un lot sera calculé au moment où on demandera le prix du lot ; il n'est pas fixé une fois pour toute et il change si le prix de l'article qui compose le lot est modifié.

Ecrivez des classes dans un paquetage "ma.gov.stock.*model*" pour représenter les différents articles.

On ne s'intéresse qu'aux différents types d'articles. Ainsi, on aura une seule instance qui représentera le type "stylo noir B2" et pas 250 instances si la société possède 250 "stylo noir B2". Si vous voulez démarrer une gestion de stock, vous pouvez ajouter le nombre d'articles en stock pour chaque type, mais ça n'est pas demandé.

Exercice 2 : Bibliothèque

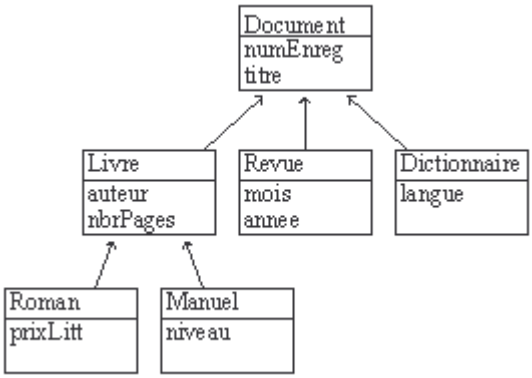
Formation Orienté Objet et Java de base

Atelier 4 : L'héritage

Encadré par M.BOULCHAHOUB

Pour la gestion d'une bibliothèque on nous demande d'écrire une application traitant des ***documents*** de nature diverse : des ***livres***, des ***revues***, des ***dictionnaires***, etc. Les livres, à leur tour, peuvent être des ***romans*** ou des ***manuels***.

Tous les documents ont un numéro d'enregistrement (un entier) et un titre (une chaîne de caractères). Les livres ont, en plus, un auteur (une chaîne) et un nombre de pages (un entier). Les romans ont éventuellement un prix littéraire (un entier conventionnel, parmi : **GONCOURT**, **MEDICIS**, **INTERALLIE**, etc.), tandis que les manuels ont un niveau scolaire (un entier). Les revues ont un mois et une année (des entiers) et les dictionnaires ont une langue (une chaîne de caractères convenue, comme **"anglais"**, **"allemand"**, **"espagnol"**, etc.).



Tous les objets en question ici (***livres***, ***revues***, ***dictionnaires***, ***romans***, etc.) doivent pouvoir être manipulées en tant que *documents*.

A. Définissez les classes **Document**, **Livre**, **Roman**, **Manuel**, **Revue** et **Dictionnaire**, entre lesquelles existeront les liens d'héritage que la description précédente suggère.

Dans chacune de ces classes définissez

- le constructeur qui prend autant arguments qu'il y a de variables d'instance et qui se limite à initialiser ces dernières avec les valeurs des arguments,

- une méthode **public String toString()** produisant une description sous forme de chaîne de caractères des objets,
- si vous avez déclaré **private** les variables d'instance (c'est conseillé, sauf indication contraire) définissez également des « accesseurs » publics *get...* permettant de consulter les valeurs de ces variables.

Écrivez une classe exécutable **TestDocuments** qui crée et affiche plusieurs documents de types différents.

B. Une bibliothèque sera représentée par un *tableau de documents* (on verra un autre jour de meilleures manières de faire, basées sur l'emploi d'une **Collection**). Définissez une classe ***Bibliotheque***, avec un tel tableau pour variable d'instance et les méthodes :

- **Bibliotheque(int capacité)** - constructeur qui crée une bibliothèque ayant la capacité (nombre maximum de documents) indiquée,
- **void afficherDocuments()** - affiche *tous* les ouvrages de la bibliothèque,
- **Document document(int i)** - renvoie le $i^{ème}$ document,
- **boolean ajouter(Document doc)** - ajoute le document indiqué et renvoie **true** (**false** en cas d'échec),
- **boolean supprimer(Document doc)** - supprime le document indiqué et renvoie **true** (**false** en cas d'échec)
- **void afficherAuteurs()** - affiche la liste des auteurs de tous les ouvrages qui ont un auteur (au besoin, utilisez l'opérateur **instanceof**)

C. Définissez, avec un effort minimal, une classe **Livrotheque** dont les instances ont les mêmes fonctionnalités que les **Bibliotheques** mais sont *entièrement constituées de livres*. Comment optimiser dans la classe **Livrotheque** la méthode **afficherAuteurs** ?

FIN DE L'ATELIER