INF1256 Informatique pour les sciences de la gestion

–Encapsulation, Interfaces, classes abstraites, Héritage –

Johnny TSHEKE, Ing. Jr.

Université du Québec à Montréal
Département d'Informatique
tsheke_shele.johnny@uqam.ca

Séance 13

Regroupement des classes en *package* (paquetage)

Un paquetage peut regrouper des

- classes
- interfaces
- enum (énumérations)
- annotations (commence avec `@` Ex : `@Entity`)
- des sous paquetages (subpackage)

Pour regrouper les types dans un package, il suffit de mettre l'instruction package suivi du nom du package tout au début de chaque fichier source :

```
//pour le package nommé  inf1256
package inf1256;
//pour le sous package nommé  hiver2017
package inf1256.hiver2017;
```

Le nom du package doit-être identique pour le type de même package.

⇒ En pratique un package donnera lieu à un sous dossier

Contrôle d'accès aux éléments d'une classes (Rappel)

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- Niveau classe : `public` si classe accessible de partout. Si pas de protection explicite alors accessible seulement dans le package.
- Niveau attribut de classe : `public` , `ptotected`, `private`, défaut (pas de protection explicite) Telle que illustré dans le tableau

⇒ voir `https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

Encapsulation de données

Consiste à protéger les données (les champs) d'un objet d'un accès direct à partir de l'extérieur et d'exiger de passer par une méthode. Pour cela, au besoin,

- protéger les champs avec le mot `private`
- définir le(s) constructeur(s) pour initialiser les données
- définir les méthodes (`setters`) pour modifier les champs
- définir les `getters` pour lire les valeurs des champs

⇒ Pour plus d'info :

`https://docs.oracle.com/javase/tutorial/java/concepts/object.html`

Exemple d'encapsulation

Exemple de la classe Citoyen : Citoyen.java

```
package inf1256s13;
// Johnny Tsheke @UQAM -- INF1256

public class Citoyen {
        private String  prenom = "";
        private String nom ="";//nom de la persoonne
        public String getNom(){ //getter
                return(this.nom);
        }

        public void setNom(String n){//setter
                this.nom = n;
        }

        String getPrenom(){
                return(this.prenom);
        }

        void setPrenom(String pre){//setter
                this.prenom = pre;
        }
}
```

Interface en Java

- Comme une classe
- Peut-être implémentée par une classe
- Ne peut-être instanciée
- Ne peut contenir que des constantes, des signatures des méthodes, des méthodes par défaut et des méthodes statiques
- Les méthodes par défaut et statiques peuvent contenir des codes (implémentation)
- On ajoute implements **NomInterface** à la déclaration de la classe

```
class NomClasse implements NomInterface {
//code de la classe ici
}
```

Exemple Interface

Exemple Interface PersonneInterface : PersonneInterface.java

```
/**
 *
 */
package inf1256s13;

/**
 * @author johnny Tsheke
 *
 */
public interface PersonneInterface {
	String COULEUR_SANG = "ROUGE";//sera public et final pcq dans Interface
	// private String COULEUR_SANG = "ROUGE";// erreur à cause du mot private
	String getNom(); //signature de la methode
	void setNom(String no);
	String getPreNom();
	void setPreNom(String pre);

}
```

Exemple utilisation Interface

Exemple classe Homme implémente l'interface PersonneInterface : Homme.java

```
public class Homme implements PersonneInterface {
        private String nom="";
        private String prenom="";

        public String getNom() {
                return this.nom;
        }

        public void setNom(String nomPers) {
                this.nom = nomPers;
        }

        public String getPreNom() {
                return this.prenom;
        }

        public void setPreNom(String prenomPers) {
                this.prenom = prenomPers;
        }

        public static void main(String[] args) {
                // TODO Auto-generated method stub
        }
}
```

Méthode par défaut

Méthode par défaut :

- Permet d'ajouter des nouvelles fonctionnalités
- Assure la compatibilité avec les applications utilisant les versions précédentes de l'interface (pas requis de changer les application)
- Pas requis d'implémenter dans la classe
- Faire précéder la déclaration du mot default et implémenter dans l'interface

```
default void nomMethode(){
//code de la méthode
}
```

Exemple Interface avec méthode par défaut

Exemple Interface PersonneInterface2 : PersonneInterface2.java

```
/**
 *
 */
package inf1256s13;

/**
 * @author johnny Tsheke
 *
 */
public interface PersonneInterface2 {
	String COULEUR_SANG = "ROUGE";//sera public et final pcq dans interface
	String getNom(); //signature de la methode
	void setNom(String no);
	String getPreNom();
	void setPreNom(String pre);
	default void afficherNomComplet(){ //méthode par défaut
		System.out.println("Nom complet: "+ this.getNom() +", "+ this.getPreNom());
	}
}
```

Exemple utilisation interface et méthode par défaut

Exemple classe Homme2 implémente l'interface PersonneInterface2 : Homme2.java

```
public class Homme2 implements PersonneInterface2 {
        private String nom="";
        private String prenom="";

        public String getNom() {
                return this.nom;
        }

        public void setNom(String nomPers) {
                this.nom = nomPers;
        }

        public String getPreNom() {
                return this.prenom;
        }

        public void setPreNom(String prenomPers) {
                this.prenom = prenomPers;
        }

        public static void main(String[] args) {
                Homme2 homme2 = new Homme2();
                homme2.setNom("Tremblay");
                homme2.setPreNom("Jean");
                homme2.afficherNomComplet(); //appel méthode par défaut
        }
}
```

Héritage en Java

La classe **B** hérite de la classe `A`

- Certaines variables et/ ou méthodes définies pour **A** peuvent-être réutilisées dans la classe **B** (sous réserve de la protection de données)
- On peut ajouter des attributs (variable, méthodes) spécifiques dans **B**
- **A** est plus générale – super-classe (*superclass*) et **B** est plus spécifique – Sous-classe (*subclass*)
- on dit aussi que **B** est dérivée (une extension, enfant) de **A** (classe parent, classe de base)
- Une classe peut-être dérivée d'une autre classe dérivée
- En Java, toute classe a un seul parent (super-classe) direct et toutes les classes ont un ancêtre commun : **Object**

→`https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`

Héritage en Java (suite)

La classe **B** hérite de la classe `A`

- Si pas d'héritage explicite → enfant de **Object**
- Tout objet de type **B** est un objet de type **A**
  Ex : Tout **Étudiant** est un **Citoyen** (Il y a évidement des citoyens qui ne sont pas des étudiants !) → **Etudiant** extension de **Citoyen**
- On ajoute extends **A** à la déclaration de la classe **B**

```
class B extends A {
//code de B ici
}
```

⇒ Une interface peut aussi hériter d'une autre Interface

Exemple d'héritage

Exemple de la classe Etudiant, une sous-classe de Citoyen : Etudiant.java

```
package inf1256s13;

public class Etudiant  extends Citoyen{
    private String universite = "";
    public Etudiant(String nomEtudiant, String prenomEtudiant){//constructeur
        this.setNom(nomEtudiant);
        this.setPrenom(prenomEtudiant);
    }
    public String getUniversite(){
        return this.universite;
    }
    protected void setUniversite(String uni){
        this.universite = uni;
    }
    public void afficherInfo(){
        System.out.println("Voici les informations de l'étudiant-e");
        System.out.println("Nom: "+this.getNom());
        System.out.println("Prénom: "+this.getPrenom());
        System.out.println("Université: "+this.getUniversite());
    }
}
```

Classes et méthodes abstraites

Classe abstraite :

- Déclaration précédée du mot `abstract`
- Ne peut pas être instanciée → Instancier les sous-classes
- Peut contenir des méthodes abstraites (et autres méthodes)

Méthode abstraite :

- déclaration précédée du mot `abstract`
- Seulement la signature suivie de ; sans implémentation

```
//ex:
public abstract int nombreMax(int[] tab);
```

- doit-être implémentée dans les sous classes

⇒`https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html`

Exemple Classe et méthodes abstraites

Exemple classe abstraite Personne : Personne.java

```
package inf1256s13;

public abstract class Personne {
        public static String COULEUR_SANG = "ROUGE";
        private String genre; //pas une constante et private ok.
    abstract String getNom(); //signature de la methode
    abstract void setNom(String no);
    abstract String getPreNom();
    abstract void setPreNom(String pre);
    public String getGenre(){//implémentation
        return(this.genre);
    }

    public void setGenre(String ge){//implémentation
        this.genre = ge;
    }
}
```

Interface Vs Classe abstraite

Tous les deux ne peuvent pas être instanciées

Interface : tous les champs sont des constantes (`public`, `static` et `final`) et toutes les méthodes sont publiques (`public`)

Classe abstraite : on peux t utiliser d'autres *modifiers* (**private**, `protected`, ...)

Une sous classe ne peut avoir que une seule super classe mais on peut implémenter plusieurs interfaces (parce que les interfaces n'ont pas de champs variables dont les valeurs déterminent l'état d'un objet)

Exemple utilisation sous classe (Héritage)

Exemple classe Femme sous classe de la classe abstraite Personne : Femme.java

```
public class Femme extends Personne {
        @Override //annotation, masque l'implementation par defaut eventuelle
        String getNom() {
                // TODO Auto-generated method stub
                return null;
        }

        @Override
        void setNom(String no) {
                // TODO Auto-generated method stub
        }

        @Override
        String getPreNom() {
                // TODO Auto-generated method stub
                return null;
        }
        @Override
        void setPreNom(String pre) {
                // TODO Auto-generated method stub
        }

        public static void main(String[] args) {
                Femme fe = new Femme();//instanciation
                //Personne fe = new Femme(); //autre façon d'instancier
            fe.setGenre("Feminin");//appele methode héritée
                System.out.println("Le genre est: "+ fe.getGenre());
        }
}
```

Redéfinition des méthodes

- Même signature dans la super-classe (ou interface pour les méthodes par défaut) et la sous-classe
- On parle de overriding si on redéfinit une méthode d'instance (sans static). Dans ce cas, faire précéder la déclaration de la redéfinition (dans la sous-classe) de l'annotation @Override
- On parle de hiding si on redéfinit une méthode de classe (avec static).
- Lors de l'exécution, la machine virtuelle Java choisit la méthode approprié comme expliqué avec le Polymorphisme (page suivante)

Polymorphisme

- Méthode de la super-classe et redéfinie dans la sous-classe
  Ex : hello() définie dans la super-classe **A** et la sous-classe **B**
- Dans le cas de overriding, La méthode réellement appelée sera celle tu type de l'objet référencé et non du type de variable

```
A obj1 = new A();
A obj2 = new B();
obj1.hello();//exécute méthode de A
obj2.hello(); //exécute méthode de B
```

- Dans le cas de hiding, C'est la méthode du type de la variable qui sera exécutée