Le langage Java

Apprentissage en lien avec le langage UML

Le langage Java

Apprentissage en lien avec le langage UML

Pourquoi réutiliser ?

- Le paradigme objet permet de structurer une application
- Des parties de celle-ci peut-être partagées ou réutilisées
  - Cela comprend la conception et le développement
- Ces parties peuvent aussi être étendues
- Cela accélère de développement et facilite la maintenance
- Mais demande plus de travail préliminaire

UML et Java

- Dans cette partie nous nous limiterons aux modèles UML structurels dédiés aux aspects statiques des objets
  - Les diagrammes de classe
  - Les diagrammes d'instance
- Nous examinerons en parallèle l'implantation possible en Java

La délégation

- Un première solution pour réutiliser en objet : La délégation
- Un classe ou une instance peut :
  - Appeler une méthode d'une autre classe
  - Instancier un objet d'une autre classe et "déléguer"

```
package coursSSI3.exemples.reutilisation;

public class DelegationPure {
        public void action() {
                Voiture v = new Voiture();
                v.demarrer();
        }
}
```

Listing 1 – coursSSI3/exemples/reutilisation/DelegationPure.java

- Il est difficile de formaliser cela : " C1 utilise C2 "

L'association en UML

- Au niveau de l'analyse un attribut ne peut être une classe
- On définit la notion d'**association** pour indiquer une relation entre deux classes
- Une association possède un **nom**, une cardinalité et éventuellement des rôles

L'aggrégation en Java

- La représentation des associations en Java se fait à l'aide d'attributs d'instance
- Dans le cas d'une cardinalité déterminée *n..m*, il est possible d'utiliser un tableau
- Plus généralement, nous utiliserons des *Collections* (cf. cours sur les Collections)
- Les contraintes exprimées sur le modèle UML :
  - Ordonné, Ou-exclusif, sous-ensemble, . . .doivent être vérifiées par le programme

La navigabilité en Java

- La navigabilité en Java est matérialisée par :
  - la présence d'un attribut
  - des accesseurs permettant d'obtenir les valeurs
- Attention, au coût et à la complexité des associations bi-directionnelles.
  - Maintien de la cohérence

L'aggrégation et la composition en Java

- L'aggrégation par rapport à la composition est plutot sémantique
- La composition impose de vérifier la contrainte de co-existance :
  - La destruction du composé impose la destruction des composants
  - Mais aussi la destruction d'un composant être interdite sans s'assurer au préalable de la destruction du composant.
- En Java, c'est le rôle du ramasse-miette, mais attention à ne pas laisser de références erronées.

L'héritage en Java

- On indique que la classe fille étend la classe mère : `extends`

```
package coursSSI3.exemples.reutilisation;

public class Vehicule {
 private String marque;
 public void setMarque(String marque) {this.marque=marque;}
 public String getMarque() {return marque;}
 public void avance() {}
}
```

Listing 2 – coursSSI3/exemples/reutilisation/Vehicule.java

```
package coursSSI3.exemples.reutilisation;

public class VehiculeAMoteur extends Vehicule {
 private String typeMoteur; /* Augmentation de l'etat Interne */
 public VehiculeAMoteur(String marque, String typeMoteur) {
  setMarque(marque);
  this.typeMoteur = typeMoteur;
 }

 public void avance() {/* Modification du comportement*/}
 public void demarre() {/* Nouveau comportement */}
}
```

Listing 3 – coursSSI3/exemples/reutilisation/VehiculeAMoteur.java

L'héritage en Java

- Si `extends` n'est pas précisé la classe étend la classe `Object`
- L'héritage multiple est interdit
- La classe fille peut
  - ajouter des variables, des méthodes, des constructeurs.
  - *redéfinir* ou *surcharger* des méthodes

Definition

La redéfinition d'une méthode consiste à définir une méthode ayant la méthode signature qu'une méthode définie dans une classe ancêtre.

L'héritage en Java

- Le principe d'utilisation de l'héritage en Java est le même que pour la conception :
  - Si `B extends A`
  - Alors toute instance `b` de `B` est un (*is a*) `A`
- Par exemple, une Voiture *v* est un Vehicule
- **Il est interdit d'utiliser l'héritage dans un autre contexte**

Les constructeurs et l'héritage

- La classe fille hérite de tous les membres (attributs et méthodes)
- **Attention** en fonction des protections (`private`), il se peut qu'elle ne puisse y accéder
  - (cf. `protected`)
- Les constructeurs ne sont pas hérités
  - Mais il est possible d'appeler ceux de la super classe avec `super()`
  - Il est toujours possible d'appeler un autre constructeur avec `this`
- Attention, les deux instructions précédentes ne peuvent chacune être que la **première instruction** d'un constructeur.

Les constructeurs et l'héritage

- Si ni `this()` ni `super()` ne sont précisés, `super()` est ajouté par défaut.
- La première instruction de tous les constructeurs est donc un appel au constructeur de la super classe.
- Quel est donc le premier constructeur appelé et pourquoi ?

Les constructeurs et l'héritage

```
package coursSSI3.exemples.reutilisation;

public class Animal {
 private String espece="";
 public Animal(String espece)
        {this.espece=espece;}
}
```

Listing 4 – coursSSI3/exemples/reutilisation/Animal.java

```
package coursSSI3.exemples.reutilisation;
public class Chien extends Animal {
        private enum Race {
                BOXER, CANICHE, DOGUE
        };
        private Race race;

        public Chien() {
                super("canide"); /* espece= canide ? */
        }
        public Chien(Race race) {
                this(); /* Que se passe-t-il sans this ? */
                this.race = race;
        }
}
```

Listing 5 – coursSSI3/exemples/reutilisation/Chien.java

Limitations imposées par Java

- Le type de retour d'une méthode surchargée doit être le même que celui de la méthode d'origine
- La nouvelle méthode ne doit pas être moins accessible
  - Par exemple une méthode `public` ne peut pas devenir `private`
  - *Quelle est la raison ?*

Le polymorphisme

Definition

Le polymorphisme est un mécanisme qui permet d'envoyer à plusieurs objets de types différents un même message chacun réagissant de façon propre.

- L'utilisation du polymorphisme est en partie liée à l'héritage mais pas seulement (cf. Interfaces)
- On distinguera donc :

Definition

Le type réel d'un objet qui est celui de l'objet effectivement créé en mémoire.

Definition

Le type déclaré d'un objet qui est celui de la référence à travers laquelle il est manipulé.

- Le type déclaré permet de savoir quel message peuvent être envoyé et le type réel quelle action **sera réellement exécutée**.

Le polymorphisme

```
package coursSSI3.exemples.reutilisation;
public class Polymorphisme {

        public class Animal {
                void crier() {System.out.println("Je␣crie.");}
        }
        class Chien extends Animal{
                void crier() {System.out.println("ouaf␣ouaf");}
        }
        class Chat extends Animal{
                void crier() {System.out.println("miaou␣miaou");}
        }

        public void crier() {
                Animal animaux[] = {new Animal(),
                                    new Chien(), new Chat()};
                for(Animal animal:animaux) animal.crier();
        }

        public static void main(String args[]) {
                new Polymorphisme().crier();
        }
}
```

Listing 6 – coursSSI3/exemples/reutilisation/Polymorphisme.java

Le transtypage

- Il possible de forcer le compilateur à considérer un objet comme étant d'un type différent :
  - de son type réél
  - de son type déclaré
- Les seuls cast possibles sont ceux entre classe filles et mères. On distingue :
  - le *upcast* vers la classe mère (Toujours possible)
  - le *downcast* vers la classe (**Attention Danger**, cf. exemple suivant)

Le transtypage

```
package coursSSI3.exemples.reutilisation;

public class Cast {

        public abstract class Animal {
                abstract void crier();
        }
        class Chien extends Animal{
                void mord() {System.out.println("grr grr");}
                void crier() {System.out.println("ouaf ouaf");}
        }
        class Chat extends Animal{
                void crier() {System.out.println("miaou miaou");}
        }

        public void attaque() {
                Animal animaux[] = {new Chat(),
                                    new Chien(), new Chat()};
                ((Chien)animaux[1]).mord();
                /* Erreur a l'execution mais pas a la compilation
                ((Chien)animaux[2]).mord(); */
        }

        public static void main(String args[]) {
                new Cast().attaque();
        }
}
```

Listing 7 – coursSSI3/exemples/reutilisation/Cast.java

Les classes abstraites

- En général la spécialisation d'une classe peut entrainer la redéfinition d'une ou plusieurs méthodes
- Dans certains cas, la définition du corps d'une méthode n'a pas de sens pour la classe mère
  - Par exemple, tous les `Animaux` peuvent `crier()`.
  - Mais on peut définir un cri général...
  - Le cri dépend de la sous-classe (`Chien`, `Chat`, ...)
- Cependant la définition de la méthode doit rester dans la classe mère :
  - Pour garantir la structuration objet
  - Pour permettre le polymorphisme (lié à l'héritage)
- On parle alors de *Classe abstraite*
  - Attention, ces classes ne peuvent être instanciées puisque les définitions de méthodes sont incomplètes.

Les classes abstraites

- En java, les méthodes dont on ne donne pas le corps ainsi que les classes concernées doit être marquées `abstract`
- Les classes qui spécialisent une classe abstraite doivent définir les méthodes abstraites ou être marquée `abstract`

```
package coursSSI3.exemples.reutilisation;

public class ClasseAbstraite {
public abstract class Animal {
        abstract void crier();
}

public abstract class Bovin extends Animal {
        abstract void ruminer();
}

public class Vache extends Bovin {
        void crier() {/* Definition */}
        void ruminer() {/* Definition */}
}
}
```

Listing 8 – coursSSI3/exemples/reutilisation/ClasseAbstraite.java

Les interfaces

- En java la définition d'une classe et l'héritage permettent de définir à la fois
  - l'état interne des instances
  - et le comportement (éventuellement par spécialisation)
- Cependant cela impose, un relation hiérarchique stricte de type "est un"
- Il est parfois utile de pourvoir imposer à des objets instances de classe sans relation d'héritage de vérifier un même comportement.
- Java comme UML propose pour cela la notion d'*Interface*

Les interfaces

Definition

Une interface est un comportement (un ensemble de signatures de méthodes) que des classes peuventt choisir de suivre (on dira d'implanter).

- En java, la déclaration d'une interface se fait avec le mot clé `interface` de la même façon que pour une classe qui ne comporterait que des méthodes abstraites et publiques.
  - L'héritage même multiples est possible entre interfaces

Les interfaces

```
package coursSSI3.exemples.reutilisation;

public class Interface {
        public interface ItrucEmpruntable {
                public void emprunte();
                public void rendu();
        }

        public class Voiture implements ItrucEmpruntable{
                boolean disponible = true;
                public void emprunte() {disponible = false;}
                public void rendu() {disponible = true;}
        }

        public class Stylo implements ItrucEmpruntable {
                boolean emprunte = false;
                public void emprunte() {emprunte = true;}
                public void rendu() {emprunte = false;}
        }

}
```

Listing 9 – coursSSI3/exemples/reutilisation/Interface.java