Héritage et polymorphisme

Mise à niveau Java

Y. Boichut & F. Moal

Notion d’héritage

- L’héritage permet d’écrire qu’une classe **B** se comporte comme une classe **A** mais avec quelques différences
- Uniquement besoin du code compilé de la classe **A**
- Code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
  - Ajouter des nouvelles méthodes
  - Modifier des méthodes de **A** - redéfinition

Vocabulaire

- La classe B hérite de la classe A
  - B est une classe fille ou sous-classe de A
  - A est la classe mère ou super-classe

L’héritage

- Particularisation - généralisation
  - Un polygone est une figure géométrique mais une figure géométrique particulière
  - la notion de figure géométrique est une généralisation de la notion de polygone
- Une classe fille offre de nouveaux services ou enrichit les services rendus par la classe mère
  - Exemple : Rectangle / RectangleColore
- Chaque langage objet a ses particularités
  - C++ et Eiffel supportent l’héritage multiple alors que ce n’est pas le cas pour C# et Java (encore que….)

Héritage en Java

- Chaque classe a **une et une** seule classe mère
- Le mot “**extends**” indique la classe mère et définit ainsi l’héritage
  - class RectangleColore extends Rectangle
- Tout objet en Java hérite de la classe **Object**

Ce que peut faire une classe fille

- La classe qui hérite peut
  - ajouter des champs, des méthodes et des constructeurs
  - redéfinir des méthodes (même signature)
  - surcharger des méthodes (même nom mais pas même signature)
- La classe qui hérite ne peut retirer aucun champ ou aucune méthode de la classe mère

Principe important

- Si “B extends A” alors *tout B est un A*
- B est un sous-type de A
  - On peut stocker une valeur de type B dans une variable de type A
  - Exemple : A a = new B();
  - L’inverse n’est pas possible sauf en forçant par Cast (pouvant échouer)

Quelques compléments sur les constructeurs

- La première instruction d’un constructeur peut être un appel
  - à un constructeur de la classe mère (pour initialiser des champs de la classe mère)
    - super(...)
  - à un autre constructeur de la classe courante
    - this(...)
- Impossible de placer ces instructions ailleurs qu’en première instruction d’un constructeur de la classe courante

Quelques choses à savoir sur les constructeurs

- Si aucun constructeur n’est spécifié pour une classe donnée
  - alors il existe un constructeur par défaut (automatique généré à la compilation)
- Si un constructeur avec paramètres est spécifié et qu’un constructeur par défaut ne l’est pas
  - alors aucun constructeur par défaut n’est généré
- Si un constructeur avec paramètres est spécifié dans une classe mère
  - alors les filles doivent également implanter un constructeur avec ces paramètres

Exercice

```java
public class Personnage {

    long pointVie;

    public Personnage(long pointVie) {
        this.pointVie = pointVie;
    }

}
```

Créez la classe Troll qui héritera de la classe **Personnage**

Exercice

Plus de constructeurs par défaut

```
public class Troll extends Personnage {

    public Troll(long pointVie) {
        super(pointVie);
    }
}
```

Appels implicite de Objet()

Génération des constructeurs - IntelliJ

Démo

Accessibilité

- private
- *package* (protection par défaut)
- protected (fortement déconseillée - encapsulation d’une classe mère)
- public

Bonnes pratiques

- Utiliser uniquement private / public et getters/setters

La classe Object

- En Java, java.lang.Object est la classe **racine** de l'arbre d'héritage
- Object n'a
  - ni variables de classes
  - ni variables d'instances
- Object a des méthodes/fonctions
  - String toString() : description de l'objet sous forme de caractère - classe & adresse par défaut
  - boolean equals(Object o) : égalité sémantique - égalité d'adresse par défaut
  - int hashCode() : valeur entière représentant l'objet - renvoie la valeur hexa de l'adresse mémoire par défaut
  - + quelques autres *dark* functions

Redéfinitions de méthodes / fonctions

- Annotation @Override

```
@Override
public boolean equals(Object x) {
        ...
}
```

- Annotation utile pour repérer les erreurs de saisie (nom différent ou profil pas identique) - Assurance d’une redéfinition effective

Bon chasseur ou mauvais chasseur ?

- Ne pas confondre **redéfinition** et **surcharge**
- Une méthode **redéfinit** une méthode héritée quand elle a la **même signature** que l'autre méthode
- Une méthode **surcharge** une méthode héritée (ou définie dans la classe) quand elle a le **même nom** mais **pas le même profil** que l'autre méthode

Exemple

Démo avec une super classe Nombre

Polymorphisme

- En Java pas 36 formes de polymorphisme
- Héritage multiple non autorisé (attention à Java 8)
- Implémentations multiples *d'interfaces* autorisées (plus tard)

- Le polymorphisme est le fait qu'une même expression peut correspondre à différents appels de méthodes
  - *late binding*

Polymorphisme - utilisation

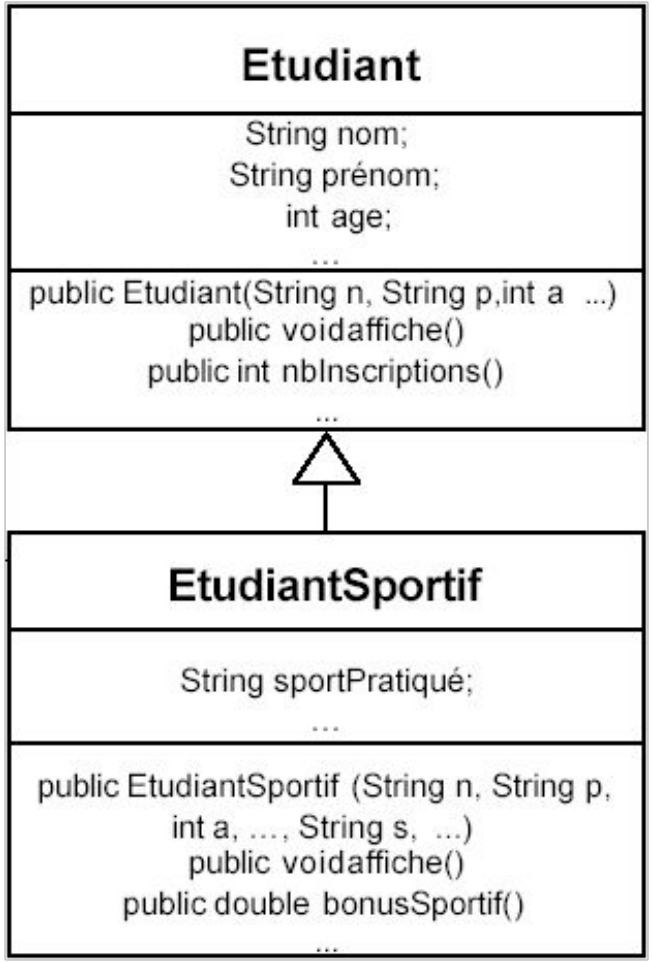
- Le polymorphisme évite de nombreux embranchements
  - Exemple : figures géométriques
- Le polymorphisme permet l’extension aisée de programme
  - Exemple : figures géométriques

Cast ou Transtypage

- *Caster* un objet = forcer le type de l'objet qui n'est pas le type déclaré ou réel de l'objet
- Les seuls cast autorisés sont ceux entre classes mères et classes filles
  - UPcast - souvent implicite
  - DOWNcast - doit être explicite et peut provoquer des erreurs à l'exécution
- Cas particulier - démo avec les double-int / Double-Integer

Jeu 1

```
EtudiantSportif es;
es = new EtudiantSportif("DUPONT",
  “fred”,25,..,"Badminton",..);
Etudiant e;
e = es; // upcasting
e.affiche();
es.affiche();
e.nbInscriptions();
es.nbInscriptions();
es.bonusSportif();
e.bonusSportif();
```



Jeu 2

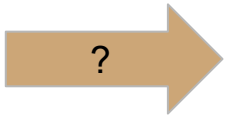
```
class A {
     int ivar = 7;
     void m1() {
          System.out.print("A m1, ");
     }
     void m2() {
          System.out.print("A m2, ");
     }
     void m3() {
          System.out.print("A m3, ");
     }
}
class B extends A {
     void m1() {
          System.out.print("B m1, ");
     }
}
```

```
class C extends B {
     void m3() {
          System.out.print("C m3, "+(ivar + 6));
     }
}

public class Melange {
     public static void main(String [] args) {
          A a = new A();
          B b = new B();
          C c = new C();
          A a2 = new C();
          ! code ici !
     }
}
```

Match ?

```
b.m1();
c.m2();
a.m3();
----
c.m1();
c.m2();
c.m3();
----
a.m1();
b.m2();
c.m3();
----
a2.m1();
a2.m2();
a2.m3();
```




```
A m1, A m2, C m3, 6

B m1, A m2, A m3,

A m1, B m2, A m3,

B m1, A m2, C m3, 13

B m1, C m2, A m3,

B m1, A m2, C m3, 6

A m1, A m2, C m3, 13
```

jeu 3

```
public class MonstreTestDrive {
  public static void main(String [] args) {
     Monstre [] ma = new Monstre[3];
     ma[0] = new Vampire();
     ma[1] = new Dragon();
     ma[2] = new Monstre();
     for(int x = 0; x < ma.length; x++) {
       ma[x].fairepeur(x);
     }
  }
}
```

```
class Monstre {
     // A
}

class Vampire extends Monstre {
     // B
}

class Dragon extends Monstre {
  boolean fairepeur(int degree) {
    System.out.println("souffler du feu");
  return true;
  }
}
```

```
% java MonstreTestDrive
mordre ?
souffler du feu
arrrgh
```

ça marche ?

```
//A
boolean fairepeur(int d) {
  System.out.println("arrrgh");
  return true;
}

// B
boolean fairepeur(int x) {
  System.out.println("mordre ?");
  return false;
}
```

```
// A
boolean fairepeur(int x) {
  System.out.println("arrrgh");
  return true;
}

// B
int fairepeur(int f) {
  System.out.println("mordre ?");
  return 1;
}
```

ça marche ?

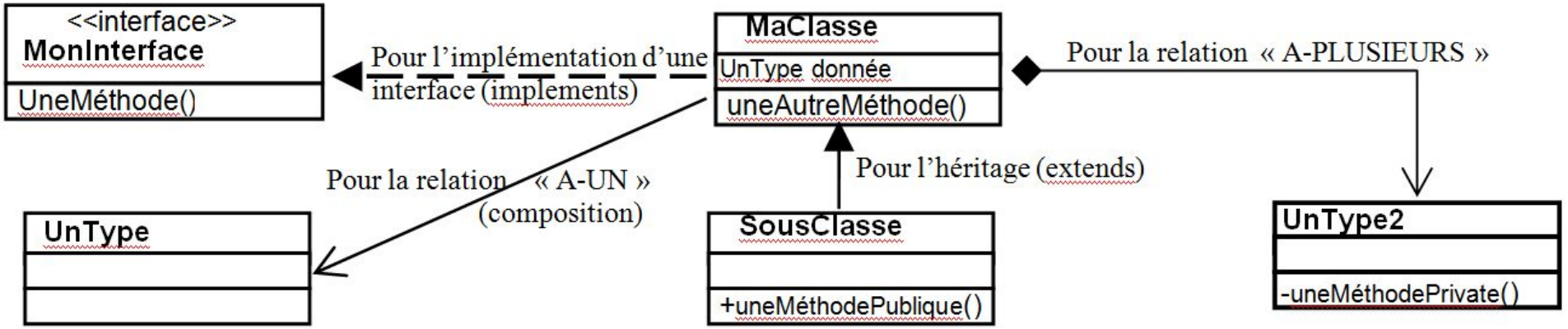
```
// A
boolean fairepeur(int x) {
  System.out.println("arrrgh");
  return false;
}

// B
boolean effrayer(int x) {
  System.out.println("mordre ?");
  return true;
}
```

```
// A
boolean fairepeur(int z) {
  System.out.println("arrrgh");
  return true;
}

// B
boolean fairepeur(byte b) {
  System.out.println("mordre ?");
  return true;
}
```

Dessin !


<<interface>>
MonInterface
UneMéthode()
Pour l'implémentation d'une interface (implements)
MaClasse
UnType donnée
uneAutreMéthode()
Pour la relation « A-PLUSIEURS »
Pour la relation « A-UN » (composition)
Pour l'héritage (extends)
UnType
SousClasse
+uneMéthodePublique()
UnType2
-uneMéthodePrivate()

A vous de jouer 4 :

```
public interface Foo { }
public class Bar implements Foo { }
```

```
public interface Vinn { }
public abstract class Vout implements Vinn { }
```

```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

```
public class Gamma extends Delta
implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma
implements Beta { }
public class Delta { }
```

Jeu 5

Un robot qui avance

Un robot qui roule

un robot qui marche

Jeu 5 - évolution

Un robot qui avance

Ce robot peut rouler quand le terrain est plat

Ce robot marche quand le terrain est accidenté

Jeu 6 sur machine

Passionné de zoologie, vous avez décidé de construire une application Java permettant de représenter différents types d'animaux. Vous avez commencé par vos animaux préférés, les canards, en créant une interface Canard possédant deux méthodes :
–void cancaner() qui permet de faire caqueter le canard
–dandiner(double distance) qui permet de faire marcher le canard sur une certaine distance

Vous avez ensuite implanté plusieurs réalisations de cette interface : la classe Colvert et la classe Mandarin

Jeu 6 sur machine

En avançant dans la construction de votre application, vous avez défini une nouvelle interface Animal plus générale avec les méthodes suivantes :
–void faireDuBruit()
–avance(double distance) qui permet de faire bouger l'animal en question

Ecrire une classe Chien et Dindon qui implémentent cette interface.

Jeu 6 sur machine

Vous vouliez pouvoir gérer des collections complètes d’animaux variés en récupérant le code de vos canards. Malheureusement, le disque dur de votre vieil ordinateur vous a lâché et vous ne disposez plus que du bytecode de l’interface Canard et de ses réalisations. Impossible donc de modifier le code source ces classes et vous ne pouvez pas les réécrire, les caquètements des canards vous ayant demandé trop de temps à implanter !
Proposez une solution permettant de considérer vos canards (Colvert, Mandarin) comme des objets de type Animal, pour pouvoir les insérer dans un tableau d'Animal ; écrire un main faisant avancer un troupeau d’animaux comprenant 2 chiens, un colvert, un didon et un mandarin.