ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École de science informatique et de génie électrique
Université d'Ottawa

Version du 25 janvier 2012

Résumé

*. Pensez-y, n'imprimez ces notes de cours que si c'est nécessaire!

Résumé

En ITI 1521, nous prendrons l'habitude de toujours déclarer les variables d'instance privées (ou protégées, voir ci-bas).

Si l'accès au contenu d'une variable est nécessaire (en lecture ou écriture), nous définirons des méthodes d'accès (setters et getters). Cette façon de faire nous donnera un très grand contrôle sur le contenu des objets et leur intégrité.

Héritage

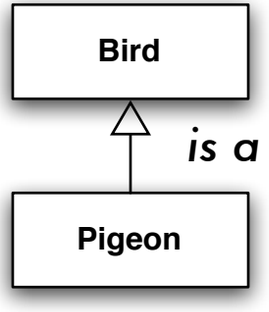
Les langages OO offrent plusieurs mécanismes afin structurer les programmes.

L'héritage est l'un de ces mécanismes et il favorise l'organisation des classes de façon hiérarchique (sous forme d'arborescence).

Lorsqu'on dit que la programmation orientée objet favorise la **réutilisation de code** on fait alors référence à la notion d'héritage.

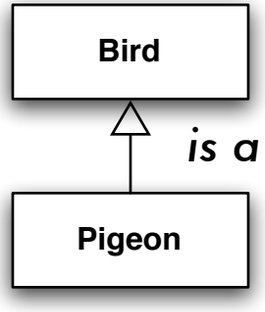
Héritage

La classe située au-dessus, dans l'arbre d'héritage, s'appelle la **superclasse** (ou **parent**), alors que la classe située au-dessous s'appelle sous-classe (on dit aussi classe dérivée).



Dans cet exemple, **Bird** est la superclasse de **Pigeon**, c'est-à-dire que **Pigeon** est une sous-classe de **Bird**.

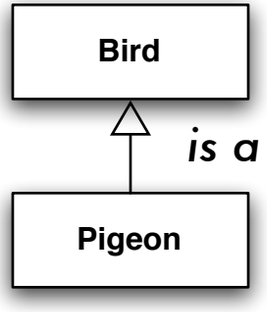
Héritage



En Java, la relation ≪is a≫ (est un) est exprimée à l'aide du mot clé réservé **extends**.

```
public class Pigeon extends Bird {
    ...
}
```

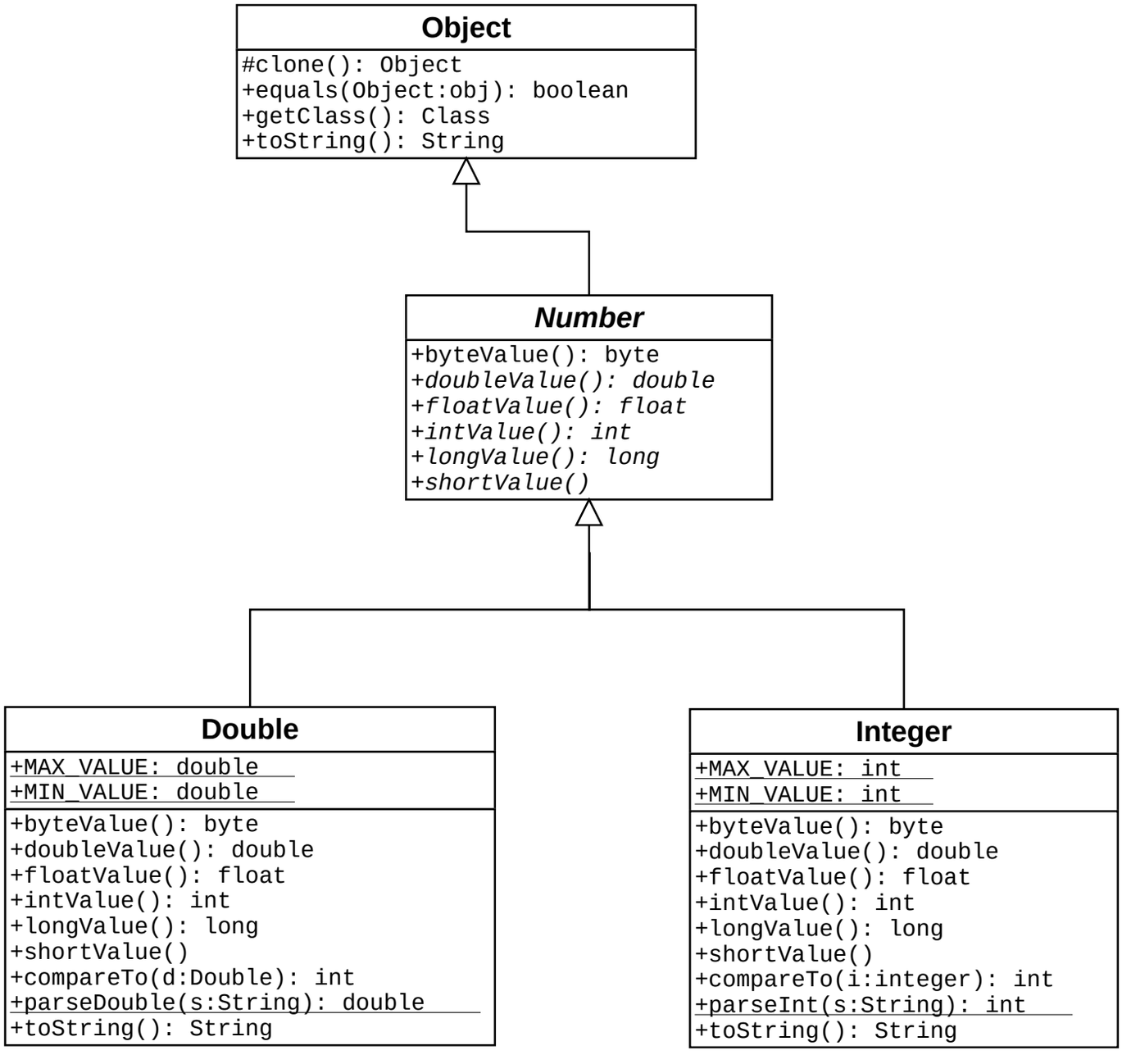
Héritage



En UML, la relation ≪is a≫ est exprimée à l'aide d'une ligne pleine connectant l'enfant à son parent et telle qu'un triangle ouvert pointe dans la direction du parent.

Héritage

En Java, les classes sont organisées sous forme d'arborescence. La classe la plus générale, celle qui est à la racine de l'arbre, s'appelle **Object**.


Object
#clone(): Object
+equals(Object:obj): boolean
+getClass(): Class
+toString(): String
Number
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
Double
+MAX_VALUE: double
+MIN_VALUE: double
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
+compareTo(d:Double): int
+parseDouble(s:String): double
+toString(): String
Integer
+MAX_VALUE: int
+MIN_VALUE: int
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
+compareTo(i:integer): int
+parseInt(s:String): int
+toString(): String

Héritage

Si la superclasse n'est pas mentionnée explicitement, **Object** est la superclasse par défaut, ainsi la déclaration suivante :

```
public class C {
    ...
}
```

est équivalente à celle-ci :

```
public class C extends Object {
    ...
}
```

Héritage

En Java, toutes les classes ont exactement un parent ; sauf la classe **Objet** qui n'en a pas.

On parle alors d'**héritage simple** par opposition à l'héritage multiple.

Ça sert à quoi ?

Une classe hérite des **caractéristiques** (variables et méthodes) de sa superclasse.

1. Une sous-classe **hérite** des méthodes et variables de sa superclasse ;
2. Une sous-classe peut **introduire/ajouter** de nouvelles méthodes et variables ;
3. Une sous-classe peut **redéfinir** les méthodes de la superclasse.

Puisqu'on ne peut qu'ajouter de nouveaux éléments, ou les redéfinir, une superclasse est plus **générale** que ses sous-classes, et inversement une sous-classe est plus spécialisée que sa superclasse.

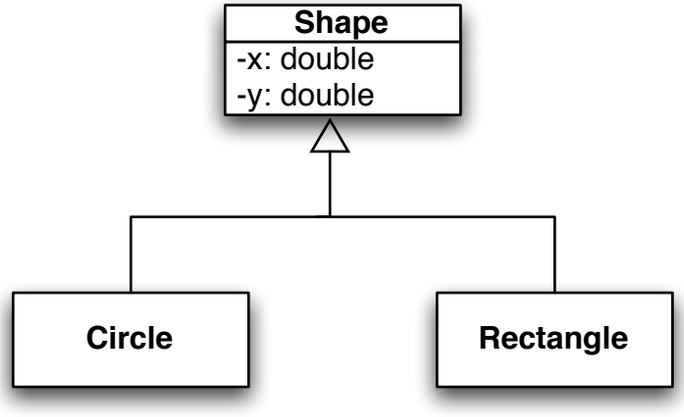
Shape

Voici l'exemple classique se trouvant dans tous les manuels de programmation orientée objet.

Problème : On doit construire un ensemble de classes afin de représenter des formes géométriques, telles que des cercles et des rectangles.

Tous les objets doivent posséder deux variables d'instance, **x** et **y**, qui représentent la position de l'objet.

Shape



Shape

De plus, **tous les objets** devraient posséder les méthodes suivantes :

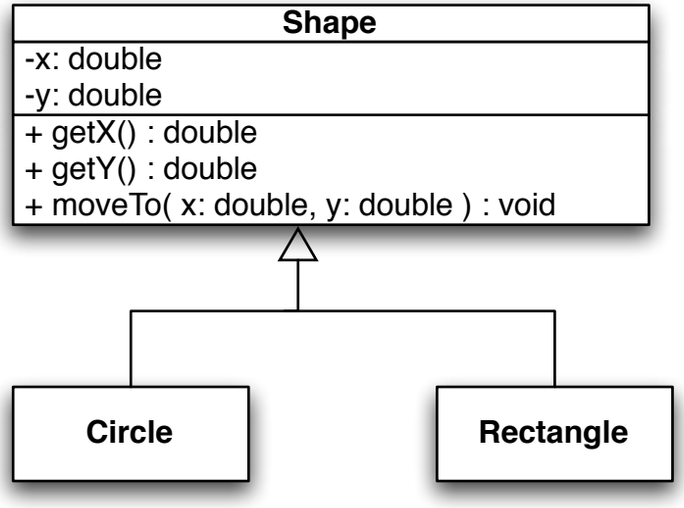
```
double getX();                          // Retourne la valeur de x

double getY();                          // Retourne la valeur de y

void moveTo(double x, double y); // Déplace l'objet vers un nouvel
                                 //   emplacement
double area();                          // Calcul l'aire de la forme

void scale(double factor);              // Transforme par un certain facteur

String toString();                      // Retourne une chaîne de caractères
```

Garder cette spécification en mémoire, nous ne serons pas en mesure de l'implémenter de façon satisfaisante lors de notre premier essai.

Shape

L'implémentation des trois premières méthodes ne pose aucun problème, elle est la même, peu importe le type de forme.

Shape


Shape
-x: double
-y: double
+ getX() : double
+ getY() : double
+ moveTo(x: double, y: double) : void
Circle
Rectangle

Shape

Par contre, le calcul de l'aire (**area**) dépend du type de forme et de même pour la méthode **scale**.

Finalement, la méthode **toString()** affichera l'information commune à toutes formes, emplacement, ainsi que l'information spécifique, le rayon dans le cas du cercle.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
}
```

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

Est-ce une déclaration valide ? Oui. Plusieurs méthodes (ou constructeurs) peuvent porter le même nom, à condition que les signatures des méthodes (constructeurs) diffèrent. Je nomme ce concept polymorphisme *ad hoc* (**overloading**).

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

À quoi sert le polymorphisme *ad hoc* (**overloading**) ?

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    // ...

    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }

}
```

Ajout de méthodes d'accès.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public final double getX() {
        return x;
    }
    public final double getY() {
        return y;
    }

}
```

Par l'utilisation du mot clé **final**, nous rendons impossible la redéfinition de méthodes.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public final double getX() { return x; }
    public final double getY() { return y; }

    public final void moveTo( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

Circle

```
public class Circle extends Shape {



}
```

La déclaration ci-haut signifie que la classe **Circle** est une sous-classe de la classe **Shape**.

Ainsi, tous les objets de la classe **Circle** auront deux variables d'instance, **x** et **y**, ainsi que les méthodes suivantes, **getX()**, **getY()** et **moveTo(double x, double)**.

Circle

```
public class Circle extends Shape {

    // Variable d'instance
    private double radius;

}
```

Private vs protected

Si les variables **x** et **y** sont déclarées privées dans la classe **Shape**, alors, bien qu'un objet de la classe **Circle** ait sa propre copie des variables, les méthodes de la classe **Circle** n'ont pas accès aux variables d'instances (définies dans la classe **Shape**).

Ceci ne fonctionne pas :

```
public Circle (double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
}
```

Le compilateur dira ≪x has private access in Shape≫ (pareillement pour y).

Private vs protected

On peut contourner ce problème en déclarant les variables **protected** dans la classe **Shape**.

```
public class Shape extends Object {

    protected double x;
    protected double y;

    ...
}
```

Private vs protected

Il est préférable de déclarer les variables **private**.

Ce qui nous forcera à utiliser les méthodes d'accès de la superclasse.

Ce qui a pour effet de concentrer toutes méthodes qui agissent sur ces variables en un même endroit.

Pour un maximum de contrôle, on déclare des variables d'instance privées et des méthodes d'accès **final**, ainsi les sous-classes ne peuvent redéfinir ces méthodes.

Circle

```
public class Circle extends Shape {

    private double radius;

    // Constructeurs

    public Circle() {
        super();
        radius = 0;
    }

    public Circle( double x, double y, double radius ) {
        super( x, y );
        this.radius = radius;
    }

}
```

super()

L'énoncé **super(. . .)** est un appel explicite au constructeur de la superclasse immédiate.

- Cette forme d'appel, super(. . .), n'apparaît que dans un constructeur ;
- Cet appel doit être le premier énoncé du constructeur ;
- Un appel de la forme **super()** est **automatiquement** inséré à moins que vous n'ajoutiez un appel **super(. . .)** vous-même ! ?

Piège !

– Un appel de la forme **super()** est **automatiquement** inséré à moins que vous n'ajoutiez un appel **super(. . .)** vous-même ! ?

Si vous n'ajoutez pas un appel explicite **super(. . .)**, Java insère **super()** automatiquement, ce constructeur d'arité 0 doit exister dans la superclasse, sinon cela causera une erreur de compilation. Souvenez-vous, le constructeur par défaut, celui d'arité 0, n'est ajouté à la classe que si vous n'avez pas défini un constructeur !

Circle

```
public class Circle extends Shape {

    private double radius;

    // Méthode d'accès

    public double getRadius() {
        return radius;
    }

}
```

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    public Rectangle() {
        super();
        width = 0;
        height = 0;
    }

    public Rectangle( double x, double y, double width, double height )
        super(x, y);
        this.width = width;
        this.height = height;
    }
```

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }
}
```

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public void flip() {
        double tmp = width;
        width = height;
        height = tmp;
    }

}
```

```
Circle d = new Circle( 100, 200, 10 );
System.out.println( d.getRadius() );

Circle c = new Circle();
System.out.println( c.getX() );

d.scale( 2 );
System.out.println ( d );

Rectangle r = new Rectangle();
System.out.println( r.getWith() );

Rectangle s = new Rectangle( 50, 50, 10, 15 );
System.out.println( s.getY() );

s.flip();
System.out.println( s.getY() );
```

Résumé

L'héritage favorise la ré-utilisation (partage) de méthodes. Les méthodes **getX()** et **getY()** ont été déclarées une seule fois.

Si on ajoute une nouvelle classe, **Triangle**, celle-ci aura automatiquement les méthodes **getX()** et **getY()** ; si elle est une sous-classe de **Shape**.

Si on corrige une erreur ou améliore une méthode d'une superclasse, toutes les sous-classes en bénéficient.