ITI 1521. Introduction à l'informatique II

Héritage : réutilisabilité

Marcel Turcotte

uOttawa

École de science informatique et de génie électrique (SIGE)
Université d'Ottawa

Version du 1er février 2017

Aperçu

Héritage : réutilisabilité

Le concept d'héritage en Java favorise la réutilisation de code. L'héritage exprime une relation de type parent-enfant entre deux classes. La sous-classe possède les attributs et méthodes de la superclasse. Celle-ci peut aussi introduire de nouveaux attributs et de nouvelles méthodes. Finalement, la sous-classe peut redéfinir certaines méthodes de la superclasse.

Objectif général :

- Cette semaine, vous serez en mesure de structurer un ensemble de classe de façon hiérarchique à l'aide de l'héritage.

Une vidéo d'introduction :

- `https://www.youtube.com/watch?v=TuPV5om-mVQ`

Objectifs d'apprentissage

- **Décrire** le fonctionnement d'une application simple utilisant les concepts d'héritage.
- **Construire** une application simple à partir de sa spécification et de diagrammes UML.
- **Critiquer** l'usage du modificateur de visibilité «protected».

Lectures :

- Pages 8–21, 27–33 de E. Koffman et P. Wolfgang.

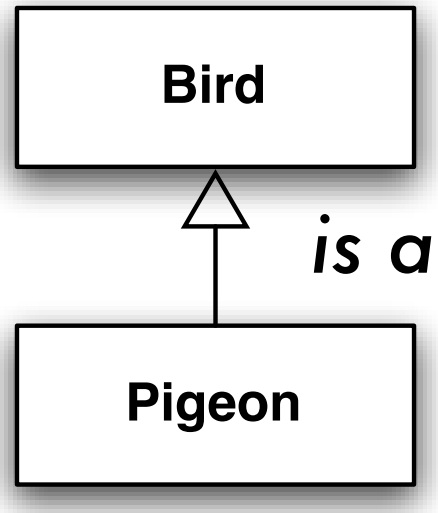
Plan

Introduction

- Les langages orienté objet 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.

Définitions : superclasse et sous-classe

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**).

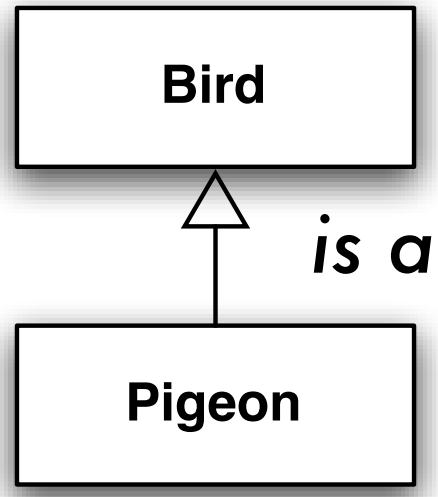


Dans cet exemple, **Bird** est la superclasse de **Pigeon**, c'est-à-dire que **Pigeon** est une sous-classe de **Bird**.

Java : extends

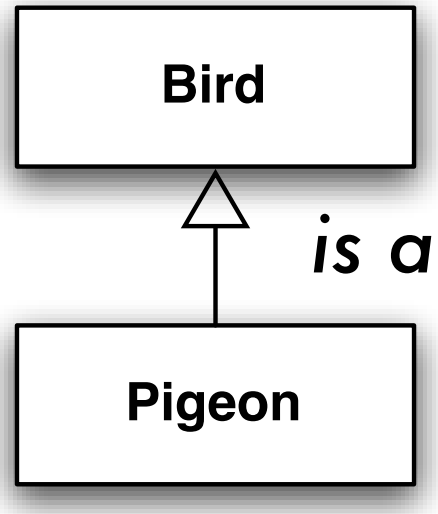
En Java, la relation «*is a*» (est un) est exprimée à l'aide du mot clé réservé **extends**.

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



UML

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.



Ça sert à **quoi** ?

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

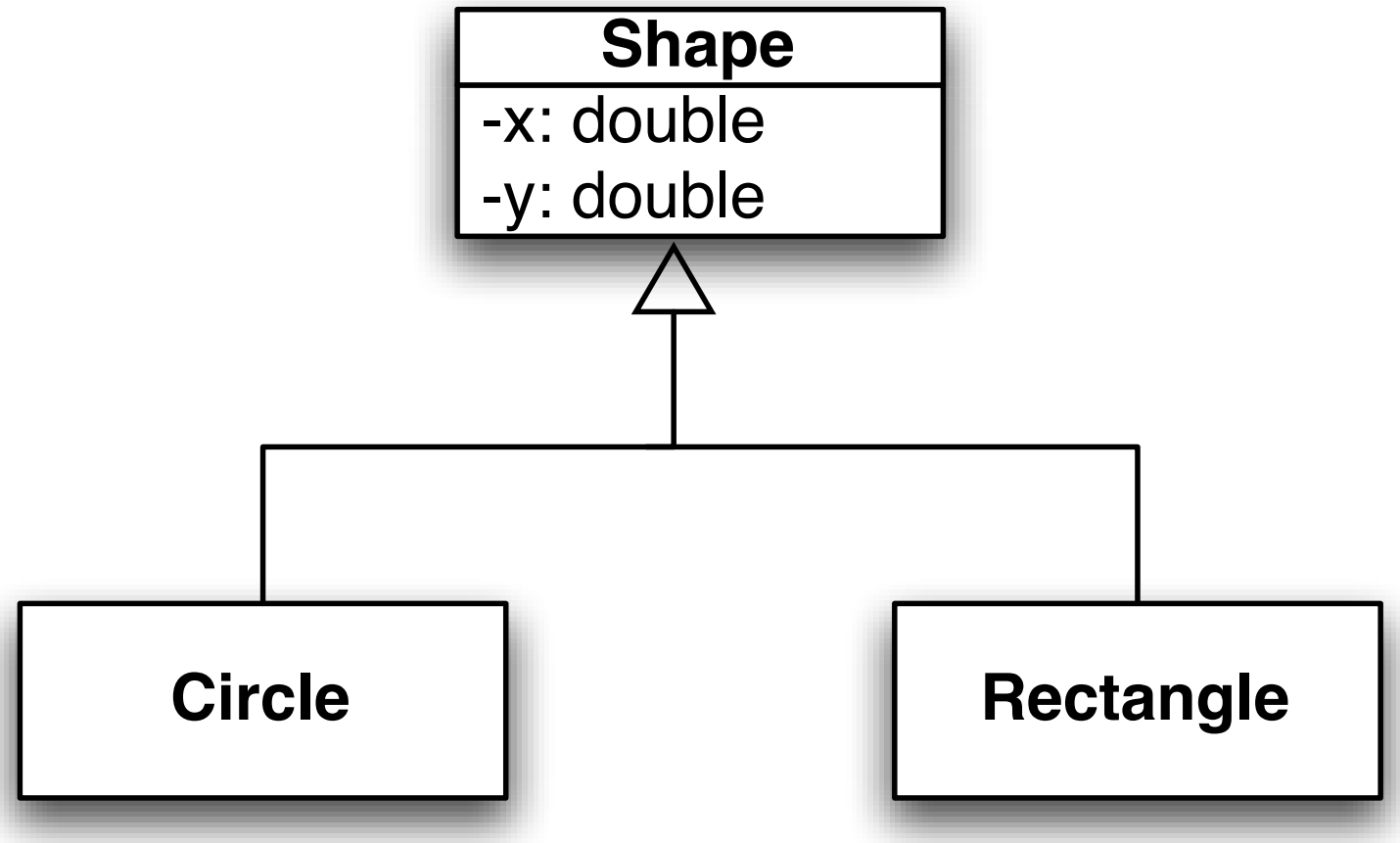
1. Une sous-classe **hérite** des méthodes et variables de sa superclasse ;
2. Une sous-classe peut **introduire/ajouter** de nouvelles méthodes et variables ;
3. Une sous-classe peut **redéfinir** les méthodes de la superclasse.

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.

Exemple : Shape

- **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.

Exemple : Shape



```java
public class Shape {

    private double x;
    private double y;

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

```java
public class Shape {

    private double x;
    private double y;

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

- Oui, oui ! Plusieurs méthodes (ou constructeurs) peuvent porter le même nom, à condition que les signatures des méthodes (constructeurs) diffèrent.
- Il s'agit de la **surcharge de nom** (**overloading**).

Méthodes d'**accès**

```
public class Shape {

    private double x;
    private double y;

    // ...

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

}
```

Méthodes **toString**

```
public class Shape {

    private double x;
    private double y;

    // ...

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

    public String toString() {
        return "Located at : (" + x + "," + y + ")";
    }

}
```

Exemple : Circle

```
public class Circle extends Shape {


}
```

- La déclaration ci-haut signifie que la classe **Circle** est une sous-classe de la classe **Shape**.
- Tous les objets de la classe **Circle** auront deux variables d'instance, **x** et **y**, ainsi que les méthodes suivantes, **getX()** et **getY()**.

Example : Circle

```
public class Circle extends Shape {

    // Variable d'instance
    private double radius;

}
```

Private versus **protected**

```
public class Circle extends Shape {

    private double radius;

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

}
```

La compilation du constructeur ci-dessus produira l'**erreur** suivante «*x has private access in Shape*» (pareillement pour y).

protected

On peut résoudre ce problème en déclarant les variables **protected** dans la superclasse **Shape**.

```
public class Shape {

    protected double x;
    protected double y;

    // ...
}
```

Private

- Je préfère garder la visibilité des variables **private**.
- Ce qui nous forcera à utiliser les **méthodes d'accès** de la superclasse.

super

```
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 !?

```java
public class Circle extends Shape {

    private double radius;

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

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

    public double getRadius() {
        return radius;
    }

}
```

Exemple : 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
        super(x, y);
        this.width = width;
        this.height = height;
    }
}
```

Exemple : Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double getWidth() {
        return width;
    }

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

Utilisation

```
Circle c, d;

d = new Circle(100.0, 200.0, 10.0);
System.out.println(d.getRadius());

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

Rectangle r, s;

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

s = new Rectangle(50.0, 50.0, 10.0, 15.0);
System.out.println(s.getY());
```

Polymorphisme

- Du grecque ***polus*** = plusieurs et ***morphê*** = formes, signifie donc **qui a plusieurs formes**.

Définitions

En informatique, le **polymorphisme** consiste à permettre l'utilisation d'un identifiant pour différentes entités (voir différents types).

1. **Polymorphisme *ad hoc* (surchage de nom)** : un même nom de méthode est associé à des blocs d'énoncés différents. Ces méthodes ont le même nom, mais elles diffèrent par leur liste de paramètres.
2. **Polymorphisme par sous-typage (par héritage)** : un identifiant est lié à des données de types différents par une relation de sous-type.
3. **Polymorphisme paramétré (générique)** : la classe possède un ou plusieurs paramètres formels de type.

Surcharge de nom

- La classe **PrintStream** utilise le polymorphisme *ad hoc* afin d'implémenter la méthode **println**.

```
println()
println(boolean value)
println(char value)
println(char[] value)
println(double value)
println(float value)
println(int value)
println(long value)
```

Surcharge de nom (suite)

- Trois méthodes ayant des **signatures** * différentes.

```
public static int somme(int a, int b, int c) {
    return a + b + c;
}
public static int somme(int a, int b) {
    return a + b;
}
public static double somme(double a, double b) {
    return a + b;
}
```

*En Java, la signature d'une méthode comprend le nom de la méthode et la liste des paramètres, mais pas la valeur de retour.

Polymorphisme par **sous-typage**

Problème : implémenter une méthode **isLeftOf** qui retourne **true** is **cette** forme est située à la gauche de son argument (une autre forme géométrique) et **false** sinon.

isLeftOf

```
Circle c1, c2;
c1 = new Circle(10, 20, 5);
c2 = new Circle(20, 10, 5);

if (c1.isLeftOf(c2)) {
    System.out.println("c1 isLeftOf c2");
} else {
    System.out.println("c2 isLeftOf c1");
}
```

isLeftOf

```
Rectangle r1, r2;
r1 = new Rectangle(0, 0, 1, 1);
r2 = new Rectangle(100, 100, 200, 400);

if (r1.isLeftOf(r2)) {
    System.out.println("r1 isLeftOf r2");
} else {
    System.out.println("r2 isLeftOf r1");
}
```

isLeftOf

```
if (r1.isLeftOf(c1)) {
    System.out.println("r1 isLeftOf c1");
} else {
    System.out.println("c1 isLeftOf r1");
}

if (c2.isLeftOf(r2)) {
    System.out.println("c2 isLeftOf r2");
} else {
    System.out.println("r2 isLeftOf c2");
}
```

Une solution **absurde** !

```
public boolean isLeftOf(Circle c) {
    return getX() < c.getX();
}
public boolean isLeftOf(Rectangle r) {
    return getX() < r.getX();
}
```

- **Pourquoi ?**

Une solution **absurde** !

```
public boolean isLeftOf(Circle c) {
    return getX() < c.getX();
}
public boolean isLeftOf(Rectangle r) {
    return getX() < r.getX();
}
```

- **Autant d'implémentation** que de variétés de formes !
- Toutes les implémentations sont **identiques** !
- Lorsqu'une nouvelle catégorie de formes est définie (**Triangle**) une nouvelle méthode **iLeftOf** doit être créée !

Solution

- Suggestions ?

```
public boolean isLeftOf("Any Shape" s) {
    return getX() < s.getX();
}
```

- “Any Shape” ?

Solution

- Implémentons la méthode **isLeftOf** dans la classe **Shape** comme suit.

```
public boolean isLeftOf(Shape s) {
    return getX() < s.getX();
}
```

isLeftOf

```
Circle c;
c = new Circle(10, 20, 5);

Rectangle r;
r = new Rectangle(0, 0, 1, 1);

if (c.isLeftOf(r)) {
    System.out.println("c isLeftOf r");
} else {
    System.out.println("r isLeftOf c");
}
```

isLeftOf

```
if (c.isLeftOf(r)) {
    // ...
```

- La méthode **isLeftOf** de l'objet désigné par la référence **c** est appelée.
- Parfait, **c** désigne un objet de la classe **Circle**, cette dernière hérite de la méthode **isLeftOf**.

isLeftOf

```
if (c.isLeftOf(r)) {
    // ...
```

- Hum, lors de l'appel, la valeur du paramètre actuel, **r**, est copiée dans le paramètre formel, **s**.
- Doit-on conclure que les énoncés suivants sont aussi valides ?

```
Shape s;
Rectangle r;
r = new Rectangle( 0, 0, 1, 1 );
s = r;
```

Types

- «A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type (§4.2) or a reference type (§4.3). A variable always contains a value that is assignment compatible (§5.2) with its type.»
- «Assignment of a value of compile-time reference type S (source) to a variable of compile-time reference type T (target) is checked as follows :
  - If S is a class type :
    - If T is a class type, then S must either be the same class as T, or S must be a subclass of T, or a compile-time error occurs."

⇒ Gosling et al. (2000) *The Java Language Specification.*

Variables

- «Une variable est un emplacement mémoire ainsi qu'un type associé, dit type de compilation, qui peut être primitif ou référence. Une variable renferme toujours une valeur qui est compatible avec son type.»
- «L'affection d'une valeur d'un type de compilation référence **S** (source) à une variable de type référence d'un type de compilation référence **T** (target/destination) est validée à l'aide de la règle suivante :»
- «Si **S** est le nom d'une classe et si **T** est aussi le nom d'une classe alors, **S** et **T** sont la même classe, ou encore **S** est une sous-classe de **T**, sinon il y aura une erreur de compilation.»

isLeftOf

En effet, cette définition confirme que les énoncés qui suivent sont valides.

```
Shape s;
Rectangle r;
r = new Rectangle( 0, 0, 1, 1 );
s = r;
```

mais pas “**r = s**” !

Polymorphique

Une variable **s** désigne un objet de la classe **Shape** ou l'une de ses sous-classes.

```
Shape s;
```

Utilisation :

```
s = new Circle(0, 0, 1);
s = new Rectangle(10, 100, 10, 100);
```

Polymorphisme

```
public boolean isLeftOf(Shape other) {
    boolean result;
    if (getX() < other.getX()) {
        result = true;
    } else {
        result = false;
    }
    return result;
}
```

Utilisation :

```
Circle c = new Circle(10, 10, 5);
Rectangle d = new Rectangle(0, 10, 12, 24);
if (c.isLeftOf(d)) { ... }
```

Exercises

```
Shape s;
Circle c;
c = new Circle(0, 0, 1);
s = c;

if (c.getX()) { ... } // valid?
if (s.getX()) { ... } // valid?

if (c.getRadius()) { ... } // valid?
if (s.getRadius()) { ... } // valid?
```

Remarques

```
Shape s;
Circle c;
c = new Circle(0, 0, 1);
s = c;
```

- L'objet désigné par **s** demeure un cercle (**Circle**). La classe d'un objet demeure la même tout au long de l'exécution du programme.

Remarques

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;

if ( s.getX() ) { ... }
```

- Lorsqu'on utilise **s** afin de désigner un cercle (**Circle**), l'objet "est vu comme" une forme géométrique (**Shape**), en ce sens qu'on n'en voit que les caractéristiques (méthodes et variables) définies dans la classe **Shape**.

Remarques

- Le polymorphisme est un concept puissant. La méthode **isLeftOf** que nous avons définie sert non seulement à traiter des cercles et rectangles, mais aussi tout objet d'une future sous-classe de la classe **Shape**.

```
public class Triangle extends Shape {
    // ...
}
```

Calcul de l'**aire**

Problème : On souhaite que **toutes** les formes géomériques (objets des sous-classes de **Shape**) possèdent une méthode pour le calcul de l'**aire**.

Qu'est-ce que tu veux dire Marcel ?

```
public class Shape {

  // ...

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return  0;
      } else {
          return  1;
      }
  }
}
```

Qu'en **pensez-vous** ?

```
public class Shape {

  // Must be redefined by the subclasses or else ...

  public double area() {
      return -1.0;
  }

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return  0;
      } else {
          return  1;
      }
  }
}
```

Abstract

```
public class Shape {

  public abstract double area();

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return  0;
      } else {
          return  1;
      }
  }
}
```

Abstract

```
public abstract class Shape {

  public abstract double area();

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return  0;
      } else {
          return  1;
      }
  }
}
```

Classes abstraites

- Une classe déclarant une **méthode abstraite** doit être **abstraite**.
- On **ne peut créer** d'objets d'une classe abstraite.
- On peut déclarer une classe abstraite, même si elle ne contient pas de méthodes abstraites.

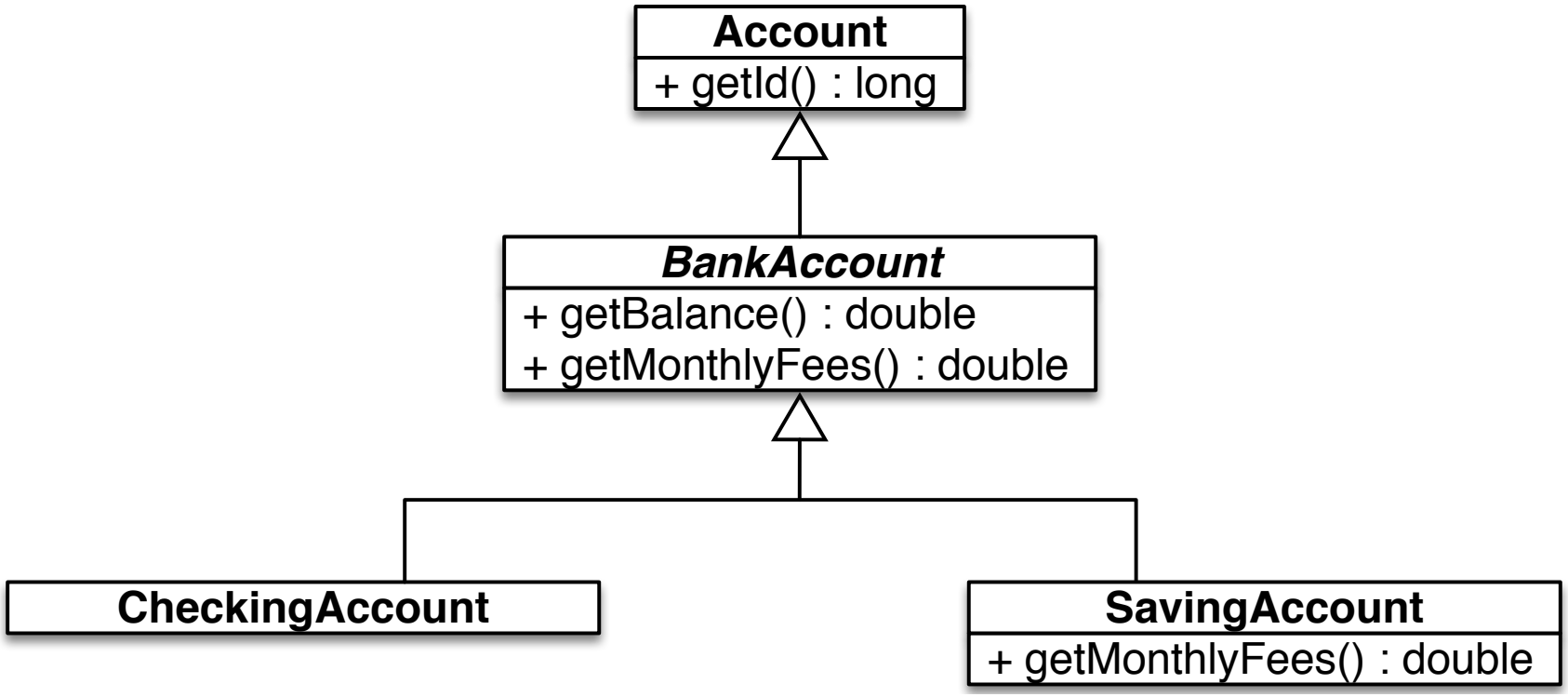
Qu'avons-nous **obtenu** ?

```
public class Circle extends Shape {

}
```

```
Circle.java:1: Circle is not abstract and
does not override abstract method area() in Shape
public class Circle extends Shape {
       ^
1 error
```

```
public class Circle extends Shape {

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }

    public void scale(double factor) {
        radius *= factor;
    }
}
```

Recherche de **nom**



- **BankAccount** et **SavingAccount** possède une méthode **getMonthlyFees**.

BankAccount :

```
public double getMonthlyFees() {
    return 25.0
}
```

SavingAccount :

```
public double getMonthlyFees() {
    double result;
    if (getBalance() > 5000.0) {
        result = 0.0;
    } else {
        result = super.getMontlhyFees();
    }
    return result;
}
```

- **Considerez** les énoncés suivants :

```
Account a;

BankAccount b;

SavingAccount s;

s = new SavingAccount();
s.getMonthlyFees();

b = s;
b.getMonthlyFees();

a = b;
a.getMonthlyFees();
```

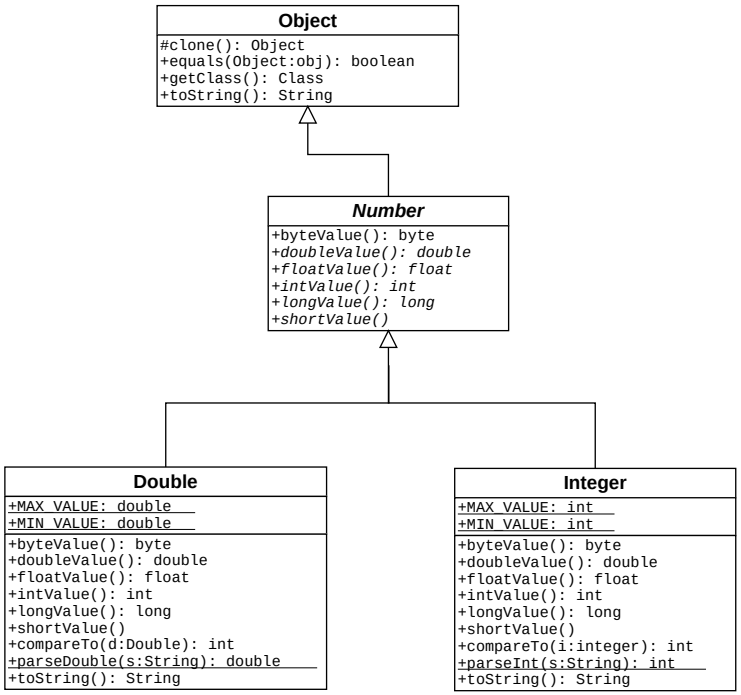
Association **dynamique**

- Soit **S** (***source***) le type de l'objet désigné par une variable de type **T** (***target***).
- À moins que la méthode ne soit **static** ou **final**, la recherche de nom :
  1. se fait au moment de l'**exécution** du programme ;
  2. la recherche débute avec la classe **S**.
     - Si la méthode est **trouvée**, elle est **exécutée** ;
     - Sinon, la recherche se poursuit avec la **superclasse immédiate** ;
     - Ce mécanisme **se poursuit** jusqu'à ce que la méthode ait été trouvée.

$\Rightarrow$ association **tardive** (**late binding**) ou associate virtuelle (*virtual binding*)

Object

- 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

- 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 {

}
```

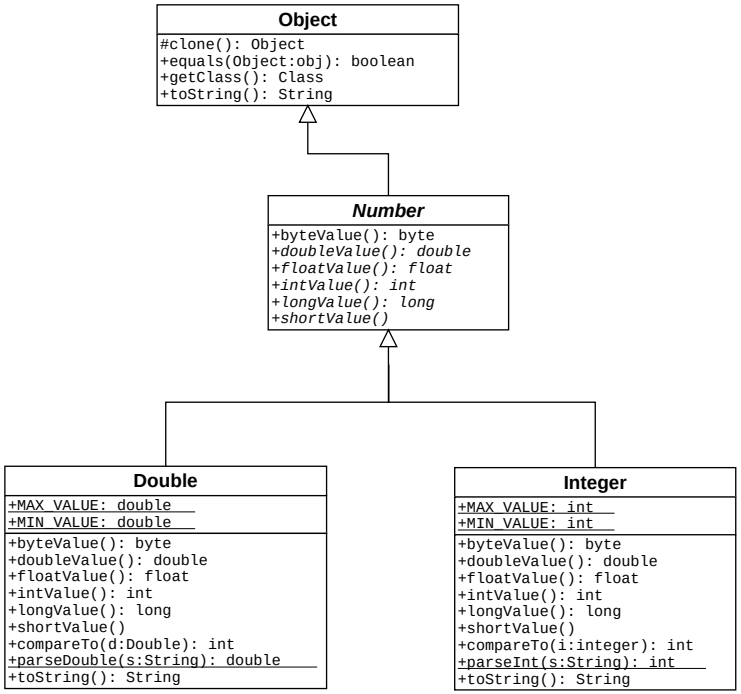
Exemple

```
import java.awt.TextField;

public class TimeField extends TextField {
    public Time getTime() {
        return Time.parseTime(getText());
    }
}
```

```
// java.lang.Object
//   |
//   +--java.awt.Component
//         |
//         +--java.awt.TextComponent
//               |
//               +--java.awt.TextField
//                     |
//                     +--TimeField
```

equals

- La classe **Object** définit une méthode **equals**.
- **Tout** objet Java possède donc une méthode **equals**
- On peut donc **toujours** écrire **a.equals(b)** si **a** et **b** sont des variables références.



equals

- Voici la méthode **equals** de la classe **Object**.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Account

```
public class Account {
    private int id;
    private String name;

    public Account(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Test

```
public class Test {
    public static void main(String[] args) {
        Account a, b;
        a = new Account(1, new String("Marcel"));
        b = new Account(1, new String("Marcel"));
        if (a.equals(b)) {
            System.out.println("a and b are equals");
        } else {
            System.out.println("a and b are not equals");
        }
    }
}
```

- Quel sera le **résultat** affiché ?

```
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) { // <---
            result = false;
        } ...
        return result;
    }
}
```

```
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) {
            result = false;
        } else if (! (o instanceof Account)) { // <---
            result = false;
        } ...
        return result;
    }
}
```

```
public class Account {
    private int id;
    private String name;
    public Account( int id, String name ) { ... }
    public boolean equals( Object o ) {
        boolean result = true;
        if ( o == null ) {
            result = false;
        } else if ( ! ( o instanceof Account ) ) {
            result = false;
        } else {
            Account other = (Account) o; // <---
            ...
        }
        return result;
    }
}
```

```
public class Account {
    private int id;  private String name;
    public Account(int id, String name) { ... }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) {
            result = false;
        } else if (! (o instanceof Account)) {
            result = false;
        } else {
            Account other = (Account) o;
            if ( id != other.id ) {
                result = false;
            } else if ( name == null && other.name != null ) {
                result = false;
            } else if ( name != null && ! name.equals( other.na
                result = false;
            }
        }
        return result;
    }
}
```

Test

```
public class Test {
    public static void main(String[] args) {
        Account a, b;
        a = new Account(1, new String("Marcel"));
        b = new Account(1, new String("Marcel"));
        if (a.equals(b)) {
            System.out.println("a and b are equals");
        } else {
            System.out.println("a and b are not equals");
        }
    }
}
```

- Quel sera le **résultat** affiché ?

Résumé

- L'héritage nous permet d'organiser les classes de façon hiérarchique
- Le mot clé **extends** dans la signature d'une classe indique son parent
- L'héritage permet la création de méthodes **polymorphiques**

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 2e edition, 2010.

Pensez-y !

L'impression de ces notes n'est probablement pas nécessaire !