ITI 1521. Introduction à l'informatique II

Héritage : polymorphisme

by

Marcel Turcotte

Version du 8 février 2020

Préambule

Préambule

Aperçu

Aperçu

Héritage : polymorphisme

Le concept d'héritage en Java favorise la réutilisation de code et supporte la notion de polymorphisme.

Objectif général :

- Cette semaine, vous serez en mesure de créer des méthodes polymorphiques.

Une vidéo d'introduction :

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

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Décrire** le polymorphisme.
- **Concevoir** une méthode polymorphique.
- **Comparer** les concepts d'interface et de classe abstraite.

Lectures :

- Pages 7–31, 39–45 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

Polymorphisme

Polymorphisme

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

Définitions

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

1. **Polymorphisme *ad hoc* (surcharge 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** si **cette** forme est située à la gauche de son argument (une autre forme géométrique) et **false** sinon.

isLeftOf

```
Circle c1, c2;
c1 = new Circle(10.0, 20.0, 5.0);
c2 = new Circle(20.0, 10.0, 5.0);

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, 0.0, 1.0, 1.0);
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(10.0, 20.0, 5.0);

Rectangle r;
r = new Rectangle(0.0, 0.0, 1.0, 1.0);

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, 0.0, 1.0, 1.0);
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, 0.0, 1.0, 1.0);
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, 0.0, 1.0);
s = new Rectangle(10.0, 100.0, 10.0, 100.0);
```

Polymorphisme

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

Utilisation :

```
Circle c = new Circle(10.0, 10.0, 5.0);
Rectangle d = new Rectangle(0.0, 10.0, 12.0, 24.0);
if (c.isLeftOf(d)) { ... }
```

Exercises

```
Shape s;
Circle c;
c = new Circle(0.0, 0.0, 1.0);
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, 0.0, 1.0);
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, 0.0, 1.0);
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 géométriques (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.

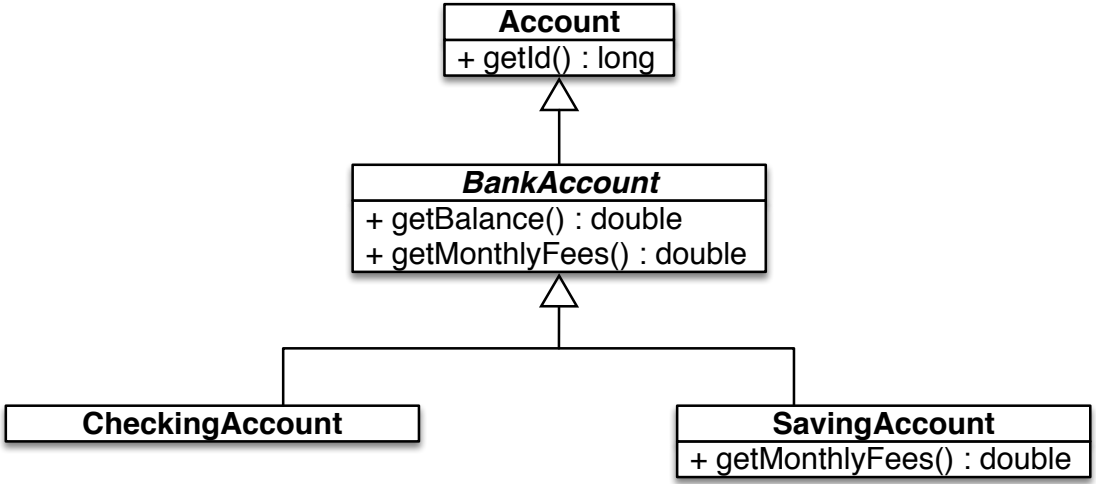
Qu'avons-nous obtenu ?

```java
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
```

```java
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** possèdent 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.getMonthlyFees();
    }
    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

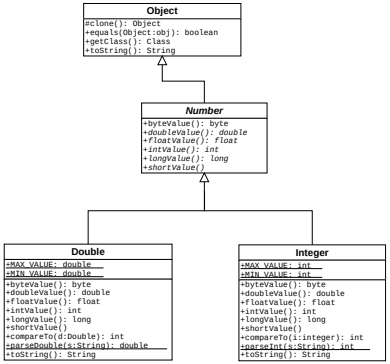
- Soit **S** (***source***) le type de l'objet désigné par une variable de type **T** (***target***).
- À moins que la méthode ne soit **static** ou **final**, la recherche de nom :
  1. se fait au moment de l'**exécution** du programme ;
  2. la recherche débute avec la classe **S**.
     - Si la méthode 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 association virtuelle (*virtual binding*)

Héritage et Java

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 :

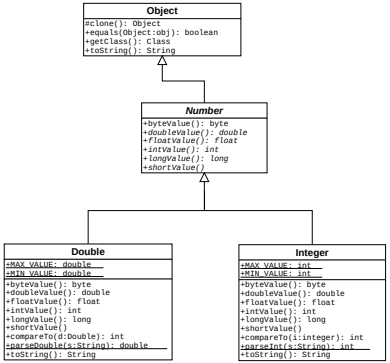
```
public class C {

}
```

est équivalente à celle-ci :

```
public class C extends Object {

}
```

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;
    }
}
```

```java
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 (this.getClass() != o.getClass()) { // <---
            result = false;
        } ...
        return result;
    }
}
```

```java
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 (this.getClass() != o.getClass()) {
            result = false;
        } else {
            Account other = (Account) o; // <---
            ...
        }
        return result;
    }
}
```

```java
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 (this.getClass() != o.getClass()) {
            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.name) ) {
                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 ?

toString()

- Puisque la class **Object** déclare une méthode **toString()**, tous les objets possèdent cette méthode.
- Soit la classe hérite d'une méthode **toString()** ou encore elle la redéfinit.
- Ainsi, l'énoncé **a.toString()** est toujours valide si **a** est une variable référence.

toString()

```
Account a;
a = new Account(101, "Marcel");
System.out.println(a);
System.out.println(a.toString());
```

System.out.println

```
public class PrintStream {

    // ...

    public void println(Object obj) {
        write(String.valueOf(obj));
    }
}
```

```
public class String {

    // ...

    public static String valueOf(Object obj) {
        return (obj == null) ? "null" : obj.toString();
    }
}
```

toString()

- Puisque la class **Object** déclare une méthode **toString()**, tous les objets possèdent cette méthode.
- Soit la classe hérite d'une méthode **toString()** ou encore elle la redéfinit.
- Ainsi, l'énoncé **a.toString()** est toujours valide si **a** est une variable référence.

```
public class Object {

    // ...

    public String toString() {
        return getClass().getName()+"@"+Integer.toHexString(hashCode());
    }
}
```

```java
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) { ... }
    // ...
}
```

toString()

```
Account a;
a = new Account(101, "Marcel");
System.out.println(a);
```

```
> java Test
Account@3fee733d
```

```java
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) { ... }
    // ...
    public String toString() {
        return "Account: id = " + id + ", name = " + name;
    }
}
```

toString()

```
Account a;
a = new Account(101, "Marcel");
System.out.println(a);
```

```
> java Test
Account: id = 101, name = Marcel
```

Exemple

```java
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
```

instanceof

- À l'occasion, on souhaite déterminer si une variable (polymorphique) désigne un objet d'une classe donnée, où l'une de ses sous-classes.
  - On utilise alors l'opérateur **instanceof** ou la méthode d'instance **isInstance**.
- Si par contre, on souhaite savoir si une variable (polymorphique) désigne un objet d'une certaine classe, mais pas l'une de ses sous-classes, utilisez alors **this.getClass() == other.getClass()**.

```java
public class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        Shape s = new Circle(100.0, 200.0, 10.0);

        shapes[0] = s;
        shapes[1] = null;
        shapes[2] = new Rectangle(50.0, 50.0, 10.0, 15.0);
        shapes[3] = new Circle();
        shapes[4] = new Rectangle();

        int count = 0;

        for (Shape shape : shapes) {
            if (shape instanceof Circle) {
                count++;
            }
        }

        System.out.println("There are " + count + " circles");
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        Shape s = new Circle(100.0, 200.0, 10.0);

        shapes[0] = s;
        shapes[1] = null;
        shapes[2] = new Rectangle(50.0, 50.0, 10.0, 15.0);
        shapes[3] = new Circle();
        shapes[4] = new Rectangle();

        int count = 0;

        for (Shape shape : shapes) {
            if (shape != null && shape.isInstanceof(s)) {
                count++;
            }
        }

        System.out.println("There are " + count + " circles");
    }
}
```

Implémentation à proscrire !

- Sur la page suivante, l'exemple utilise **getClass().getName().equals("Circle")**.
- Cette solution n'offre aucune **sécurité de type** («*type safety*»).
  - Si je fais une faute de frappe dans le nom de la classe comme paramètre à la méthode **equals**, il s'agit quand même d'une chaîne bien formée, elle sera compilée, mais le programme ne fonctionnera pas comme prévu.
    - Avec les deux premières approches, cette erreur est détectée au moment de la compilation.
  - Plus tard, si je change le nom de la classe («*refacton*») pour **Cercle** («circle» en français), avec les deux premières approches, le compilateur trouvera tous les cas où j'utilise "**ref instanceof Circle**", mais pas **getClass().getName().equals("Circle")**.

```java
public class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        Shape s = new Circle(100.0, 200.0, 10.0);

        shapes[0] = s;
        shapes[1] = null;
        shapes[2] = new Rectangle(50.0, 50.0, 10.0, 15.0);
        shapes[3] = new Circle();
        shapes[4] = new Rectangle();

        int count = 0;

        for (Shape shape : shapes) {
            if (shape.getClass().getName.equals("Circle")) {
                count++;
            }
        }

        System.out.println("There are " + count + " circles");
    }
}
```

getClass()

- Le contrat de la méthode **equals** requiert que la méthode soit symétrique. C'est-à-dire que **a.equals(b)** et **b.equals(a)** donne le même résultat.
- Si on utilisait **instanceof**, cette propriété pourrait ne pas être vérifiée dans le contexte d'une hiérarchie de classes où la méthode **equals** est redéfinie dans une sous-classe.
- Il est donc préférable d'utiliser **this.getClass() == other.getClass()**, tel qu'illustré sur la page suivante.
- `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html`

```java
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 (this.getClass() != o.getClass()) {
            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.name) ) {
                result = false;
            }
        }
        return result;
    }
}
```

Prologue

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**

Prochain module

- Interface utilisateur graphique **GUI**

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa