ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 23 janvier 2011

Résumé

– Héritage (partie 2)
  – Polymorphisme

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Circle

Complétons l'implémentation de la classe **Circle**.

Où crée-t-on l'implémentation de la méthode **area()** ?

Elle sera déclarée dans la classe **Shape** ou la classe **Circle** ?

Circle

```
public class Circle extends Shape {

    private double radius;

    public double getRadius() { return radius; }

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

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

Rectangle

De même, complétons l'implémentation de la classe **Rectangle**.

Où crée-t-on l'implémentation de la méthode **area()** ?

Elle sera déclarée dans la classe **Shape** ou la classe **Rectangle** ?

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double area() {
        return width * height;
    }

    public void scale(double factor) {
        width = width * factor;
        height = height * factor;
    }

}
```

Il ne faudrait pas avoir l'impression que l'héritage se limite aux classes qu'on définit soi-même. Au contraire, l'héritage est souvent utilisé afin de spécialiser les classes de la bibliothèque de 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
```

Polymorphisme

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

1. **Polymorphisme** ***ad hoc*** (polymorphisme des traitements, surcharge de nom) : un même nom de méthode est associé à des blocs d'énoncés différents
2. **Polymorphisme universel** (polymorphisme de données) : un identificateur (une variable référence) est lié à des données de types différents par une relation de sous-type

Surcharge (≪ overloading ≫)

Java nous permet la déclaration de plusieurs méthodes ayant le même nom, mais ayant des signatures différentes (la signature d'une méthode est constituée du nom de la méthode ainsi que de la liste des paramètres formels ; elle ne contient cependant pas le type de la valeur de retour).

On l'utilise souvent pour les constructeurs :

```
Shape() {
  x = 0.0;
  y = 0.0;
}
Shape( int x, int y ) {
  this.x = x;
  this.y = y;
}
```

⇒ On appelle parfois ceci le polymorphisme *ad hoc*.
(*ad hoc* expressément conçu en vue d'un usage particulier.)

Surcharge (suite)

Certains opérateurs, tel le ≪+≫, le sont.

Java utilise la signature de la méthode afin de déterminer la méthode à utiliser.

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

Surcharge (suite)

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

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

Polymorphisme (suite)

Pros : les méthodes ayant un comportement similaire ont un même nom.

Cons : on doit fournir une implémentation pour chaque comportement.

Le ≪ vrai ≫ polymorphisme : motivation 1

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

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();
}
```

Qu'est-ce qui est absurde ?

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 ?

Toutes les formes ont une méthode **getX()** !

```
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**" !

Variable 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 );
```

Méthode polymorphique : le ≪ vrai ≫ 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 ) ) { ... }
```

Variable polymorphique (suite)

```
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?
```

Variable polymorphique (suite)

```
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.

Variable polymorphique (suite)

```
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**.

Variable polymorphique (suite)

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

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

Ici, **s** est de type **Shape**, la méthode **getX()** est définie dans la classe **Shape**.

Variable polymorphique (suite)

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

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

C'est logique, **s** désigne un objet de la classe **Shape** ou de l'une de ses sous-classes. Ces objets auront les caractéristiques de la classe **Shape**.

Variable polymorphique (suite)

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

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

Le dernier énoncé n'est pas valide. Pourquoi ? La méthode **getRadius()** n'est pas définie dans la classe **Shape** (ou ses parents).

Variable polymorphique (suite)

1) Le type d'une variable référence définit l'ensemble des classes dont les objets peuvent être désignés par cette référence.

2) Le type type d'une variable référence définit l'ensemble des opérations valides (appels de méthode, accès à une variable, etc.).

Polymorphisme

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 {
   // ...
}
```

Le ≪ vrai ≫ polymorphisme : motivation 2

Problème : on souhaite définir une méthode pour comparer l'aire de deux formes géométriques.

Une solution absurde !

Un solution non souhaîtable consiste à créer plusieurs méthodes ; une méthode pour chaque paire possible :

(`Circle`, `Circle`), (`Circle`, `Rectangle`), (`Rectangle`, `Circle`) et (`Rectangle`, `Rectangle`).

- Autant d'implémentation que de paires de formes !
- Toutes les implémentations sont identiques !
- Lorsqu'une nouvelle catégorie de formes est définie (Triangle) de nouvelles méthodes **compareTo** doivent être créées !

Solution

Propositions ? Que pensez-vous de ceci ?

```
public class Shape {

  // ...

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

Solution

```
public class Shape {
  // ...

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

Ça ne compile pas ! Pourquoi ? Parce que la (super) classe **Shape** n'a pas de méthode **area()**.

Solution

Proposition ? Solution casse coup, ajoutons une méthode bidon, **area()**.

```
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;
  }
```

Solution

C'est trop dangereux ! Aucun mécanisme ne force la redéfinition de la méthode dans la sous-classe.

```
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;
  }
```

Solution : abstract

La solution idéale consiste à déclarer la méthode **area()** comme étant **abstraite**. Une méthode abstraite est déclarée à l'aide du mot clé **abstract**, d'une signature, mais sans corps.

```
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;
  }
```

Cette définition, hélas, ne compile pas !

Solution : 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;
  }
```

Imaginez qu'on puisse créer un objet à partir de cette classe, ces objets auraient une méthode **area()** mais aucun énoncé ne s'y rattache.

Solution : classe abstraite

```
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;
  }
```

Une **classe** ayant une méthode abstraite doit être **abstraite** ! On ne peut créer un objet à partir d'une classe abstraite. L'énoncé ≪ new Shape() ≫ causerait une erreur de compilation.

Classe abstraite

- Une classe ayant une méthode abstraite (déclarée dans cette classe ou héritée d'un parent) doit être abstraite ;
- On ne peut créer un objet à partir d'une classe abstraite ;
- On peut déclarer une classe abstraite sans y déclarer des méthodes abstraites. C'est une question de design. Voir Employee, SalariedEmployee, HourlyEmployee.

Solution : classe abstraite

Qu'avons-nous réussi ?

```
public class Triangle extends Shape {

}
```

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

La création d'une sous-classe concrète sans méthode **area()** est maintenant **impossible** !

Solution : classes et méthodes abstraites

La déclaration d'une méthode abstraite (**abstract method**) forces toutes sous-classes (concrètes) à fournier une implémentation.

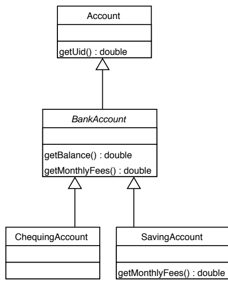
```
public abstract class Shape {
  // ...

  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;
  }

}
```

Late binding (a.k.a. dynamic binding, virtual binding)



Les classes **BankAccount** et **SavingAccount** contiennent toutes deux une définition de la méthode **getMonthlyFees**();

Par exemple, on retrouvera cette définition dans la classe **BankAccount**.

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

Et cette re-définition dans la classe **SavingAccount**.

```
public double getMonthlyFees() {
    double result;
    if ( getBalance() > 5000 ) {
       result = 0.0;
    } else {
       result = super.getMontlyFees();
    }
    return result;
}
```

```
Account a;
BankAccount b;
SavingAccount s;

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

b = s;
b.getMontlyFees();

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

Late binding (a.k.a. dynamic binding, virtual binding)

Soit S (source) de type de l'objet désigné par une variable référence de type T (target/destination).

À moins que la méthode ne soit static ou final, la recherche de la méthode i) se fait au moment de l'exécution du programme, ii) débute par la classe S : si la méthode si trouve elle est exécutée, sinon on regarde dans la superclasse immédiate, ce processus se poursuit jusqu'à ce que la méthode ait été trouvée.

Autrement, la sélection de la méthode se fait au moment de l'exécution du programme et regardant la classe de l'objet et non celle de la référence.