ITI 1521. Introduction à l'informatique II*

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

Version du 1er février 2012

Résumé

– Exemples de polymorphisme :
  – Object : toString ;
  – Structure de données ≪ générique ≫ : Pair ;
  – Object : equals.

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

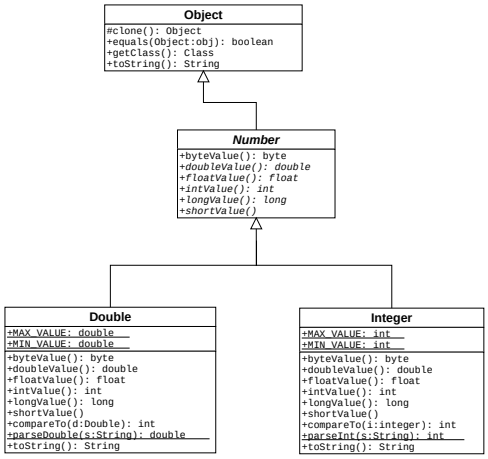
Object

Qu'en pensez-vous ?

```
Object o;
```

Une référence de type **Object** peut désigner tout objet dont la classe est une sous-classe de la classe **Object**, donc tout objet.

Qui a-t-il dans la classe **Object** ?



La ≪ magie ≫ derrière la méthode print

Utilisation :

```
System.out.print( 10 );

System.out.print( true );

System.out.print( 'c' );

System.out.print( "vote today" );

System.out.print( new Time( a, b, c ) );
```

La ≪ magie ≫ derrière la méthode print

```
System.out.print( ... );
```

D'abord, puisque les conventions d'écriture ont été respectées, nous savons qu'on utilise la méthode **print** de l'objet désigné par la variable de classe **out**, de la classe **System**.

La variable de classe **out** désigne un objet de la classe **PrintStream**.

La ≪ magie ≫ derrière la méthode print

Ensuite, la classe **PrintStream** nous présente un bon exemple de polymorphisme *ad hoc* (surcharge du nom **print**).

```
public static void print( boolean b ) {
  if ( b ) {  print( "true" );  }
  else { print( "false"); }
}
public static void print( char c ) {
  print( String.valueOf( c ) );
}
public static void print( int i ) {
  print( Interger.toString( i ) )
}
...
```

Une déclaration pour chaque type primitif.

La ≪ magie ≫ derrière la méthode print

Mais aussi, une méthode polymorphique pouvant traiter tout objet !

```
public static void println( Object obj ) {
  if ( obj == null ) {
       print( "null" );
  else {
       print( obj.toString() ); // <---
  }
}
public static void println( String s ) {
  ...
}
```

Qu'est-ce que cette méthode **toString()** ?

print

Qu'est-ce que cette méthode **toString()** ?

C'est une méthode définie dans la classe **Object** de sorte que **obj.toString()** est toujours valide si **obj** est une référence.

toString() est **héritée** ou **redéfinie**.

Considérez cet exemple.

```
Account a;
a = new Account( 1, "Marcel");
System.out.print( a );
```

Affiche,

```
Account@863399
```

print

Affiche,

```
Account@863399
```

La méthode **toString** de la classe **Object** est définie comme suit :

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

Hum . . .ça me semble pas très utile !

En effet, il est toujours préférable de redéfinir cette méthode.

print

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

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

```
Account a;
a = new Account( 1, "Marcel");
System.out.print( a );
```

Affiche :

```
id = 1; name = Marcel
```

Structures de données et polymorphisme

Problème : Vous devez concevoir une classe afin de sauvegarder deux objets, disons des objets de la classe **Time**.

Quelles seront les variables d'instance ? et les méthodes ?

Time Pair

```
public class Pair {
  private Time first;
  private Time second;
  public Pair( Time first, Time second ) {
    this.first = first;
    this.second = second;
  }
  public Time getFirst( ) {
      return first;
  }
  public Time getSecond( ) {
      return second;
  }
}
```

⇒ `new Pair( new Time( 14, 30 ), new Time( 16, 0 ) );`

Shape Pair

```
public class Pair {
  private Shape first;
  private Shape second;
  public Pair( Shape first, Shape second ) {
    this.first = first;
    this.second = second;
  }
  public Shape getFirst( ) {
      return first;
  }
  public Shape getSecond( ) {
      return second;
  }
}
```

⇒ `new Pair( new Circle( 0, 0, 0 ), new Rectangle( 1, 1, 1, 1 );`

Structures de données ≪ génériques ≫

Concevoir une classe afin de sauvegarder une paire d'objets.

Attributs ?

Pair

```
public class Pair {
  private ______ first;
  private ______ second;
  public Pair( ______ first, ______ second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( ______ first, ______ second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( Object first, Object second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {  // type de la valeur de retour?
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( Object first, Object second ) {
    this.first = first;
    this.second = second;
  }
  public Object getFirst( ) {
      return first;
  }
  public Object getSecond( ) {
      return second;
  }
}
```

Pair

Utilisation :

```
Pair p;

String a;
a = "King";
String b;
b = "Edward";

p = new Pair( a, b );

a = p.getFirst();
b = p.getFirst();
```

Problèmes ?

Pair

```
Pair p;

String a;
a = "King~≫;
String b;
b = "Edward";

p = new Pair( a, b );

a = (String) p.getFirst();
b = (String) p.getFirst();
```

La classe **Object** est plus générale que la classe **String** ! Il faut donc forcer le compilateur à faire cette affectation.

Structures de données génériques avant 1.5

Dans les versions précédentes de Java, celles avant 1.5, on utilisait des variables de type **Object** afin de créer des structures de données génériques.

L'**avantage** c'est que ces structures de données servent à sauvegarder tout objet pouvant être créé en Java.

Le **désavantage** c'est que le type de la valeur de retour des méthodes d'accès est **Object**, il fallait donc toujours forcer l'affectation.

On perd la vérification statique des types.

Structures de données génériques avant 1.5

```
Time t;
Pair p;
...
t = (Time) p.getFirst();
```

C'est dangereux, on dégage le compilateur de l'une de ses responsabilités très importante, la vérification des types, ainsi, on court le risque de causer des erreurs d'exécution ! ! !

Structures de données génériques et 1.5

Les types génériques (**Generics**) introduisent la définition de paramètres de classe.

```
public class Pair<T> {
   ...
}
```

Cette déclaration (ce paramètre) représente le type des objets qui seront sauvegardés dans cette **Pair**.

Structures de données génériques et 1.5

La valeur de ce paramètre est spécifiée au moment de la déclaration de de type.

```
Pair<String> name;
Pair<Integer> range;
```

ainsi que pour la création des objets.

```
name = new Pair<String>( "Hilary", "Clinton" );
```

```
Integer min;
min = new Integer( 0 );
Integer max;
max = new Integer( 100 );

range = new Pair<Integer>( min, max );
```

Structures de données génériques et 1.5

Qu'avons nous obtenu ?

Nous n'avons plus à forcer le type de la valeur de retour.

```
Pair<Integer> range;
range = new Pair<Integer>( new Integer( 0 ), new Integer( 100 ) );
Integer i;
i = range.getFirst();
```

Structures de données génériques et 1.5

Maintenant, une paire de type **Pair**<**Integer**> ne peut contenir que des objets de la classe **Integer**.

L'énoncé,

```
range.setFirst( "Voila" );
```

causera l'erreur de compilation suivante,

```
Test.java:20: setFirst(java.lang.Integer)
in Pair<java.lang.Integer> cannot be applied to (java.lang.String)
        range.setFirst( "Voila" );
              ^
1 error
```

Structures de données génériques et 1.5

De même, une référence de type **Pair**<**Integer**> ne peut désigner un objet de type **Pair**<**String**>.

L'énoncé,

```
range = new Pair<String>( "Hilary", "Clinton" );
```

causera l'erreur de compilation suivante,

```
Test.java:22: incompatible types
found   : Pair<java.lang.String>
required: Pair<java.lang.Integer>
        range = new Pair<String>( "Hilary", "Clinton" );
                ^
1 error
```

Structures de données génériques et 1.5

C'est le meilleur des deux mondes.

La structure de données **Pair** a une implémentation unique bien qu'elle permette la création de paires servant à sauvegarder des objets génériques.

Tout ça sans compromettre l'intégrité des types (détections de plusieurs types d'erreurs au moment de la compilation).

Type générique et type paramétré

≪ A **generic type** is a type with formal type parameters. A **parameterized type** is an instantiation of a generic type with actual type arguments. ≫

Un **type générique** est un type ayant paramètre formel de type. Un **type paramétré** est l'instanciation d'un type générique à l'aide d'un type actuel de paramètre.

Type générique et type paramétré

Définir un type générique.

Un **type générique** est un **type référence** ayant un ou plusieurs paramètres de type.

Un **type générique**, c'est une **classe** ayant un ou plusieurs paramètres de type.

Définir un type générique

```
public class Pair<X,Y> {

    private X first;
    private Y second;

    public Pair( X a, Y b ) {
        first = a;
        second = b;
    }

    public X getFirst() { return first; }

    public Y getSecond() { return second; }

    public void setFirst( X arg ) { first = arg; }

    public void setSecond( Y arg ) { second = arg; }
}
```

Créer un type paramétré

Lorsqu'on utilise un type générique, ici **Pair**, un **argument de type** est donné pour chaque **paramètre de type**.

```
public class Test {
    public static void main( String[] args ) {

        Pair<String, Integer> p;

        String attribut;
        attribut = new String( "height" );

        Integer value;
        value = new Integer( 100 );

        p = new Pair<String, Integer>( attribut, value );

    }
}
```

Question : quels énoncés sont valides ?

```
Pair<String,Integer> p;


p = new Pair<String,Integer>();


p.setFirst( "session" );


p.setSecond( 12345 );
```

Est-ce que ces énoncés sont valides ?

```
public class T1 {
    public static void main( String[] args ) {

        Pair<String,Integer> p;

        p = new Pair<Integer,String>();

    }
}


// > javac T1.java
// T1.java:6: incompatible types
// found   : Pair<java.lang.Integer,java.lang.String>
// required: Pair<java.lang.String,java.lang.Integer>
//      p = new Pair<Integer,String>();
//              ^
// 1 error
```

Est-ce que ces énoncés sont valides

```
public class T2 {
    public static void main( String[] args ) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>();
        p.setFirst( 12345 );
        p.setSecond( "session" );
    }
}


T2.java:5: setFirst(java.lang.String) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (int)
       p.setFirst( 12345 );
         ^
T2.java:6: setSecond(java.lang.Integer) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (java.lang.String)
       p.setSecond( "session" );
         ^
2 errors
```

Est-ce que ces énoncés sont valides ?

```
public class T3 {
    public static void main( String[] args ) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>( "session", 12345 );
        Integer s = p.getFirst();
    }
}


// > javac T3.java
// T3.java:8: incompatible types
// found   : java.lang.String
// required: java.lang.Integer
//      Integer s = p.getFirst();
//                              ^
// 1 error
```

Types génériques

Les types génériques sont de nouveaux outils pour nous aider à détecter des erreurs dès la compilation !

C'est comme si quelqu'un regardait par-dessus votre épaule !

Implémentation

Sans aller dans les détails, les génériques sont implémentés à l'aide d'une technique qui s'appelle **effacement de type** (type erasure) , en conséquence, les paramètres de type n'existent qu'au moment de la compilation.

A.java

```
public class A {

    public static void main( String[] args ) {

        Pair<String, Integer> p;
        p = new Pair<String, Integer>( "Orange", new Integer( 1 ) );

        String s;
        s = p.getFirst();

    }
}
```

B.java

```
public class B {

    public static void main( String[] args ) {

        Pair p;
        p = new Pair( "Orange", new Integer( 1 ) );

        String s;
        s = (String) p.getFirst();

    }
}
```

Effacement de type

A.java :

```
Pair<String> p;
p = new Pair<String, Integer>( "Orange", 1 );

String s;
s = p.getFirst();
```

B.java :

```
Pair p;
p = new Pair( "Orange", 1 );

String s;
s = (String) p.getFirst();
```

Effacement de type

A.java et **B.java** produiront le même code-octet. Les incrédules pourront le vérifier à l'aide du déssassembleur de classe (**javap -c**).

≪ Generics implicitly perform the same cast that is explicitly performed without generics. ≫

≪ Cast-iron guarantee : the implicit casts added by the compilation of generics never fail. ≫

[NW07, page 5]

Effacement de type

– Simplicité et compatibilité avec les versions antérieures ;
– Économique : une seule copie du code-octet pour plusieurs usages.

Classes enveloppantes

Avant 1.5, il fallait soit même envelopper les objets dans un objet (Integer, Double, . . .).

```
Pair p;
p = new Pair( 0, 100 );
```

Classes enveloppantes

```
Pair p;
p = new Pair( new Integer( 0 ), new Integer( 100 ) );
```

Java 1.5

C'est maintenant automagique (auto boxing/unboxing).

```
Pair<Integer> range;

range = new Pair<Integer>( 0, 10 );

int i;

i = range.getSecond();
```

Object/equals

La classe **Object** définit une méthode **equals**.

Ainsi, pour toutes variables références **a** et **b**, on peut toujours écrire,

```
if ( a.equals( b ) ) { ... }
```

C'est vrai pour les classes prédéfinies, telles que **String** et **Integer**, mais aussi pour toutes les classes que l'on définit soit même.

Toutes les classes sont des sous-classes de la classe **Object** !

Comparaison

Soient deux références, **a** et **b**, et leurs objets désignés.

- La **comparaison par identité** vérifie seulement que **a** et **b** désignent ou non le même objet (i.e. `a == b`) ;
- Souvent, on souhaite déterminer si le **contenu** des objets désignés par **a** et **b** est le même, on utilise alors la méthode **equals** ; **comparaison logique**.

equals

Peut-on vraiment définir une méthode **equals** qui fonctionnerait pour tout type d'objet ?

La méthode **equals** de la classe **Object** est définie comme suit :

```
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

```
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 message sera affiché ?

≪ a and b are not equals ≫

Solution : redéfinir la méthode **equals** dans la classe **Account**.

Recette de la méthode equals (1/4)

Utiliser == afin de s'assurer que le paramètre **o** n'est pas **null**.

```
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;
    }
}
```

Recette de la méthode equals (2/4)

Utiliser **instanceof** afin de s'assurer que l'objet désigné est de la bonne classe.

```
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;
    }
}
```

Recette de la méthode equals (3/4)

Puisque le paramètre **o** est non **null** et désigne un objet de la classe **Account**, utilisons une variable de type **Account** afin de le désigner.

```
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;
    }
}
```

Recette de la méthode equals (4/4)

Il faut comparer les attributs un à un. Utilisez == pour les types primitifs et **equals** pour les références (attention aux références **null**).

```
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.name ) ) {
                result = false;
            }
        }
        return result;
    }
}
```

instanceof

```
Shape s;
...
if ( s instanceof Circle ) {
   Circle c;
   c = (Circle) s;
   double radius = c.getRadius();
}
```

Toujours précédé d'une vérification de type.

```
long l;
...
if ( ( Integer.MIN_VALUE <= l ) && ( l <= Integer.MAX_VALUE ) ) {
   int i;
   i = (int) l;
   ...
}
```

instanceof

Étant donné l'expression,

```
s instanceof T
```

L'opérateur **instanceof** retourne **false** si la valeur de **s** est **null**, sinon, l'opérateur retourne **true** si la classe de l'objet désigné par **s**, au moment de l'exécution, est compatible avec le type **T** ; soit il s'agit de la même classe ou encore, la classe de l'objet désigné par **s** est une sous-classe de la classe **T**.

instanceof

```
if ( s instanceof Circle ) {
   double radius = ( (Circle) s ).getRadius();
}
```

Références

[NW07] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly, 2007.