Objet prédéfinis

Rémi Forax
forax@univ-mlv.fr

Plan

- java.lang.Object
- Les tableaux
- Chaîne de caractères
- Les wrappers

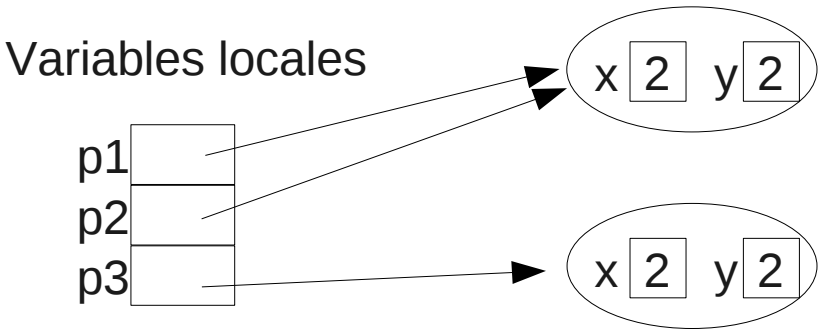
La classe Objet

- Classe mère de toutes les classes.
- Possède des méthodes de base qu'il est possible de redéfinir :
  - **toString()**
  - **equals() & hashCode()**
  - **getClass()**
  - **clone()**
  - **finalize()**

Tests d'égalité

- Les opérateurs de comparaison **==** et **!=** tests les valeurs des variables :
  - Pour les types primitifs, on test leurs valeurs
  - Pour les types objets, on test les valeurs leurs références

```
Point p1;
p1=new Point(2,2);
Point p2;
p2=p1
Point p3;
p3=new Point(2,2);
p1==p2; // true
p1==p3; // false
p2==p3; // false
```



La méthode equals()

- Il existe déja une méthode equals(Object) dans Object
- Mais son implantation test les références

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);

p1==p3;          // false
p1.equals(p3); // false
```

- Pour comparer structurellement deux objet, il faut changer (on dit **redéfinir**) le code de la méthode equals()

Redéfinir equals()

- Pourquoi equals ?
  Car elle sert à cela.
- La plupart des classes de l'API redéfinissent la méthode equals

```
public class Point {
  private final int x,y;
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public boolean equals(Point p) {
    return x==p.x && y==p.y;
  }
}
```

CELA NE MARCHE PAS !!!

Redéfinir equals()

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);
p1==p3;          // false
p1.equals(p3); // true

ArrayList points=new ArrayList();
points.add(p1);
points.contains(p3); // false
// pourtant contains utilise Object.equals(Object) ??
```

- La VM ne fait pas la liaison entre Object.equals(Object) et Point.equals(Point)

Redéfinir equals()

- Ce n'est pas le même equals(), car il n'y a pas eut de redéfinition mais une autre définition (on dit **surcharge**)

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);
p1.equals(p3); // true : Point.equals(Point)

Object o1=p1;
Object o3=p3;
o1.equals(o3); // false : Object.equals(Object)
```

- Point possède deux méthodes equals (equals(Object) et equals(Point))

Redéfinir equals()

- Il faut définir equals() dans Point de telle façon qu'elle remplace equals de Object
- Pour cela equals doit avoir la méme signature que equals(Object) de Object

```
public class Point {
  private final int x,y;
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public boolean equals(Object o) {
    return x==p.x && y==p.y; // ce code ne marche plus
  }
}
```

Utiliser @Override

- @Override est une annotation qui demande au compilateur de vérifier que l'on redéfinie bien une méthode

```
public class Point {
  private final int x,y;
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  @Override public boolean equals(Point p) {
    ...
  } // the method equals(Point p) in Point must override
    // a superclass method
}
```

Redéfinir equals

- On demande dynamiquement à voir une référence à Object comme à Point
  (car on le sait que c'est un Point)

```
public class Point {
  ...
  @Override public boolean equals(Object o) {
    Point p=(Point)o; // ClassCastException si pas un Point
    return x==p.x && y==p.y;
  }
}
```

- Mais equals doit renvoyer false si o n'est pas un Point et pas lever une CCE

Redéfinir equals

- On utilise instanceof qui renvoie vrai si une référence est d'un type particulier

```
public class Point {
  ...
  @Override public boolean equals(Object o) {
    if (!(o instanceof Point)) // marche aussi avec null
      return false;
    Point p=(Point)o;
    return x==p.x && y==p.y;
  }
}
```

- **null instanceof** WhatYouWant renvoie false

int hashCode()

- Renvoie un entier qui peut être utilisé comme valeur de hachage de l'objet
- Permet au objet d'être utiliser dans les tables de hachage

```
public class Point {
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  @Override public int hashCode() {
    return x ^ Integer.rotateLeft(y,16);
  }
  @Override public boolean equals(Object o) {
    ...
}
```

hashcode et equals

- Tout objet redéfinissant **equals** doit redéfinit **hashcode** si l'objet doit être utilisé dans les **Collection** (donc tout le temps)

- **equals** et **hashcode** doivent vérifier
  - **equals** est symétrique, transitive, réflexive
  - x.equals(y) implique
    x.hashcode() == y.hashcode()

hashcode et equals

- Les valeurs de **hashcode** doivent de préférence être différentes pour les objets du programme
- **hashcode** doit être rapide à calculer (éventuellement précalculée).

String toString()

- Affiche un objet sous-forme textuelle.
- Cette méthode **doit** être utilisée que pour le deboggage

```
public class MyInteger {
  public MyInteger(int value) {
    this.value=value;
  }
  public String toString() {
    return Integer.toString(value); // ou ""+value;
  }
  private final int value;
  public static void main(String[] args) {
    MyInteger myi=new MyInteger(3);
    System.out.println(myi); // appel toString()
  }
}
```

Class<?> getClass()

- permet d'obtenir un objet Class représentant la classe d'un objet particulier

```
String s="toto";
Object o="tutu";
s.getClass()==o.getClass(); // true
```

- Cette méthode est **final**
- Il existe une règle spéciale du compilateur indiquant le type de retour de getClass().
(cf cours Reflection & Types Paramétrés)

Class<?> getClass()

- Permet d'obtenir un objet Class représentant la classe d'un objet particulier
- Un objet Class est un objet qui correspond à la classe de l'objet à l'exécution

```
String s="toto";
Object o="tutu";
s.getClass()==o.getClass(); // true
```

- Cette méthode est **final**
- Il existe une règle spéciale du compilateur indiquant le type de retour de getClass().
  (cf cours Reflection & Types Paramétrés)

.class, getClass() et instanceof

- **instanceof** test aussi les sous-classes

```
String s="toto";
Class<?> StringClass=String.class;
Class<?> CharSeqClass=CharSequence.class;

s.getClass()==StringClass;  // true
s.getClass()==CharSeqClass; // false

s instanceof String;        // true
s instanceof CharSequence;  // true
```

- CharSequence est une superclass de String
- String**.class** correspond à la classe de String

Clonage

- Permet de dupliquer un objet
- Mécanisme pas super simple à comprendre :
  - clone() a une visibilité **protected**
  - clone() peut lever une exception **CloneNotSupportedException**
  - **Object.clone()** fait par défaut une copie de surface si l'objet implante l'interface **Cloneable**
  - L'interface **Cloneable** ne **définie pas** la méthode clone()

Clonage (2)

- Cloner un objet sans champs contenant un objet mutable

```
public class MyInteger implements Cloneable { // nécessaire
  public MyInteger(int value) {                // pour Object.clone()
    this.value=value;
  }
  public @Override MyInteger clone() {
    //return new MyInteger(value);        // Mal si héritage
    return (MyInteger)super.clone();    // shallow copy
  }                                     // mais cast MyInteger
  private final int value;
  public static void main(String[] args) {
    MyInteger i=new MyInteger(3);
    MyInteger j=i.clone();
    System.out.println(i==j);         // false
    System.out.println(i.equals(j)); // true
  }
}
```

Clonage (3)

- Cloner un objet avec des champs contenant des objets mutables

```
public class MyHolder implements Cloneable { // nécessaire
  public MyHolder(Mutable mutable) {         // pour Object.clone()
    this.mutable=mutable;
  }
  public @Override MyHolder clone() {
    MyHolder holder=(MyHolder)super.clone();   // shallow copy
    holder.mutable=this.mutable.clone();     // clone le mutable
    return holder;
  }
  private final Mutable mutable;
}
```

- On appel **clone()** sur l'objet mutable

void finalize()

- Méthode testamentaire qui est appelé juste avant que l'objet soit réclamé par le GC

```
public class FinalizedObject {
  protected @Override void finalize() {
    System.out.println("ahh, je meurs");
  }

  public static void main(String[] args) {
    FinalizedObject o=new FinalizedObject();
    o=null;
    System.gc(); // affiche ahh, je meurs
  }
}
```

- Cette méthode est **protected** et peut lever un **Throwable**

wait()/notify()

- **wait()** permet d'attendre sur le verrou associé à l'objet.
- **notify()** permet de libérer quelqu'un en attente sur le verrou associé à l'objet

- Pour plus d'infos, voir cours sur la concurrence

Les tableaux

- Il existe deux types de tableaux :
  - Les tableaux d'objets, héritant de Object[],
    ils contiennent des référence sur des objets
  - Les tableaux de types primitifs, héritant de Object
    ils contiennent des types primitifs

- Tous les tableaux implantent les interfaces **Cloneable** et **Serializable**

Initialisation et accès

- Syntaxes d'initialisation :
  new type[taille]
  new type[]{valeur$_1$,valeur$_2$,...,valeur$_{n-1}$}

```
double[] values=new double[5];
int[] array=new int[]{2,3,4,5};
```

- [] permet d'accèder en lecture/écriture

```
values[0]=values[1]+2.0;
values[2]=array[3];
```

- L'accès est protégé (pas de buffer overflow) et les tableaux sont tous mutables

Longueur et boucle

- Les tableaux sont utilisables dans une boucle **for-each**

```
long[] array=new int[]{2,3,4,5};
for(long l:array)
  System.out.print(l);
```

- **.length** sur un tableau permet d'obtenir sa taille

```
long[] array=new int[]{2,3,4,5};
for(int i=0;i<array.length;i++)
  System.out.print(array[i]);
```

CharSequence

- Représente une suite de caractère :
  - char **charAt**(int index)
    retourne le n-ième caractère
  - int **length**()
    retourne la taille de la chaîne
  - CharSequence **subSequence**(int start, int end)
    retourne une sous-chaîne
  - String **toString**()
    transforme en chaîne immutable

Les chaînes de caractères

- L'interface **CharSequence** est implantée par les 4 classes
  - **String**, chaîne de caractère immutable
  - **StringBuilder**, buffer de caractère mutable et auto-expensif
  - **StringBuffer**, pareil que StringBuilder mais synchronizé
  - **CharBuffer**, buffer de caractères mutable de taille fixe qui peut être alloué par malloc
    (pratique pour read/write avec le système)

String

- Chaîne de caractère **immutable**
- Initialisation dans le langage
- Concaténation avec +

```
public class StringExample {
  public static void main(String[] args) {
    String t="toto";
    String s=args[0]+t+args.length;
    for(int i=0;i<s.length();i++)
      System.out.println(s.charAt(i));
  }
}
```

- La plus utilisée car immutable

Les méthodes de String

- Méthodes habituellement utilisée
  - toUpperCase()/toLowerCase()
  - equals()/equalsIgnoreCase()
  - compareTo()/compareToIgnoreCase()
  - startsWith()/endsWith()
  - indexOf()/lastIndexOf()
  - matches(regex)/split(regex)
  - trim()/split()
  - format() [équivalent de sprinft]

String et intern()

- Par défaut, la VM utilise un même objet pour la même chaine de caractères littérale

```
public class InternStringExample {
  public static void main(String[] args) {
    String s="toto";
    System.out.println(s=="toto"); // true
    String t=new String(s);
    System.out.println(t=="toto"); // false
    String u=t.intern();
    System.out.println(u=="toto"); // true
  }
}
```

- La méthode **intern()** permet de récupérer une instance de String partagée

StringBuilder

- Equivalent mutable de **String**, possède un buffer de caractère qui s'aggrandit tous seul
- Est utilisé pour créer une chaîne de caractère qui sera après rendu immutable

```
public class StringBuilderExample {
  public static void main(String[] args) {
    StringBuilder builder=new StringBuilder();
    for(String s:args)
      builder.append(s).append(" ").append(s.length());
    String result=builder.toString();
    System.out.println(result);
  }
}
```

- Les méthodes **append()** sont chaînables

La concaténation

- Par défaut, le compilateur utilise un StringBuilder si l'on fait des + sur des String

```
public class ConcatExample {
  public String toString() {
    return "toto"+value;
  }
  private final int value;
}
```

compile ↓

bytecode

décompile →

```
public class ConcatExample {
  public String toString() {
    return new StringBuilder().append("toto").
      append(value).toString();
  }
  private final int value;
}
```

- Dans quels cas utiliser un **StringBuilder** alors ?

La concaténation (2)

- Lorsqu'il y a une boucle

```
public class StringBuilderExample {
  public static void main(String[] args) {
    String result="";
    for(String s:args)
      result+=s+" "+s.length();
    System.out.println(result);
  }
}
```

compile

bytecode

décompile

```
public class StringBuilderExample {
  public static void main(String[] args) {
    String result="";
    for(String s:args) {
      result=new StringBuilder(result).append(s).
        append(" ").append(s.length()).toString();
    }
    System.out.println(result);
  }
}
```

- Le code générer alloue un StringBuilder par tour de boucle (pas super optimisé)

StringBuffer

- Possède la même implantation que StringBuilder mais est synchronizé par défaut
- Utilisé avant que StringBuilder existe, c-a-d avant la version 5.0 (1.5).

- Ne doit plus être utilisé sauf par compatibilité avec d'anciennes bibliothèques

java.nio.CharBuffer

- Correspond à un tableau de caractère de taille fixe ainsi qu'à deux pointeurs sur ce tableau
- Très pratique et rapide lors des opérations de lecture et écriture

- Cf cours sur les entrées/sorties

Type primitif et Objet

- Il existe des classes wrapper (enveloppes) qui permettent de voir un type primitif comme un objet

```
List list=...
Holder holder=new Holder();
list.add(holder);

int i=3;
list.add(new Integer(i));
```

```
public class List {
  public void add(Object o) {
    ...
  }
}
```

- Cela permet de réutiliser un code marchant pour des objets avec des types primitifs

Les wrappers

- Un wrapper est juste un objet stockant un type primitif

```
public class Integer {
  private final int value;
  public Integer(int value) {
    this.value=value;
  }
  ...
}
```

- boolean -> Boolean, byte -> Byte
  short -> Short, char -> Character,
  int -> Integer, long -> Long,
  float -> Float, double -> Double

Conversion type primitif/wrapper

- Les wrappers possèdent deux méthodes qui permettent de faire la conversion entre un type primitif et le wrapper
- ***Wrapper*.valueOf(*primitif*)**

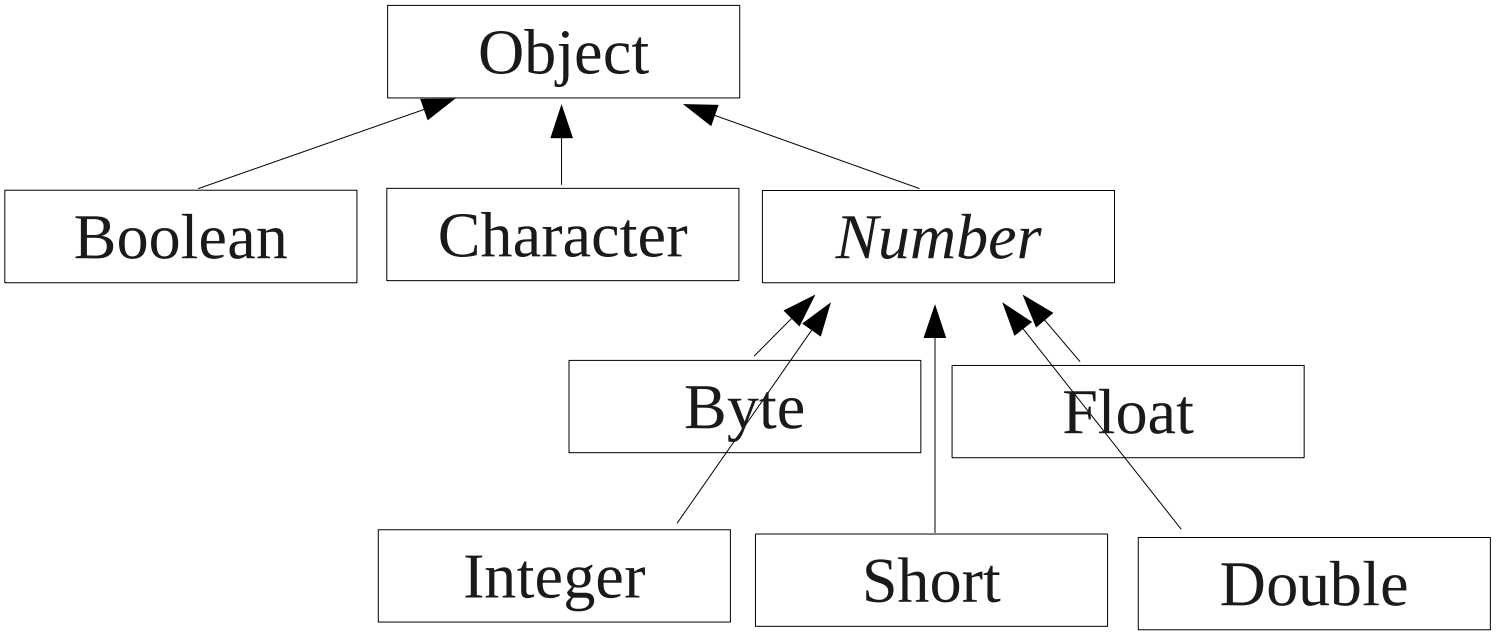
```
int value=3;
Integer i=Integer.valueOf(value);
```

- ***wrapper*.*primitif*Value**

```
Integer value=...
int i=value.intValue();
```

Hiérarchie des wrappers

- Les relations de sous-typage entre les wrappers ne sont pas les mêmes que celles des types primitifs associés



Auto boxing/unboxing

- En fait, la conversion est automatique
- Auto-boxing :

```
int value=3;
Integer i=value;

Object o=3.0 // passage en Double
```

- Auto-unboxing :

```
Integer value=new Integer(3);

int i=value;
double d=value;  // passage en int puis promotion en double
```

Auto boxing/unboxing (2)

- Les conversions de boxing/unboxing sont faites **avant** les conversions classiques mais pas **après**

```
int value=3;
Integer bValue=value;

double d=bValue; // Integer -> int puis promotion en double

Double wd=value; // erreur pas de convertion Integer -> Double
                 // et pas possible de faire int -> double -> Double
Double wd=(double)value;
```

Appel de méthode

- Auto-boxing, unboxing s'applique aussi lors de l'appel de méthode
  - Sur les arguments
  - Sur la valeur de retour

```
List list=...
list.add(3);

int value=list.get(0);
```

```
public class List {
  public void add(Object o) {
    ...
  }
  public Object get(int index) {
    ...
  }
}
```

Opérateurs et auto[un]boxing

- De même que pour les méthodes, l'auto[un]boxing s'applique avec les opérateurs numériques <,<=,>=,>, &, |, ^, ~

```
Integer i=235; // boxing
if (i>4)       // unboxing
  System.out.println(i);

Integer j=235; // boxing
if (i==j)      // false
               // compare les références !!!
  System.out.println("arg !");
```

- Il ne s'applique pas avec == et !=.

Boxing et cache

- Pour faire le boxing, on utilise la méthode *Wrapper*.valueOf() et non new *Wrapper*() pour permettre un partage des *wrappers*

- Pour les types **byte**, **short**, **char**, **int** et **long**, les valeurs entre **-128 et 127** sont mises dans un **cache**.

Boxing et cache (2)

- Donc, les *wrappers* possèderont la **même référence**

```
Long a=17;
Long b=17;

Integer i=235;
Integer j=235;

System.out.println(a==b); // true
System.out.println(i==j); // false

b=new Long(17);
System.out.println(a==b); // false
```

- **Eviter les ==, != sur les *wrappers***

Unboxing et **null**

- Si la référence sur un wrapper est **null**, un unboxing provoque une exception **NullPointerException**

```
Integer i=null;
if (randomBoolean())
  i=3;

int value=i; // risque de NPE
```

- Cette exception est **très dure** à trouver car on ne la voie pas dans le code source

Boxing et tableau

- Il n'y a pas de boxing/unboxing entre un tableau de type primitif et un tableau de wrapper

```
public class ArrayBoxing {
  public void sort(Object[] array) {
    ...
  }
  public static void main(String[] args) {
    int[] array=new int[]{2,3};
    sort(array);// sort(java.lang.Object[])
                // cannot be applied to (int[])
  }
}
```