Object builtin
Les objets de base de Java

Plan

- java.lang.Object
- java.lang.Class
- Les tableaux
- Chaînes de caractères
- Les wrappers

La classe Objet

- Super classe de toutes les classes.
- Possède des méthodes de base qu'il est possible de redéfinir :
  - **equals(), hashCode() & toString()**
    cf cours sur Object
  - **clone()**
  - **finalize()**
  - **getClass()**

Clonage

De temps en temps, il est nécessaire de dupliquer un objet pour en créer un nouveau ayant les mêmes attributs

Une solution assez simple consiste à ajouter un constructeur dit constructeur par copie qui prend une instance du même type que la classe et qui fait la copie des champs

- solution du C++

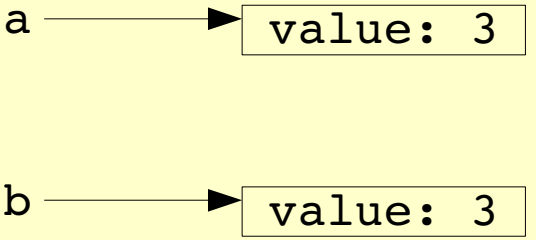
Constructeur par Copie

```
public class Int  {
  private final int value;

  public Int(int value) {
    this.value = value;
  }
  public Int(Int anInt) {
    this(anInt.value);
  }
  public boolean equals(Object o) {
    if (!(o instanceof Int)) {
      return false;
    }
    Int anInt = (Int)o;
    return value == anInt.value;
  }

  public static void main(String[] args) {
    Int a = new Int(3);
    Int b = new Int(a);
    System.out.println(a == b);        // false
    System.out.println(a.equals(b));   // true
  }
}
```

Problème:
ce code ne fait
pas ce qu'on veut !

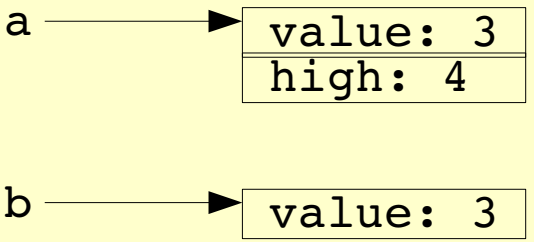


Quel est le problème ?

```
public class LongInt extends Int {
  private final int high;

  public LongInt(int low, int high) {
    super(low);
    this.high = high;
  }
  public boolean equals(Object o) {
    if (!(o instanceof LongInt)) {
      return false;
    }
    LongInt aLongInt = (LongInt)o;
    return high == aLongInt.high &&
           super.equals(aLongInt);
  }

  public static void main(String[] args) {
    Int a = new LongInt(3, 4);
    Int b = new Int(a);
    System.out.println(a == b);        // false
    System.out.println(a.equals(b));   // false
  }
}
```



En effet: le constructeur de Int n’a pas cloné l'objet LongInt

Comment résoudre le problème ?

On peut essayer d'utiliser la redéfinition !

```
public class Int  {
  private final int value;

  public Int(int value) {
    this.value = value;
  }



  public Int duplicate() {
    return new Int(value);
  }
}
```

```
public class LongInt extends Int {
  private final int high;

  public LongInt(int low, int high) {
    super(low);
    this.high = high;
  }

  @Override
  public LongInt duplicate() {
    return new LongInt(value, high);
  }
}
```

pas le même type de retour

Ce code ne marche pas !

Ce code ne compile pas et ne respecte pas la séparation des fonctionalités !

Comment faire alors ?

Utiliser Object.clone()

```
public class Int {
  private final int value;

  public Int(int value) {
    this.value = value;
  }


  public Int clone() {
    Int anInt = (Int)super.clone();
    anInt.value = value;
    return anInt;
  }
}
```

```
public class LongInt extends Int {
   private final int high;

   public LongInt(int high, int low) {
     super(low);
     this.high = high;
   }


   public LongInt clone() {
     LongInt aLongInt =
       (LongInt)super.clone();
     aLongInt.high = high;
     return aLongInt;
   }
 }
```

ce code marche presque :(

Clonage

Mécanisme pas super simple à comprendre :

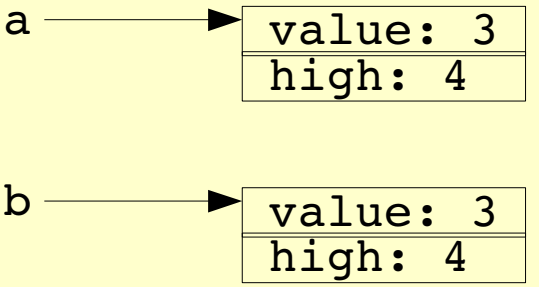
- Object.clone() a une visibilité **protected**
- clone() lève **CloneNotSupportedException** si la classe n'implante pas **Cloneable**
- L'interface Cloneable est vide et donc **ne définit pas** la méthode **clone()**
- **Object.clone()** fait par défaut une copie de surface

Exemple

```
public class Int implements Cloneable {
  private final int value;

  public Int(int value) {
    this.value = value;
  }
  @Override
  public Int clone() {
    try {
      return (Int)super.clone();   // shallow copy
    } catch(CloneNotSupportedException e) {   // ne devrait pas arriver
      throw new AssertionError(e);
    }
  }

  public static void main(String[] args) {
    Int a = new LongInt(3, 4);
    Int b = a.clone();
    System.out.println(a == b);         // false
    System.out.println(a.equals(b));    // true
  }
}
```



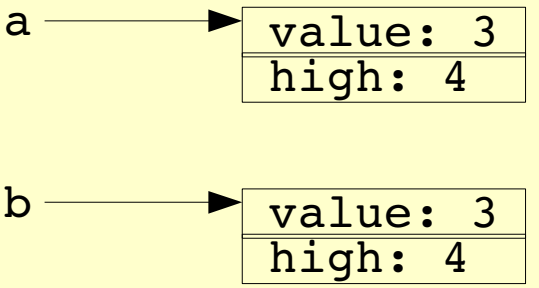
Marche enfin !

Exemple

```
public class LongInt extends Int {
  private final int value;

  public Int(int value) {
    this.value = value;
  }
  @Override
  public LongInt clone() {
    return (LongInt) super.clone();   // shallow copy
  }

  public static void main(String[] args) {
    LongInt a = new LongInt(3, 4);
    LongInt b = a.clone();
    System.out.println(a == b);         // false
    System.out.println(a.equals(b));    // true
  }
}
```



Marche enfin !

Clone et copie défensive

Il faut faire une copie défensive si la classe possède des champs mutables.

On appelle **clone()** sur l'objet mutable

```
public class Holder implements Cloneable {
  private Mutable mutable;
  public Holder(Mutable mutable) {
    this.mutable = mutable.clone();           // copie défensive
  }
  @Override
  public Holder clone() {
    Holder holder;
    try {
      holder = (Holder)super.clone();       // shallow copy
    } catch(CloneNotSupportedException e) {
      throw new AssertionError(e);
    }
    holder.mutable = this.mutable.clone(); // copie défensive
    return holder;
  }
}
```

void finalize()

Méthode testamentaire qui est appelée juste avant que l'objet ne 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**

Précaution d'utilisation

finalize() est appelée une seule fois par objet

finalize() empêche les objets d'être réclamés plus tôt car il doivent vivre même si ils devraient être garbage-collectés pour appeler finalize()

finalize() n'est pas appelé par la thread main cf cours de master

System.gc() est très très lent, au point qu'il existe un argument de la VM pour le désactiver

java.lang.Class

Représente la classe d'un objet **à l'exécution**

Il existe 3 façon d'obtenir une instance de la classe java.lang.Class

- Class.forName("java.lang.String")
  - Pratique pour un système de plugin
- String.class
  - Même appel mais la classe est vérifiée par le compilateur et non initialisée
- La méthode getClass() sur une Instance
  - Renvoie la classe de l'instance

Type et classe

Ne pas confondre un type et une classe

```
Object o = new Integer(3);
```

Le type de o est l'information pour le compilateur (ici Object)

La classe de o est l'information pour la machine virtuelle (ici Integer)

getClass() permet d'obtenir la classe à l'exécution

```
o.getClass() == Integer.class   // true
```

Class<?> Object::getClass()

Cette méthode est **final** et 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
```

La classe java.lang.Class est typé comme Class<?> (cf cours de master)

.getClass() et instanceof

Quelle différence entre

```
Object o = ...
if (o.getClass() == Number.class) {
  // ...
}
```

et

```
Object o = ...
if (o instanceof Number) {
  // ...
}
```

Indication: Number est une classe abstraite donc un des deux if est toujours faux

.getClass() vs instanceof

instanceof teste si la classe de l'instance est la classe demandée **ou une sous-classe**

.getClass() == est juste une égalité entre classes

Faire un o.getClass() == Foo avec Foo une interface ou une classe abstraite est idiot

Les tableaux

- Il existe deux types de tableaux :
  - Les tableaux d'objets, héritent de Object[],
    ils contiennent des références sur des objets
  - Les tableaux de types primitifs, héritent 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 caractères immutable
  - **StringBuilder**, buffer de caractères 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 caractères **immutable**
- Syntaxe “foo” crée un objet de la classe java.lang.String
- Concaténation avec +

```
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));
  }
}
```

String constante

La VM utilise un même objet pour la même chaine de caractères **littérale**

```
public static void main(String[] args) {
  String s = "toto";
  System.out.println(s == "toto"); // true

  String t = new String(s);        // a ne pas utiliser
  System.out.println(t == "toto"); // false

  String u = new Scanner(System.in).next();
  System.out.println(u == "toto"); // false
}
```

Si les String sont constantes == suffit, sinon il faut utiliser equals

String.intern()

La méthode **intern()** permet de récupérer l'instance de String constante

```
public static void main(String[] args) {
  String s="toto";
  System.out.println(s=="toto"); // true

  String t = new String(s);      // a ne pas utiliser
  String u = t.intern();
  System.out.println(u=="toto"); // true
}
```

Switch sur les strings

Le switch sur un String est un switch sur le hashCode() + un equals()

```
public static void main(String[] args) {
  switch(args[0]) {
    case "-a":
      ...
      break;
    ...
    default:
      ...
  }
}
```

Traduction par le compilateur

```
public static void main(String[] args) {
  String s = args[0];
  switch(s.hashCode()) {
    case 1492:   // "-a".hashCode()
      if (s.equals("-a")) {
        ...
        break;
      }
      goto default;
    ...
    default:
      ...
  }
}
```

Les méthodes de String

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

StringBuilder

Equivalent mutable de **String**, possède un buffer de caractères qui s'aggrandit tout seul

Est utilisé pour créer une chaîne de caractères qui sera finalement rendue immutable

```
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 →

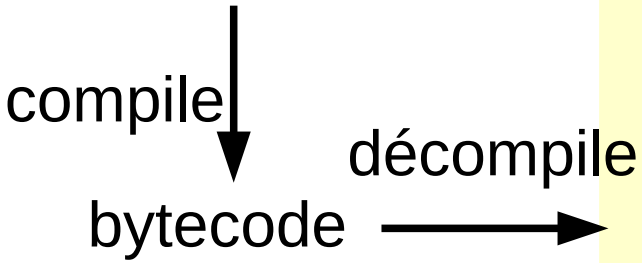
```
public class ConcatExample {
  public String toString() {
    return new StringBuilder().append("toto").
      append(value).toString();
  }
  private final int value;
}
```

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

La concaténation (2)

Lorsqu'il y a une boucle

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




```
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 généré alloue un StringBuilder par tour de boucle, pas top :(

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 caractères 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

Wrapper en mémoire

Un type primitif est “alloué” sur la pile,
un wrapper est alloué dans le tas, la pile contient une référence

Utiliser un wrapper à la place d'un type primitif a donc un coût

- Plus gros en mémoire (header d'un objet + alignement)
- Plus de pression sur le GC

Wrapper, cas d'utilisation

Un wrapper est intéressant car on peut réutiliser du code qui fonctionne avec un objet

- Par exemple, les collections

Un wrapper n'est pas intéressant en tant que champs, utiliser un type primitif plutôt !

```
public class Stupid {
  private final Integer value;

  public Stupid(int value) {
    this.value = value;   // boxing !
  }
}
```

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*)**

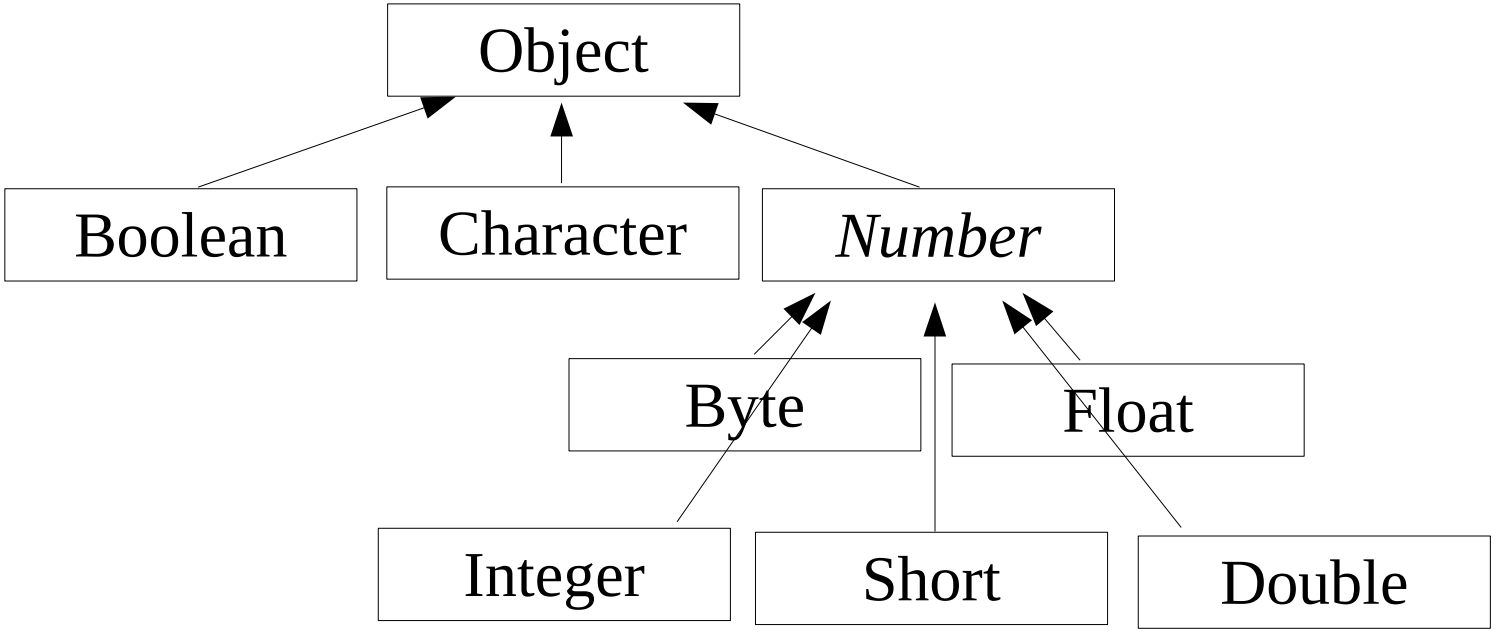
```
int value = 3;
Integer i = Integer.valueOf(value);
```

***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

Depuis Java 1.5, la conversion est automatique, et dans les deux sens

- Auto-boxing :

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

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

- Auto-unboxing :

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

int i = value;  // unboxing
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 =
 (Integer)list.get(0);
```

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

Autoboxing & Objects.requireNonNull()

Il n'y a pas de moyen de désactiver l'auto-boxing même si de temps en temps, le compilateur devrait générer une erreur

```
public class Objects {
  public static <T> T requireNonNull(T obj) { ... }
}
```

```
public class OhNo {
  private final int value;

  public OhNo(int value) {
    this.value = Objects.requireNonNull(value); // compile !
  }
}
```

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 ou true
                 // 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**.

- pour les autres valeurs on ne sait pas :(

Boxing et cache (2)

donc, les *wrappers* possèderont la **même référence**

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

Integer i = 235;
Integer j = 235;

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

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

Il est interdit de faire == ou != 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 voit pas facilement 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[])
  }
}
```