

# Object, sous-typage & polymorphisme

# Plan

- java.lang.Object
- Sous-typage
- Polymorphisme
- Redéfinir equals & hashCode

# Type, référence et objet

En Java, il existe deux sortes de types

Les types primitifs qui sont manipulés par leur valeur

- `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`

Les types objets qui sont manipulés par leur référence

- `Object`, `String`, `int[]`, `StringBuilder`, etc.

La taille d'une case mémoire correspondant à une variable locale ou à un champ n'excède donc jamais 64 bits

# java.lang.Object

En Java, toutes les classes héritent directement ou indirectement de `java.lang.Object`

```
public class Person {  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

Hérite implicitement de `java.lang.Object`



# toString(), equals() & hashCode()

`java.lang.Object` définit 3 méthodes  
“universelles”

- `toString()`
  - Chaîne permettant un affichage **de debug** d'un objet
- `equals()`
  - Qui renvoie vrai si deux objets sont égaux structurellement (si leurs champs sont égaux).
- `hashCode()`
  - Renvoie un “résumé” d'un objet sous forme d'un entier

# toString(), equals() & hashCode()

Chacune de ces méthodes possède une implantation par défaut (définie dans Object)

```
Person person = new Person("Mark", 23);
```

```
person.toString()
```

Person@73d16e93, car par défaut : class+"@"+hashCode()

```
person.equals("hello")
```

false, car par défaut équivalent à == (égalité primitive)

```
person.equals(person)
```

true, car par défaut équivalent à ==

```
person.hashCode()
```

73d16e93, valeur aléatoire calculée une fois (sur 24bits)

# Magique ?

```
Person person = new Person("Mark", 23);  
System.out.println(person); // Person@73d16e93
```

- Comment se fait-il que l'on puisse appeler une méthode déjà écrite dans le JDK avec un objet inconnu lors de la création de cette méthode ?

`System.out` est de type `PrintStream`  
or, dans cette classe la méthode qui existe est  
`PrintStream::println(Object)`

- Comment ça fonctionne avec `println(person)` ?

# Plan

- java.lang.Object
- Sous-typage
- Polymorphisme
- Redéfinir equals & hashCode



# Sous Typage

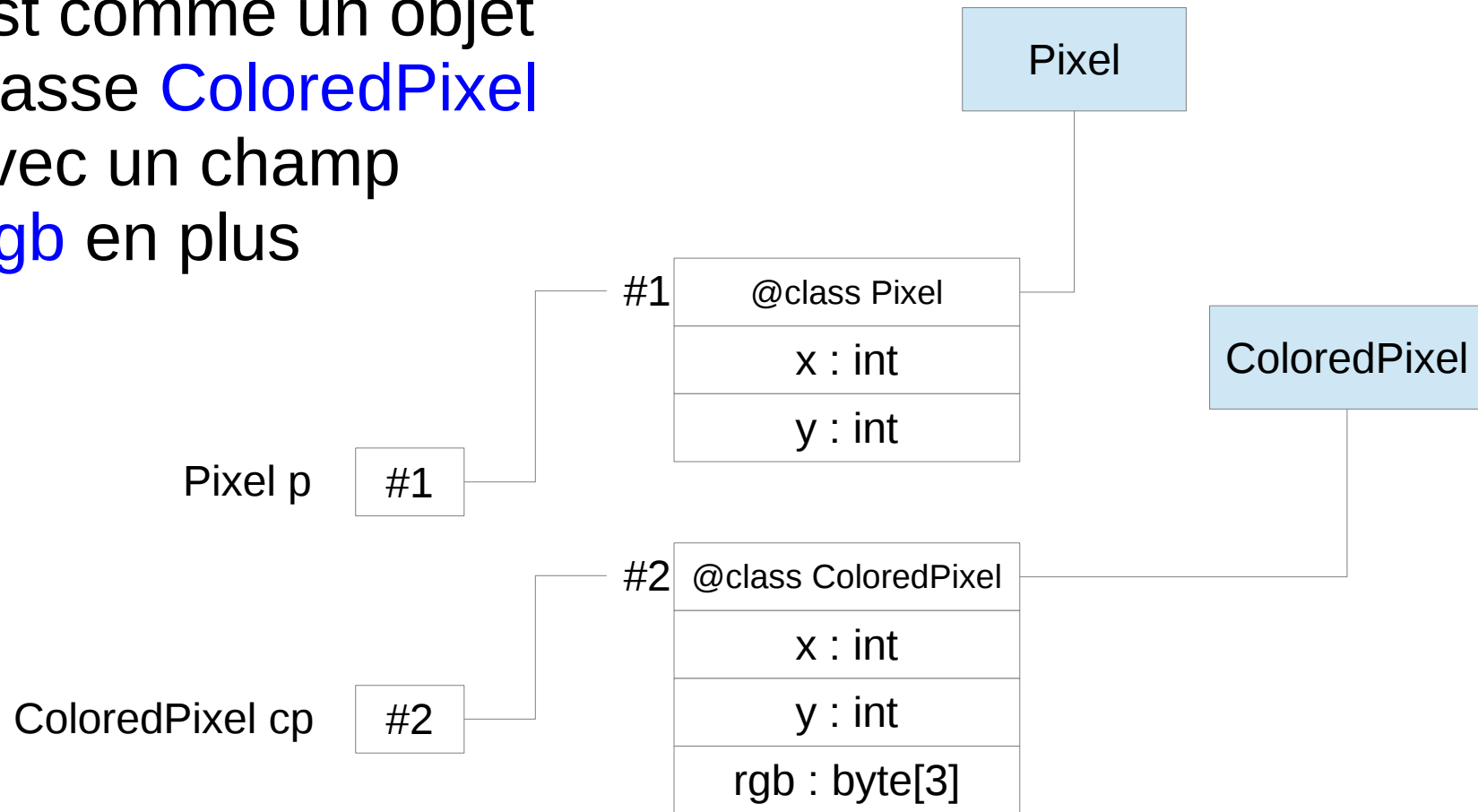
- En fait, **Person** est un **sous-type** de **Object**
    - La relation de **sous-typage** est liée à la notion d'héritage (entre-autres)
      - Il y a plusieurs façons d'obtenir du sous-typage en Java (héritage, interface, conversion de types primitifs)
    - **Ce qu'on sait faire sur un Type, on doit également savoir le faire sur un Sous-Type**
- => introduisons les éléments syntaxiques de l'héritage

# Héritage : un exemple

- On a la classe `Pixel`, dont les instances ont un `x` et un `y`.
  - Sur tout `Pixel`, on peut faire `moveTo(int nx, int ny)` pour « déplacer » ce `Pixel`
- On voudrait maintenant avoir une classe `ColoredPixel`, dont les instances ont un `x`, un `y`, et une couleur sous la forme d'un `byte[3] rgb`
  - Sur tout `ColoredPixel`, on peut faire `moveTo(int nx, int ny)`, mais aussi récupérer par des getter les composantes RGB : `getRed()`, `getGreen()`, `getBlue()`

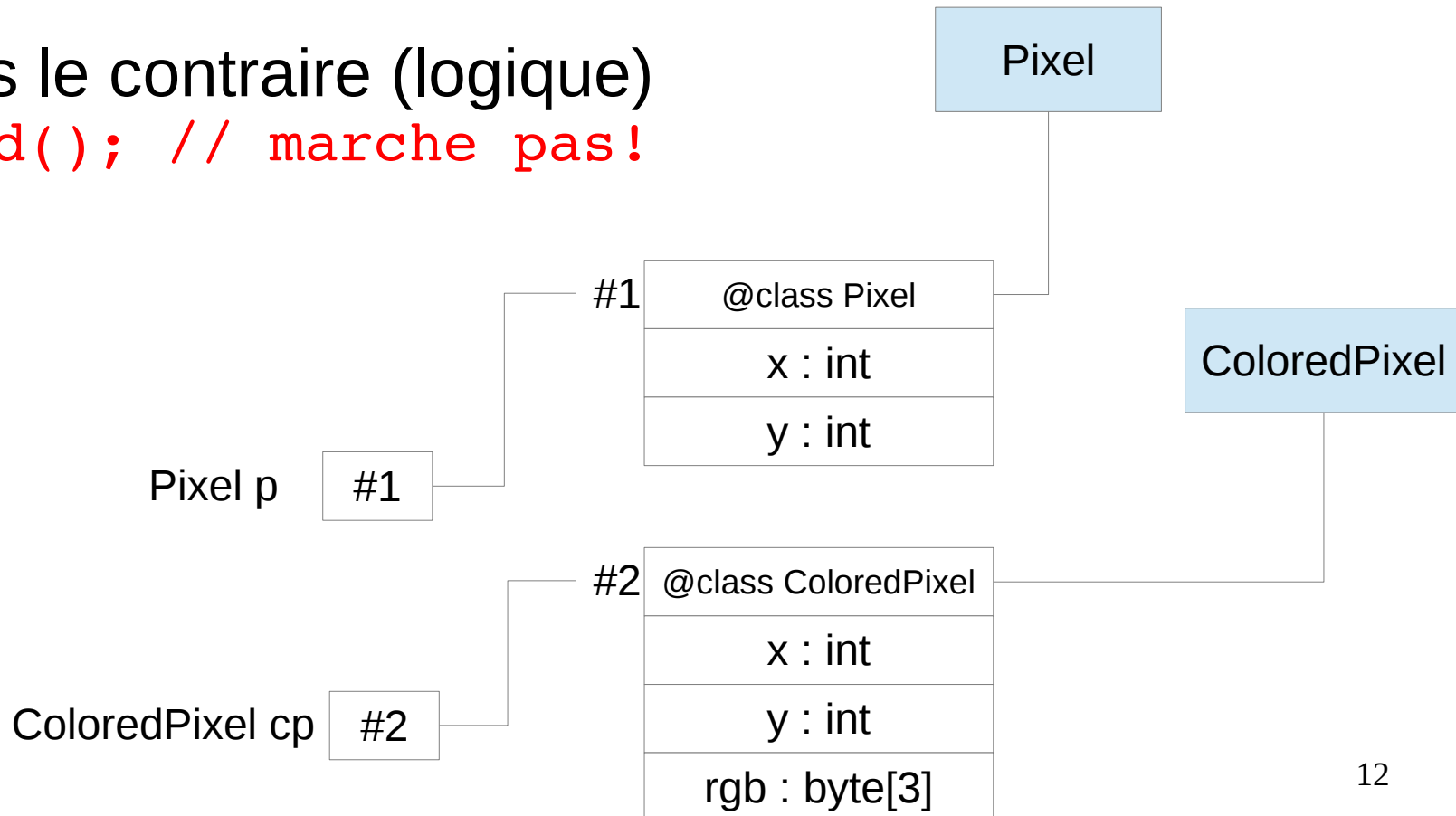
# Héritage : un exemple

- Rq1 : un objet de la classe **Pixel** est comme un objet de la classe **ColoredPixel** mais avec un champ **byte[] rgb** en plus



# Héritage : un exemple

- Rq2 : tout ce qu'on sait faire (en terme de méthodes) sur un **Pixel**, on peut le faire sur un **ColoredPixel**  
`cp.moveTo(2,2);`
- Mais pas le contraire (logique)  
`p.getRed(); // marche pas!`

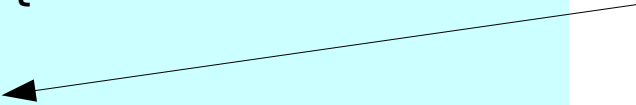


# Si on le fait... sans héritage

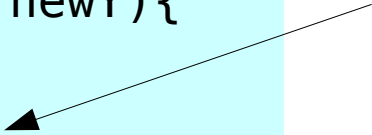
```
public class Pixel {  
    private int x;  
    private int y;  
    public void moveTo(int newX, int newY){  
        this.x = newX;  
        this.y = newY;  
    }  
}
```

```
public class ColoredPixel {  
    private int x;  
    private int y;  
    private byte[] rgb;  
    public void moveTo(int newX, int newY){  
        this.x = newX;  
        this.y = newY;  
    }  
    public byte getRed() { return rgb[0]; }  
    public byte getGreen() { return rgb[1]; }  
    public byte getBlue() { return rgb[2]; }  
}
```

Duplication des  
champs déclarés



Duplication  
des méthodes



# Si on le fait... avec héritage

```
public class Pixel {  
    private int x;  
    private int y;  
    public void moveTo(int newX, int newY){  
        this.x = newX;  
        this.y = newY;  
    }  
}
```

```
public class ColoredPixel extends Pixel {  
    private byte[] rgb;  
    public byte getRed() { return rgb[0]; }  
    public byte getGreen() { return rgb[1]; }  
    public byte getBlue() { return rgb[2]; }  
}
```

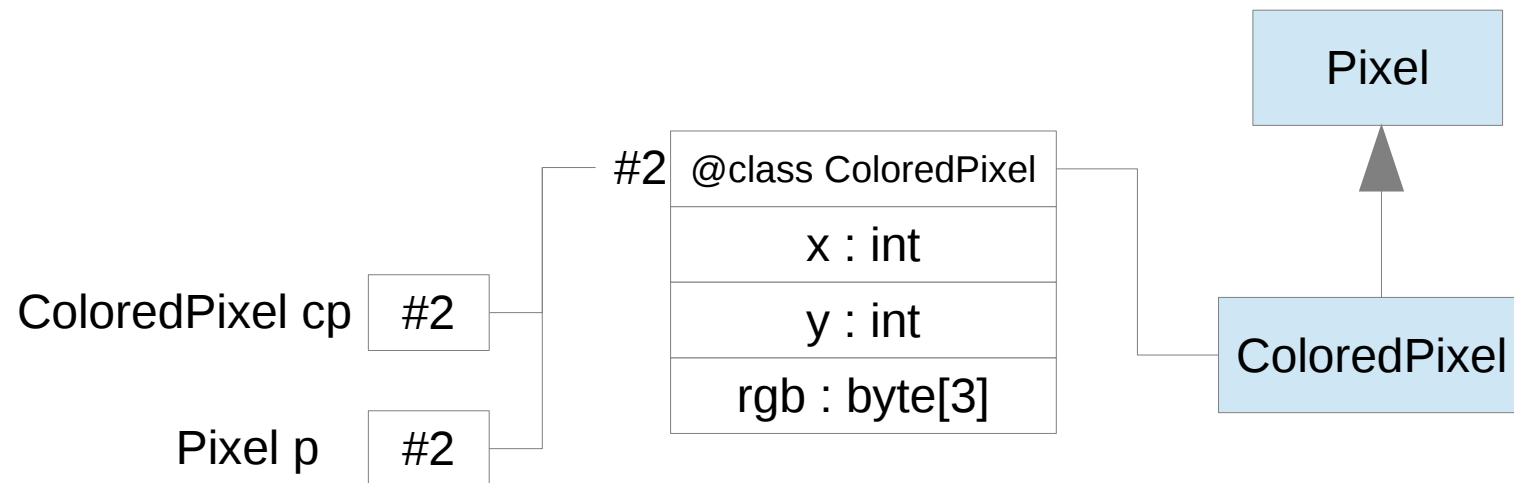
extends : « hérite de »

pas besoin de répéter  
les champs

on « hérite » des  
méthodes

# Et on obtient du sous-typage !

```
public static void main(String[] args) {  
    ColoredPixel cp = new ColoredPixel(); // cp est un ColoredPixel  
    Pixel p = cp; // mais on peut le manipuler comme un Pixel  
    // ColoredPixel est un sous-type de Pixel  
    p.moveTo(1,1); // va déplacer le ColoredPixel... à suivre !  
}
```



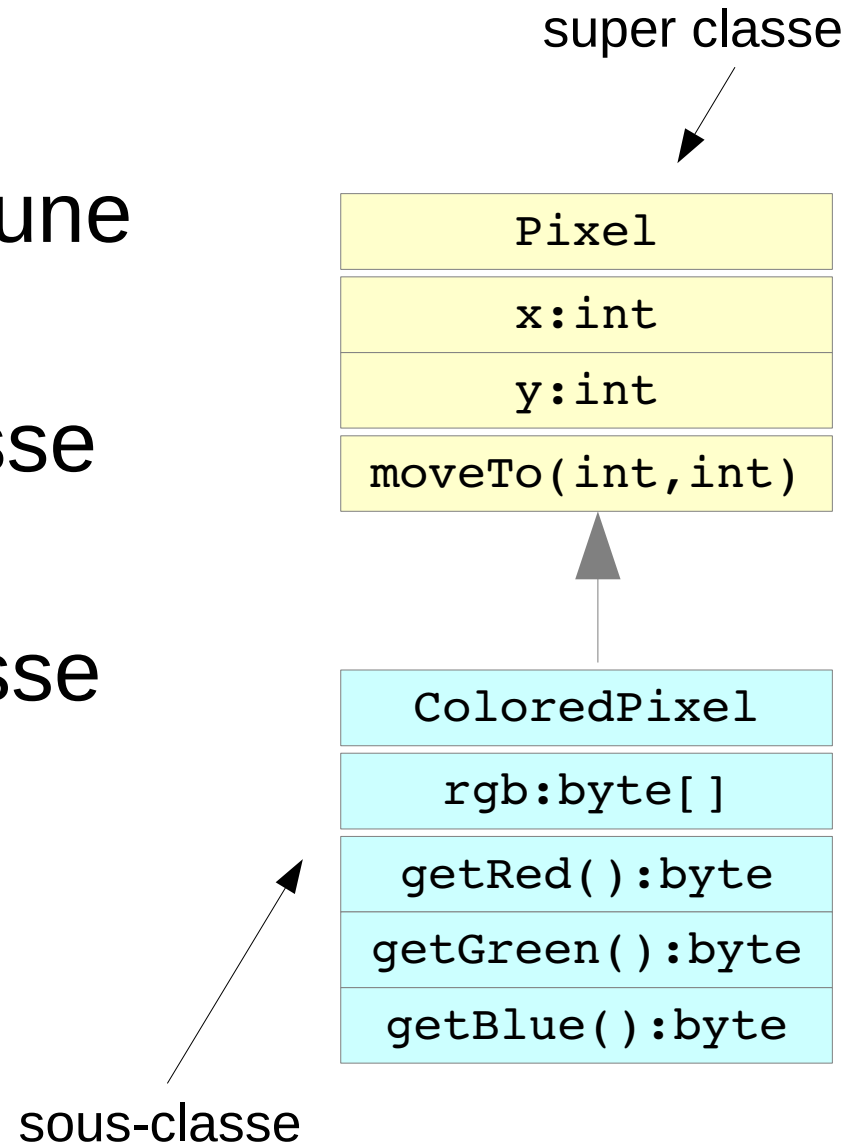
Le compilateur accepte d'utiliser une variable d'un sous-type en lieu et place d'une variable d'un super-type

# Vocabulaire

Une sous-classe hérite d'une super-classe

La sous-classe est la classe qui hérite

la super-classe est la classe dont on hérite



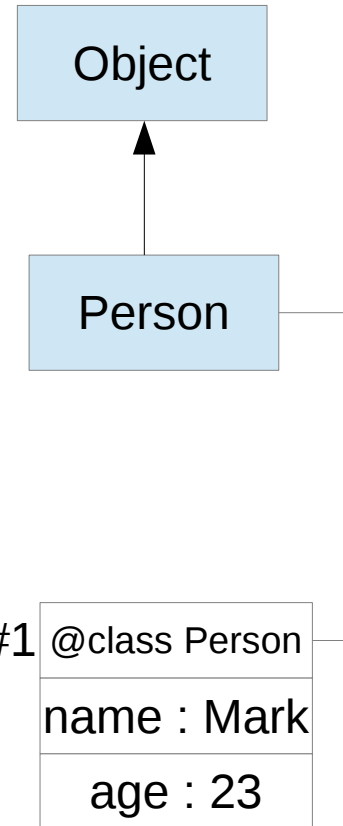


# TOUT est sous-type d'Object

- 1) la classe **Person** hérite de la classe **Object**
- 2) on doit pouvoir faire sur les instances de **Person** tout ce qu'on sait faire sur les instances d'**Object**  
`p.toString()`, `p.equals()`, `p.hashCode()`...
- 3) une référence à une instance de **Person** peut être stockée dans une variable déclarée de type **Object**
- 3bis) autrement dit, les objets de la classe **Person** peuvent être manipulés par des références sur **Object** :

```
Object o = new Person("Mark", 23); // ok !
```

=> On dit que **Person** est un **sous-type** de **Object**  
(un type plus précis)  
**Object** est un **super-type** de **Person**



# PrintStream::println(Object)

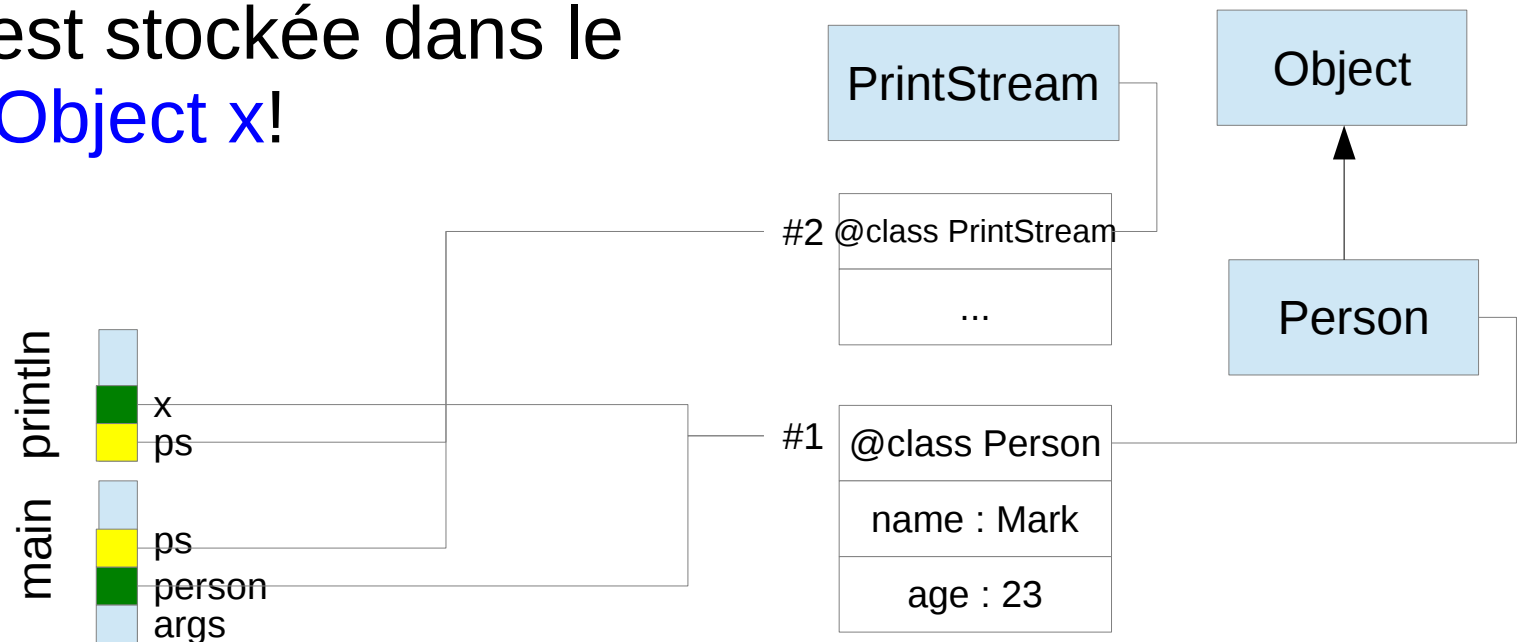
- ```
package java.io;
public class PrintStream {
    ...
    public void println(Object x) {
        ...
    }
}
```
- ```
public class Person {
    ...
    public static void main(String[] args) {
        Person person = new Person("Mark", 23);
        PrintStream ps = System.out;
        ps.println(person);
    }
}
```

??



# PrintStream.println(Object)

En mémoire, lors de l'exécution,  
la référence (#1) est identique  
lorsqu'elle est stockée dans le  
paramètre **Object x**!



Le sous-typage est un mécanisme pour le  
compilateur pas à l'exécution !

# PrintStream.println(Object)

- ```
package java.io;
public class PrintStream {
    ...
    public void println(Object x) {
        String s = String.valueOf(x);
        ...
    }
}
```

appel



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

Object::toString



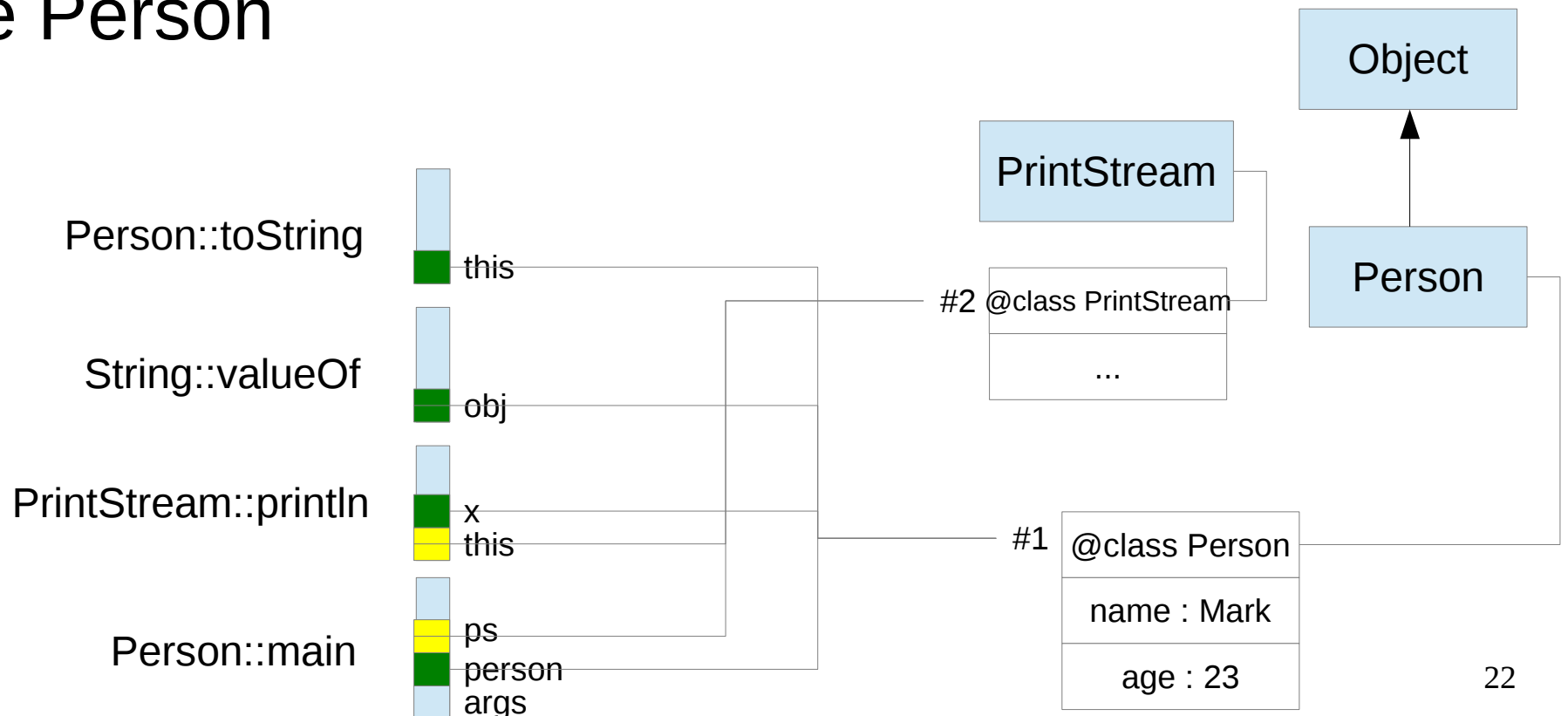
# Magique, Magique!

Et si on écrit une méthode toString() dans Person ?

```
public class Person {  
    private final String name;  
    private final int age;  
    ...  
    public String toString() {  
        return name + ' ' + age;  
    }  
  
    public static void main(String[] args) {  
        Person person = new Person("Mark", 23);  
        System.out.println(person);  
        // Mark 23                <----- TADA !  
    }  
}
```

# Pourquoi ?

En mémoire, lors de l'appel `obj.toString()`, `obj` est à l'exécution un objet de la classe `Person` donc la méthode `toString()` appelée est celle de `Person`



# Plan

- `java.lang.Object`
- Sous-typage
- Polymorphisme
- Redéfinir `equals` & `hashCode`

# Polymorphisme

A la compilation, le compilateur voit  
`Object::toString`

A l'exécution, la méthode `Object::toString` est appelée avec en premier paramètre une référence sur un objet de la classe `Person`, dans ce cas, la VM appelle la méthode `Person::toString`

Ce mécanisme est appelé le polymorphisme !



# Polymorphisme

Le polymorphisme est le fait de substituer à l'exécution un appel de méthode par un autre en fonction de la classe du receveur

```
receveur.methode(param1, param2)
```

```
Object[] array = new Object[] {  
    "hello", Person("Mark", 23) };  
for(Object value: array) {  
    System.out.println(o);  
}
```

Appelle String::toString puis Person::toString

# Sous-typage et polymorphisme

- Le sous-typage permet de réutiliser un code (dit “générique”) qui a été écrit en typant les références avec un super-type (ici Object) et en appelant le code avec un sous-type.
- Le polymorphisme est le fait que lors de l'exécution du code “générique” avec des références sur une sous-classe, les appels de méthodes sur le super-type appellent les méthodes définies sur le sous-type.

# Redéfinition de méthode

- Une méthode **redéfinit** une autre si elle est substituable par polymorphisme

Par ex, `Person::toString` est une redéfinition de `Object::toString`

- Une méthode redéfinie est une façon de remplacer un code d'une méthode existante par un nouveau code

# Méthodes appelables

```
public class Person {  
    private final String name;  
    private final int age;  
    ...  
  
    public static void main(String[] args) {  
        Person person = new Person("Mark", 23);  
        person.method(...)  
    }  
}
```

Ici, method peut être soit `Object::equals`,  
`Object::hashCode` et `Object::toString` !

# Méthodes appelables (2)

```
public class Person {  
    private final String name;  
    private final int age;  
    ...  
    public String toString() {  
        return name + ' ' + age;  
    }
```

Remplace Object::toString

```
    public static void main(String[] args) {  
        Person person = new Person("Mark", 23);  
        person.method(...)  
    }  
}
```

Ici, method peut être soit Object::equals, Object::hashCode et **Person::toString** !

# Redéfinition

Le mécanisme de polymorphisme est mis en œuvre automatiquement par la machine virtuelle (on ne peut pas l'empêcher)

Mais il n'y a pas polymorphisme s'il n'y a pas redéfinition

- Si la méthode est statique (pas d'objet, pas de classe à l'exécution)
- Si la méthode est privé (pas visible)
- Si la signature de la méthode n'est pas identique (pas complètement vrai cf cours redéfinition/surcharge)

# Non-redéfinition

```
public class Person {  
    private final String name;  
    private final int age;
```

```
    ...
```

```
    public String toString() {  
        return name + ' ' + age;  
    }  
}
```

Oups !

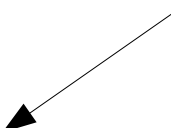


Dans ce cas, Person contient deux méthodes :  
Object::toString et Person::toString

# @Override

@Override est une annotation qui demande au compilateur de vérifier qu'il existe une méthode à redéfinir dans le super-type

```
public class Person {  
    private final String name;  
    private final int age;  
    ...  
    public @Override String toStrrrring() {  
        return name + ' ' + age;  
    }  
}
```

Compile pas !!!! 



# @Override

Attention !, @Override est une annotation pour le compilateur pas pour la machine virtuelle

Le mécanisme de polymorphisme marche sans le @Override

Le @Override permet au compilateur de signaler une erreur si la signature de la méthode est pas identique à une méthode existante

# Plan

- `java.lang.Object`
- Sous-typage
- Polymorphisme
- Redéfinir `equals` & `hashCode`

# Redéfinir equals()

Object::equals permet de tester si deux objets sont égaux structurellement (champs à champs)

Comme equals est définie sur java.lang.Object la signature de la méthode qui doit être redéfinie est

boolean equals(Object o)

Attention !



# Equals et ==

**==** test l'**identité** d'une référence

- Est ce que deux références contiennent la même adresse mémoire

**equals(Object)** test l'**égalité** de deux objets

- Si deux objets ont le même contenu
- L'implantation doit être compatible avec ==  
l'implantation, par défaut, celle dans  
`java.lang.Object.equals()` fait juste un ==

# Pourquoi redéfinir `Object.equals()`

L'API des collections (structures de données) de `java.util` utilise `Object.equals(Object)` pour savoir si un objet est déjà stocké dans la collection

Si on ne redéfinit pas `equals`, tester si un objet est déjà présent ou rechercher un objet risque de ne pas fonctionner correctement

# Transtypage, type déclaré et type « réel »

- Le transtypage de référence est le fait de considérer explicitement (forcer) une référence comme étant d'un type donné (qui n'est pas nécessairement le type de l'objet accessible via cette référence)
- La machine virtuelle vérifiera, à l'exécution, que le type en question est bien compatible et que voir cette référence comme étant de ce type là est possible; dans le cas contraire, l'exécution provoque une **ClassCastException**

```
class A { }  
class B extends A { }  
class C extends B { }
```

```
B b = new B();  
A a = b;  
// B b2 = a; // incompatible types  
B b2 = (B) a; // OK  
C c = (C) a; // ClassCastException
```

```
Object o;  
if(Math.random() > 0.5)  
    o = "toto";  
else  
    o = new Object();  
String s = (String) o;  
// Compile toujours mais a une  
// chance sur deux de lever une  
// ClassCastException...
```

# L'opérateur instanceof

- Il est possible d'assurer un transtypage sans exception en utilisant l'opérateur **x instanceof T**
  - **x** doit être une (variable contenant une) référence ou **null**
  - **T** doit représenter un type
  - Le résultat vaut **true** si **x** n'est pas **null** et s'il peut être affecté dans **T** sans **ClassCastException**; sinon c'est **false**.

```
class A { }  
class B extends A { }  
class C extends B { }
```

```
A ab = null;  
System.out.println(ab instanceof A); // false  
ab = new B();  
System.out.println(ab instanceof A); // true  
System.out.println(ab instanceof B); // true  
System.out.println(ab instanceof C); // false
```

```
Object o; String s;  
if(Math.random()>0.5)  
    o = "toto";  
else  
    o = new Object();  
if (o instanceof String)  
    s = (String) o; // OK...
```

# Equals et classe

Equals doit renvoyer false si l'objet passé en argument est d'une classe incompatible avec l'objet courant.

On utilise l'opérateur instanceof pour faire le test dynamiquement

- o **instanceof Foo**

renvoie true si la classe de o est une sous-classe de Foo

- `null instanceof Foo` est toujours false
- donc `o instanceof Object` est équivalent à `o != null`
  - Car toute instance est d'une sous classe de Object



# Instanceof vs getClass

`Object::getClass()` permet d'obtenir la classe d'une référence à l'exécution

`o.getClass() == Foo.class`

- Teste si `o` est une instance de `Foo`  
(pas une sous-classe contrairement à `instanceof`)
- Si `o` est `null` => `NullPointerException`
- `o.getClass() == Interface.class` ou  
`o.getClass() == ClassAbstraite.class` est idiot !

=> rarement ce que l'on veut pour `equals` !

# instanceof vs cast

La sémantique du cast “(Foo)” et de “instanceof Foo” est quasiment équivalente

- Les deux testent les sous-classes
- instanceof renvoie faux là où le cast lève l'exception ClassCastException si la classe de la référence n'est pas un sous-type

Mais null est traité différemment

- null instanceof Foo renvoie false
- (Foo)null est toujours valide

# Code de equals

```
public class Car {  
    private final String owner;  
    private final int numberOfWheels;
```

```
    ...
```

```
    @Override
```

```
    public boolean equals(Object o) {
```

```
        if (!(o instanceof Car)) {
```

```
            return false;
```

```
        }
```

```
        Car car = (Car)o;
```

```
        return ...;
```

```
    }
```

```
}
```

doit être Object  
sinon pas de redéfinition



On teste la classe  
(et les sous-classes)

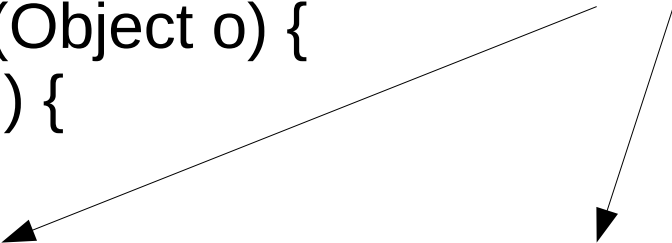


# Pas de getter

Un getter peut être **redéfini** dans une sous classe, ce qui va changer la sémantique du equals !

```
public class Car {  
    private final int numberOfWheels;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        if (!(o instanceof Car)) {  
            return false;  
        }  
        Car car = (Car)o;  
        return getNumberOfWheels() == car.getNumberOfWheels()  
        &&  
        ...  
    }  
}
```

ahhhhh



# Code de equals

```
public class Car {  
    private final String owner;  
    private final int numberOfWheels;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        if (!(o instanceof Car)) {  
            return false;  
        }  
        Car car = (Car)o;  
        return numberOfWheels==car.numberOfWheels  
            && owner.equals(car.owner);  
    }  
}
```

type objet

type primitif

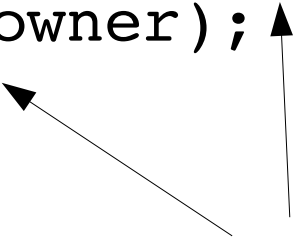
type primitif

type objet

private veut dire à l'intérieur de la classe et pas que pour this

# Astuces habituelles

```
public class Car {  
    private final String owner;  
    private final int numberOfWheels;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        if (!(o instanceof Car)) {  
            return false;  
        }  
        Car car = (Car)o;  
        return numberOfWheels==car.numberOfWheels  
            && owner.equals(car.owner);  
    }  
}
```



&& est paresseux donc on  
teste les primitifs d'abord

# equals() et null

`a.equals(b)` n'est pas symétrique dans le cas où `a` ou `b` est `null`

- Si `a` est `null`
  - lève une `NullPointerException`
- Si `b` est `null`
  - Renvoie `false`

La méthode `java.util.Objects.equals(a,b)` permet de tester si deux objets sont égaux ou tout les deux `null`

# Equals et performance

Attention: l'appel à equals peut être assez coûteux en temps d'exécution

par exemple:

- `String::equals`, teste les caractères des deux chaînes de caractères deux à deux
- `Car::equals`, teste les différents champs, dont un qui est lui-même une `String`



# Redéfinir hashCode()

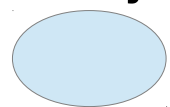
hashCode() doit renvoyer un entier qui “résume” l'objet sur lequel il a été appelé

comme hashCode() est utilisé par l'API des collections basée sur des tables de hachage (java.util.HashMap et java.util.HashSet) il est important que la valeur de hachage couvre l'ensemble des entiers possibles, sinon la table de hachage se transforme en liste chaînée (  $O(1)$   $\rightarrow$   $O(n)$  )

# Rappel table de hachage

- `ArrayList.remove()` et `ArrayList.contains()` en  $O(n)$ 
  - Scan de l'ensemble des éléments
- `HashSet.add()`, `remove()` et `contains()` en  $O(1)$ 
  - On essaye de ranger les éléments à la bonne case

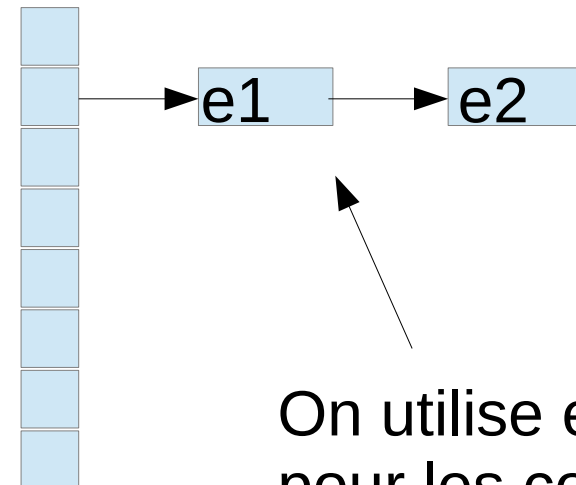
@Object



hashCode()

int

% array.length



On utilise `equals()`  
pour les collisions<sup>50</sup>

# hashCode et mutabilité

hashCode doit être sans effet de bord

- sinon on ne le retrouvera pas dans la table de hachage

définir hashCode sur un objet mutable est dangereux

- car si l'objet est modifié la valeur de hachage change !!

# hashCode et equals

Deux objets o1, o2 tel que  
o1.equals(o2) doivent vérifier que  
o1.hashCode() == o2.hashCode()

L'inverse est faux,  
si o3.hashCode() == o4.hashCode(),  
o3 et o4 peuvent ne pas être égaux

Si on **redéfinit** equals, on **doit redéfinir**  
hashCode (et vice versa) !

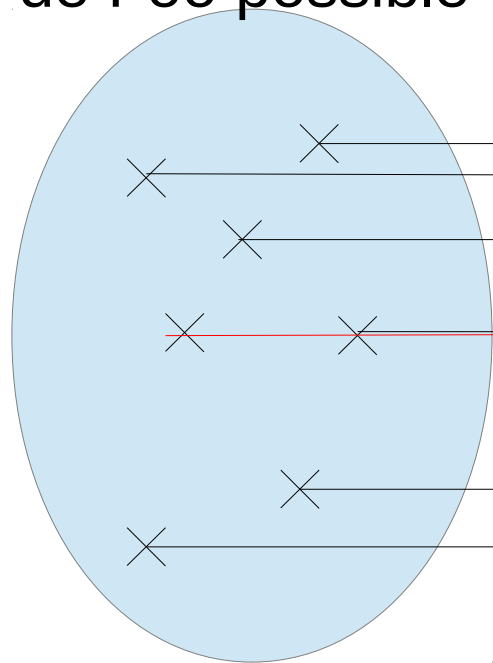
# hashCode et equals

Si on **redéfinit** equals,  
on **doit** redéfinir hashCode !  
et vice versa !

# Répartition des valeurs de hashCode

Projection de l'ensemble des instances d'une classe vers l'ensemble des entiers

Ensemble des instances  
de Foo possible



Foo

int

collision

# Example

```
public class Car {  
    private final String owner;  
    private final int numberOfWheels;  
    ...
```

type objet →

type primitif →

```
@Override  
public int hashCode() {  
    return numberOfWheels ^ owner.hashCode();  
}
```

ou exclusif →

- @Override type primitif  
public boolean equals(Object o) {  
 ...  
 if (!(o instanceof Car)) {  
 return false;  
 }  
 Car car = (Car)o;  
 return numberOfWheels == car.numberOfWheels &&  
 owner.equals(car.owner);  
}

type objet →

# Implanter hashCode

hashCode doit être rapide !

pas d'allocation (pas de new !)

on utilise un ^ (ou exclusif) entre les différentes valeurs de hachage si celle-ci sont bien répartie

On peut utiliser Integer.rotateLeft(), Integer.rotateRight() pour décaler des bits de façon circulaire

```
return first.hashCode() ^  
Integer.rotateLeft(second.hashCode(), 16);
```

sinon on utilise une expression à base d'un nombre premier

```
char val[] = value;  
for (int i = 0; i < value.length; i++) {  
    h = 31 * h + val[i];  
}
```



# Switch sur les Strings

Il est possible de faire un switch sur des Strings en Java

```
String sizeName = ...
```

```
switch(sizeName) {
```

```
    case "big":
```

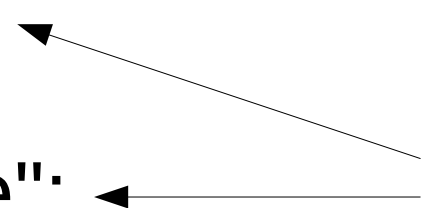
```
        ....
```

```
    case "large":
```

```
        ....
```

```
}
```

Le compilateur calcule  
les valeurs de hashCode  
pour chaque chaîne (constante)



# Switch sur les Strings

Puis equals() est appelée car deux String différentes peuvent avoir la même valeur de hashCode().

```
String sizeName = ...
switch(sizeName.hashCode()) {
    case 0x17d00:        // "big".hashCode()
        if (sizeName.equals("big")) {
            ....
        }
        break;
    case 0x61fbb3b:      // "large".hashCode()
        if (sizeName.equals("large")) {
            ....
        }
        break;
}
```

La même méthode hashCode doit être utilisée à la compilation et à l'exécution