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'excéde dont jamais 64 bits

java.lang.Object

En Java, toutes les classes hérite 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;
  }
}
```

toString(), equals() & hashCode()

java.lang.Object définie 3 méthodes “universelle”

- toString()
  - Chaine 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 possèdent une implantation par défaut

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

- person.toString()
  - Person@73d16e93 // class + "@" + hashCode()
- person.equals("hello")
  - false, car équivalent à ==
- person.equals(person)
  - true, car équivalent à ==
- person.hashCode()
  - 73d16e93, valeur aléatoire calculé une fois (sur 24bits)

Magic ?

```
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 inconnue lors de la création de cette méthode ?

```
System.out -> PrintStream
System.out.println(person)
                   -> PrintStream::println(Object)
```

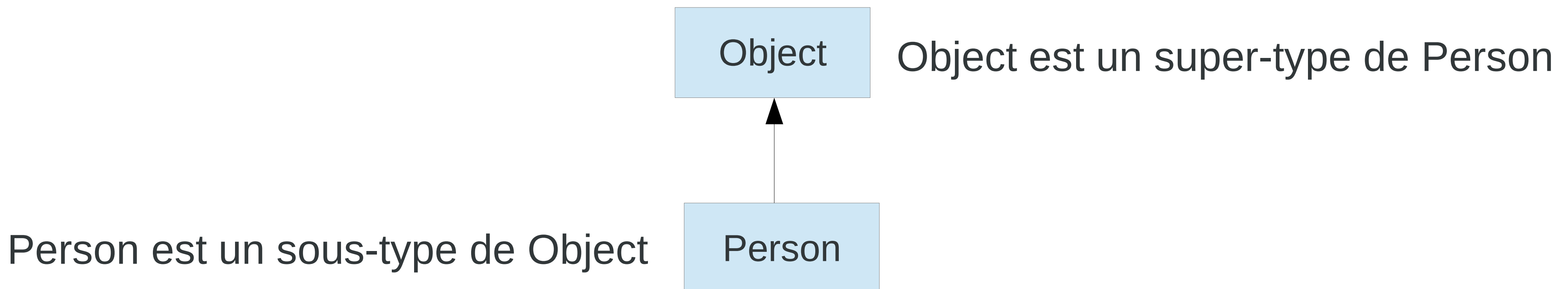
Sous-typage

Comme la classe Person hérite de la classe java.lang.Object

- Les objets de la classe Person peuvent être manipuler par des références sur java.lang.Object

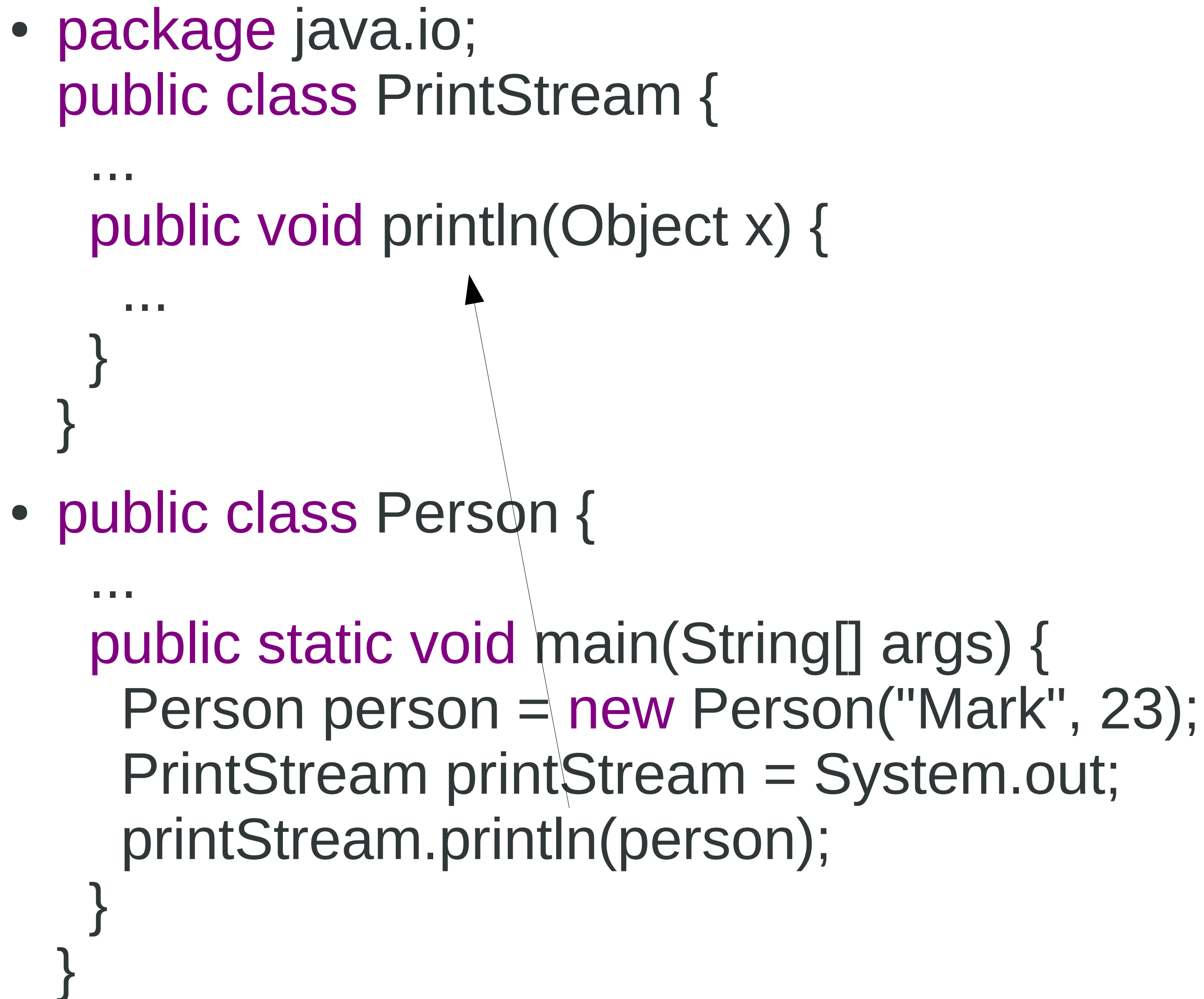
  Object o = new Person("Mark", 23);   // Ok !

- On dit que Person est un sous-type (un type plus précis) de Object



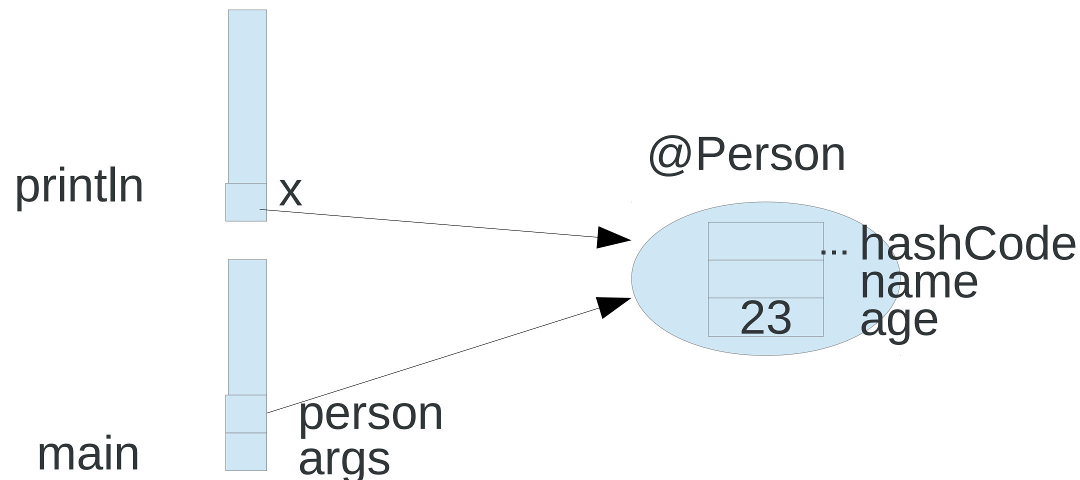
PrintStream.println(Object)

- ```
  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 printStream = System.out;
      printStream.println(person);
    }
  }
  ```



PrintStream.println(Object)

En mémoire, lors de l'exécution, la référence est identique !



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);
    ...
  }
}
```

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

Magic, Magic !

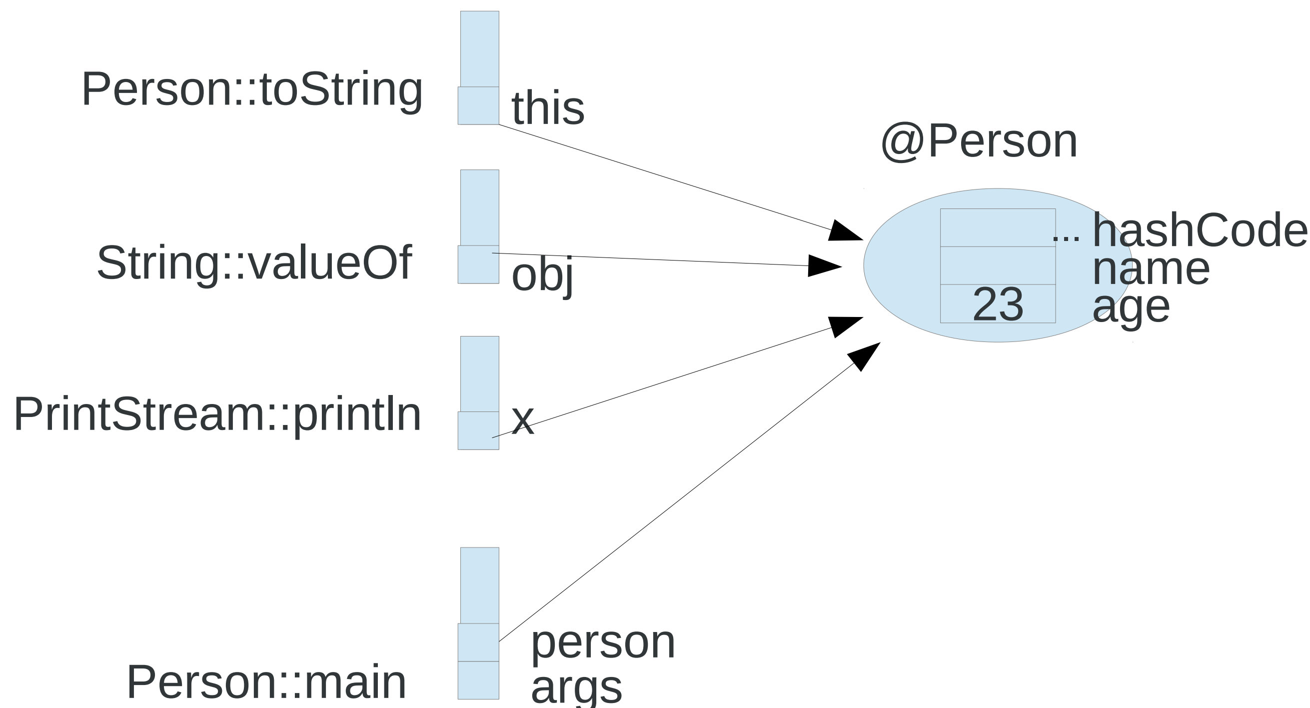
Et si on écrit une méthode toString() dans Person ?

```
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       <------------- ahhhhhh
  }
}
```

Pourquoi ?

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



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 appel la méthode String::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);
}
```

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) 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 appels les méthodes définie sur le sous-type.

Redéfinition de méthode

- Une méthode redéfinie 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;
  }

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

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

Redéfinition

Le mécanisme de polymorphisme est fait automatiquement par la machine virtuelle (sans possibilité de dire, non, je n'en veut pas)

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 est pas identique (pas complètement vrai cf cours redefinition/surcharge)

Non-redéfinition

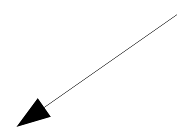
```
public class Person {
  private final String name;
  private final int age;
  ...
  public String toStrrrring() {
    return name + ' ' + age;
  }
}
```

Dans ce cas, Person contient en Object::toString et Person::toStrrrring

@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 lancer 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)

Equals et ==

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

- Est ce que deux réfèrences 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 (structure de données) de java.util utilise)Object.equals(Object pour savoir si un objet est déjà stocké dans la collection

Si on ne définie pas equals, tester si un objet est déja présent ou rechercher un objet risque de ne pas fonctionner correctement

Equals et classe

Equals doit renvoyer false si l'objet passé en argument n'est pas de la même classe que l'objet courant

On utilise l'opérateur instanceof pour faire le test dynamiquement

- o instanceof Foo renvoie vrai 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

Instanceof vs getClass

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

o.getClass() == Foo.class

- Test si o est une instance de Foo
  (pas une sous-classe contrairement à instanceof)
- Si o est null => NPE
- 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 tests les sous-classes
- Instanceof renvoie faux là et 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

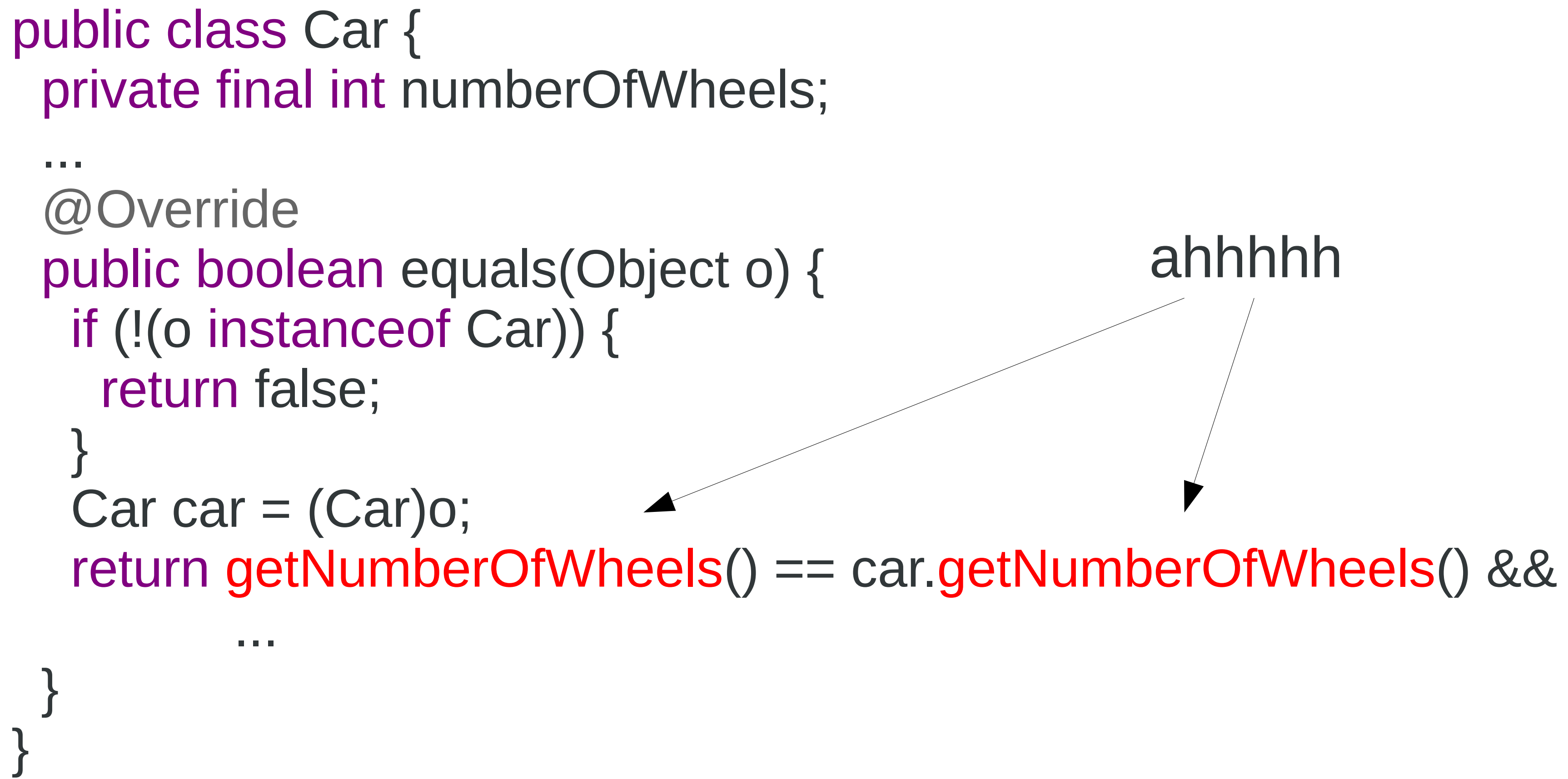
```
public class Car {
  private final String owner;
  private final int numberOfWheels;
  ...
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Car)) {
      return false;
    }
    Car car = (Car)o;
    return ...;
  }
}
```

doit être Object
sinon pas de redéfinition

On test la classe
(et les sous-classes)

Pas de getter

Un getter peut être **redéfinie** dans un 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() &&
              ...
  }
}
```

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

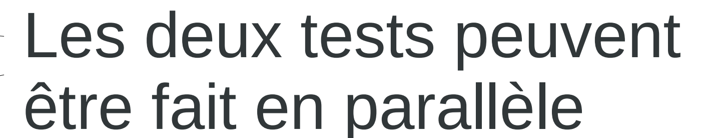
type primitif

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 (this == o) {
      return true;     // short-cut
    }
    if (!(o instanceof Car)) {
      return false;
    }
    Car car = (Car)o;
    return numberOfWheels == car.numberOfWheels &&
           owner.equals(car.owner);
  }
}
```



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 null

Equals et performance

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

par exemple:

- String.equals, test les caractères des deux chaines de caractères deux à deux
- Car.equals, test les différents champs, dont un qui est lui-même une String

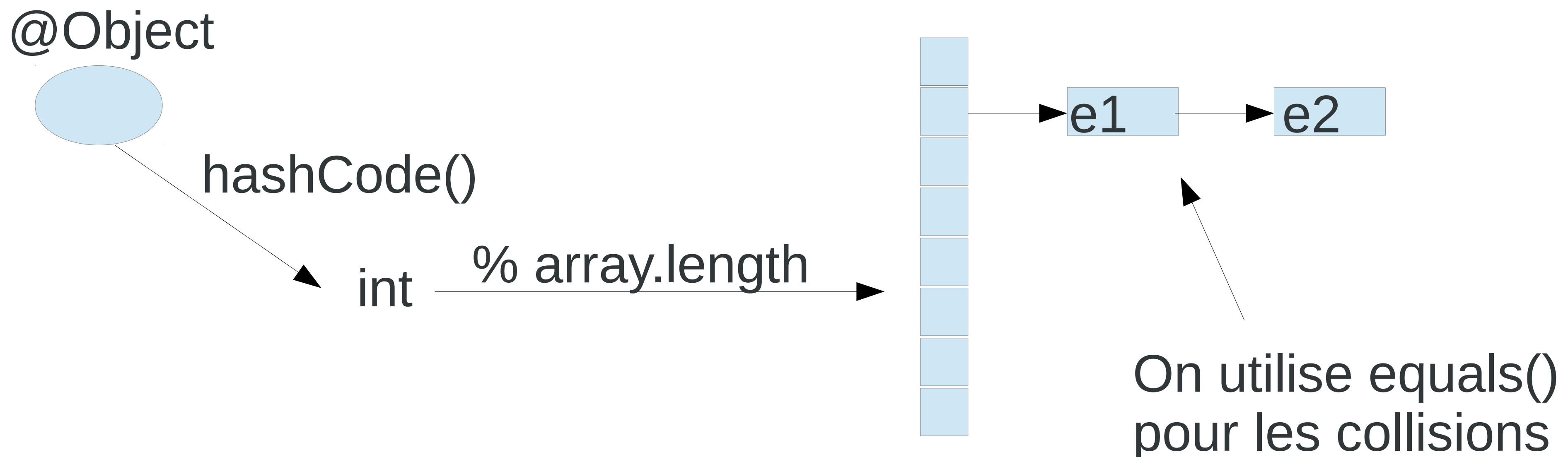
Redéfinir hashCode()

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

comme hashCode() est utilisé par l'API des collections basé 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 chainée (O(1) -> 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 élements à la bonne case



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()

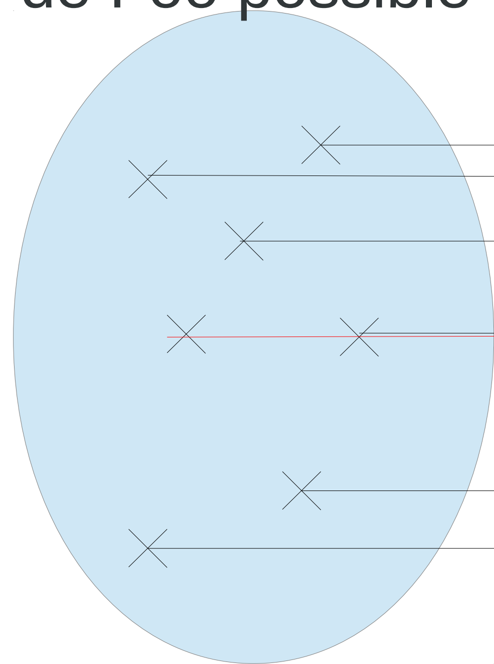
L'inverse est faux,
si o3.hashCode() == o4.hashCode(),
o3 et o4 peuvent ne pas être égaux

Si on **redéfinie** 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



collision

Foo

int

Exemple

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

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);
```

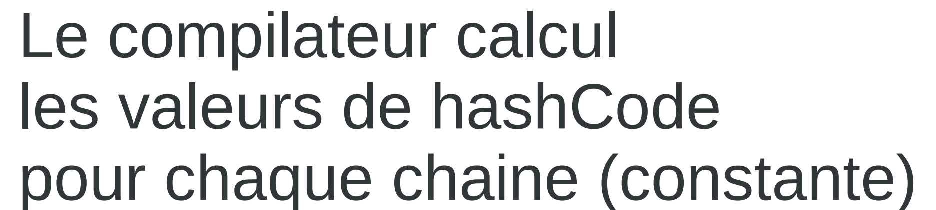
sinon on utilise une expression à base d'un nombre premier

```
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":
    ....
}
```



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;
}
```