

Introduction à la programmation objet



Objectifs du module

Approche objet de la programmation

- Connaître et maîtriser les concepts de la programmation objet
 - ▶ Classe, instance/objet, attribut, méthode, constructeur, encapsulation, héritage, interface, polymorphisme, liaison tardive
- Adopter le "penser objet"
 - ▶ Savoir décomposer un problème en classes et objets
 - ▶ Savoir expliquer ce qui différencie le paradigme objet des autres paradigmes
- Connaître les principes ouvert-fermé, de substitution de Liskov et *KISS (Keep It Simple, Stupid)*, et savoir les appliquer

Objectifs du module

Le langage Java

- Connaître les principaux éléments de la syntaxe du langage Java et pouvoir expliquer clairement leur rôle et leur sémantique
 - ▶ `new`, `this`, `super`, `public`, `private`, `protected`, `static`, `final`, `extends`, `implements`, `package`, `import`, `enum`, `throws`, `throw`
- Savoir écrire (et corriger) un programme dans le langage Java
 - ▶ Maîtriser les "outils" pour développer en Java : `javac`, `java` (et `classpath`), `javadoc`, `jar`, *IDE*, débogueur
 - ▶ Comprendre le transtypage (*upcast/downcast*)
 - ▶ Être en mesure de choisir une structure de données appropriée et savoir utiliser les types Java `List`, `Set`, `Map` et `Iterator`
 - ▶ Savoir gérer les exceptions et connaître la différence entre la capture et la levée d'une exception

Au menu

- 1 Introduction
- 2 Éléments syntaxiques de base
- 3 Classes et objets

Au menu

1 Introduction

2 Éléments syntaxiques de base

3 Classes et objets

Paradigme de programmation

- Est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation (source : Wikipédia)
- Exemples de paradigme de programmation
 - ▶ Paradigme impératif (e.g., Pascal, C)
 - ▶ Paradigme objet (e.g., Java)
 - ▶ Paradigme fonctionnel (e.g., Lisp)
 - ▶ Paradigme logique (e.g., Prolog)
 - ▶ *etc.*

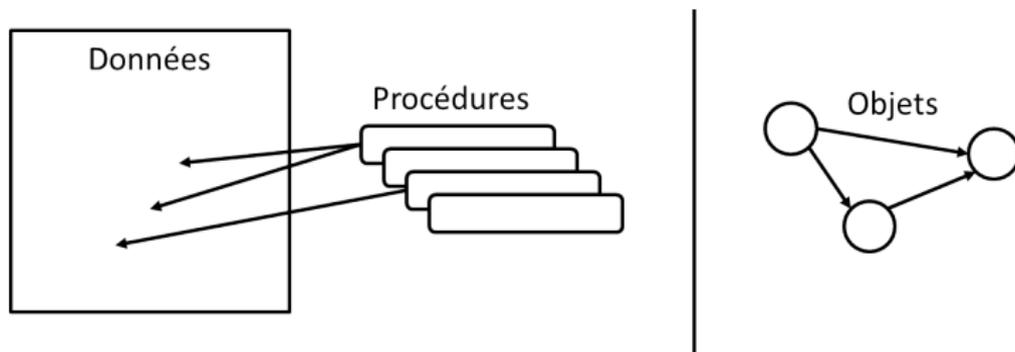
Programmation impérative et modulaire

- Programmation impérative
 - ▶ Un programme est une séquence d'instructions exécutées par un ordinateur pour modifier son état
 - ▶ Instructions : affectations, séquences, structures conditionnelles et itératives
- Programmation modulaire
 - ▶ Un programme est décomposé en éléments plus simples afin de faciliter son développement et permettre la réutilisation
 - ★ Principe diviser pour mieux régner
 - ▶ Éléments : procédures, fonctions, modules, unités
- Exemples de langage : Pascal, C...

Programmation objet

- Programmation objet
 - ▶ Un programme est un ensemble d'objets qui interagissent
 - ▶ Reprend et prolonge la démarche modulaire : décomposition d'un problème en parties simples
 - ▶ La programmation des traitements reste impérative
 - ▶ Plus intuitive car s'inspire du monde réel pour une modélisation plus naturelle
 - ▶ Facilite la réutilisation et la conception de grandes applications
- Exemples de langage : Java, C++, C#, Python, PHP5...

Programmation impérative vs programmation objet



Historique des langages de programmation objet

- 1967 : Simula
 - ▶ Langage à classes
- 1972 : Smalltalk
 - ▶ Langage "pur" objet
- 1983 : C++
- 1986 : Eiffel
- 1988 : CLOS (*Common Lisp Object System*)
- 1991 : Python

Historique de Java

- Début des années 90 : langage Oak (chêne)
 - ▶ Créé par James Gosling (*Sun Microsystems*)
 - ▶ Destiné à la programmation des systèmes embarqués
 - ▶ Objectif principal : améliorer le C++
 - ▶ Rebaptisé Java en 1994
- 1995 : Java 1.0
 - ▶ Lien avec le Web (applet)
 - ▶ Versions 1.1 (JavaBeans, RMI, JDBC)
- 1998 : Sun appelle Java 2 les versions de Java
 - ▶ 1.2 (Swing, optimisation JVM, collections), 1.3, 1.4 (assertions, regexp)
- 2004 : Java 5.0 (annotations, types génériques, enum, foreach)
 - ▶ Changement dans le système de numérotation (mais encore JDK 1.5)
- 2006 : Java 6 (amélioration de l'API, intégration SGBD)
- 2010 : Oracle rachète Sun
- 2011 : Java 7
- 2014 : Java 8 (version courante)

Caractéristiques de Java (1)

- Simple et familier
 - ▶ Basé sur C/C++, sans certaines caractéristiques compliquées ou mal utilisées (e.g., pas de pointeur, pas de gestion explicite de la mémoire)
- Orienté objet
 - ▶ Modèle objet propre tout en fournissant un accès à des types primitifs (`int`, `float`, etc.)
 - ★ Approche hybride adoptée pour des raisons de performance qui sont aujourd'hui largement obsolètes
 - ▶ Héritage simple + interfaces
 - ▶ Vaste bibliothèque standard (réutilisation)
- Portabilité du code source et des fichiers binaires (*bytecode*)
 - ▶ *Write once, run anywhere*
 - ★ Un code Java peut s'exécuter partout (i.e., quels que soient le matériel et le système d'exploitation) où il existe une machine virtuelle Java (du moins en théorie)
 - ▶ Bibliothèque standard indépendante
 - ▶ Définition (sémantique) précise du langage

Caractéristiques de Java (2)

- Sûr

- ▶ Fortement typé
 - ★ Vérification de type statique
- ▶ Transtypage contrôlé
- ▶ Contrôle de l'accès à la mémoire
 - ★ Pas de risque d'écrasement, pas de dépassement de tampon
 - ★ Pas d'arithmétique des pointeurs
 - ★ Vérification des bornes d'un tableau
- ▶ Gestion automatique de la mémoire (ramasse-miettes)
 - ★ Pas de fuite mémoire

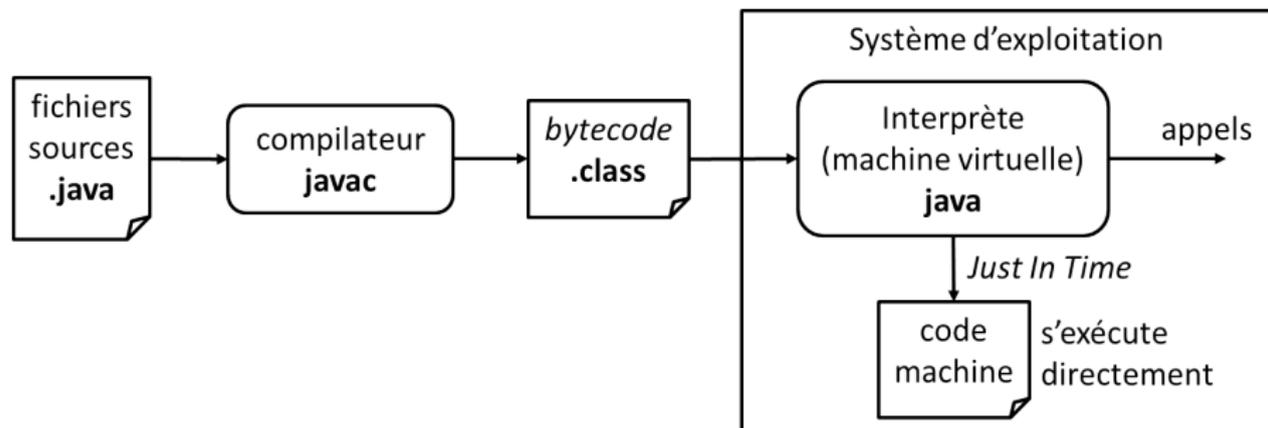
- Sécurisé

- ▶ Interprété, s'exécute sur une machine virtuelle protégée
- ▶ Bac à sable
 - ★ Par exemple, le code d'un applet ne peut pas accéder à la machine (sauf par des moyens clairement définis)
- ▶ Vérification du *bytecode*, signature de code, autorisation d'accès

Caractéristiques de Java (3)

- Dynamique
 - ▶ Chargement de classes et édition des liens dynamiques
 - ★ Permet d'étendre des systèmes à l'exécution
 - ★ Minimise les re-compilations et facilite la modularité
 - ▶ Introspection
 - ★ Capacité d'un programme à examiner et modifier sa structure et son comportement à l'exécution
- Distribué
 - ▶ Applets, servlets, RMI, Corba
- *Multi-threadé*
 - ▶ Exécution parallèle dans le même espace d'adressage

Comment marche Java ?



- Java est un langage compilé et interprété !
- *Bytecode* est un code intermédiaire pour la JVM, indépendant de la plate-forme, qui ne peut pas être directement exécuté par la machine
 - ▶ Quelle que soit la plate-forme (Windows, Linux, MacOS, etc.) :
 - ★ Est obtenu par compilation identique
 - ★ S'exécute à l'identique
 - ▶ Moins performant que le code natif (e.g., .exe) ?

Performances de Java

- Java a souffert des problèmes de performance pendant de nombreuses années par rapport à d'autres langages qui ont été directement compilé pour une plate-forme/machine particulière
 - ▶ Par exemple, C/C++
- Aujourd'hui, l'utilisation de la compilation à la volée (*Just In Time*) a largement éliminé ces problèmes
- La JVM est continuellement améliorée avec de nouvelles techniques
 - ▶ Interfaces de code natif (accès à des bibliothèques C) pour gagner en vitesse si nécessaire
 - ▶ Cache mémoire pour éviter le chargement (et la vérification) multiple d'une même classe
 - ▶ Ramasse-miettes : processus indépendant de faible priorité
- Java fournit d'excellentes performances pour de nombreux *frameworks* dans de nombreux domaines
- Minecraft est développé en Java + OpenGL

Que faut-il pour faire du Java ?

- Un éditeur de texte : emacs, vi, Notepad++, *etc.*
- Un kit de développement : JDK (*Java Development Kit*)
 - ▶ javac : compilateur
 - ▶ java : machine virtuelle pour une plate-forme particulière
 - ▶ javadoc : générateur de documentation HTML
 - ▶ jar : constructeur d'archives
 - ▶ jdb : débogueur
 - ▶ *etc.*
- Des outils d'automatisation : Ant, Makefile, *etc.*
- Un environnement de développement (IDE - *Integrated Development Environment*) : Eclipse, IntelliJ, Netbeans, *etc.*

Différentes plate-formes

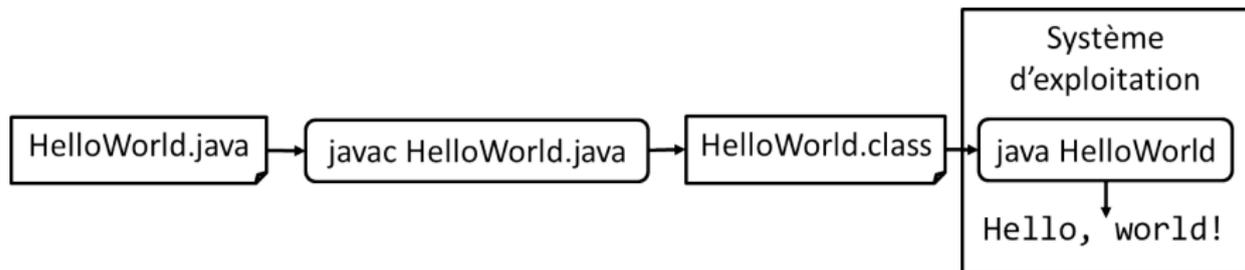
- *Java Platform, Standard Edition* (Java SE)
 - ▶ *Java Runtime Environment* (JRE) : environnement d'exécution
 - ★ Java API, JVM, etc. pour exécuter une application/applet Java
 - ▶ *Java Development Kit*(JDK) : kit de développement
 - ★ JRE + outils de développement (compilateur, etc.)
- *Java Platform, Enterprise Edition* (Java EE)
 - ▶ Développement d'applications d'entreprise multi-couches (client/serveur) orientées composants (JavaBeans), services Web (servlet, JSP, XML), etc.
 - ▶ Inclus Java SE
- *Java Platform, Micro Edition* (Java ME)
 - ▶ Développement d'applications pour les téléphones mobiles, PDA et autres systèmes embarqués
 - ▶ Optimisé pour la mémoire, la puissance de traitement et les entrées/sorties

Premier programme : HelloWorld.java

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello ,_world!");  
    }  
}
```

- Une classe par fichier
- Le nom de la classe est le même que celui du fichier

Compilation et exécution



- **Compilation** : `> javac HelloWorld.java`
 - ▶ Détermine les dépendances et compile tous les fichiers nécessaires
 - ★ Il suffit donc de compiler la classe principale
 - ▶ Produit autant de fichiers `.class` qu'il y a de classes (ici, `HelloWorld.class`)
- **Exécution** : `> java HelloWorld`
 - ▶ Lance la JVM en exécutant la méthode `main` de la classe `HelloWorld`
 - ★ Attention à ne pas mettre d'extension derrière le nom de la classe !
 - ★ Peut être suivie d'arguments
 - ▶ Affiche dans la console : `Hello, world!`

Classpath

- Par défaut, les outils du JDK cherchent les classes dans le répertoire courant
- Si les classes sont dans plusieurs répertoires, utiliser le classpath :
 - ▶ Soit avec l'option `-classpath` des outils du JDK
 - ▶ Soit avec la variable d'environnement `CLASSPATH`
- Nécessaire dès que des bibliothèques (e.g., JUnit, Log4J) qui sont dans des répertoires ou fichiers d'archive (.jar) propres sont utilisées
- Exemples :
 - ▶ Unix : `javac -classpath /foo/junit.jar:. HelloWorld.java`
 - ▶ Windows : `java -classpath \foo\junit.jar;. HelloWorld`
 - ▶ Les classes sont cherchées dans `junit.jar`, puis dans le répertoire courant (`.`)

La méthode principale `main`

```
public static void main(String [] args) {  
    ...  
}
```

- Est publique et statique
- Ne retourne pas de valeur (`void`)
- Prend un seul paramètre : un tableau de chaîne de caractères correspondant aux arguments de la ligne de commande
- Est le point de départ de l'exécution du programme
- Chaque classe peut ou non définir sa méthode principale

Un peu de lecture

- James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, Addison-Wesley, 3rd Edition, 2005
- Kathy Sierra and Bert Bates, *Head First Java*, O'Reilly Media, 2nd Edition, 2005
- Bruce Eckel, *Thinking in Java*, Prentice-Hall, 4th Edition, 2006
- Joshua Bloch, *Effective Java*, Addison-Wesley, 2nd Edition, 2008
- Ben Evans and David Flanagan, *Java in a Nutshell*, O'Reilly Media, 6th Edition, 2014
- Java API : <http://docs.oracle.com/javase/7/docs/api/>

Au menu

1 Introduction

2 Éléments syntaxiques de base

3 Classes et objets

Commentaires

```
public class HelloWorld {  
    /*  
     * Ceci est un commentaire en bloc  
     */  
    public static void main(String[] args) {  
        // Ceci est un commentaire en pleine ligne  
        System.out.println("Hello ,_world!"); // fin de ligne  
    }  
}
```

- Les commentaires en bloc peuvent s'étendre sur plusieurs lignes
 - ▶ À utiliser avant des classes ou des méthodes
- Les commentaires sur une ligne s'étendent jusqu'à la fin de la ligne
 - ▶ À utiliser à l'intérieur des méthodes
- Rappel : les commentaires ne doivent pas paraphraser le code

Javadoc

- Outil fourni dans le JDK : `> javadoc HelloWorld.java`
- Permet de produire automatiquement une documentation des classes Java au format HTML à partir des commentaires de leur code source
 - ▶ La documentation est directement rédigée dans le code source Java
 - ★ Facilite sa mise à jour
 - ★ Favorise (mais ne garantit pas) sa cohérence
- Permet une présentation standardisée de la documentation des classes Java

Commentaires structurés

```
/**  
 * Exemple de documentation de classe avec <i>javadoc</i>  
 * @author JML  
 * @version 1.0  
 */  
public class HelloWorld {  
    ...  
}
```

- Sont exploités par l'outil javadoc
- Sont placés avant l'élément (*i.e.*, classe, méthode, attribut) à documenter
- Peuvent contenir :
 - ▶ Des tags commençant par @
 - ★ @author : nom du développeur
 - ★ @param : documente un paramètre de méthode
 - ★ @return : documente la valeur de retour
 - ★ *etc.*
 - ▶ Des éléments HTML

Exemple de Javadoc

All Classes
Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Class HelloWorld

java.lang.Object
HelloWorld

```
public class HelloWorld
extends java.lang.Object
```

Exemple de documentation de classe avec *javadoc*

Version:
1.0

Author:
JML

Constructor Summary

Constructors

| Constructor and Description |
|-----------------------------|
| HelloWorld() |

Method Summary

Methods

| Modifier and Type | Method and Description |
|-------------------|-------------------------------|
| static void | main(java.lang.String[] args) |

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Types primitifs

- `boolean` (1 octet) : `true` et `false`
 - ▶ En Java, aucun type ne peut être casté en booléen (`int` inclus)
 - ▶ Un booléen ne peut pas non plus être casté en un autre type
- Entier non signé
 - ▶ `char` (2 octets) : 0 à 0xffff (pour les caractères Unicode)
- Entier signé
 - ▶ `byte` (1 octet) : -128 à 127
 - ▶ `short` (2 octets) : -32768 à 32767
 - ▶ `int` (4 octets) : -2147483648 à 2147483647
 - ▶ `long` (8 octets) : -9223372036854775808 à 9223372036854775807
- Nombre à virgule flottante signé
 - ▶ `float` (4 octets) : 2^{-149} à $2^{128} - 2^{104}$
 - ▶ `double` (8 octets) : 2^{-1074} à $2^{1024} - 2^{971}$

Autres types

- Type référence
 - ▶ Fait référence à un objet en mémoire (pas un type primitif)
 - ▶ Peut être `null` si la référence ne se réfère à rien
 - ▶ Exemples : `String`, `Integer`
- En Java, les tableaux sont aussi des types référence
 - ▶ `int[] numArray; // à préférer`
 - ▶ `int numArray[]; // marche aussi`

Littéraux

- Un booléen est simplement `true` ou `false`
- Une valeur entière est directe
 - ▶ `int i = 22`
- Toutefois, un littéral de type `long` utilise le suffixe `"L"`
 - ▶ `long secondsInYear = 31556926L;`
 - ▶ Éviter le `"l"` minuscule car ressemble à un 1 dans beaucoup de polices
- Le type par défaut d'une valeur décimale est `double`
 - ▶ `double pi = 3.14159265358979323;`
- Un littéral de type `float` utilise le suffixe `"F"`
 - ▶ `float goldenRatio = 1.618f;`
 - ▶ `"F"` ou `"f"` convient

Caractères et chaînes de caractères

- Les caractères peuvent être des caractères entre guillemets simples ou des nombres entre 0 et 65535
 - ▶ `char capA = 'A';` // à préférer
 - ▶ `char capA = 65;` // plus dur à maintenir
 - ▶ `'A' + 20 = ?`
- Les chaînes de caractères sont entre guillemets doubles
 - ▶ `String name = "Toto";`
- Les caractères spéciaux doivent être protégés
 - ▶ `String msg = "Il a dit : \"Java, c'est super!\"";`
 - ▶ Caractères spéciaux les plus utiles :
 - ★ `\t` (tabulation), `\r` (retour charriot), `\n` (nouvelle ligne),
`\\` (*backslash*), `\'` (guillemet simple), `\"` (guillemet double)

Convention de nommage

- Améliore la lisibilité des programmes
 - ▶ Utiliser des noms d'identifiant significatifs!
- Noms doivent commencer par une lettre et peuvent inclure uniquement des lettres et des chiffres
 - ▶ `_` et `$` sont considérés comme des "lettres" en Java
 - ▶ Ne pas utiliser `$` car il est utilisé par le compilateur pour les noms auto-générés
- Les majuscules sont très importantes dans le style de codage Java
 - ▶ Les attributs et les méthodes commencent par une minuscule, puis une majuscule à l'initiale de chaque mot d'un nom composé
 - ★ `exampleOfMethodName`
 - ▶ Les classes et les interfaces commencent par une majuscule, puis une majuscule à l'initiale de chaque mot d'un nom composé
 - ★ `ExampleOfClassName`
 - ▶ Les noms des *packages* doivent être tout en minuscules
 - ★ `java.lang`

Déclaration et initialisation de variables

- La déclaration des variables est similaire au C/C++ :

```
int i;  
boolean error = false;  
String name = "Toto";
```

- Les variables locales n'ont pas de valeur initiale par défaut

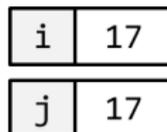
```
int i;  
i = i + 1;
```

⇒ Erreur à la compilation : la variable `i` n'a pas été initialisée
(En C/C++, ce code compilerait sans erreur)

Variables de type primitif et référence

- La différence entre les types primitif et référence est où est réellement stockée la valeur
- Type primitif :

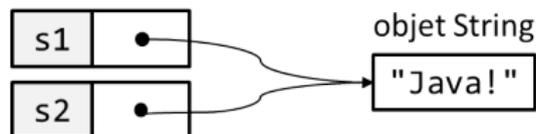
```
int i = 17;
int j = i;
```



- ▶ Chaque variable stocke sa propre valeur (pile)

- Type référence :

```
String s1 = "Java!";
String s2 = s1;
```



- ▶ La valeur est stockée dans la mémoire principale (tas)
- ▶ Chaque variable contient l'adresse en mémoire de l'objet
- ▶ Les deux variables font référence au même objet

Opérateurs arithmétiques et de comparaison

- Même ensemble d'opérateurs qu'en C/C++
- Arithmétique simple : + - * / %
- Affectation composée : += -= *= /= *etc.*
- Incrémentation/décrémentation : ++ -- (pré et post)

```
int i = 5;  
int j = ++i; // j = 6, i = 6  
int k = i++; // k = 6, i = 7
```

- Comparaison : == != > >= < <=
 - ▶ Ces opérateurs produisent des valeurs booléennes

Opérateurs de logique booléenne

- Encore comme en C/C++
 - ▶ `&&` (ET logique) `||` (OU logique) `!` (NON logique)
- Ces opérateurs requiert des valeurs booléennes et produisent des valeurs booléennes
- Évaluation paresseuse (*lazy evaluation*)
 - ▶ `name != null && name.equals("Toto");`
 - ★ `name.equals("Toto")` uniquement évalué si `name != null`
 - ▶ Réciproquement, `name == null || !name.equals("Toto");`
- Ordre de priorité : `!` `&&` `||`

Opérateur de chaîne de caractères (String)

- Concaténation : + (comme l'opérateur d'addition)

```
String name = "Toto";
System.out.println("Hello_" + name);
```

- Au moins un opérande doit être une chaîne de caractères pour que l'opérateur + soit l'opérateur de concaténation et non d'addition
 - ▶ L'opérateur + est évalué de gauche à droite

```
int i = 5;
int j = 7;
System.out.println("i_" + i); // Affiche "i = 5"
System.out.println(i + j); // Affiche "12"
System.out.println("i_+_j_" + i + j); // "i + j = 57"
System.out.println(i + j + "_=" + i + j); // "12 = i + j"
```

Flot de contrôle

- Instructions conditionnelles et itératives quasiment identiques au C/C++

```

if (condition)
    instruction ;
else if (condition)
    instruction ;
else
    instruction ;
  
```

```

while (condition)
    instruction ;

do
    instruction ;
while (condition);
  
```

- Différence : *condition* doit produire une valeur booléenne!
- Les blocs d'instructions sont mis entre accolades, comme en C/C++

```

if (condition) {
    instruction1 ;
    instruction2 ;
    ...
}
  
```

Instruction conditionnel `switch`

```
switch (expression scalaire) {  
  case valeur1 :  
    instructions ;  
    break ;  
  case valeur2 :  
    instructions ;  
    break ;  
  ...  
  default :  
    instructions ;  
}
```

- *expression scalaire* doit être un entier signé ou non (caractère)
- Ne marche pas avec des chaînes de caractères
- Si `break` est omis, les instructions du `case` suivant sont aussi exécutées
 - ▶ Factorisation de traitement

Opérateur conditionnel ternaire

`condition ? valeur_vrai : valeur_faux`

- *condition* doit produire une valeur booléenne
- Si *condition* est vrai, le résultat retourné est *valeur_vrai*, sinon c'est *valeur_faux*
 - ▶ Exemple : `statut = (age >= 18) ? "majeur" : "mineur"`

Boucle for (1)

- Très similaire au C++
 - ① Initialiser (et possiblement déclarer) une ou plusieurs variables de boucle
 - ② Tester certaines conditions avant chaque itération de la boucle
 - ③ Appliquer une ou plusieurs mises à jour aux variables de boucle

```

for (init; condition; update) statement;
for (init; condition; update) {
    statement1;
    ...
}

```

- Équivalente à une boucle `while`, mais en plus compacte

```

int i = 1;
while (i <= 10) {
    sum += i;
    i++;
}

for (i = 1; i <= 10; i++)
    sum += i;

```

Boucle for (2)

- Peut spécifier plusieurs valeurs initiales

```
int i, sum;  
for (i = 1, sum = 0; i <= 10; i++)  
    sum += i;
```

- Peut déclarer les variables de boucle directement dans la boucle for

```
int sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;
```

- ▶ i est uniquement visible à l'intérieur de la boucle for
- ▶ Autrement dit, la portée de i est limitée à la boucle for

Boucle for (3)

- Peut spécifier plusieurs opérations de mise à jour

```
int sum = 0;  
for (int i = 1; i <= 10; sum += i , i++) /* rien */ ;
```

- ▶ La boucle for n'a pas besoin d'un corps!

- Encore plus compacte

```
int sum = 0;  
for (int i = 1; i <= 10; sum += i++) /* rien */ ;
```

- ▶ Difficile à maintenir, mieux vaut éviter!

Au menu

1 Introduction

2 Éléments syntaxiques de base

3 Classes et objets

Terminologie : classes et objets

- Java est un langage de programmation objet
 - ▶ Les programmes Java sont entièrement composés de classes
- Un objet
 - ▶ A un état
 - ★ Ensemble de valeurs de données (attributs) qui caractérisent l'objet
 - ▶ A un comportement
 - ★ Ensemble d'opérations (méthodes) qui manipulent ces données d'une manière cohérente
 - ▶ A une identité
 - ★ Permet de s'adresser à l'objet
 - ★ Est unique (deux objets différents ont des identités différentes)
 - ★ Peut être nommée pour faire référence à l'objet
 - ★ Plusieurs références possibles pour une seule identité (*i.e.*, un seul objet)
 - ▶ Est une instance d'une classe
 - ★ N'a de réalité qu'à l'exécution du programme
- Une classe
 - ▶ Est un moule à objets
 - ▶ Définit l'état et le comportement des objets de cette classe
 - ▶ Définit un nouveau type dans le langage

Terminologie : attributs et méthodes

- Une classe est composée de membres
- Les attributs sont des variables typées associées à la classe
 - ▶ Ils stockent l'état de la classe qui peut évoluer dans le temps
- Les méthodes sont des opérations que la classe peut effectuer
 - ▶ Elles spécifient le comportement de la classe
 - ▶ Elles impliquent généralement, mais pas toujours, les attributs de la classe

Méthodes Java

| | | | | |
|-------------------------|----------------------------|-------------------|----------------------|-------------------------|
| <u>public</u> | <u>static</u> | <u>void</u> | <u>main</u> | <u>(String[] args)</u> |
| modificateur d'accès | modificateur de méthode | type de retour | nom de la méthode | liste des paramètres |

- Retournent une valeur du type spécifié
- Ou ne retournent pas de valeur, indiqué par le mot clé `void`
- Peuvent prendre un nombre quelconque d'arguments/paramètres
 - ▶ "Aucun argument" est indiqué avec des parenthèses vides `()`, et non avec `void`
- La signature d'une méthode inclut son nom et sa liste de paramètres (les types)
- Peuvent être associées à des modificateurs
 - ▶ Tout comme les attributs
- Le corps (*i.e.*, le code) d'une méthode est son implémentation

Modificateurs d'accès

- Peuvent être utilisés sur des classes, des méthodes et des attributs
- Quatre modificateurs d'accès en Java :
 - ▶ `public` : n'importe qui peut y accéder
 - ▶ `protected` : les classes du même *package* et les sous-classes (dérivées) peuvent y accéder (cf. héritage)
 - ▶ Niveau d'accès par défaut si aucun modificateur n'est spécifié : seules les classes du même *package* peuvent y accéder
 - ★ Appelé accès *package-private* (cf. *package*)
 - ▶ `private` : seule la classe peut y accéder
- Protégez les détails d'implémentation en utilisant des modificateurs d'accès dans votre code !
 - ▶ Masquage d'information

Abstraction et encapsulation

Concepts clé de la programmation objet

- Abstraction

- ▶ Présenter une interface propre et simplifiée
- ▶ Cacher les détails inutiles aux utilisateurs de la classe (e.g., les détails d'implémentation)
 - ★ En général, ils ne se soucient pas de ces détails
 - ★ Mieux vaut qu'ils se concentrent sur le problème qu'ils sont en train de résoudre

- Encapsulation

- ▶ Permettre à un objet de protéger son état interne des accès externes et des modifications
- ▶ L'objet contrôle lui-même tous les changements d'état interne
 - ★ En déclarant l'état `private`, il ne pourra être modifié que *via* les méthodes `public`
 - ★ Les méthodes garantissent les changements d'état valides (contrôle de la cohérence)

4 images, 1 mot



Quelles caractéristiques ? Quels comportements ?

Exemple : Car.java

```
public class Car {  
    private String make;  
    private String model;  
    private String numberPlate;  
    private short speed;  
    private int nbMiles;  
    ...  
  
    public void start() {...}  
    public void stop() {...}  
    public void move(short s) {...}  
    public boolean isMoving() {...}  
    ...  
}
```

- Variables d'instance
 - ▶ Déclarées en début de classe
 - ▶ En dehors de toute méthode
 - ▶ Non publiques
- Méthodes d'instance

Exemple : Car.java

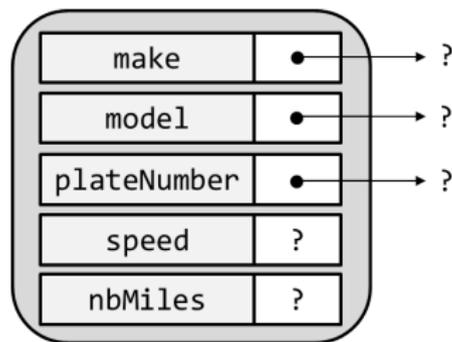
```

public class Car {
    private String make;
    private String model;
    private String numberPlate;
    private short speed;
    private int nbMiles;
    ...

    public void start() {...}
    public void stop() {...}
    public void move(short s) {...}
    public boolean isMoving() {...}
    ...
}

```

- Variables d'instance
 - ▶ Déclarées en début de classe
 - ▶ En dehors de toute méthode
 - ▶ Non publiques
- Méthodes d'instance



Méthodes spéciales : les constructeurs

- Créent de nouvelles instances d'une classe
 - ▶ Initialisent toutes les variables d'instance de manière cohérente
- Portent le nom de la classe
- Peuvent prendre des arguments, mais ce n'est pas obligatoire
- N'ont pas de type de retour (pas même `void`)
- Toutes les classes ont au moins un constructeur
 - ▶ Par défaut (si aucun constructeur n'est défini), la classe a un constructeur sans paramètre qui ne fait rien
- Pas de destructeur en Java !
 - ▶ Ramasse-miettes (*garbage collector*)

Exemple : Car.java

```
public class Car {  
    private String make;  
    private String model;  
    private String numberPlate;  
    private short speed;  
    private int nbMiles;  
    public Car(String mk, String mdl, String np) {  
        make = mk;  
        model = mdl;  
        numberPlate = np;  
        speed = 0;  
        nbMiles = 0;  
    }  
    public Car(String mk, String mdl) {  
        this(mk, mdl, ""); // Chainage de constructeurs  
    }  
    ...  
}
```

Méthodes spéciales : les accesseurs en lecture et en écriture

- Accesseurs en lecture (*accessors/getters*)
 - ▶ Permettent de récupérer les données internes (*i.e.*, l'état de l'objet)
 - ▶ Permettent de contrôler comment les données sont exposées
- Accesseur en écriture (*mutators/setters*)
 - ▶ Permettent de modifier les données internes (*i.e.*, l'état de l'objet)
 - ▶ Permettent de contrôler comment et quand des modifications peuvent être effectuées
- Les classes n'ont pas toutes des accesseurs en lecture et en écriture

Exemple : Car.java

```
public class Car {  
    private String make;  
    private String model;  
    private String numberPlate;  
    private short speed;  
    private int nbMiles;  
    ...  
    // Accesseurs en lecture  
    public String getMake() { return make; }  
    public String getModel() { return model; }  
    ...  
    // Accesseurs en ecriture  
    public void setNumberPlate(String np) { numberPlate = np; }  
    public void setSpeed(short s) {  
        speed = (s >= 0) ? s : 0;  
    }  
    ...  
}
```

Convention de nommage des accesseurs

- Les accesseurs en écriture commencent généralement par set
 - ▶ `void setNumberPlate(String)`
 - ▶ `void setSpeed(short)`
- Les accesseurs en lecture commencent généralement par get
 - ▶ `String getMake()`
 - ▶ `String getModel()`
- Les accesseurs en lecture qui retournent un booléen commencent souvent par is
 - ▶ `boolean isStarted()`
 - ▶ `boolean isMoving()`
- Des exceptions sont autorisées lorsque is n'a pas de sens
 - ▶ `boolean contains(Occupant)`
 - ▶ `boolean intersects(Object)`

S'affranchir de la représentation mémoire

- Variables d'instance `private`
- Les constructeurs accèdent directement aux variables d'instance
- Les accesseurs accèdent directement aux variables d'instance
- Les autres méthodes (appelées services) utilisent les accesseurs pour accéder (indirectement) à l'état de l'objet
 - ▶ Elles demeurent correctes lors d'un changement de représentation mémoire

```
public class Car {  
    ...  
    public void move(short s) {  
        setSpeed(s);  
        ...  
    }  
    public boolean isMoving() { return getSpeed() > 0; }  
    ...  
}
```

Utiliser un objet

- Créer un nouvel objet en utilisant l'opérateur `new`

```
Car c0 = new Car(); // erreur  
Car c1 = new Car("DeLorean", "DMC-12", "OUTATIME");  
Car c2 = new Car("Volkswagen", "Coccinelle");
```

- Faire appel aux méthodes de l'objet

```
c1.move(88);  
System.out.println("c1 est une " + c1.getMake() + " "  
    + c1.getModel() + " et roule à " + c1.getSpeed()  
    + " miles à l'heure");
```

Objets et références

- Qu'est-ce que c1 et c2 ?
 - ▶ Ce sont des références à des objets Car
 - ▶ Ce ne sont pas les objets eux-mêmes

- Jongle de références :

```
Car c3 = c1; // Il n'y a toujours que deux objets  
c1 = null; // Les deux objets sont encore accessibles  
c2 = null; // Un des objets n'est plus accessible
```

- La JVM suit les objets qui ne sont plus accessibles
 - ▶ Si un objet n'est désigné par aucune référence, le ramasse-miettes libère l'espace qu'il occupe

Constructeurs par copie

- Copie superficielle

- ▶ L'objet initialisé partage potentiellement des données avec un autre

```
private Engine engine;  
public Car(Car c) {  
    ...  
    engine = c.engine;  
}
```

- Copie profonde

- ▶ L'objet initialisé a sa propre copie de l'information, indépendante de tout autre objet

```
private Engine engine;  
public Car(Car c) {  
    ...  
    engine = new Engine(c.engine.getXXX(), ...);  
}
```

Arguments objet d'une méthode

- Que se passe-t-il lorsque un objet est passé en paramètre d'une méthode ?
 - ▶ Exemple : `public static void foo(Car c)`
- Pour rappel, `c` est une référence à l'objet
- La référence est copiée dans `c`, mais pas l'objet `Car` auquel il réfère
 - ▶ Passage par copie de la référence
 - ▶ Les variables de type primitif sont quant à elles passées par valeur
- Des effets de bord et des erreurs peuvent alors facilement arriver !

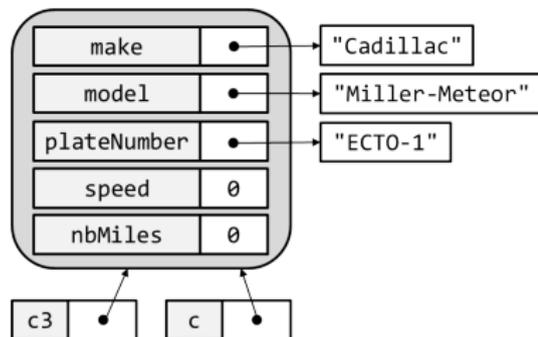
Passer des objets (1)

```

public static void foo(Car c) {
    System.out.println("Plaque: "
        + c.getNumberPlate());
    c.setNumberPlate("ECTO-2"); //??
}

public static void main(String[] a) {
    Car c3 = new Car("Cadillac",
        "Miller-Meteor", "ECTO-1");
    foo(c3);
}

```



Passer des objets (2)

```

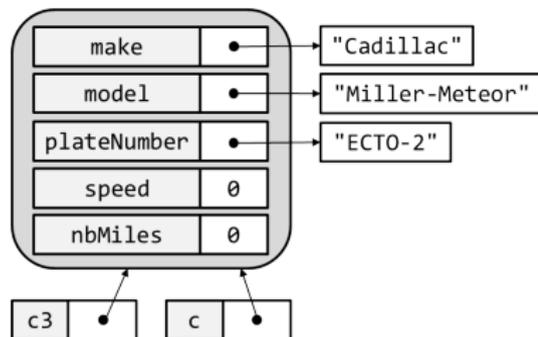
public static void foo(Car c) {
    System.out.println("Plaque: "
        + c.getNumberPlate());
    c.setNumberPlate("ECTO-2");
    c = null; //??
}

```

```

public static void main(String[] a) {
    Car c3 = new Car("Cadillac",
        "Miller-Meteor", "ECTO-1");
    foo(c3);
}

```



Passer des objets (3)

```

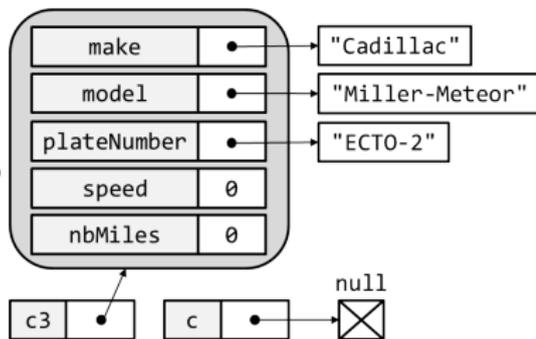
public static void foo(Car c) {
    System.out.println("Plaque: "
        + c.getNumberPlate());
    c.setNumberPlate("ECTO-2");
    c = null;
    c = new Car("Ford", "Explorer"); //??
}

```

```

public static void main(String[] a) {
    Car c3 = new Car("Cadillac",
        "Miller-Meteor", "ECTO-1");
    foo(c3);
}

```



Passer des objets (4)

```

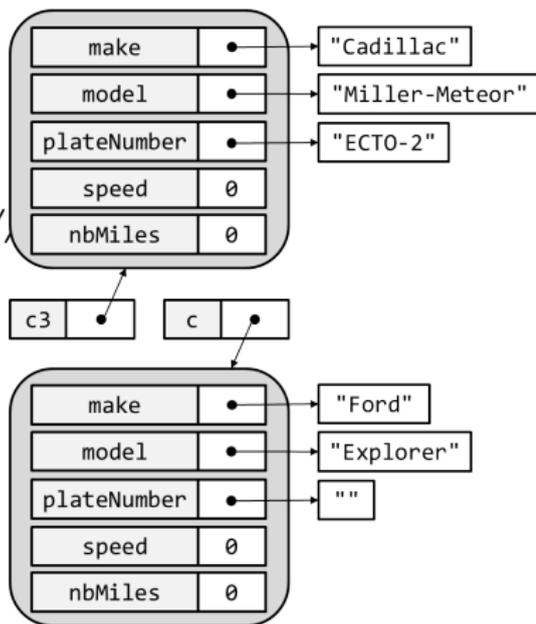
public static void foo(Car c) {
    System.out.println("Plaque: "
        + c.getNumberPlate());
    c.setNumberPlate("ECTO-2");
    c = null;
    c = new Car("Ford", "Explorer");
    c.setNumberPlate("Jurassic Park");
}

```

```

public static void main(String[] a) {
    Car c3 = new Car("Cadillac",
        "Miller-Meteor", "ECTO-1");
    foo(c3);
}

```



Passer des objets (5)

```

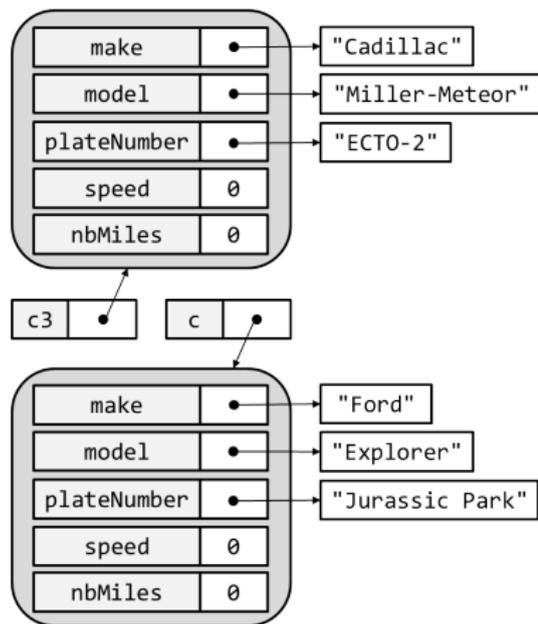
public static void foo(Car c) {
    System.out.println("Plaque: "
        + c.getNumberPlate());
    c.setNumberPlate("ECTO-2");
    c = null;
    c = new Car("Ford", "Explorer");
    c.setNumberPlate("Jurassic Park");
}

```

```

public static void main(String[] a) {
    Car c3 = new Car("Cadillac",
        "Miller-Meteor", "ECTO-1");
    foo(c3);
}

```



Moralité

- Il faut être très prudent avec les références d'objet
 - ▶ Si une méthode modifie accidentellement un objet, cela peut être très difficile à retrouver
- Une solution : rendre les objets immuables
 - ▶ Java n'a pas d'équivalent au mot clé `const` de C++
 - ▶ Un objet est immuable s'il ne fournit pas d'accessor en écriture
 - ★ Définir l'état de l'objet à sa construction
 - ★ Ne fournir aucun moyen de modifier l'état

Mot clé `this` (1)

- Les méthodes d'instance ont un paramètre implicite `this` qui est une référence à l'objet sur lequel elles sont appelées (*i.e.*, l'objet receveur)
- À ne pas confondre avec `this(...)` qui permet l'appel d'un autre constructeur de la même classe (chaînage de constructeurs)
- Est implicitement utilisé lorsque les attributs ou les méthodes d'instance sont accédés à l'intérieur d'une autre méthode

```
public short getSpeed() {  
    return speed; // Identique a "return this.speed;"  
}  
public void move(short s) {  
    setSpeed(s); // Identique a "this.setSpeed(s)"  
    ...  
}
```

Mot clé `this` (2)

- Permet de résoudre des ambiguïtés
 - ▶ Par exemple, si le nom d'un paramètre est le même que celui d'un attribut
 - ★ Ce qui est généralement le cas dans les constructeurs ou encore les accesseurs en écriture
 - ▶ En général, il faut éviter les ambiguïtés inutiles qui peuvent mener à des erreurs très subtiles...

```
void setNumberPlate(String numberPlate) {  
    // numberPlate est le parametre  
    // this.numberPlate est l'attribut de l'objet  
    this.numberPlate = numberPlate;  
}
```

Méthodes statiques

- Aussi appelées méthodes de classe
- Sont appelées sur la classe
 - ▶ Par exemple, la classe `Math` de Java a uniquement des méthodes statiques

```
public static double atan2(double y, double x);  
double tangent = Math.atan2(yComp, xComp);
```

- Ne nécessitent pas une instance particulière pour être appelées
 - ▶ Ne peuvent donc pas utiliser la référence `this`

Attributs statiques

- Aussi appelées variables de classe
- Servent à représenter des données générales qui ne sont pas liées à une instance particulière
 - ▶ Un seul exemplaire est stocké au niveau de la classe
 - ▶ Pas de duplication pour chaque instance
- Sont accédés en préfixant avec le nom de la classe
 - ▶ `System.out`

Console Java : entrée/sortie

- `System.out` est le flot de sortie standard
 - ▶ `System.out.println(...)` va à la ligne
 - ▶ `System.out.print(...)` reste sur la même ligne
- `System.err` est le flot d'erreur standard
 - ▶ À utiliser pour signaler des erreurs
- `System.in` est le flot d'entrée standard

```
try {  
    BufferedReader buffer = new BufferedReader(  
        new InputStreamReader(System.in));  
    String str = buffer.readLine();  
} catch (IOException e) {}
```

System.out.println()

- Accepte différents types de paramètre :
 - ▶ `System.out.println(String x)`
 - ▶ `System.out.println(boolean x)`
 - ▶ `System.out.println(char x)`
 - ▶ `System.out.println(float x)`
 - ▶ `System.out.println(int x)`
 - ▶ `System.out.println(Object x)`
 - ★ Fait appel à la méthode `toString` de la classe `Object`
 - ▶ `System.out.println()`
 - ▶ Et quelques-unes de plus...
- Ce sont des méthodes surchargées
 - ▶ Même nom, mais signature différente

Méthode toString

- Est automatiquement appelée lorsqu'un objet doit être converti en chaîne de caractères (String)
 - ▶ Par exemple, dans une concaténation de chaîne de caractères :
 - ★ `String msg = "La voiture est une " + c;` est automatiquement traduit par le compilateur comme `String msg = "La voiture est une " + c.toString();`
- Par défaut, toutes les classes ont une méthode `toString` qui retourne une chaîne de caractères correspondant à l'adresse de l'objet
 - ▶ Héritée de la classe `Object` (cf. héritage)
 - ▶ Si elle n'est pas redéfinie, `c.toString` retournera `"Car@e22a17"`
- Prendre l'habitude de la redéfinir

@Override

```
public String toString() {
    return getMake() + " " + getModel()
        + " immatriculée " + getNumberPlate();
}
```

Égalité

- Pour les types primitifs, `==` compare leur valeur
- Pour les types référence, `==` compare les références elles-mêmes (*i.e.*, si elles désignent le même objet)!

```
Car c1 = new Car("Chevrolet", "Camaro");  
Car c2 = new Car("Chevrolet", "Camaro");  
Car c3 = c1;
```

- ▶ Les voitures `c1` et `c3` sont les mêmes objets
 - ★ `c1 == c3` est vrai
 - ★ `c1 == c2` est faux, même si les valeurs sont les mêmes
- Utiliser la méthode `equals` pour tester l'égalité de deux objets d'un point de vue sémantique

Méthode equals

```
@Override
public boolean equals(Object obj) {
    return ...;
}
```

- Retourne vrai si obj est "égal à" l'objet this
 - ▶ Dépend de ce que la classe représente
 - ▶ Si obj est null, la réponse est toujours faux
- Noter que obj est une référence à un objet générique Object
 - ▶ Il pourrait être de n'importe quel type référence!
 - ▶ L'opérateur instanceof permet de vérifier cela
- Par défaut, toutes les classes ont une méthode equals dont le comportement est équivalent à celui de ==
 - ▶ Héritée de la classe Object (cf. héritage)
- Prendre l'habitude de la redéfinir en fournissant une implémentation raisonnable

Est-ce que la méthode equals a du sens ?

- Réflexive
 - ▶ `a.equals(a)` doit retourner vrai
- Symétrique
 - ▶ `a.equals(b)` doit être identique à `b.equals(a)`
 - ▶ Ceci peut être compliqué parfois...
- Transitive
 - ▶ Si `a.equals(b)` est vrai et `b.equals(c)` est vrai alors `a.equals(c)` doit aussi être vrai
- Nullité
 - ▶ `a.equals(null)` doit être faux

Opérateur instanceof

- Permet de tester le type d'un objet (*i.e.*, sa classe)
- Retourne faux si la référence est null
 - ▶ Il n'est donc pas nécessaire de vérifier si le paramètre `obj` de la méthode `equals` est null

Égalité entre deux voitures

```
@Override
public boolean equals(Object obj) {
    // obj est de type Car ?
    // Si non, obj.getMake() est interdit
    if (obj instanceof Car) {
        // Cast vers le type Car, puis compare
        Car c = (Car) obj;
        if (getMake().equals(c.getMake())
            && getModel().equals(c.getModel())) {
            return true;
        }
    }
    return false;
}
```