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

Au menu

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.*

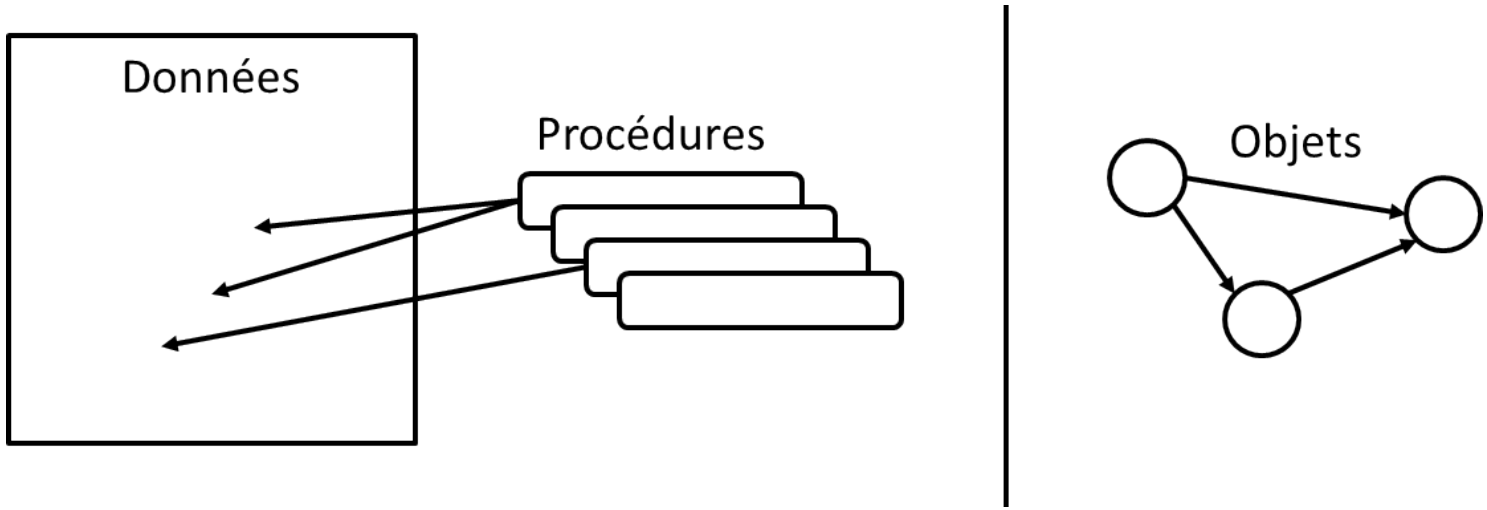
Programmation impérative et modulaire

- 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

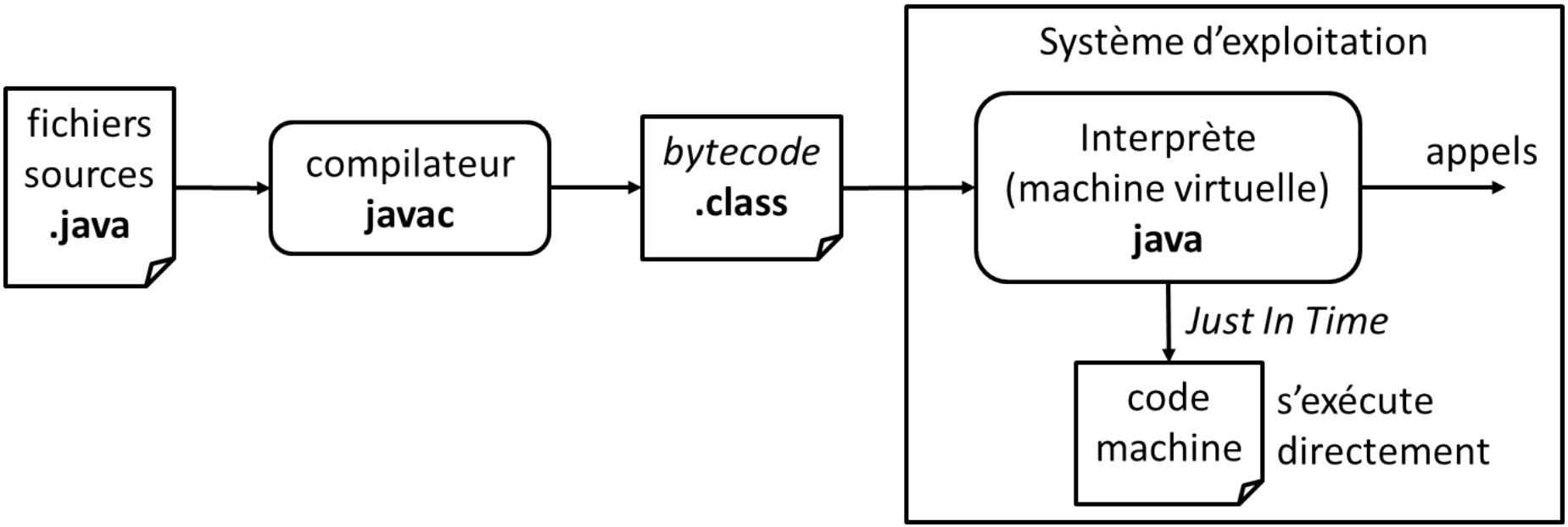
Caractéristiques de Java (2)

- 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

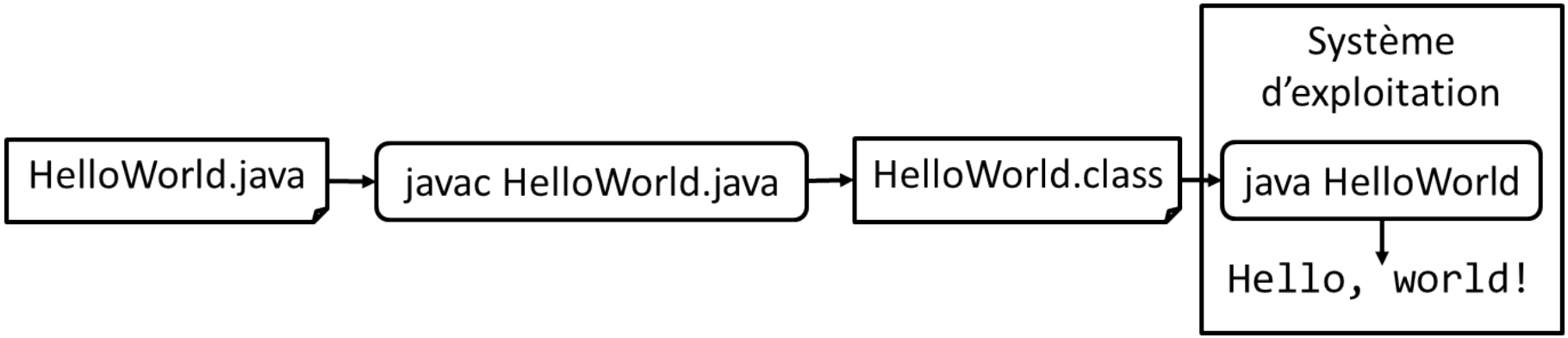
- *Java Platform, Standard Edition* (Java SE)
  - *Java Runtime Environment* (JRE) : environnement d'exécution
    - ★ Java API, JVM, *etc.* pour exécuter une application/applet Java
  - *Java Development Kit*(JDK) : kit de développement
    - ★ JRE + outils de développement (compilateur, *etc.*)
- *Java 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

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

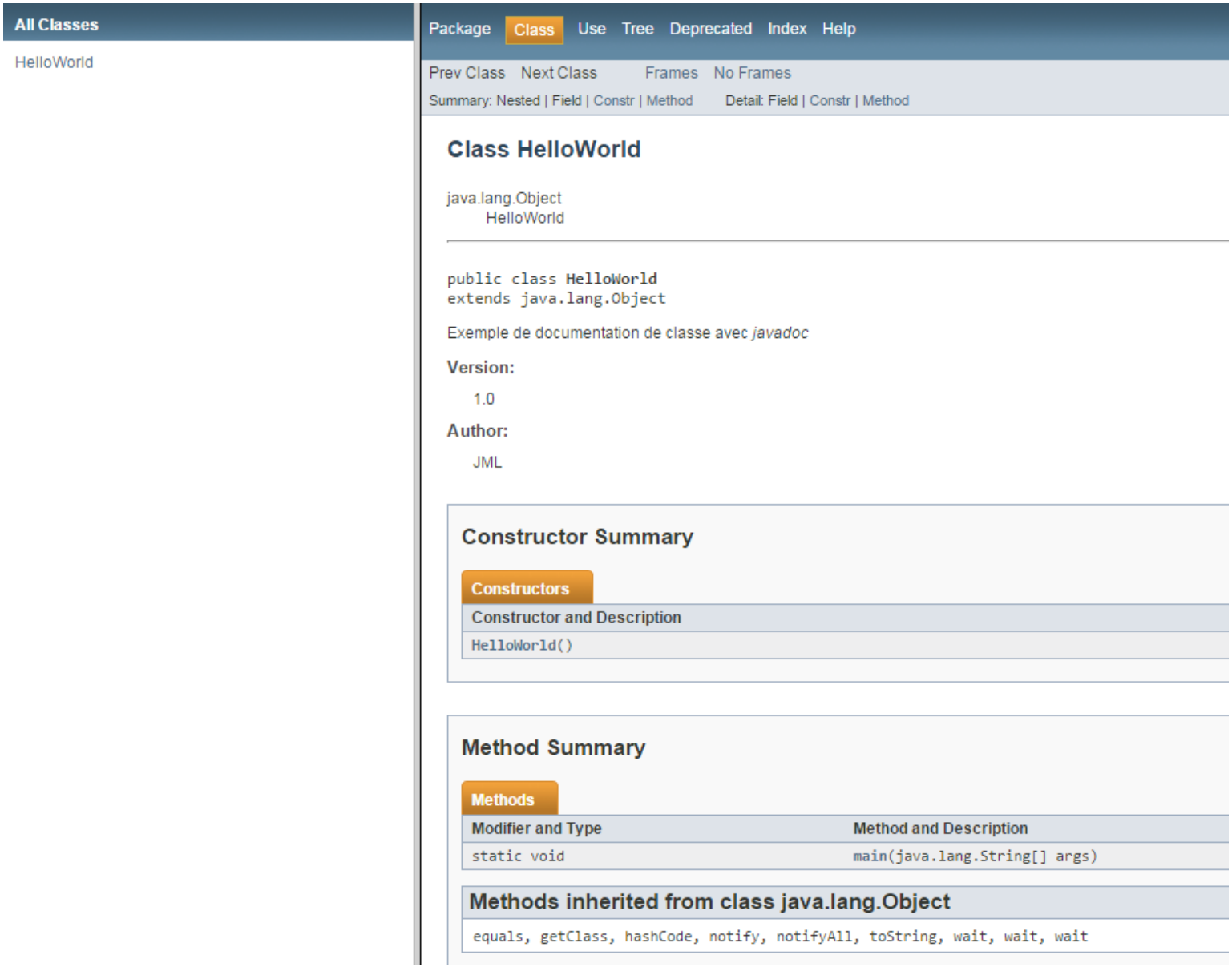
- Outil fourni dans le JDK : `> javadoc HelloWorld.java`
- Permet de produire automatiquement une documentation des classes Java au format HTML à partir des commentaires de leur code source
  - La documentation est directement rédigée dans le code source Java
    - ★ Facilite sa mise à jour
    - ★ Favorise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

HelloWorld

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

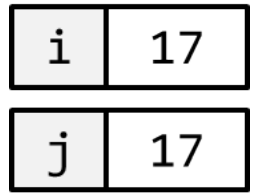
```
int i;
i = i + 1;
```

⇒ 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

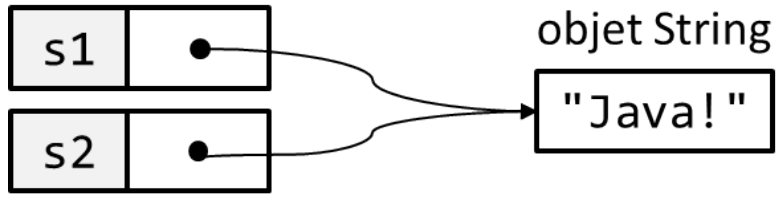
- La différence entre les types primitif et référence est où est réellement stockée la valeur
- Type primitif :

```
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

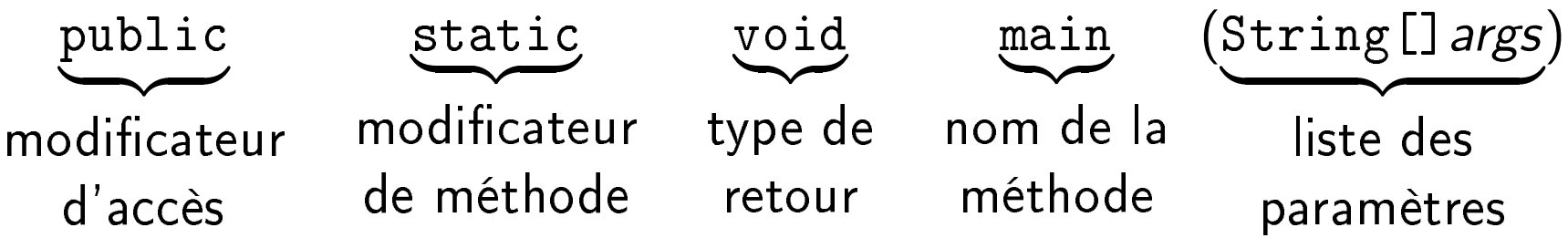
Terminologie : classes et objets

- 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



- 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

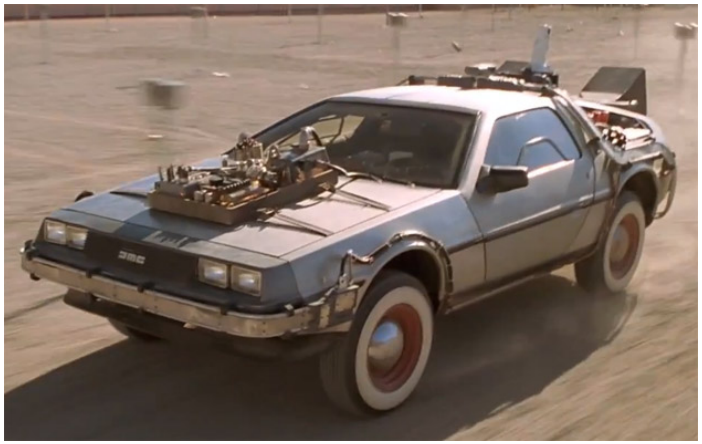
- Peuvent être utilisés sur des classes, des méthodes et des attributs
- Quatre modificateurs d'accès en Java :
  - `public` : n'importe qui peut y accéder
  - `protected` : les classes du même *package* et les sous-classes (dérivées) peuvent y accéder (cf. héritage)
  - Niveau d'accès par défaut si 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

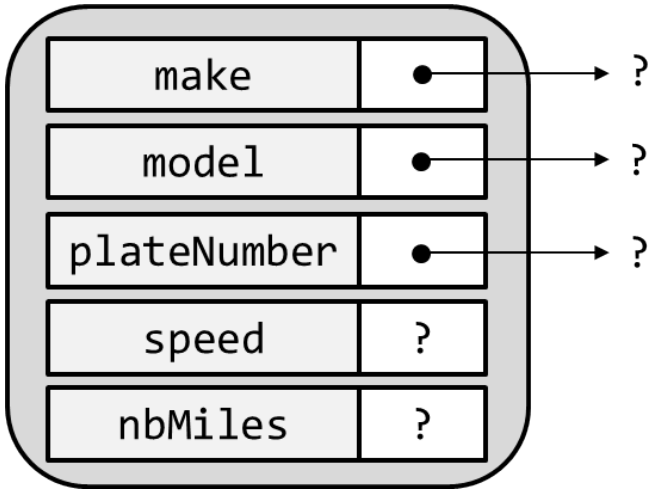
```
public class Car {
   private String make;
   private String model;
   private String numberPlate;
   private short speed;
   private int nbMiles;
   ...

   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 (*acccessors*/*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 a " + c1.getSpeed()
  + " miles a 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 recopie

- 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

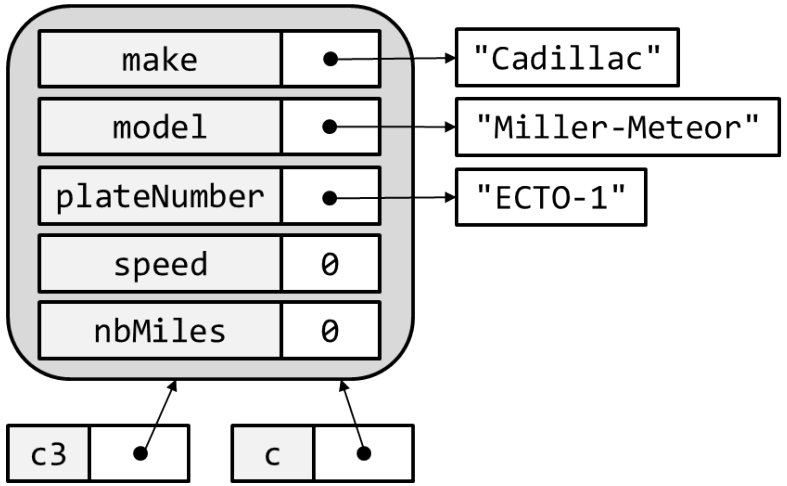
```
private Engine engine;
public Car(Car c) {
  ...
  engine = new Engine(c.engine.getXXX(), ...);
}
```

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 !

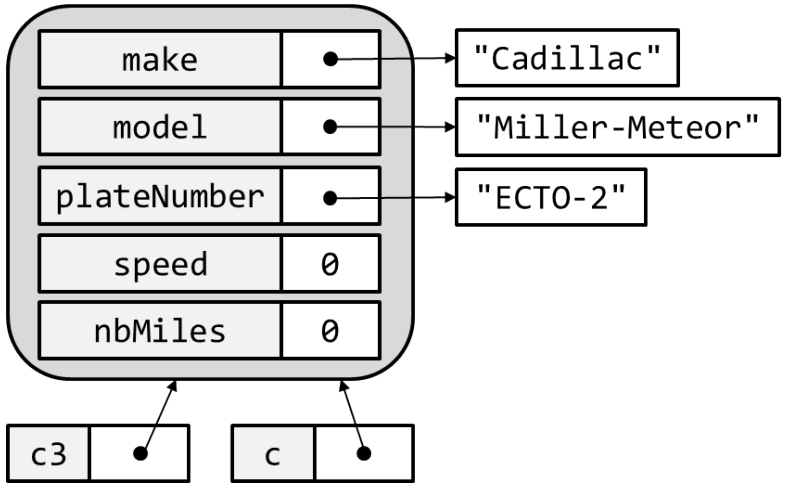
Passer des objets (1)

```
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);
}
```



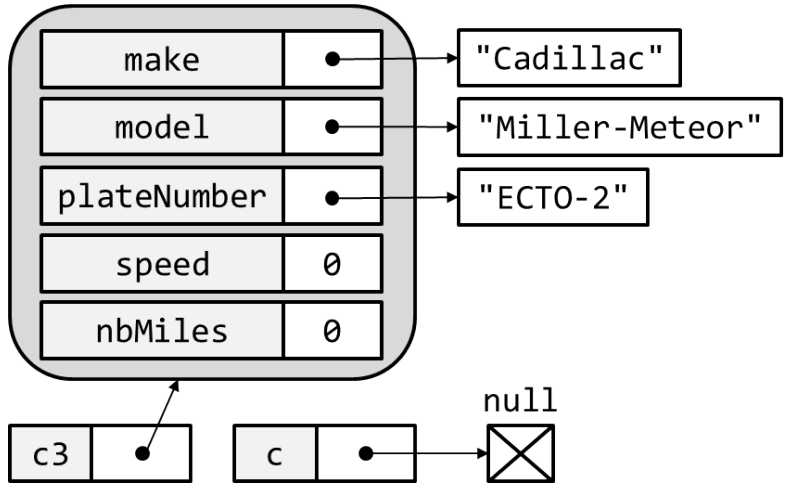
Passer des objets (2)

```
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);
}
```



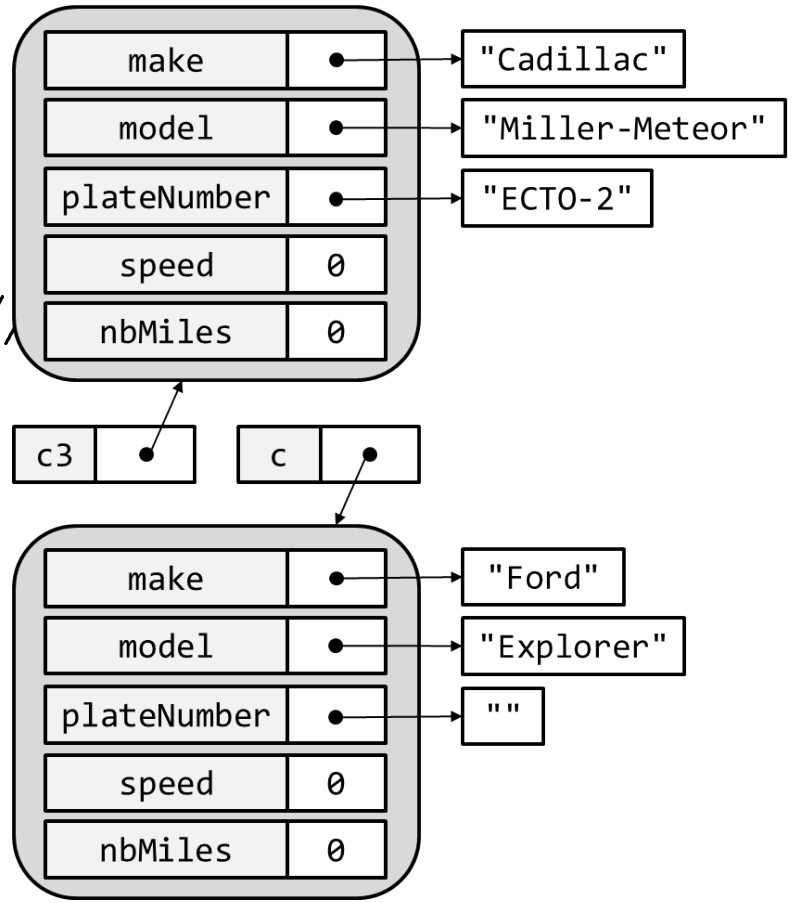
Passer des objets (3)

```
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);
}
```



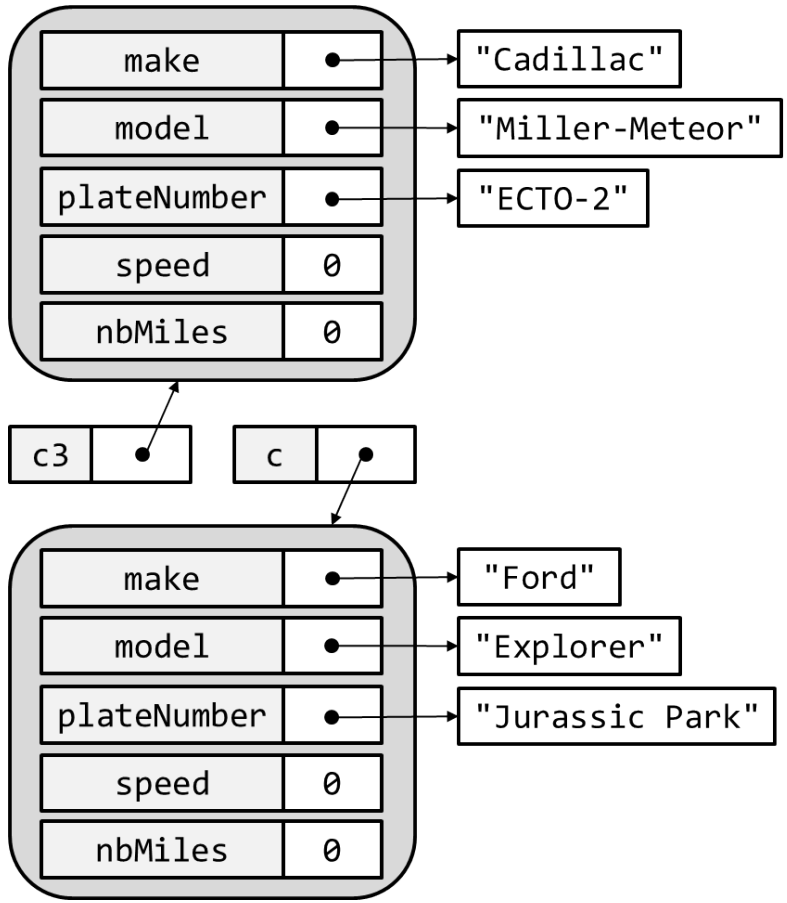
Passer des objets (4)

```
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'accesseur 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 : entrées/sorties

- `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()
    + "␣immatriculee␣" + 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;
}
```

Tableaux

- En Java, les tableaux sont aussi des objets
  - Bien qu'ils aient une syntaxe différente

```
// Alloue un tableau pour 10 entiers
int[] tab = new int[10];
for (int i = 0; i < tab.length; i++) {
  tab[i] = 100 * i; // Stocke des entiers dedans
}
```

- Les tableaux sont tous alloués dynamiquement
- Les tableaux ont un attribut `length`, désignant leur taille
  - `length` est (bien entendu) en lecture seule
- Les éléments d'un tableau sont accessibles en utilisant des crochets [*index*] (comme en C/C++)
  - *index* doit être compris entre 0 et `length`-1, sinon provoque une erreur

Déclarer un tableau

- Les variables tableau sont déclarées avec des crochets après leur type, non après leur nom
  - `String[] names;` vs `String names[];`
  - La dernière forme est tout de même acceptée, mais est déconseillée
- Elles peuvent être déclarées sans être initialisées
  - boolean[] flags ; // Tableau de booleens
- Elles doivent être initialisées avant utilisation

Initialiser un tableau

- Allouer un nouveau tableau avec `new` *type*`[`*size*`]` où *size* est la taille du tableau (*i.e.*, son nombre d'éléments) et *type* est le type des éléments du tableau
  - *size* peut être égal à zéro : tableau vide
- Assigner un tableau existant
  - Les tableaux sont essentiellement des objets avec de la syntaxe supplémentaire
- Définir à `null`
- Assigner des valeurs spécifiques

```
String[] colors = {"vert", "bleu", "jaune", "violet"};
// colors.length == 4
```

  - Sucre syntaxique pour les opérations d'initialisation
  - De tels tableaux peuvent toujours être réassignés et réinitialisés
    - ★ `colors` est une référence à un tableau d'objets de type `String`

Tableaux d'objets

- Contiennent initialement des valeurs `null`
  - L'initialisation d'un tableau n'initialise pas les références-objet
  - Doit être fait dans une étape à part
- Exemple :

```
// Alloue un tableau de 15 references-voiture
Car[] cars = new Car[15];

// Cree un nouvel objet Car pour chaque element
for (int i = 0; i < cars.length; i++)
  cars[i] = new Car(...);
```

Tableaux de tableaux (1)

- Les tableaux peuvent contenir d'autres tableaux

```
int[][] matrix; // Array of arrays of ints.
matrix = new int[20][];
for (int i = 0; i < matrix.length; i++)
   matrix[i] = new int[50];
```

  - D'abord, le tableau de tableaux est alloué
    - ★ Chaque élément de `matrix` est de type `int[]`
  - Ensuite, chaque sous-tableau est alloué

- Pour les tableaux à deux dimensions, Java fournit un raccourci

```
int[][] matrix = new int[20][50]; // Meme chose !
```

Tableaux de tableaux (2)

- Les sous-tableaux peuvent être de tailles différentes

```
int[][] reducedMatrix;
reducedMatrix = new int[20][];
for (int i = 1; i <= reducedMatrix.length; i++)
  reducedMatrix[i - 1] = new int[i];
```

  - Impossible de faire la même chose avec la syntaxe raccourci
- Ils peuvent aussi être spécifiés avec des valeurs initiales nichées

```
int[][] reducedMatrix = {{1, 2, 3}, {4, 5}, null, {6}};
```

Copier un tableau

- Utiliser `System.arraycopy()` pour copier un tableau dans un autre efficacement
- Utiliser la méthode `clone` pour dupliquer un tableau

```
int[] nums = new int[33];
...
int[] numsCopy = (int[]) nums.clone();
```

  - Le type de retour de la méthode est `Object`
    - ★ Le résultat doit être casté dans le bon type
  - La copie est superficielle, seul le tableau de plus haut niveau est copié !
    - ★ Si c'est un tableau d'objets, les objets ne sont pas clonés
    - ★ Si c'est un tableau de tableaux, les sous-tableaux ne sont pas non plus clonés

Types énumérés

- Un type énuméré est une classe qui représente un ensemble prédéfini et fixe de constantes
  - Ces constantes sont en fait des instances de la classe
- Il est défini en utilisant le mot clé `enum`

```
public enum Fuel {
  GASOLINE, DIESEL, LPG; // constantes en majuscules
}
```

- Les valeurs du type sont ces constantes (*i.e.*, des objets de la classe créée)

```
Fuel f = Fuel.DIESEL;
```

  - `Fuel` est une classe qui a (et n'aura) que 3 instances
  - `Fuel.DIESEL` désigne l'une des instances de `Fuel`

Méthodes d'un type énuméré E

- Le compilateur ajoute automatiquement certaines méthodes au type énuméré créé
- Méthodes d'instance
  - `String name()` retourne la chaîne de caractères correspondant au nom de l'objet receveur (sans le nom du type)
    - ★ `f.name()` retourne la chaîne de caractères `"DIESEL"`
  - `int ordinal()` retourne l'indice de l'objet receveur dans l'ordre de déclaration du type énuméré (à partir de 0)
    - ★ `f.ordinal()` retourne 1
- Méthodes de classe
  - `static E valueOf(String s)` retourne, si elle existe, l'instance dont la référence (sans le nom du type) correspond à la chaîne `s`
    - ★ `Fuel.valueOf("LPG")` retourne une référence à l'objet `Fuel.LPG`
  - `static E[] values()` retourne le tableau des valeurs du type dans leur ordre de déclaration
    - ★ `Fuel.values()` retourne le tableau `{ Fuel.GASOLINE, Fuel.DIESEL, Fuel.LPG }`

Égalités des types énumérés

- Pour un objet e d'un type énuméré `E` :
  - `E.valueOf(e.name()) == e`
  - `E.values()[e.ordinal()] == e`
- Utiliser == pour tester l'égalité de valeurs entre deux références d'un même type énuméré
  - Pourquoi ?

Classe générée par le compilateur

```
public class Fuel {
  private String name;
  private int index;
  private Fuel(String theName, int idx) {
    this.name = theName;
    this.index = idx;
  }
  public static final Fuel GASOLINE = new Fuel("GASOLINE", 0);
  public static final Fuel DIESEL = new Fuel("DIESEL", 1);
  public static final Fuel LPG = new Fuel("LPG", 2);
  public String name() { return this.name; }
  public int ordinal () { return this.index; }
  public static Fuel[] values() {
    return { Fuel.GASOLINE, Fuel.DIESEL, Fuel.LPG };
  }
  public static Fuel valueOf(String s) { // grosso modo
   if (s.equals("GASOLINE") { return Fuel.GASOLINE; }
   else if ... // idem pour DIESEL et LPG
  }
}
```

Type énuméré et `switch`

```
Fuel fuel = ...;
switch (fuel) {
   case GASOLINE:
      ...
   case DIESEL:
      ...
   case LPG:
      ...
   default:
      ...
}
```

Exemple : Fuel.java

- Beaucoup plus puissant que les types énumérés d'autres langages
  - Possibilité d'inclure des constructeurs, méthodes et autres attributs

```
public enum Fuel {
  GASOLINE (43.8), DIESEL (42.5), LPG (46.1);
  private final double heatingValue; // en MJ/kg
  private Fuel(double heatingValue) {
    this.heatingValue = heatingValue;
  }
  public double getHeatingValue() {
    return this.heatingValue;
  }
}
...
for(Fuel f : Fuel.values())
  System.out.println(f.name() + ": " +
    f.getHeatingValue());
```

Packages (paquetages en français)

- Sont une collection de types liés
- Permettent de regrouper des classes
  - C'est optionnel, mais généralement très utile !
- Forment une hiérarchie
  - `package1.package2.package3`
- Fournissent une gestion de l'espace de noms (*namespace*)
  - Deux classes du même *package* ne peuvent pas avoir le même nom
  - Autrement dit, des classes peuvent avoir le même nom si elles sont dans des *packages* différents
- Par défaut, une classe est dans le "*default package*"
  - *Default package* n'a pas de nom !
  - Prendre l'habitude de créer un nouveau *package*
- Utiliser le mot clé `package`pour spécifier un *package* différent
  - `package car;`
  - Doit être la première instruction dans le fichier `.java`
  - Détermine où les fichiers `.java` et `.class` doivent être placés

Utiliser une classe d'un *package*

- Trois possibilités :
  - Faire référence à la classe avec le nom qualifié (évite les conflits de noms)

```
java.util.Date d1 = new java.util.Date();
java.sql.Date d2 = new java.sql.Date(...);
```

  - Importer la classe elle-même

```
import java.util.Date;
...
Date date = new Date();
```

  - Importer le *package* entier

```
import java.util.*;
...
Date date = new Date();
```

import static

- Permet d'alléger le code pour l'utilisation des variables et des méthodes statiques d'une classe

```
import static java.lang.Math.*;
...
x = max(sqrt(abs(y)), sin(y)); // au lieu de Math.max..
```

- À utiliser avec précaution car il peut être plus difficile de savoir d'où vient une variable ou une méthode de classe

Packages et API Java

- Toutes les classes de l'API Java sont dans des *packages*
- Les classes dans `java.lang` sont automatiquement importées
  - Pas besoin d'importer explicitement le contenu de `java.lang`
- Pour importer des classes Java qui ne sont pas dans le *package* `java.lang` :

```
import java.util.ArrayList;
import java.util.HashSet;
...
```

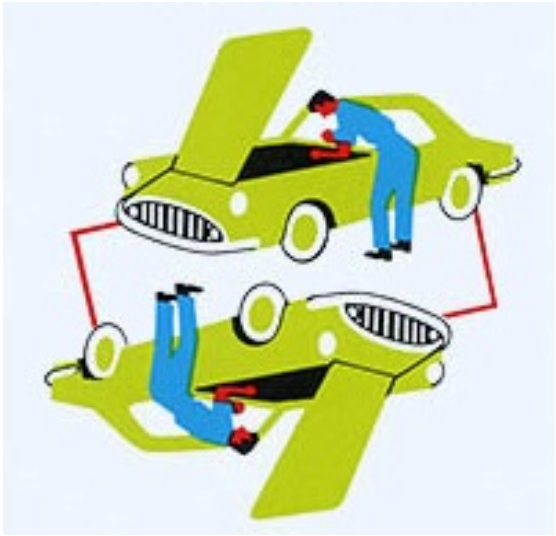
OU

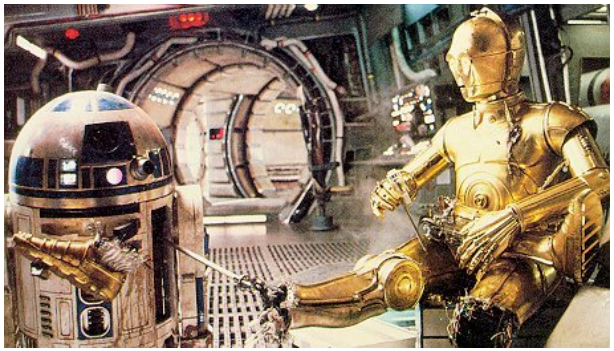
```
import java.util.*;
```

- L'importation d'un *package* n'est pas récursive !
  - Importer `java.*` ne mènera nulle part

Au menu

4 images, 1 mot





Réparation

- Voitures, montres, droïdes et Groot sont des "objets" différents, ayant des comportements différents
  - Faire le plein d'une voiture, remonter une montre, *etc.*
- Mais ils sont tous réparables (même si les processus sont différents)
- D'un point de vue programmation objet, `Car`, `Watch`, `Droid` et `Alien` sont des classes d'objets différentes, proposant des fonctionnalités (méthodes) différentes
  - `fillUp()` pour `Car`, `wind()` pour `Watch`, *etc.*
- Mais elles proposent toutes `repair()`
  - Avec un traitement propre à chacune

Problème

```
T[] repairs = new T[2]; // objets reparables
repairs[0] = new Car(...);
repairs[1] = new Watch(...);
for (int i = 0; i < repairs.length; i++) {
  repairs[i].repair();
}
```

- Quel est le type `T` des éléments du tableau `repairs` ?
  - `T` accepte la méthode `repair()`
  - Le tableau doit pouvoir contenir à la fois des voitures, des montres, *etc.*

Très mauvaise solution

```
Object[] repairs = new Object[2]; // objets reparables
repairs[0] = new Car(...);
repairs[1] = new Watch(...);
for (int i = 0; i < repairs.length; i++) {
  if (repairs[i] instanceof Car) {
    ((Car) repairs[i]).repair();
  } else if (repairs[i] instanceof Watch) {
      ((Watch) repairs[i]).repair();
  } else if ...
}
```

Solution

- Conserver les différentes classes : `Car`, `Watch`, *etc.*
  - Le processus de réparation est propre à chaque classe
  - L'implémentation de `repair()` est donc différente dans chaque classe
- Créer un type commun
  - Il faut pouvoir traiter les objets sans les différencier par leur classe
  - Il faut pouvoir considérer leurs instances comme des objets du type "est réparable", c'est-à-dire "accepte la méthode `repair()`"
    - ★ Ne considérer qu'une "facette" de l'objet indépendamment des autres
    - ★ Réaliser une "projection" de l'objet sur ce type
    - ★ "multi-typage" des objets = polymorphisme des objets

Interfaces

- Une interface déclare un ensemble de signatures de méthodes publiques (comportements), sans corps
  - Elle déclare uniquement des comportements, mais ne les définit pas
  - Elle ne contient ni implémentation de méthode, ni variable d'instance
- Une classe implémente (ou réalise) une interface afin de signifier qu'elle fournit cet ensemble de comportements (contrat à respecter)
  - Elle doit définir un comportement pour chacune des méthodes déclarées dans l'interface
  - Les instances de la classe pourront être vues et manipulées comme étant du type de l'interface
- Les interfaces sont donc des types de données abstraits, vus uniquement au travers de ses méthodes
  - Pas de constructeur
  - Versus les classes = types concrets

Déclarer des interfaces

- Les interfaces sont déclarées comme des classes, mais en utilisant le mot clé `interface`

```
public interface Reparable {
  void repair();
  ...
}
```

- Fichier `Reparable.java`
- Pas de modificateur d'accès de méthode
  - L'accès est publique !

Implémenter des interfaces

- Les classes peuvent implémenter des interfaces en utilisant le mot clé `implements`
- Lorsqu'une classe implémente une interface, elle doit déclarer les méthodes comme `public`

```
public class Car implements Reparable {
   ...
   @Override
   public void repair() {
      ...
   }
}
```

- N'importe qui peut appeler l'implémentation de la classe de l'interface, puisque qu'elle est publique

Interfaces et classes

- Une classe peut implémenter plusieurs interfaces

```
public class Car implements Reparable, Vehicle { ... }
```

  - C'est une version plus simple et plus propre de l'héritage multiple
  - En Java, il n'y a pas d'héritage multiple de classes
  - Quels sont les types possibles pour une instance de `Car` ?
- Les interfaces ne peuvent pas être instanciées (pas de constructeur)
  - Elles doivent être implémentées par une classe, et ensuite la classe est instanciée
- Les variables peuvent être d'un type interface, tout comme elles peuvent être d'un type classe

```
Reparable[] repairs = new Reparable[2];
repairs[0] = new Car(...); // projection des instances
repairs[1] = new Watch(...); // sur le type Reparable
for (int i = 0; i < repairs.length; i++)
  repairs[i].repair(); // meme signature de methode
                       // mais traitements differents
```

Utiliser des interfaces (1)

- Les interfaces peuvent exprimer des propriétés (en termes de services fournis) que les types ont
  - L'interface `Reparable` exprime la propriété qu'un objet "est réparable"
- Souvent, il y a des situations où :
  - Il y a un unique et bien défini ensemble de comportements
  - Mais avec beaucoup d'implémentations différentes possibles
- Les interfaces permettent de découpler les composants du programme
  - En particulier, lorsqu'un composant peut être implémenté de multiples façons
  - Les autres composants interagissent avec le type interface général, et pas avec des implémentations spécifiques

Utiliser des interfaces (2)

- Exemple :

```
public interface Vehicle {
  void move(short speed);
  void turn(Direction direction);
  boolean isMoving();
  ...
}
```

- Fournir de multiples implémentations

```
public class Car implements Vehicle { ... }
public class Motorcycle implements Vehicle { ... }
```

- Écrire du code pour une interface, pas pour des implémentations

```
public static void simulation(Vehicle v) {
  v.move(88);
  v.turn(Direction.LEFT);
  ...
}
```

Utiliser des interfaces (3)

- Pouvoir changer les détails d'implémentation si nécessaire
  - Aussi longtemps que la définition de l'interface reste la même
- Si l'implémentation de l'interface est importante et complexe :
  - Un autre code peut utiliser une implémentation temporaire de l'interface, jusqu'à ce que la version complète soit terminée

```
public class FakeVehicule implements Vehicle {
  public void move(short speed) {
    // Ne fait rien
  }
  public boolean isMoving() {
    return false;
  }
  ...
}
```

  - Le développement de composants dépendants peut être fait en parallèle

Polymorphisme

Concept clé de la programmation objet

- Les interfaces et l'héritage permettent d'écrire du code polymorphe, pouvant être utilisé avec différents types
- C'est ce qui différencie la programmation objet de la programmation impérative et modulaire

```
Reparable r;
if ((int)(Math.random()*2)%2 == 0) {
   r = new Car(...);
} else {
   r = new Watch(...);
}
r.repair();
```

  - Quel code sera exécuté lors de l'appel `r.repair()` ?
    - ★ Impossible de savoir à la compilation quel objet désignera `r`
    - ★ La méthode `repair()` appelée est *a priori* non connue
  - Ce code compile et le résultat dépend de `(int)(Math.random()*2)%2`
  - Comment la bonne méthode `repair()` est-elle appelée ?

Type statique et type dynamique

- Le type statique d'une référence correspond à celui de sa déclaration
  - Il définit les appels de méthodes autorisés
  - Il est connu dès la compilation
- Le type dynamique d'une référence correspond au type de l'objet référencé (lequel peut évoluer au cours de l'exécution)
  - Il définit le traitement exécuté
  - Il n'est connu qu'à l'exécution

```
Reparable r;
if ((int)(Math.random()*2)%2 == 0) {
   r = new Car(...);
   // r.move(88) // erreur
} else {
   r = new Watch(...);
}
r.repair();
```

Liaison tardive

- Liaison anticipée (*early binding*)
  - Le compilateur génère un appel à une fonction en particulier et le code à exécuter est précisément déterminé lors de l'édition de liens à la compilation
- Liaison tardive (*late binding*)
  - Le code à exécuter lors de l'appel d'une méthode sur un objet n'est déterminé qu'à l'exécution
- Résolution du choix de la méthode à appeler
  1. Le compilateur détermine la signature de méthode à rechercher en se fondant sur les types statiques des références
     - ★ Il vérifie seulement la validité de l'appel
  2. Mais ce n'est qu'à l'exécution que le code à exécuter est recherché dans la classe du type dynamique de l'objet receveur

Transtypage (*Cast*)

- Une référence n'a qu'un seul type, un objet peut en avoir plusieurs
  - Objets polymorphes
    - ★ Les objets sont des instances d'une classe, donc du type de cette classe, mais aussi du type de chacune des interfaces implémentée par la classe
    - ★ Différents points de vue possibles (facettes) sur un même objet
- Caster/transtyper
  - Conversion explicite d'une donnée dans un autre type
  - À partir d'une référence sur un objet, en créer une autre d'un autre type, vers le même objet
- Généralisation (*upcast*) : changer vers une classe moins spécifique (toujours possible vers `Object`)
  - Naturelle et implicite
  - Vérifiée à la compilation
  - Sûre
- Spécialisation (*downcast*) : changer vers une classe plus spécifique
  - Explicite
  - Vérifiée à l'exécution
  - À risque

Illustration

```
Reparable[] repairs = new Reparable[2];
repairs[0] = new Car(...);     // upcast implicite
repairs[1] = new Watch(...); // Car/Watch -> Reparable
for (int i = 0; i < repairs.length; i++)
  ((Car) repairs[i]).repair(); // downcast explicite
                               // Reparable -> Car
                             // erreur a l'execution qd i=1
```

Étendre des interfaces

- Une interface peut être étendue en utilisant le mot clé `extends`

```
public interface Spacecraft extends Vehicle {
    void travelThroughHyperspace(...);
    ...
}
```

  - Cette interface hérite de toutes les déclarations de méthodes de `Vehicle`
  - Encore une fois, elles sont toutes publiques

Au menu

Concevoir du logiciel réutilisable

- C'est une des préoccupations majeures des programmeurs
- Pour faciliter le développement logiciel et augmenter la productivité
  - Ne pas refaire ce qui a déjà été fait (souvent de manière efficace)
    - ★ API et bibliothèques de composants logiciels existantes
  - Diminuer les sources d'erreurs
    - ★ Les API et les bibliothèques ont été *a priori* maintes fois éprouvées

Réutiliser un type

- Définir un comportement propre à son contexte d'utilisation, tout en respectant des interfaces prédéfinies, et ainsi s'insérer dans un cadre préétabli (*framework*)
- En Java : utilisation des interfaces

Réutiliser un comportement

- Définir un attribut de la classe dont on veut récupérer le comportement (association/agrégation/composition)
- Définir des méthodes correspondant aux comportements que l'on veut récupérer (réutilisation partielle), en les ajustant à leur contexte
  - Le corps de ces méthodes consiste en un appel des méthodes de l'attribut (délégation) avec une adaptation éventuelle

```
public class Car {
  private Wheel[] wheels;
  ...
  public void brake() {
    for (Wheel w : this.wheels) {
      w.brake();
    }
  }
}
```

Association, agrégation, composition, dépendance

- Association
  - Lien structurel entre deux classes
    - ★ Possède une référence vers
  - Les deux classes ont le même niveau conceptuel
    - ★ Aucune n'étant plus importante que l'autre
  - Relation "connaît"
- Agrégation
  - Association entre une classe composite (*i.e.*, le tout) et une classe agrégée (*i.e.*, la partie)
  - Relation "a un"
- Composition
  - Agrégation forte dans laquelle la partie ne se conçoit pas sans son tout et où une partie ne peut figurer que dans un seul objet composite
- Dépendance
  - Une méthode d'une classe prend en paramètre des objets d'une autre classe

Réutiliser un type vs réutiliser un comportement

- Réutiliser un type
  - Conformité de type de la classe créée
  - Polymorphisme
  - Mais obligation de récrire pour chaque classe implémentant l'interface le code de toutes les méthodes, y compris si celui-ci est le même pour plusieurs classes
    - ★ Duplication de code
    - ★ Gênant dans le cas de modifications qu'il faudra reporter plusieurs fois
- Réutiliser un comportement
  - Pas besoin de récrire le code des méthodes
  - Mais pas de compatibilité de type et donc pas de polymorphisme
    - ★ Les instances de la classe créée ne sont pas du type de la classe de l'attribut

Factorisation de code ?

```
class Car implements Vehicle {
  private Engine engine;
  ...
  public void start() {
    this.engine.start();
  }
  ...
  public void turn(Direction d) {
    ... // traitement
  }
}
```

```
class Motorcycle implements Vehicle {
  private Engine engine;
  ...
  public void start() {
    this.engine.start();
  }
  ...
  public void turn(Direction d) {
    ... // traitement different
  }
}
```

- Attributs et code répétés
- Mais la méthode `turn` est différente pour les deux classes
- Comment concilier la factorisation de code et les différences ?

Héritage de classe (1)

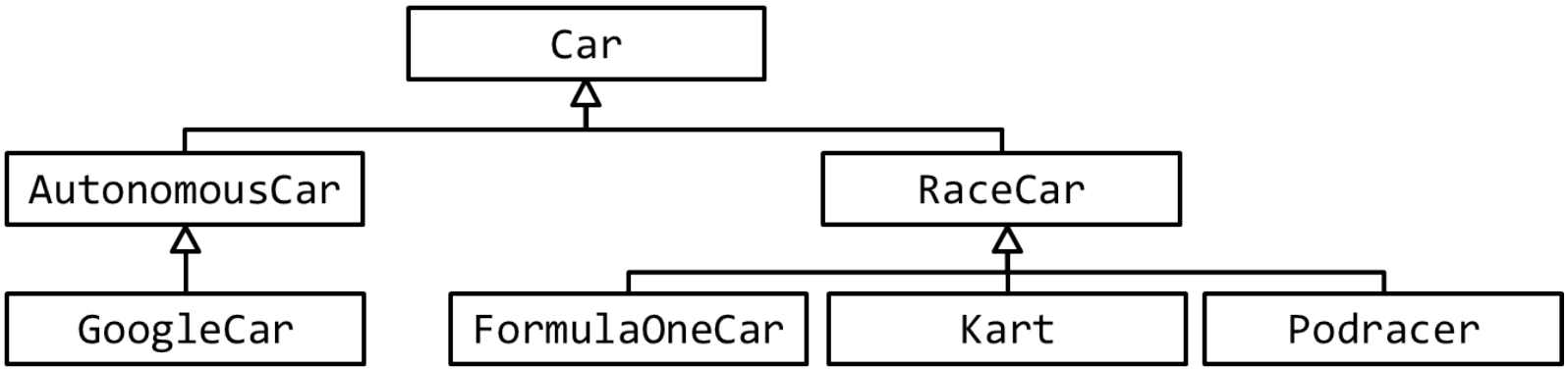
- Une classe peut étendre une autre classe en utilisant le mot clé `extends`

```
public class RaceCar extends Car { ... }
```

- Terminologie :
  - Classe mère/parent, super-classe ou classe de base (ici, `Car`)
  - Classe fille/enfant, sous-classe ou classe dérivée (ici, `RaceCar`)
  - Relation "est un"
- Les classes enfant héritent de toutes les méthodes et attributs (accessibles) de la classe parent
  - Elles peuvent ajouter de nouvelles fonctionnalités
  - Elles peuvent également redéfinir des méthodes héritées
- Les classes enfant sont une spécialisation de la classe parent
  - Elles peuvent être traitées comme la classe parent puisqu'elles ont (au moins) les mêmes membres
  - Elles sont un sous-type de la classe parent
  - Une instance de classe enfant est donc aussi du type de la classe parent (polymorphisme)

Héritage de classe (2)

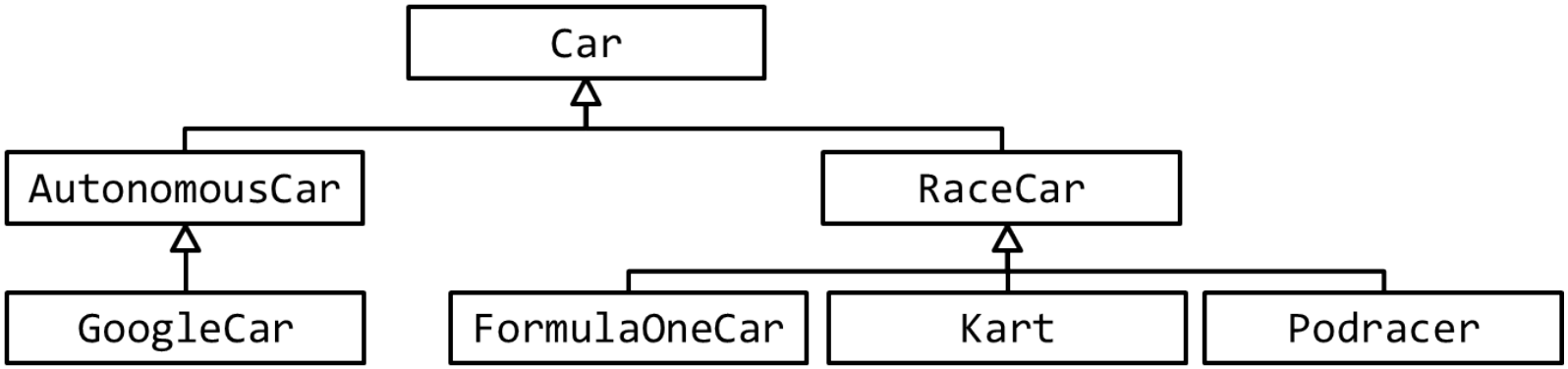
- En Java, il n'y a pas d'héritage multiple
  - Une classe ne peut hériter que d'une seule classe (qui elle-même peut hériter d'une autre classe, et ainsi de suite)
- Les liens d'héritage permettent de définir des hiérarchies de classes



  - Une instance de `FormulaOneCar` est aussi une `RaceCar` et une `Car`
  - Les interfaces publiques définies dans `RaceCar` et dans `Car` font partie de l'interface publique d'un objet `FormulaOneCar`

Héritage de classe (3)

- Les liens d'héritage permettent de définir des hiérarchies de classes



  - Par héritage :
    - ★ Une *Google car* est une voiture sans conducteur
    - ★ Une Formule 1 est une voiture
    - ★ Une voiture de course est une voiture
  - Ces déclarations sont clairement fausses :
    - ★ Une *Google car* est une voiture de course
    - ★ Une voiture sans conducteur est une Formule 1
  - Qu'en est-il de ces déclarations ?
    - ★ Une voiture est une *Google car*
    - ★ Une voiture de course est une Formule 1
  - Dépend du véhicule considéré !
    - ★ Nécessite d'examiner un véhicule spécifique pour vérifier la déclaration

Classe `Object`

- Si une classe ne précise pas de classe parent, elle hérite par défaut de la classe `Object`
- En Java, toutes les classes héritent donc directement ou indirectement (*via* leur classe parent) de la classe `Object`
  - Tout objet peut être traité comme une instance d'`Object`
  - Tout objet peut utiliser les méthodes définies par la classe `Object`
    - `toString()`, `equals(Object)`, `clone()`, `hashCode()`, `getClass()`
    - Penser à les redéfinir pour avoir un traitement spécifique à la classe
- Ainsi, les liens d'héritage forment un arbre ayant la classe `Object` pour racine

Factorisation du comportement

- Les comportements (accessibles) définis dans une classe sont directement disponibles pour les instances des classes qui en héritent (même indirectement)

```
public class RaceCar extends Car {
  public int getNbSeats() {
    return 1;
  }
}
public class Kart extends RaceCar { ... }
Kart k = new Kart();      // un Kart peut utiliser les
int n = k.getNbSeats(); // methodes publiques de ses
Class c = k.getClass(); // classes parent
RaceCar r = k; // upcast autorise vers RaceCar
Car c = k;       // ou vers Car
```

  - Les méthodes des classes parent peuvent être appelées sans syntaxe spéciale
  - Un objet `Kart` est aussi une `RaceCar` et peut donc appeler toutes les méthodes déclarées et/ou implémentées de `RaceCar`

Factorisation de l'état

- Les attributs des classes parent sont des attributs de la classe enfant

```
public class Car {
  private String make;
  public Engine engine; // uniquement pour illustrer
  ...
}
public class RaceCar extends Car { ... }
public class Kart extends RaceCar { ... }
...
Kart k = new Kart();
k.engine.start(); // engine est un attribut de Kart
```

- Mais les attributs privés restent toujours inaccessibles

```
public class Kart extends RaceCar {
  public void foo() {
    // S.o.p(this.make); // erreur : acces interdit
    S.o.p(this.getMake()); // l'attribut make existe
  }                        // donc  bien pour Kart
}
```

Extension du comportement

- Une classe enfant peut fournir ses propres méthodes (*i.e.*, ajouter de nouveaux comportements)
- Une classe enfant est donc une extension de la classe parent

```
public class RaceCar extends Car {
  public int getNbSeats() {
    return 1;
  }
}
public class Kart extends RaceCar {
  public void drifting() { ... }
}
```

  - La classe `Kart` étend les fonctionnalités de la classe `RaceCar`
  - Un objet `Kart` peut appeler les méthodes `getNbSeats` et `drifting`

Spécialisation du comportement

- Une classe enfant peut redéfinir une méthode héritée
  - On parle de redéfinition lorsqu'une classe enfant définit une méthode portant le même nom et les mêmes paramètres qu'une méthode héritée
  - Cela permet de préciser un autre comportement que celui hérité
  - C'est ce comportement qui est utilisé par les instances de la classe

```
public class KartDoubleDash extends Kart {
  public int getNbSeats() {
    return 2;
  }
}
...
KartDoubleDash mario = new KartDoubleDash();
System.out.println(mario.getNbSeats()); // affiche 2
Kart yoshi = new Kart();
System.out.println(yoshi.getNbSeats()); // affiche 1
Kart luigi = mario; // upcast
System.out.println(luigi.getNbSeats()); // affiche 2
                                        // liaison tardive
```

Recherche de méthodes (1)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p();
    this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h();
    this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p();
    this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
}
```

```
public class C extends B {
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new A();
ref.f(); // affiche ?
```

Recherche de méthodes (2)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p();
    this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h();
    this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p();
    this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
}
```

```
public class C extends B {
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new A();
ref.f(); // affiche :
// A.f
// A.p
// A.g
// A.h
// A.p
```

Recherche de méthodes (3)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p();
    this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h();
    this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p();
    this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
}
```

```
public class C extends B {
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new B();
ref.f(); // affiche ?
```

Recherche de méthodes (4)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p();
    this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h();
    this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p();
    this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
}
```

```
public class C extends B {
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new B();
ref.f(); // affiche :
// B.f
// B.p
// A.g
// B.h
// A.p
```

Recherche de méthodes (5)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p();
    this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h();
    this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p();
    this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
}
```

```
public class C extends B {
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new C();
ref.f(); // affiche ?
```

Recherche de méthodes (6)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p();
    this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h();
    this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p();
    this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
}
```

```
public class C extends B {
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new C();
ref.f(); // affiche :
// B.f
// B.p
// A.g
// C.h
// A.p
```

Résumé

- Méthode publique
  - Le code à exécuter lors de l'invocation d'une méthode publique n'est déterminé qu'à l'exécution (liaison tardive)
  - La recherche commence dans la classe du type dynamique de l'objet receveur
  - Et remonte de classe parent en classe parent jusqu'à trouver la signature identifiée à la compilation
- Méthode privée
  - Le code à exécuter lors de l'invocation d'une méthode privée est celui défini dans la classe du type statique de l'objet receveur

Masquage d'attribut

```
public class Kart extends RaceCar {
  public int nbSeats = 1; // uniquement pour illustrer
  public int getNbSeats() { return this.nbSeats; }
}
public class KartDoubleDash extends Kart {
  public int nbSeats = 2; // uniquement pour illustrer
}
...
KartDoubleDash mario = new KartDoubleDash();
System.out.println(mario.nbSeats); // affiche 2
Kart luigi = mario;
System.out.println(luigi.nbSeats); // affiche 1
System.out.println(mario.getNbSeats()); // affiche 1
```

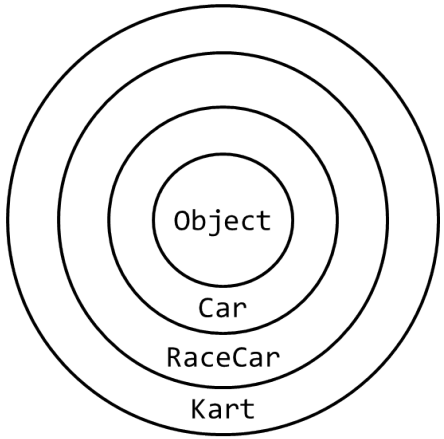
- Masquage de la variable héritée, mais les deux continuent d'exister
- Attention, il est possible de masquer un attribut en changeant de type
- Moralité : éviter de déclarer une variable portant le même nom qu'une variable héritée !

Ce à quoi les classes enfant n'ont pas accès

- Les classes enfant ne peuvent pas accéder aux membres privés des classes parent
- Le modificateur d'accès `protected` permet à la classe enfant d'accéder aux membres de la classe parent
  - Uniquement accessibles à l'intérieur de la classe et dans toutes ses sous-classes, ainsi que dans les classes du même *package*
  - Plus lâche que `private`, mais toujours pas `public` !
  - Les attributs "factorisables" peuvent donc être définis comme `protected` (en conservant les accesseurs pour les autres classes)
- Les classes enfant n'héritent pas non plus des attributs et des méthodes statiques
  - Ils peuvent être accédés, mais ne sont pas hérités

Plusieurs couches objet

«Les ~~ogres~~ objets sont comme les oignons... Ils ont des couches !»



- Un objet de type `Kart` est composé d'un noyau défini par `Object`, étendu par une couche `Car`, puis une couche `RaceCar` et enfin `Kart`
- À chaque couche, il est possible d'utiliser tout ce qui est accessible aux couches intérieures
- Dans le cas d'une redéfinition, c'est la définition de la couche la plus extérieure qui est considérée
- Un *upcast* revient à supprimer des couches, c'est-à-dire supprimer l'accès aux définitions des couches enlevées (*e.g.*, `Car c=new Kart();`)

Construction d'objet de classe enfant

- Il faut construire les différentes couches
  - Utilisation des constructeurs pour chaque couche
- Les constructeurs de la classe parent ne sont pas hérités !
- Si la classe parent n'a pas de constructeur par défaut, il faut explicitement appeler l'un des constructeurs de la classe parent, en utilisant le mot clé `super` avec d'éventuels paramètres
  - Chaînage de constructeurs : il ne s'agit plus d'appeler un constructeur de la même classe mais de la classe parent, d'où le `super(...)` au lieu d'un `this(...)`
- Le constructeur d'une classe enfant doit commencer par un appel à un constructeur de la classe parent pour initialiser la partie héritée de l'état, puis il initialise directement les variables d'instance déclarées

Illustration

```
public class Car { // extends Object implicite
  private int nbSeats;
  public Car(int nbSeats) {
    // utilise implicitement super() de Object
    this.nbSeats = nbSeats;
  } ...
}
public class RaceCar extends Car {
  private String tournament;
  public RaceCar(String tournament) {
    super(1); // appel du constructeur de la classe parent
    this.tournament = tournament;
  } ...
}
public class Kart extends RaceCar {
  public Kart() {
    super("Mario Kart"); // idem
  } ...
}
```

Mot clé super

- `super` est une référence à l'objet receveur (*i.e.*, sur lequel la méthode est appelée)
  - `super == this`
- Au sein d'une classe enfant, l'appel à une méthode héritée se fait sur `this`, sauf si on est en train de redéfinir la méthode héritée, auquel cas l'appel se sait sur `super`

```
public class Car {
  public String toString() {
    return "voiture";
  }
}
public class AutonomousCar extends Car {
  public String toString() {
    return super.toString() + " sans conducteur";
  }
}
...
S.o.p(new AutonomousCar()); // voiture sans conducteur
```

Recherche de méthodes avec `super`

- Différente de celle avec `this`
- Méthode publique
  - La recherche commence dans la classe parent de la classe définissant la méthode utilisant `super`
  - Le processus de chaînage reste ensuite le même
- `super` ne fait pas commencer la recherche dans la classe parent de l'objet !
  - `this` est dynamique, `super` est statique

Recherche de méthodes avec super (1)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p(); this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h(); this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
  public void k() {S.o.p("A.k");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p(); this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
  public void k() {
    S.o.p("B.k"); super.k();
  }
```

```
public class C extends B {
  public void f() {
    S.o.p("C.f"); super.f();
  }
  public void g() {
    S.o.p("C.g"); super.g();
  }
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new B();
ref.k(); // affiche ?
```

Recherche de méthodes avec super (2)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p(); this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h(); this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
  public void k() {S.o.p("A.k");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p(); this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
  public void k() {
    S.o.p("B.k"); super.k();
  }
```

```
public class C extends B {
  public void f() {
    S.o.p("C.f"); super.f();
  }
  public void g() {
    S.o.p("C.g"); super.g();
  }
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new B();
ref.k(); // affiche :
// B.k
// A.k
```

Recherche de méthodes avec super (3)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p(); this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h(); this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
  public void k() {S.o.p("A.k");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p(); this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
  public void k() {
    S.o.p("B.k"); super.k();
  }
```

```
public class C extends B {
  public void f() {
    S.o.p("C.f"); super.f();
  }
  public void g() {
    S.o.p("C.g"); super.g();
  }
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new C();
ref.f(); // affiche ?
```

Recherche de méthodes avec super (4)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p(); this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h(); this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
  public void k() {S.o.p("A.k");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p(); this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
  public void k() {
    S.o.p("B.k"); super.k();
  }
```

```
public class C extends B {
  public void f() {
    S.o.p("C.f"); super.f();
  }
  public void g() {
    S.o.p("C.g"); super.g();
  }
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new C();
ref.f(); // affiche :
// C.f
// B.f
// B.p
// C.g
// A.g
// C.h
// A.p
```

Recherche de méthodes avec super (5)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p(); this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h(); this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
  public void k() {S.o.p("A.k");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p(); this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
  public void k() {
    S.o.p("B.k"); super.k();
  }
```

```
public class C extends B {
  public void f() {
    S.o.p("C.f"); super.f();
  }
  public void g() {
    S.o.p("C.g"); super.g();
  }
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new C();
ref.k(); // affiche ?
```

Recherche de méthodes avec super (6)

```
public class A {
  public void f() {
    S.o.p("A.f");
    this.p(); this.g();
  }
  public void g() {
    S.o.p("A.g");
    this.h(); this.p();
  }
  public void h() {S.o.p("A.h");}
  private void p() {S.o.p("A.p");}
  public void k() {S.o.p("A.k");}
}
public class B extends A {
  public void f() {
    S.o.p("B.f");
    this.p(); this.g();
  }
  public void h() {S.o.p("B.h");}
  private void p() {S.o.p("B.p");}
  public void k() {
    S.o.p("B.k"); super.k();
  }
```

```
public class C extends B {
  public void f() {
    S.o.p("C.f"); super.f();
  }
  public void g() {
    S.o.p("C.g"); super.g();
  }
  public void h() {S.o.p("C.h");}
  private void p() {S.o.p("C.p");}
}
...
A ref;
ref = new C();
ref.k(); // affiche :
// B.k
// A.k
```

Attention

Lors de la sélection d'une méthode polymorphe, c'est le type de la référence passée en paramètre qui compte et pas celui de l'objet référencé

```
public class A {
  public void f() { S.o.p("A.f"); }
}
public class B extends A {
  public void f() { S.o.p("B.f"); }
}
public class C extends A { }
public class D {
  public void foo(A a) { a.f(); }
  public void foo(B b) { S.o.p("foo(B)"); }
  public void foo(C c) { S.o.p("foo(C)"); }
  public void bar(A a) { this.foo(a); }
  public static void main(String[] args) {
    A a = new A(); B b = new B();
    C c = new C(); D d = new D();
    d.foo(a); d.foo(b); d.foo(c);
    d.bar(a); d.bar(b); d.bar(c);
  }
}
```

```
// affiche :
// A.f
// foo(B)
// foo(C)
// A.f
// B.f
// A.f
```

Étapes de la création d'un objet

1. Chargement de la classe, si cela n'a pas encore été fait
   - Et donc chargement des éventuelles classes parent selon le même principe
2. Initialisation des attributs statiques (avec valeur par défaut)
   - Une seule fois au moment du chargement de la classe
3. Appel du constructeur de la classe parent
4. Initialisation des attributs ayant une valeur par défaut
5. Exécution du reste du code du constructeur

Illustration (1)

```
public class V {
  public V(int i) { S.o.p("V" + i); }
}
public class A {
  private static V v0 = new V(0);
  public A() { S.o.p("A"); }
}
public class B extends A {
  private static final V v1 = new V(1);
  private V v2 = new V(2);
  private V v3;
  public B() {
    System.out.println("B");
    this.v3 = new V(3);
  }
}
...
B b1 = new B(); // affiche ?
```

Illustration (2)

```
public class V {                                    // affiche :
  public V(int i) { S.o.p("V" + i); }               // V0
}                                                   // V1
public class A {                                    // A
  private static V v0 = new V(0);                   // V2
  public A() { S.o.p("A"); }                        // B
}                                                   // V3
public class B extends A {
  private static final V v1 = new V(1);
  private V v2 = new V(2);
  private V v3;
  public B() {
    System.out.println("B");
    this.v3 = new V(3);
  }
}
...
B b1 = new B();
```

Illustration (3)

```
public class V {
  public V(int i) { S.o.p("V" + i); }
}
public class A {
  private static V v0 = new V(0);
  public A() { S.o.p("A"); }
}
public class B extends A {
  private static final V v1 = new V(1);
  private V v2 = new V(2);
  private V v3;
  public B() {
    System.out.println("B");
    this.v3 = new V(3);
  }
}
...
B b1 = new B();
B b2 = new B(); // affiche ?
```

Illustration (4)

```
public class V {                                    // affiche :
  public V(int i) { S.o.p("V" + i); }               // V0
}                                                   // V1
public class A {                                    // A
  private static V v0 = new V(0);                   // V2
  public A() { S.o.p("A"); }                        // B
}                                                   // V3
public class B extends A {                          // A
  private static final V v1 = new V(1);             // V2
  private V v2 = new V(2);                          // B
  private V v3;                                     // V3
  public B() {
    System.out.println("B");
    this.v3 = new V(3);
  }
}
...
B b1 = new B();
B b2 = new B(); // affiche ?
```

Mort d'un objet

- *A priori*, il n'y a pas à s'en occuper
- Le ramasse-miettes recycle si nécessaire les objets qui ne sont plus référencés et libère la mémoire associée
- Mais il n'y a aucune assurance qu'un objet sera effectivement collecté
- La méthode `finalize` est appelée par le ramasse-miettes (donc pas forcément appelée !)
  - Elle permet un traitement spécifique lors de la libération de la mémoire
  - La correction ne doit pas dépendre de l'appel à `finalize`
- Ramasse-miettes $\neq$ destruction (*e.g.*, en C++)
  - Il n'est pas systématique ni spécifié
  - Il concerne uniquement la libération de ressources mémoire

Méthode `finalize`

- Elle est peut être nécessaire si de la mémoire a été allouée autrement que par Java (*e.g.*, par un programme C/C++ *via* JNI)
- Elle n'est appelée qu'une unique fois pour un objet
- Ramasse-miettes en deux passes :
  1. Détermine les objets qui ne sont plus référencés et appelle la méthode `finalize` pour ces objets
  2. Libère effectivement la mémoire
- Donc, si l'on veut un traitement particulier (autre que mémoire) lors de la fin de vie d'un objet (*e.g.*, fermer des flux), il est préférable de définir et appeler explicitement une méthode dédiée (pas `finalize`)

Illustration

```
public class Value {
  private static int cpt = 0;
  private int i;
  public Value() { this.i = cpt++; }
  public void finalize() { S.o.p(this.i + " finalized"); }
}
public class TestFinalize {
  public static void main(String[] args) {
    for (int i=0; i < Integer.parseInt(args[0]); i++)
      new Value();
  }
}
```

```
> java TestFinalize 100000
> java TestFinalize 110000
0 finalized
[...]
10 finalized
```

Factorisation de code ?

```
class Car implements Vehicle {
  private Engine engine;
  ...
  public void start() {
    this.engine.start();
  }
  ...
  public void turn(Direction d) {
    ... // traitement
  }
}
```

```
class Motorcycle implements Vehicle {
  private Engine engine;
  ...
  public void start() {
    this.engine.start();
  }
  ...
  public void turn(Direction d) {
    ... // traitement different
  }
}
```

- Attributs et code répétés
- Mais la méthode `turn` est différente pour les deux classes
- Comment concilier la factorisation de code et les différences ?

Début de solution

```
public class MotorVehicle implements Vehicle {
  protected Engine engine; ...
  public void start() { this.engine.start(); }
  public void turn(Direction d) { } // ne fait rien
}
public class Car extends MotorVehicle {
  ...
  public void turn(Direction d) { ... }  // traitement
}
public class Motorcycle extends MotorVehicle {
  ...
  public void turn(Direction d) { ... } // traitement
}                                       // different
```

- Un certain nombre de méthodes et d'attributs peuvent être définis de manière commune pour tous les véhicules motorisés
- En fait, cela n'a pas beaucoup de sens pour `MotorVehicle` d'avoir une implémentation (vide !) de `turn`
  $\Rightarrow$ Faire de `MotorVehicle` une classe abstraite

Classes abstraites

- Une classe abstraite déclare un ensemble de comportements, mais le définit seulement partiellement
  - Elle déclare des méthodes abstraites, c'est-à-dire sans comportement attaché (implémentation)
  - Elle peut toujours comporter des attributs et des méthodes non abstraites
  - C'est une classe intermédiaire entre l'interface et la classe concrète
  - Elle est utilisée pour factoriser du code entre les classes concrètes en utilisant le mécanisme d'héritage
- Une classe abstraite ne peut pas être instanciée
  - Elle peut néanmoins déclarer des constructeurs
  - Elle doit être dérivée afin d'implémenter les fonctionnalités manquantes
  - Les classes enfants doivent fournir une implémentation de toutes les méthodes abstraites de la classe parent
    - ★ Si non, elles doivent aussi être déclarées abstraites

Déclarer des classes abstraites

- Les classes abstraites sont déclarées comme des classes, mais en utilisant le mot clé `abstract` avant `class`
- Les méthodes abstraites sont déclarées en utilisant le mot clé `abstract` avant le type de retour, et sans corps

```
public abstract class MotorVehicle implements Vehicle {
  protected Engine engine;
  ...
  public void start() {
    this.engine.start();
  }
  public abstract void turn(Direction d);
  ...
}
```

Exemple : Car.java

```
public class Car extends MotorVehicle {
   ...
   public void turn(Direction d) {
      ... // traitement
   }
   ...
}
```

- La classe `Car` ne redéfinit pas la méthode `turn`
  - Il n'y a rien à redéfinir !
  - `Car` implémente `turn`
  - Il n'est évidemment pas possible de faire appel à `super.turn(...)`
- Encore une fois, les signatures de méthode doivent correspondre
  - Sans le modificateur `abstract`, bien sûr !

Retour sur les types énumérés

```
public enum Fuel {
  GASOLINE {
    public double evalCO2Emissions(double qty) { ... }
  },
  DIESEL {
    public double evalCO2Emissions(double qty) { ... }
  },
  LPG {
    public double evalCO2Emissions(double qty) { ... }
  };
  public abstract double evalCO2Emissions(double qty);
}
```

- Méthodes différenciées pour chaque constante

Mot clé `final`

- Il est possible d'interdire l'héritage d'une classe en utilisant le modificateur `final` avant `class`
  - Par exemple, les classes `String` ou encore `Boolean`
- Il est possible d'interdire la redéfinition d'une méthode en utilisant le modificateur `final` avant le type de retour
- `final` ∼ "constant"

```
public final class Podracer extends MotorVehicle {...}
public class Kart extends RaceCar {
  public final void beep() { ... }
}
public class KartDoubleDash extends Kart {
  // ne peut pas redefinir public void beep()
}
```

- Quels intérêts de définir une méthode `final` ?
  - Fermer le code à l'extension
  - Garantir le code exécuté indépendamment de tout héritage possible

Héritage multiple

- Rappel, l'héritage multiple de classes n'existe pas en Java
- Quel en serait l'intérêt ?
  - Faire hériter une classe simultanément de plusieurs classes et récupérer ainsi les comportements de chacune

```
public class Car extends MotorVehicle {
  public void move(short speed) { ... }
}
public class TimeMachine {
  public void backToTheFuture(Date date) { ... }
}
// interdit en Java
public class DeLorean extends Car, TimeMachine { ... }
...
DeLorean d = new DeLorean();
d.move(88);
d.backToTheFuture(new Date(21, 10, 2015));
Car c = d;              // upcast possible vers l'une ou
TimeMachine tm = d; // l'autre des classes
```

Problèmes de l'héritage multiple

- Quelle définition choisir en cas d'une déclaration multiple dans les classes parent ?

```
public class Car extends MotorVehicle {
  public void fillUp(double qty) { ... }
}
public class TimeMachine {
  public void fillUp(double qty) { ... }
}
// interdit en Java
public class DeLorean extends Car, TimeMachine { ... }
...
DeLorean d = new DeLorean();
d.fillUp(1.21); // Quelle definition de methode ?
```

- Comment résoudre ce problème ?
  - Une solution : imposer le transtypage

```
(Car) d.fillUp(1.21);
ou (TimeMachine) d.fillUp(1.21);
```

Problème du diamant (1)

- Est-ce suffisant ?

```
public abstract class MotorVehicle implements Vehicle {
  public abstract void fillUp(double qty);
}
public class Car extends MotorVehicle {
  public void fillUp(double qty) { ... }
  ...
}
public class TimeMachine extends MotorVehicle {
  public void fillUp(double qty) { ... }
  ...
}
// interdit en Java
public class DeLorean extends Car, TimeMachine { ... }
...
MotorVehicle v = new DeLorean();
v.fillUp(1.21); // Que se passe-t-il ? -> ambiguite !
```

Problème du diamant (2)

- Le problème du diamant est lié à l'héritage multiple de 2 classes (ici, `Car` et `TimeMachine`) qui héritent d'une même classe parent (ici, `MotorVehicle`) lors de l'*upcast* vers cette classe parent commune
  - Quelle définition de `fillUp(double qty)` choisir ?
- Le problème apparaît aussi pour d'éventuels attributs dans la classe `MotorVehicle`
  - Quelle copie de l'attribut considérer ?
  - Y a-t-il 2 copies ?
- En Java :
  - Héritage simple de classe
  - Implémentation multiple des interfaces

Solution Java : combiner interfaces et héritage

- Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre classe qu'il faut récupérer

```
public abstract class MotorVehicle implements Vehicle {
  public abstract void fillUp(double qty);
}
public class Car extends MotorVehicle {
  public void fillUp(double qty) { ... } ...
}
public interface TimeMachine {
  void backToTheFuture(Date date); ...
}
public class DeLorean extends Car implements TimeMachine {
  public void backToTheFuture(Date date) { ... }  ...
}
...
DeLorean d = new DeLorean();
MotorVehicle v = d;
v.fillUp(1.21);
Car c = d;
TimeMachine tm = d;
```

Intérêts de combiner interfaces et héritage

- Accroître le polymorphisme tout en évitant les problèmes liés à l'héritage multiple
- Accroître l'abstraction grâce à l'*upcast*

```
public interface TimeMachine {
  void backToTheFuture(Date date);
}
public class DeLorean extends Car implements TimeMachine {
  public void backToTheFuture(Date date) { ... } ...
}
public class Tardis extends PoliceBox implements TimeMachine {
  public void backToTheFuture(Date date) { ... } ...
}
public class Traveller {
  public void timeTravel(TimeMachine tm, Date date) {
    tm.backToTheFuture(date);
  }
}
...
(new Traveller()).timeTravel(new Tardis(), new Date(15,10,2015));
```

  - N'importe quelle classe peut potentiellement implémenter l'interface `TimeMachine` et ainsi être passée en paramètre de `timeTravel`

Héritage et composition

- Deux manières de réutiliser :
  - Héritage (relation "est un")
  - Composition (relation "a un")

Réutilisation par héritage

- Avantages :
  - Liaison dynamique et polymorphisme (utilisation de l'*upcast*)
  - Modification de code facilitée en remplaçant un objet de classe `C` par une instance d'une sous-classe de `C`
- Mais un changement de l'interface de la classe parent peut corrompre le code qu'utilise les sous-classes
  - La sous-classe dépend de l'implémentation de toutes ses classes parent
  - Encapsulation faible

Illustration

```
public class RaceCar extends Car {
  private String driver;
  ...
  public String getDriver() {
    return this.driver;
  }
}
public class Kart extends RaceCar { ... } // heritage
...
Kart k = new Kart();
String name = k.getDriver();
```

Changement de l'interface

```
public class Driver {
  private String name;
  ...
  public String getName() { return this.name; }
}
public class RaceCar extends Car { // interface modifiee
  private Driver driver;
  ...
  public Driver getDriver() { // type de retour modifie
    return this.driver;
  }
}
public class Kart extends RaceCar {...} // rien n'a change
...
Kart k = new Kart();
String name = k.getDriver(); // erreur : code corrompu
```

Réutilisation par composition

- Au lieu d'hériter d'une classe `C`, une classe peut :
  1. Détenir une référence à une instance de `C`
  2. Encapsuler les méthodes à récupérer de `C` dans de nouvelles méthodes
- Il n'y a pas de récupération automatique de toute l'interface publique de `C`
  - Elle doit être explicite : délégation
  - `C` devient une classe *back-end*
  - Encapsulation plus forte
    - ★ L'interface de la classe *front-end* (*i.e.*, la classe englobante) ne peut pas être changée en changeant celle de `C`

Illustration

```
public class RaceCar extends Car {
  private String driver;
  ...
  public String getDriver() {
    return this.driver;
  }
}
public class Kart {
  private RaceCar car = new RaceCar(); // composition
  ...
  public String getDriver() {
    return this.car.getDriver(); // delegation
  }
}
...
Kart k = new Kart();
String name = k.getDriver();
```

Changement de l'interface

```
public class Driver {
  private String name; ...
  public String getName() { return this.name; }
}
public class RaceCar extends Car { // interface modifiee
  private Driver driver; ...
  // type de retour modifie
  public Driver getDriver() { return this.driver; }
}
public class Kart {
  private RaceCar car = new RaceCar(); ...
  public String getDriver() { // code a changer
    return (this.car.getDriver()).getName();
  }
}
...
Kart k = new Kart();
String name = k.getDriver(); // ok
```

Composition > Héritage

- Il est plus facile de changer l'interface d'une classe *back-end* que d'une classe parent
  - Permet de conserver l'interface de la classe *front-end*
- Il est plus facile de changer l'interface d'une classe *front-end* que d'une sous-classe
  - Il n'est pas toujours possible de changer l'interface d'une méthode héritée (*e.g.*, le type de retour)
- Il est possible de retarder la création des objets agrégés
  - Avec l'héritage, la classe parent est créée dès la création de l'objet
- Le changement de l'implémentation (pas de l'interface) est aussi facile dans les deux cas

Héritage > composition

- Il est plus facile d'ajouter des sous-classes par héritage que d'ajouter des classes *front-end* par composition
  - Grâce au polymorphisme, il est possible de facilement passer d'une (instance de) super-classe à sa sous-classe sans changer de code
- La délégation d'invocation de méthode de la composition a un coût, surtout si elle est systématique
- Le lien par héritage est plus facile à appréhender que le lien par composition
- Le changement de l'implémentation (pas de l'interface) est aussi facile dans les deux cas

Héritage et composition

- Deux manières de réutiliser :
  - Héritage (relation "est un")
    - ★ Plus facile d'hériter des sous-classes
  - Composition (relation "a un")
    - ★ Plus de flexibilité
    - ★ Risque de corruption de code limité
- Par rapport à l'héritage, la composition diminue le couplage
- Ne pas utiliser l'héritage juste pour "récupérer" du code, préférer la composition
  - La relation "est un" est-elle durable/fiable pour l'application/API ?
    - ★ Par exemple, on peut avoir une `Person` "est un" `Driver`, mais *quid* si la `Person` perd tous les points de son permis de conduire ?
- Penser à combiner interfaces et composition

Combiner interfaces et composition

```
public class Drivable {
  public Driver getDriver();
}
public class RaceCar extends Car {
  private String driver; ...
  public String getDriver() {
    return this.driver;
  }
}
public class Kart implements Drivable {
  private RaceCar car = new RaceCar();
  ...
  public Driver getDriver() {
    return new Driver(this.car.getDriver());
  }
}
public class Game() {
  public void foo(Drivable d) { S.o.p(d.getDriver()); }
}
...
Game g = new Game();
g.foo(new Kart());
```

- Le problème de la modification de la classe *back-end* est pris en charge par la composition
- Le problème de l'ajout de sous-classe est prise en charge par l'interface `Drivable`

Au menu

Gestion des erreurs à l'exécution

- Parfois, le code peut détecter une erreur, mais pas nécessairement la résoudre
  - Par exemple, un code peut détecter qu'un fichier ne peut pas être ouvert, mais que doit-il faire ?
- Plusieurs manières d'indiquer des erreurs à l'appelant
  - Retourner une valeur d'erreur spéciale
    - ★ Sauf si c'est un constructeur, lequel ne peut pas retourner de valeur !
    - ★ Possiblement difficile à traiter
  - Lever/lancer une exception pour signaler l'erreur

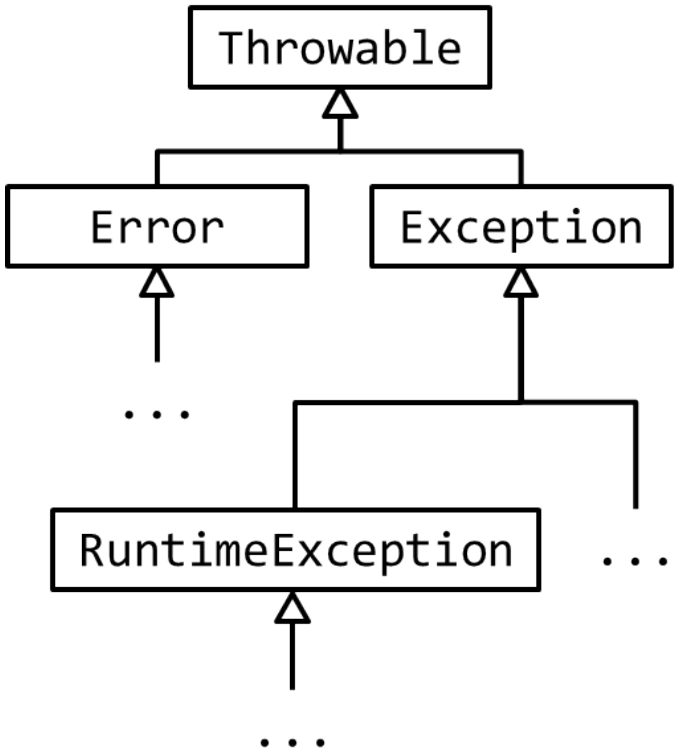
Exceptions

- Une exception est un signal déclenché en cas de problème
- Idée :
  - Anticiper le code susceptible de produire une erreur
  - Séparer le traitement des cas normaux de celui des cas exceptionnels
    - ★ Un cas exceptionnel est une situation qui ne correspond au fonctionnement normal du programme
- En Java, une exception est un objet de type `Exception`
  - Les sous-classes d'`Exception` se nomment par convention *XXX*`Exception`

Exemples de cas exceptionnels

- Appel d'une méthode sur une référence `null`
  - `NullPointerException`
- Accès à des cases d'un tableau en dehors des indices autorisés
  - `ArrayIndexOutOfBoundsException`
- Transtypage impossible
  - `ClassCastException`
- Division par zéro
  - `ArithmeticException`
- Ouverture en lecture d'un fichier qui n'existe pas
  - `FileNotFoundException`
- *etc.*

Hiérarchie d'exceptions Java



- `Throwable`
  - Classe de base pour tous les objets pouvant être lancés
    - `void printStackTrace()`
    - `String getMessage()`
    - `String getLocalizedMessage()`
- `Error`
  - Sérieux problèmes dans la JVM (*e.g.*, manque de ressource) que les applications ne traiteront généralement pas
- `Exception`
  - Problèmes ordinaires que les applications pourraient vouloir traiter
- `RuntimeException`
  - Les applications peuvent ou non les traiter
  - En général, elles indiquent des erreurs de programmation

Créer une classe d'exception

- Définir une classe dérivant d'`Exception` ou de l'une de ses sous-classes déjà définies, en la nommant de la forme *XXX*`Exception`
- Spécialiser un type d'erreur, ajouter une information dans l'objet exception...
  - Exceptions spécifiques à l'application
  - Sinon utiliser les exceptions existantes

```
public class OutOfFuelException extends Exception {
  public OutOfFuelException(String msg) {
    super(msg);
  }
}
```

Mécanisme de gestion des exceptions

- La gestion des exceptions se décompose en deux phases :
  1. Lever (ou lancer) une exception pour signaler l'erreur (*i.e.*, là où un problème se pose dans le code)
  2. Capturer (ou attraper) une exception pour proposer une solution/alternative et ainsi rétablir une situation stable (*e.g.*, quitter le programme proprement)
     - ★ Les exceptions sont transmises au gestionnaire d'exceptions (*exception handler*) afin d'être traitées

Lever des exceptions (1)

- Pour lever explicitement une exception dans un code :
  1. Créer un objet exception (de la classe d'exception voulue)
  2. Lever l'exception en utilisant le mot clé `throw`
- Il est possible de spécifier un message d'erreur pour les exceptions
  - Il doit indiquer ce qui est attendu et ce qui est réellement arrivé

```
...
if (this.getFuelQuantity() == 0) {
  throw new OutOfFuelException("demarrage de la
voiture impossible , pas de carburant");
}
this.engine.start();
...
```

Lever des exceptions (2)

- Lorsqu'une exception est levée, l'exécution est immédiatement transférée au gestionnaire de cette exception
  - Cela casse le flot de contrôle du programme
  - Le calcul courant est interrompu, mais pas le programme !
- Dans cet exemple, lorsqu'une exception est levée, plus aucun code après n'est exécuté, retour à l'appelant
  - Le moteur n'est pas démarré

```
...
if (this.getFuelQuantity() == 0) {
  throw new OutOfFuelException("demarrage de la
voiture impossible , pas de carburant");
}
this.engine.start();
...
```

Gestionnaire d'exceptions (1)

- Lorsqu'un code est susceptible de lever une exception, il est possible d'essayer d'exécuter ce code et de capturer l'exception pour la traiter, en utilisant la séquence d'instructions `try/catch/finally`

```
...
try {
   this.engine.start(); // peut lever une exception
   System.out.println("Moteur OK");
} catch (EngineException e) {
   System.err.println("Probleme : " + e.getMessage());
} finally {
   System.out.println("Fin du demarrage");
}
```

  - Bloc `try` : code susceptible de lever des exceptions
  - Bloc `catch` : code de traitement pour un type particulier d'exception
    - ★ Il peut y avoir plusieurs blocs `catch` pour un même bloc `try`
  - Bloc `finally` (optionnel) : code à exécuter dans tous les cas, exception levée ou non
  - Il faut au moins un bloc `catch` ou `finally` pour chaque bloc `try`

Gestionnaire d'exceptions (2)

- Si la méthode `start()` lève une exception, l'exécution est immédiatement transférée au bloc `catch` avec le même type d'exception
- Le bloc `catch (EngineException e)` capture toute exception de type `EngineException` et ses sous-classes

```
...
try {
   this.engine.start(); // peut lever une exception
   System.out.println("Moteur OK");
} catch (EngineException e) {
   System.err.println("Probleme : " + e.getMessage());
} finally {
   System.out.println("Fin du demarrage");
}
```

  - `"Moteur OK"` n'est pas affichée sur la sortie standard
  - L'erreur est affichée à la place sur la sortie d'erreur
  - `"Fin du programme"` est affichée sur la sortie standard

Gestionnaire d'exceptions (3)

- Un bloc `try` peut uniquement traiter des exceptions qui se produisent à l'intérieur de ce bloc de code
- Un même bloc `try` peut être susceptible de lever plusieurs exceptions
  - Il est possible de les traiter séparément ou globalement
- Le type de l'exception détermine quel bloc `catch` traite réellement l'exception
  - Spécifier un ou plusieurs blocs `catch` immédiatement après le bloc `try`
  - Le premier bloc `catch` avec le type correspondant traitera l'exception
    - ★ Attention à l'ordre des blocs `catch`
  - Après l'exécution du bloc `catch`, l'exécution reprend après l'instruction `try/catch`
    - ★ Seulement un bloc `catch` est exécuté
  - Le bloc `try` n'est pas ré-exécuté après le traitement de l'exception
    - ★ Même si la cause de l'erreur a été corrigée par ce traitement

Ordre des blocs `catch`

- L'ordre des blocs `catch` est important à cause de l'héritage entre les classes d'exceptions
  - Il faut placer les sous-classes avant leur classe parent
- Sinon, le compilateur génère une erreur

```
// ordre incorrect
try {
  this.engine.start();
} catch (Exception e) {
  // capture Exception et
  // ses sous-classes
  ...
} catch (EngineException e) {
  // erreur, EngineException
  // deja capturee
  ...
}
```

```
// ordre correct
try {
  this.engine.start();
} catch (EngineException e) {
  ...
} catch (Exception e) {
  // capture Exception
  // et ses sous-classes
  ...
}
```

Relancer une exception

- Une exception peut être partiellement traitée, puis relancée

```
try {
  this.engine.start();
} catch (EngineException e) {
  ... // traitement partiel de l'exception
  throw e; // relance l'exception
}
```

- Il est également possible de relancer une exception d'un autre type
  - Cette dernière ayant l'exception originale comme cause

```
try {
  this.engine.start();
} catch (EngineException e) {
  throw new RuntimeException(...);
}
```

Trace d'appels

- Une exception est remplie avec une trace d'appels de la pile d'exécution (*stack trace*)

```
Exception in thread "main" java.lang.EngineException:
at Engine.start(Engine.java:5)
at Car.start(Car.java:17)
at Car.main(Car.java:22)
```

  - Spécifie où l'objet exception a été créé, mais pas où il a été lancé
    - La méthode `printStackTrace()` affiche la localisation de la création de l'instance
    - Pour mettre à jour la pile d'appels d'une exception pré-existante (*e.g.*, pour relancer une exception), utiliser la méthode `fillInStackTrace()`

```
try {
  this.engine.start();
} catch (EngineException e) {
  ... // traitement partiel de l'exception
  e.fillInStackTrace(); // maj de la pile d'appels
  throw e; // relance l'exception
}
```

  - Le mieux est de créer l'exception juste au moment où elle est levée

Exceptions vérifiées (*checked exceptions*) (1)

- N'importe quelle sous-classe d'`Exception` qui ne dérive pas de `RuntimeException`
- Une exception peut être traitée directement par la méthode dans laquelle elle est levée, mais elle peut aussi être envoyée à la méthode appelante, car la méthode ne souhaite pas ou ne peut pas la traiter
- Dans ce cas, les méthodes doivent déclarer quels types d'exception vérifiée elles lèvent, en utilisant le mot clé `throws`
- Javadoc : tags `@exception`, `@throws`

```
public void start() throws OutOfFuelException,
                            EngineException {
  if (this.getFuelQuantity() == 0) {
    throw new OutOfFuelException("demarrage de la
voiture impossible, pas de carburant");
  }
  this.engine.start(); // peut lever des exceptions
}                      // de type EngineException
```

Exceptions vérifiées (*checked exceptions*) (2)

- Le compilateur Java vérifie le code de la méthode par rapport aux spécifications
  - Un autre aspect de Java pour imposer la correction de programme
  - Force les programmes à traiter les exceptions, ou à explicitement déclarer ce qui pourrait être lancé
- Si une exception de type `OutOfFuelException` ou `EngineException` est levée durant l'exécution de `start()`, l'exception sera envoyée à la méthode appelant `start()`, qui devra la traiter

```
public static void main(String[] args) {
  Car c = new Car(...);
  try {
    c.start();
    c.move(88);
  } catch (OutOfFuelException e) { ... }
  } catch (EngineException e) { ... }
}
```

Exceptions vérifiées (*checked exceptions*) (3)

- Une méthode peut spécifier une classe de base de ce qu'elle lève

```
public void start() throws OutOfFuelException,
                            EngineException {
   ...
}
```

  - Toutes ces exceptions dérivent de la classe `EngineException`
    - ★ `EngineOilException` : niveau d'huile moteur insuffisant
    - ★ `CoolantException` : surchauffe du moteur
    - ★ *etc.*
  - La méthode `start()` pourrait également les lever sans changer sa spécification d'exception
- Un code peut aussi capturer le type de la classe de base

```
try { ... } catch (EngineException e) { ... }
```

  - Capture et traite toutes les exceptions ci-dessus
  - Traitement global

throws et sous-classes

- Problème non détecté à la compilation

```
public void start() throws OutOfFuelException,
                           EngineException {
  if (this.getFuelQuantity() == 0) {
    throw new OutOfFuelException("demarrage de la
voiture impossible, pas de carburant");
  }
  try {
    this.engine.start();
  } catch (Exception e) { // capture Exception
    ...                   // et ses sous-classes
  }
}
```

  - Cette méthode ne lèvera jamais d'exception de type `EngineException` car cette sous-classe d'`Exception` est déjà capturée

Exceptions et redéfinition de méthode

- Lors de la redéfinition d'une méthode, la signature doit être rigoureusement la même que celle de la classe parent, jusqu'aux exceptions !
- Toutefois, il est possible d'affiner les exceptions levées par la méthode par des sous-classes des exceptions originales

```
public class EngineException extends Exception { ... }
public class RaceCarEngineException extends EngineException{...}
public class Car {
  public void start() throws EngineException { ... }
}
public class RaceCar extends Car {
  public void start() throws RaceCarEngineException { ... }
}
...
Car c = new RaceCar(...);
try { c.start(); }
catch(RaceCarEngineException e) { ... } // attention a l'ordre
catch(EngineException e) { ... }   // de capture des exceptions
```

Exceptions non vérifiées (*unchecked exceptions*)

- Certaines exceptions sont levées implicitement par la machine virtuelle
  - `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, *etc.*
- Les exceptions non vérifiées héritent de `RuntimeException`
  - Elles n'ont pas besoin d'être obligatoirement capturées
  - Elles n'ont pas besoin d'être déclarées avec l'instruction `throws`
  - Elles ne sont pas censées être lancées par une méthode codée et utilisée correctement
- Sans ce type d'exception, il faudrait déclarer `throws ArrayIndexOutOfBoundsException` chaque fois qu'une méthode utilise un tableau, ou encore `throws NullPointerException` chaque fois qu'un objet accède à ses membres

Exceptions et API Java

- La documentation de l'API Java indique quelles exceptions sont levées et aussi quand elles sont levées
- Les bibliothèques d'entrées/sorties et de réseau peuvent lever beaucoup d'exceptions
- C'est toujours très important de traiter proprement les exceptions, afin de rendre les applications robustes !

Cloner des objets

- Comment dupliquer un objet pour avoir deux versions de cet objet susceptibles d'évoluer différemment ?
  - Constructeur par recopie
  - Méthode `clone()` de la classe `Object`
  - L'appel de la méthode `clone()` est résolue dynamiquement (cf. polymorphisme), contrairement au constructeur par recopie

Méthode `clone`

```
protected Object clone() throws CloneNotSupportedException {
   ...
}
```

- Réservation de l'espace mémoire : copie bit à bit
  - C'est la référence des attributs qui est copiée
  - Copie superficielle
- `protected`

```
...
Car c = new Car(...);
Car clone = (Car) c.clone(); // illegal, clone() non accessi
```

- Type de retour `Object` : *downcast*
- Exception de type `CloneNotSupportedException` levée si la classe de l'objet à cloner n'implémente pas l'interface `Cloneable`

Interface `Cloneable`

- Pour permettre le clonage des objets, la classe doit implémenter l'interface `Cloneable`
  - Permet de typer les objets clonables (test avec `instanceof`)
  - Permet aux développeurs d'avoir des classes d'objets non clonables
    - ★ `Object.clone()` vérifie si la classe implémente `Cloneable`
- Déclarer `public` la méthode `clone()`
- Appeler systématiquement `super.clone()`

```
public class Car extends MotorVehicle implements Cloneable{
  ...
  public Object clone() throws CloneNotSupportedException {
    Car clone = (Car) super.clone(); // downcast
    ...
    return clone;
  }
}
```

Copie superficielle vs profonde des références

```
// copie profonde
public Object clone() throws CloneNotSupportedException {
  Car clone = (Car) super.clone(); // downcast
  clone.engine = new Engine(this.engine); // constructeur
                                          // par recopie
  // ou utiliser clone() sur engine s'il est clonable
  ...
  return clone;
}
...
public static void main(String[] args) {
  Car c = new Car(...);
  Car clone = null;
  try {
    clone = (Car) c.clone();
  } catch(CloneNotSupportedException e) { }
  S.o.p("Meme moteur ? " + (c.engine == clone.engine));
}
```

Au menu

Flux d'entrées/sorties

- Java fournit un mécanisme d'entrées/sorties fondé sur la notion de flux (*stream*)
  - Un flux est canal de communication entre un *écrivain* et un *lecteur*
- Package `java.io` : `java.io.InputStream`, `java.io.OutputStream`
  - Classes de base abstraites qui spécifient toutes les opérations (de bas niveau) de lecture (entrée) et d'écriture (sortie) que les flux doivent fournir
  - Ce sont sont des flux d'octets
    - ★ Les valeurs effectivement transférées sont des octets
    - ★ Souvent, ils ne conviennent pas pour des données textuelles
- Les interfaces `java.io.Reader` et `java.io.Writer` utilisent des caractères
  - Essentiellement les mêmes opérations que `InputStream` et `OutputStream`, mais avec des valeurs de type `char` (Unicode)

Méthodes de `InputStream` et de `OutputStream`

- Méthodes de `InputStream`
  - `read()` lit un ou plusieurs octets
    - ★ Méthode bloquante : ne retournera pas jusqu'à ce que davantage de données soient disponibles, ou si elle sait qu'une lecture va certainement échouer
  - `available()` retourne combien d'octets peuvent être lus sans bloquer
  - `close()` ferme le flux d'entrée et libère les ressources associées au flux
- Méthodes de `OutputStream`
  - `write()` écrit un ou plusieurs octets
  - `flush()` vide la mémoire tampon (*buffer*) en écrivant les données dans le flux de sortie
  - `close()` ferme le flux de sortie

Différents types de flux

- `InputStream[Reader|Writer]`
  - Flux qui permet de transformer des caractères en octets et *vice versa*
- `Data[Input|Output]Stream`
  - Flux qui permet de lire/écrire des données de types primitifs à partir d'un autre flux d'une manière indépendante de la machine
- `Object[Input|Output]Stream`
  - Flux qui permet de lire/écrire des objets sérialisés
  - La lecture implique la reconstruction d'objets
- `Buffered[Input|Output]Stream/Buffered[Reader|Writer]`
  - Flux qui ajoute une mémoire tampon à un autre flux pour améliorer les performances
- `Piped[Input|Output]Stream/Piped[Reader|Writer]`
  - Flux qui fonctionne par paire, utilisé notamment pour la communication entre *threads*
- `File[Input|Output]Stream/File[Reader|Writer]`
  - Flux qui permet de lire/écrire dans un fichier

Composition de flux (1)

- L'API Java de flux d'entrées/sorties supporte la composition de flux
- Exemple : lire les lignes d'un fichier texte

```
FileInputStream fis = new FileInputStream("foo.txt");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
String line;
// lecture de la ligne
while ((line = br.readLine()) != null) { ... }
// ou (new BufferedReader(new InputStreamReader(
//       new FileInputStream("foo.txt")))).readLine();
// ou (new BufferedReader(new FileReader("foo.txt")))
//       .readLine();
```

  - `FileInputStream` dérive de `InputStream`
  - Envelopper le flux d'entrée avec un `Reader` pour lire les caractères
  - Ajouter de la mémoire tampon au lecteur afin de pouvoir lire des lignes de texte entières

Composition de flux (2)

- Exemple : écrire dans un fichier texte

```
FileWriter fw = new FileWriter("foo.txt");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter pw = new PrintWriter(bw);
pw.println("Une ligne de texte");
// ou (new PrintWriter(new BufferedWriter(FileWriter(
//       "foo.txt")))).println("Une ligne de texte");
```

- Penser à fermer les flux ouverts avec la méthode `close()`
- Exemple : lire les entrées de l'utilisateur

```
(new BufferedReader(new InputStreamReader(System.in)))
   .readLine();
```

ou

```
new java.util.Scanner(System.in).nextLine();
```

Fichiers (1)

- Java représente les fichiers avec la classe `java.io.File`
  - Soit les chemins absolus, soit les chemins relatifs
- Les chemins absolus commencent au répertoire racine du système de fichiers
  - Windows : `"C:\Documents and Settings\jml\Desktop\foo.txt"`
    - ★ Rappel : le caractère '\' doit être protégé dans les chaînes de caractères
  - Unix : `"/home/jml/Desktop/foo.txt"`
- Les chemins relatifs commencent à partir du répertoire courant
  - Utiliser `.` pour désigner le répertoire courant
  - Utiliser `..` pour désigner le répertoire parent du répertoire courant

Fichiers (2)

- `java.io.File` fournit plusieurs constantes :
  - `File.separator` est une chaîne de caractères contenant le séparateur qui apparaît dans les chemins
    - Windows : `"\\"`
    - Unix : `"/"`
  - `File.separatorChar` est identique à `File.separator`, mais de type `char`
  - `File.pathSeparator` est une chaîne de caractères contenant le séparateur de chemins
    - Windows : `";"`
    - Unix : `":"`
  - `File.pathSeparatorChar` est identique à `File.pathSeparator`, mais de type `char`
- Utiles lorsqu'un programme doit générer un *classpath* ou une autre collection de chemins de fichiers/répertoires

Créer des objets `File`

- Constructeurs de `File` :
  - `File(String pathname)`
    - ★ Spécifie un chemin relatif ou absolu vers le fichier
  - `File(File parent, String child)`
    - ★ Suppose que `parent` est un répertoire
    - ★ Crée un nouvel objet `File` pour référencer un fichier `child` dans le répertoire `parent`
  - `File(String parent, String child)`
    - ★ Identique au constructeur précédent
- Ces constructeurs ne testent pas si les fichiers existent réellement !

Examiner des objets `File`

- De nombreuses méthodes utiles pour examiner les fichiers :
  - `boolean exists()`
    - ★ Existe-t-il un fichier ou un répertoire du système de fichiers correspondant à l'objet `File` ?
  - `boolean isFile()`
    - ★ Est-ce que l'objet `File` est un fichier "normal" ?
    - ★ Vérifie que ce n'est pas un répertoire et effectue certaines vérifications spécifiques au système
  - `boolean isDirectory()`
    - ★ Est-ce que l'objet `File` est un répertoire ?
  - `boolean canRead()`
    - ★ Est-ce que le fichier existe et peut-il être lu par l'application ?
  - `boolean canWrite()`
    - ★ Est-ce que le fichier existe et peut-il être écrit par l'application ?
  - `long length()`
    - ★ Retourne la taille du fichier (en octets)

Manipuler des objets `File`

- Il est possible d'effectuer des opérations de base sur les fichiers :
  - `boolean delete()`
    - ★ Supprime un fichier ou un répertoire (s'il est vide)
    - ★ Retourne `true` en cas de succès, `false` sinon
  - `boolean renameTo(File dest)`
    - ★ Renomme un fichier ou un répertoire vers un autre emplacement
    - ★ Peut ne pas réussir si par exemple le fichier de destination existe déjà

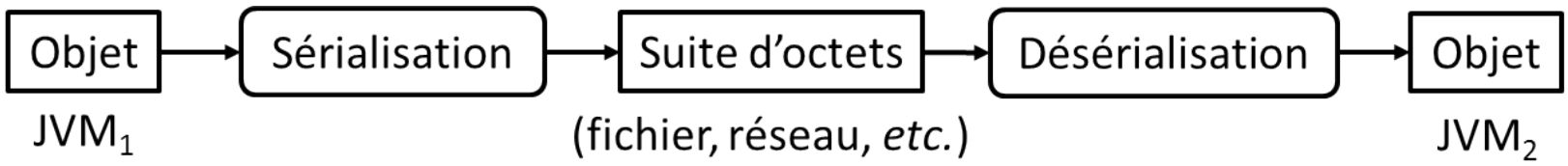
Naviguer dans le système de fichiers

- Il est possible d'utiliser `File` pour naviguer dans le système de fichiers :
  - `File[] File.listRoots()`
    - ★ Méthode statique qui retourne un tableau d'objets `File` spécifiant les répertoires racine du système
  - `File[] listFiles()`
    - ★ Méthode d'instance qui retourne un tableau d'objets `File` désignant les fichiers dans le répertoire
    - ★ Il est aussi possible de spécifier un filtre en paramètre
    - ★ Pour cela, implémenter l'interface `FilenameFilter` ou `FileFilter` afin d'exclure des fichiers en fonction de certains critères

Flux d'entrées/sorties et exceptions

- Les objets `File` signalent certains échecs avec une valeur de retour booléenne
  - `boolean delete()`
  - `boolean renameTo(File dest)`
- La plupart des opérations d'entrées/sorties signalent des échecs en levant des exceptions
  - En général `java.io.IOException`, ou certaines sous-classes de cette exception

Sérialisation/désérialisation



- Sérialisation :
  - Transformer un objet en mémoire en une suite d'octets le représentant
- Cette suite d'octets peut être utilisée pour :
  - La sauvegarde (persistance entre sessions)
  - Le transport sur le réseau (*e.g.*, *proxy*, *RMI*)
- Désérialisation :
  - Transformer cette suite pour créer une copie conforme de l'objet d'origine
- Il est possible de sérialiser et désérialiser entre JVM de systèmes d'exploitation différents

Sérialiser/désérialiser un objet

- La classe de l'objet doit implémenter `java.io.Serializable`
- Sérialisation par défaut
  - Méthode `writeObject` de la classe `ObjectOutputStream`
- Désérialisation par défaut
  - Méthode `readObject` de la classe `ObjectInputStream`

```
public final void writeObject(Object obj) throws IOException
```

- Écrit l'objet `obj` dans le flux de sortie `ObjectOutputStream`
  - La classe de l'objet, la signature de la classe et les valeurs des attributs non `transient` et non `static` de la classe et de toutes ses classes parent sont écrites
- Réalise la clôture transitive des dépendances sur l'objet sérialisé
  - Les objets référencés par cet objet sont écrits transitivement afin qu'un graphe d'objets équivalent complet puisse être reconstruit par un `ObjectInputStream`
  - Si un objet o à sérialiser possède des références à des objets o_1 et o_2, ces derniers seront aussi sérialisés (et donc y compris les objets qu'ils référencent : clôture transitive du graphe de dépendances des objets)
- Peut lever des exceptions :
  - `InvalidClassException`
    - ★ Quelque chose ne va pas avec une classe utilisée par la sérialisation
  - `NotSerializableException`
    - ★ Certains objets à sérialiser n'implémente pas l'interface `Serializable`
  - `IOException`
    - ★ N'importe quelle exception levée par le flux de sortie

`public final Object readObject() throws OptionalDataException, ClassNotFoundException, IOException`

- Lit un objet à partir du flux d'entrée `ObjectInputStream`
  - La classe de l'objet, la signature de la classe et les valeurs des attributs non `transient` et non `static` de la classe et de toutes ses classes parent sont lues
- Les objets référencés par cet objet sont lus transitivement afin qu'un graphe d'objet complet équivalent soit reconstruit par `readObject`
  - L'objet racine est complètement restauré lorsque l'ensemble de ses champs et les objets qu'il référence sont complètement restaurés
- Évidemment le fichier `.class` doit être accessible à la JVM d'accueil
- Peut lever des exceptions :
  - `ClassNotFoundException`
    - La classe d'un objet sérialisé ne peut être trouvé
  - `InvalidClassException`
    - Quelque chose ne va pas avec une classe utilisée par la désérialisation
  - `StreamCorruptedException`
    - L'information de contrôle dans le flux est incohérente
  - `OptionalDataException`
    - Des données primitives ont été trouvé dans le flux à la place d'objets
  - `IOException`

Illustration (1)

```
public class Value implements java.io.Serializable {
  private int i;
  private Value v = null;

  public Value(int i) {
    this.i = i;
  }

  public Value(int i, int j) {
    this(i);
    this.v = new Value(j);
  }

  public String toString() {
    return "" + this.i + (v==null ? "" : " et " + this.v);
  }
}
```

Illustration (2)

```
...
Value v1 = new Value(2, 2);
Value v2 = new Value(1, 7);
try {
  ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("foo.dat"));
  out.writeObject(v1); out.writeObject(v2);
  out.close();
} catch(Exception e) { }

... // dans une autre classe independante
try {
  ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("foo.dat"));
  Value v1 = (Value) in.readObject();
  Value v2 = (Value) in.readObject();
  in.close();
  System.out.println(v1); // affiche 2 et 2
  System.out.println(v2); // affiche 1 et 7
} catch (Exception e) { }
```

Modificateur d'attribut `transient`

- Il spécifie que l'attribut ne doit pas être sérialisé (ni donc restauré)
- L'attribut aura une valeur `null` à la désérialisation
- C'est le programme qui doit prendre en charge l'attribut

Personnaliser la sérialisation/désérialisation

- La sérialisation/désérialisation par défaut d'une classe peuvent être surchargées, en utilisant les méthodes `writeObject` et `readObject`
- Ajouter à une classe implémentant l'interface `Serializable` les méthodes :
  - `private void writeObject(ObjectOutputStream out) throws IOException`
  - `private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException`
- Ces méthodes sont privées !
- Elles sont appelées automatiquement par les méthodes `writeObject` de `ObjectOutputStream` et `readObject` de `ObjectInputStream`
- Les méthodes `defaultWriteObject` de `ObjectOutputStream` et `defaultReadObject` de `ObjectInputStream` gèrent tous les attributs non `transient` et non `static`

Illustration

```
public class Value implements java.io.Serializable {
  private int i;
  private Value v = null;
  private transient String s;
  public Value(int i) {
    this.i = i;
    this.s = ""+i;
  }
  public Value(int i, int j) {
    this(i);
    this.v = new Value(j);
  }
  private void writeObject(ObjectOutputStream out) throws
    IOException {
   out.defaultWriteObject();
   out.writeObject((new CaesarCipher()).encrypt(s));
  }
  private void readObject(ObjectInputStream in) throws
    IOException, ClassNotFoundException {
    in.defaultReadObject();
    this.s = (new CaesarCipher()).decrypt((String)in.readObject());
  }
}
```

Cohérence de la sérialisation

- Problème :
  - Deux objets o_1 et o_2 partagent une référence sur l'objet o_3
  - On sérialise o_1 et o_2 (qui chacun sérialise o_3) *via* le même flux
  - Qu'en sera-t-il lors de leur désérialisation vis-à-vis de o_3 ?

Cohérence de la sérialisation

- Problème :
  - Deux objets o_1 et o_2 partagent une référence sur l'objet o_3
  - On sérialise o_1 et o_2 (qui chacun sérialise o_3) *via* le même flux
  - Qu'en sera-t-il lors de leur désérialisation vis-à-vis de o_3 ?
- Les dépendances d'objets sont conservées !

Illustration

```
public class Value implements java.io.Serializable {
  private int i;
  private Value v = null;
  public Value(int i) { this.i = i; }
  public Value(int i, Value v) {
    this(i);
    this.v = v;
  }
  public boolean sameV(Value v) {
   return this.v == v.v;
  }
}
...
Value v = new Value(0);
Value v1 = new Value(1, v);
Value v2 = new Value(2, v);
... // Serialisation, puis ouverture d'un flux d'entree in
Value v1 = (Value) in.readObject();
Value v2 = (Value) in.readObject();
System.out.println(v1.sameV(v2)); // true
```