



Programmation Orientée Objet application au langage Java

Version Août 2009

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

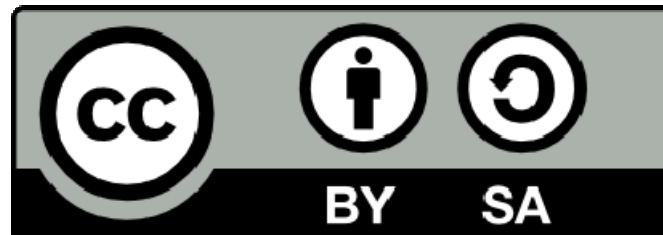
Licence

Creative Commons

Contrat Paternité

Partage des Conditions Initiales à l'Identique

2.0 France



<http://creativecommons.org/licenses/by-sa/2.0/fr>

Rapide historique concernant la POO

- La POO a une « longue » histoire
 - POO ? : Programmation Orientée Objet
 - Début en Norvège à la fin des années 60
 - Simula, programmation des simulations par événements discrets
 - Programmation structurée pas adaptée
- Dans les années 70
 - Développement par Xérox du premier système Fenêtres, Icônes et Souris
 - **SmallTalk**, « archétype » des langages objets
 - Développement par Apple des premiers Mac

Rapide historique concernant la POO

- Au cours des années 80
 - Développement des ordinateurs personnels
 - Intérêt pour les interfaces graphiques
 - Apparition de nouveaux langages
 - **Eiffel**, fortement typé, entièrement OO
 - **C++**, extension de C, pas totalement OO
 - **Object Pascal (Delphi)** développé par Borland
- Dans les années 90, vers une maturité des concepts objets
 - Standardisation de C++
 - Apparition de langages comme **Java** ou **Python**

Programmation Structurée VS POO

➤ Objectifs de la POO

- Facilité la réutilisation de code, encapsulation et abstraction
- Facilité de l'évolution du code
- Améliorer la conception et la maintenance des grands systèmes
- Programmation par « composants ». Conception d'un logiciel à la manière de la fabrication d'une voiture

➤ Programmation Structurée

- Unité logique : le module
- Une zone pour les variables
- Une zone pour les fonctions
- Chaque fonction résout une partie du problème
- Structuration « descendante » du programme

Principes POO : programmation par objets

➤ Unité logique : l'objet

➤ Objet est défini par

➤ un état

➤ un comportement

➤ une identité

➤ État : représenté par des attributs (variables) qui stockent des valeurs

➤ Comportement : défini par des méthodes (procédures) qui modifient des états

➤ Identité : permet de distinguer un objet d'un autre objet

<u>maVoiture</u>
- couleur = bleue
- vitesse = 100

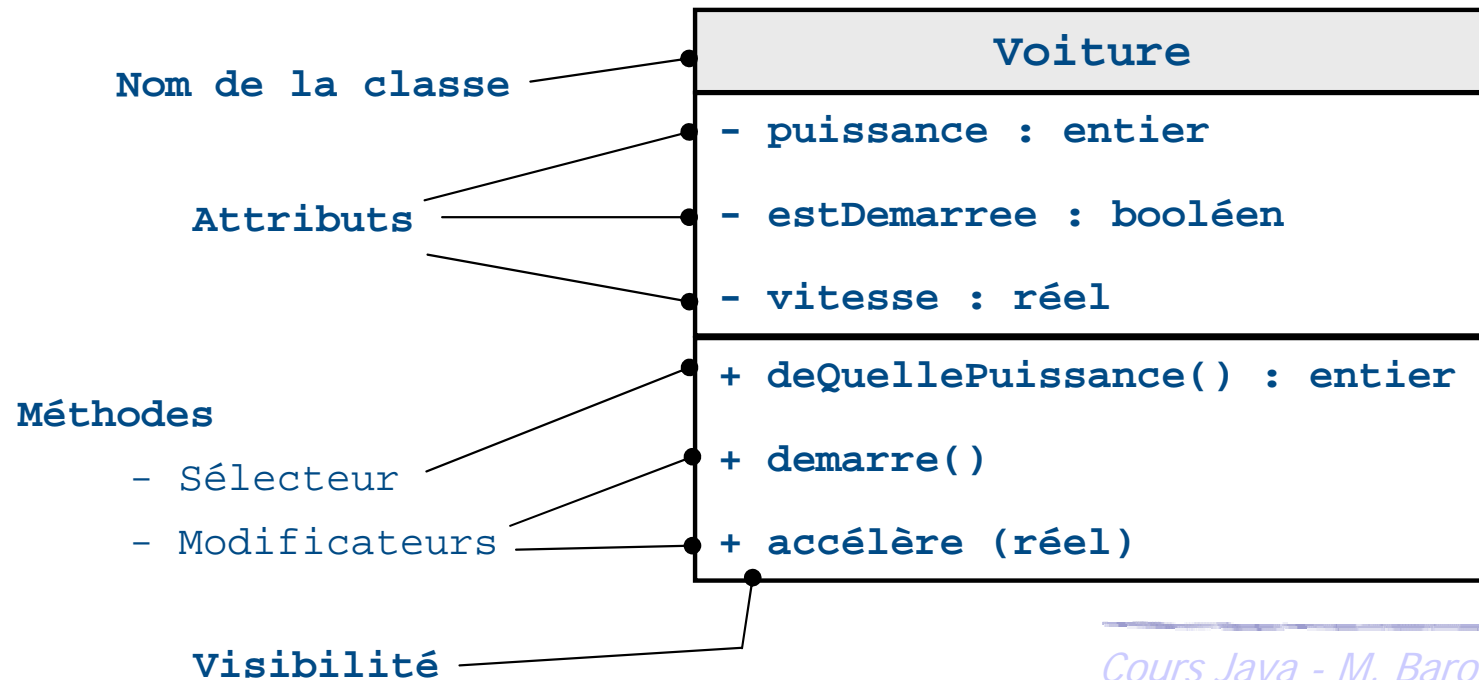
Principes POO

- Les objets communiquent entre eux par des messages
- Un objet peut recevoir un message qui déclenche
 - une méthode qui modifie son état
et / ou
 - une méthode qui envoie un message à un autre objet



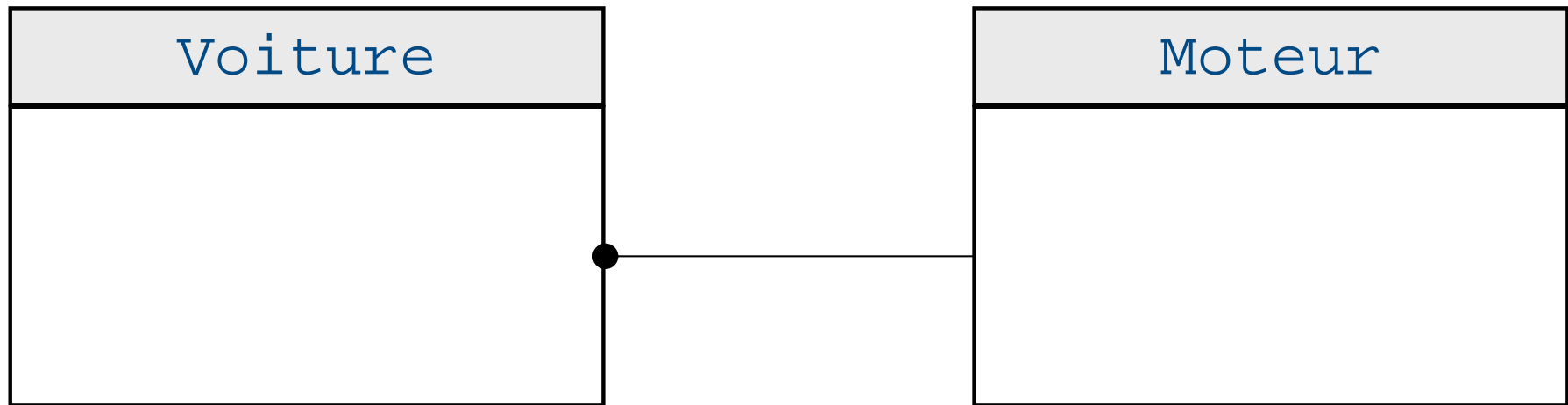
Principes POO : notion de classe

- Les objets qui ont les mêmes états et les mêmes comportements sont regroupés : c'est une **classe**
- Les classes servent de « moules » pour la création des objets
Un objet est une **instance** d'une classe
- Un programme OO est constitué de classes qui permettent de créer des objets qui s'envoient des messages



Principes POO

- L'ensemble des interactions entre les objets défini un algorithme
- Les relations entre les classes reflètent la décomposition du programme



Déroulement du cours

- Structuration du cours
 - Présentation des concepts
 - Illustration avec de nombreux exemples
 - Des bulles d'aide tout au long du cours :



Ceci est une alerte



Ceci est une astuce

- Mise en place du cours
 - Cours de Francis Jambon (ancien MdC à l'Université de Poitiers)
 - Livre : Programmer en Java 2^{ème} édition – Claude Delannoy - Eyrolles
 - Internet : *www.developpez.com*
- Remerciements pour les relectures
 - Laurent Guittet, ENSMA, Futuroscope de Poitiers
 - Developpez.com : Jérémie Habasque, Néo Kimz

Organisation ...

- Partie 1 : Introduction au langage JAVA
- Partie 2 : Bases du langage
- Partie 3 : Classes et objets
- Partie 4 : Héritage
- Partie 5 : Héritage et polymorphisme
- Partie 7 : Les indispensables : package, collections et exception



Disponible également en version
espagnole à l'adresse :
mbaron.developpez.com/javase/java



Programmation Orientée Objet application au langage Java

Introduction au langage Java

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Rapide historique de Java

➤ Origine

- Créé par Sun Microsystems
- Cible : les systèmes embarqués (véhicules, électroménager, etc) utilisant des langages dédiés incompatibles entre eux

➤ Dates clés

- **1991** : Introduction du langage « Oak » par James Gosling
- **1993** : Montée en puissance du Web grâce à Mosaic (l'idée d'adapter Java au Web fait son chemin)
- **1995** : Réalisation du logiciel HotJava en Java permettant d'exécuter des *applets*
- **1996** : Netscape™ Navigator 2 incorpore une machine virtuelle Java 1.0 en version « beta »
- **1997** : Un premier pas vers une version industrielle Java 1.1
- **1999** : Version industrielle de Java

Sun voit Java comme ...

➤ Références

- Wikipedia : fr.wikipedia.org/wiki/java_%28technologie%29
- White papers : java.sun.com/docs/white/index.html

➤ Sun définit le langage Java comme

- Simple
- Sûr
- Orienté objet
- Portable
- Réparti
- Performant
- Interprété
- Multitâches
- Robuste
- Dynamique ...



Principe de fonctionnement de Java

- Source Java
 - Fichier utilisé lors de la phase de programmation
 - Le seul fichier réellement intelligible par le programmeur!
- Byte-Code Java
 - Code objet destiné à être exécuté sur toute « Machine Virtuelle » Java
 - Provient de la compilation du code source
- Machine Virtuelle Java
 - Programme interprétant le Byte-Code Java et fonctionnant sur un système d'exploitation particulier
 - *Conclusion* : il suffit de disposer d'une « Machine Virtuelle » Java pour pouvoir exécuter tout programme Java même s'il a été compilé avec un autre système d'exploitation

Machines Virtuelles Java ...

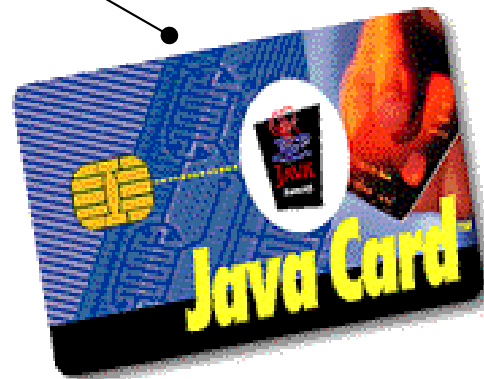
➤ Navigateurs Web, Stations de travail, Network Computers

➤ WebPhones

➤ Téléphones portables

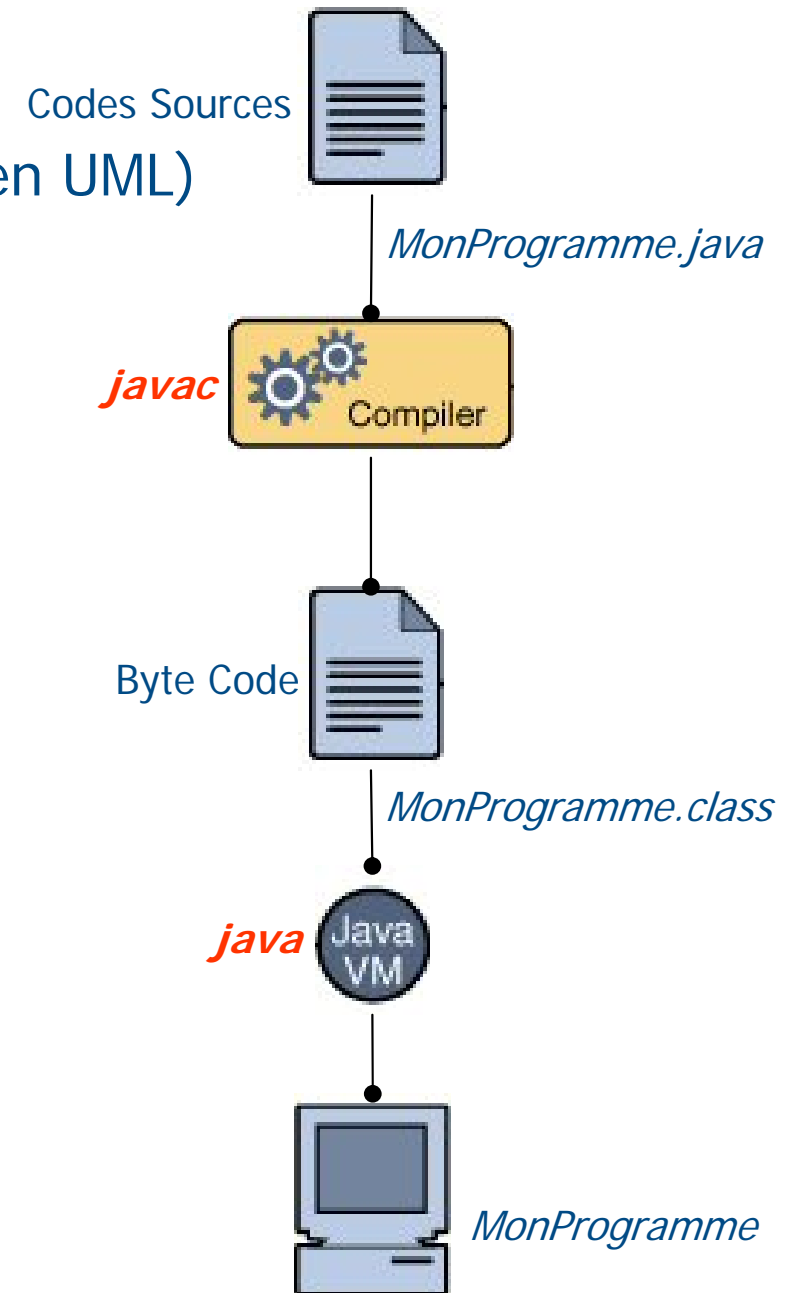
➤ Cartes à puces

➤ ...



Principales étapes d'un développement

- Création du code source
 - A partir des spécifications (par exemple en UML)
 - Outil : éditeur de texte, IDE
- Compilation en Byte-Code
 - A partir du code source
 - Outil : compilateur Java
- Diffusion sur l'architecture cible
 - Transfert du Byte-Code seul
 - Outils : réseau, disque, etc
- Exécution sur la machine cible
 - Exécution du Byte-Code
 - Outil : Machine Virtuelle Java



Java et ses versions ...

- Différentes versions de la machine virtuelle
 - Java 2 Micro Edition (Java ME) qui cible les terminaux portables
 - Java 2 Standard Edition (Java SE) qui vise les postes clients
 - Java 2 Enterprise Edition (Java EE) qui définit le cadre d'un serveur d'application



Dans la suite du cours, on va s'intéresser principalement aux API fournies par Java SE

- Différentes finalités
 - SDK (Software Development Kit) fournit un compilateur et une machine virtuelle
 - JRE (Java Runtime Environment) fournit uniquement une machine virtuelle. Idéal pour le déploiement de vos applications.

- Version actuelle de Java
 - Actuellement « Java SE 6.0 » ou encore appelée « JDK 5.0 »
 - Bientôt Java SE 7.0 (nom de code Dolphin)

Les outils ...

➤ Simples éditeurs ou environnements de développement

- Eclipse
- NetBeans
- JBuilder
- IntelliJ
- ...

Affichage des résultats du sondage: Quel EDI Java utilisez vous en 2007 ?

Eclipse		<u>246</u>	61,81%
NetBeans		<u>93</u>	23,37%
JBuilder 2007		<u>4</u>	1,01%
JBuilder (<= 2006)		<u>10</u>	2,51%
IntelliJ		<u>11</u>	2,76%
JDeveloper		<u>8</u>	2,01%
Sun Java Studio Creator		<u>2</u>	0,50%
JCreator		<u>6</u>	1,51%
WSAD		<u>3</u>	0,75%
JBoss Eclipse IDE		<u>2</u>	0,50%
BEA Workshop Studio		<u>1</u>	0,25%
Editeurs de texte avancés (Emacs, VI, JEdit, UltraEdit, ...)		<u>10</u>	2,51%
Autre (précisez)		<u>2</u>	0,50%

➤ Les ressources sur Java

- Site de Java chez Sun : java.sun.com
- API (référence) : java.sun.com/j2se/1.5.0
- Tutorial de Sun : java.sun.com/doc/bookstutorial
- Cours et exemples : java.developpez.com
- Forum : fr.comp.lang.java

L'API de Java



Packages

Java™ 2 Platform
Std. Ed. v1.4.2

[All Classes](#)

Packages

- [java.applet](#)
- [java.awt](#)
- [java.awt.color](#)
- [java.awt.datatransfer](#)
- [java.awt.dnd](#)
- [java.awt.event](#)
- [java.awt.font](#)
- [java.awt.geom](#)
- [java.awt.im](#)
- [java.awt.im.spi](#)
- [java.awt.image](#)
- [java.awt.image.renderable](#)

Classes

All Classes

- [ARG_IN](#)
- [ARG_INOUT](#)
- [ARG_OUT](#)
- [AWTError](#)
- [AWTEvent](#)
- [AWTEventListener](#)
- [AWTEventListenerProxy](#)
- [AWTEventMulticaster](#)
- [AWTException](#)
- [AWTKeyStroke](#)
- [AWTPermission](#)
- [AbstractAction](#)
- [AbstractBorder](#)
- [AbstractButton](#)
- [AbstractCellEditor](#)
- [AbstractCollection](#)
- [AbstractColorChooserPanel](#)
- [AbstractDocument](#)
- [AbstractDocument.AttributeContext](#)
- [AbstractDocument.Content](#)
- [AbstractDocument.ElementEdit](#)
- [AbstractInterruptibleChannel](#)
- [AbstractLayoutCache](#)
- [AbstractLayoutCache.NodeDimensions](#)
- [AbstractList](#)
- [AbstractListModel](#)
- [AbstractMap](#)
- [AbstractMethodError](#)
- [AbstractPreferences](#)
- [AbstractSelectableChannel](#)
- [AbstractSelectionKey](#)
- [AbstractSelector](#)
- [AbstractSequentialList](#)
- [AbstractSet](#)
- [AbstractSpinnerModel](#)
- [AbstractTableModel](#)
- [AbstractUndoableEdit](#)
- [AbstractWriter](#)
- [AccessControlContext](#)
- [AccessControlException](#)
- [AccessController](#)
- [AccessException](#)
- [Accessible](#)
- [AccessibleAction](#)
- [AccessibleBundle](#)
- [AccessibleComponent](#)

Overview Package Class Use Tree Deprecated Index Help

REV NEXT FRAMES NO FRAMES

Java™ 2 Platform
Std. Ed. v1.4.2

Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification

This document is the API specification for the Java 2 Platform, Standard Edition, version 1.4.2.

ee:
[Description](#)

Java 2 Platform Packages

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing <i>beans</i> -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.nio.channels	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
java.nio.channels.spi	Service-provider classes for the java.nio.channels package.
java.nio.charset	Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters.
java.nio.charset.spi	Service-provider classes for the java.nio.charset package.
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garbage-collection (DGC).
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server side of RMI.

Description
Attributs
Méthodes

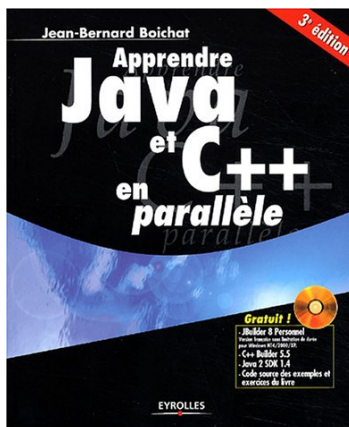
Ouvrages d'initiation



- Programmer en Java (2^{ème} édition)
 - Auteur : Claude Delannoy
 - Éditeur : Eyrolles
 - Edition : 2002 - 661 pages - ISBN : 2212111193



- Java en action
 - Auteur : Ian F. Darwin
 - Éditeur : O'Reilly
 - Edition : 2002 - 836 pages - ISBN : 2841772039



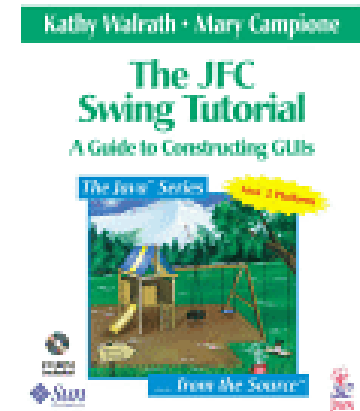
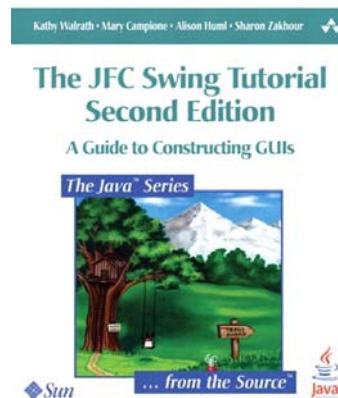
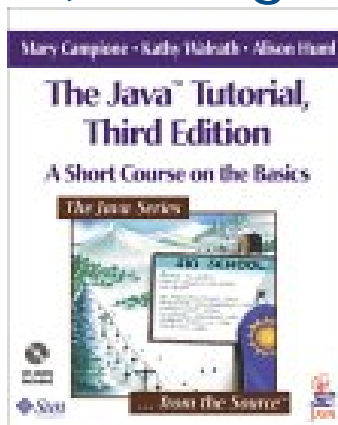
- Apprendre Java et C++ en parallèle
 - Auteur : Jean-Bernard Boichat
 - Éditeur : Eyrolles
 - Edition : 2003 - 742 pages - ISBN : 2212113277

Ouvrages de référence

- Ouvrages thématiques aux éditions O'Reilly sur une sélection des Packages Java (certains traduits en Français)



- Ouvrages de référence de SUN aux éditions Paperback (en anglais uniquement)





Programmation Orientée Objet application au langage Java

Bases du langage

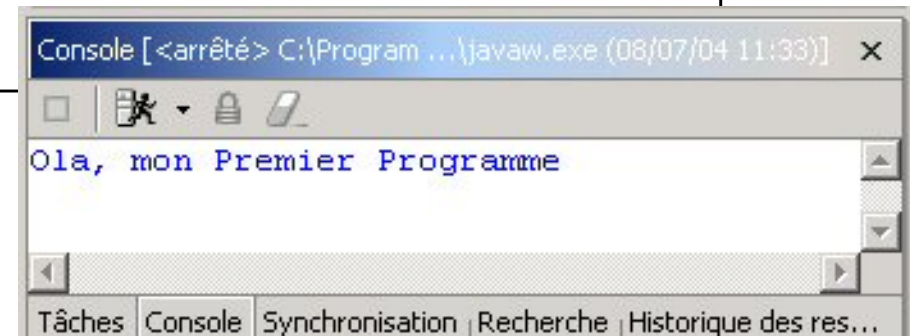
Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Premier exemple de programme en Java

```
public class PremierProg {

    public static void main (String[] argv) {
        System.out.println("Ola, mon Premier Programme");
    }

}
```



- *public class PremierProg*
 - Nom de la classe
- *public static void main*
 - La fonction principale équivalent à la fonction *main* du C/C++
- *String[] argv*
 - Permet de récupérer des arguments transmis au programme au moment de son lancement
- *System.out.println("Ola ... ")*
 - Méthode d'affichage dans la fenêtre console

Mise en œuvre

- Pas de séparation entre définition et codage des opérations
 - Un seul fichier « NomDeClasse.java »
 - Pas de fichier d'en tête comme C/C++

➤ Compilation

Nom de la classe = Nom du fichier java

- *javac* NomDeClasse.java ou *javac* *.java quand plusieurs classes
- Génération d'un fichier Byte-Code « NomDeClasse.class »
- Pas d'édition de liens (seulement une vérification)

➤ Exécution

- *java* NomDeClasse
- Choisir la classe principale à exécuter

Ne pas mettre l'extension .class pour l'exécution

Types primitifs de Java

- Ne sont pas des objets !!!
- Occupent une place fixe en mémoire réservée à la déclaration
- Types primitifs
 - Entiers : **byte** (1 octet) - **short** (2 octets) - **int** (4 octets) - **long** (8 octets)
 - Flottants (norme IEEE-754) : **float** (4 octets) - **double** (8 octets)
 - Booléens : **boolean** (true ou false)
 - Caractères : **char** (codage Unicode sur 16 bits)
- Chacun des types simples possède un alter-ego objet disposant de méthodes de conversion (à voir dans la partie Classes et Objets)
- L'autoboxing introduit depuis la version 5.0 convertit de manière transparente les types primitifs en références

Initialisation et constantes

➤ Initialisation

- Une variable peut recevoir une valeur au moment de sa déclaration :

```
int n = 15;  
boolean b = true;
```

- Cette instruction joue le même rôle :

```
int n;  
n = 15;  
boolean b;  
b = true;
```



**Penser à l'initialisation
au risque d'une erreur de
compilation**

```
int n;  
System.out.println(" n = " + n);
```

```
Console [ <carre> C:\Program Files\jzsd\1.4.2_04\bin\javaw.exe (06/07/04 16:59)]  
java.lang.Error: Problème de compilation non résolu :  
  La variable locale i peut ne pas avoir été initialisée  
  
  at PremierProg.main(PremierProg.java:18)  
Exception in thread "main"
```

➤ Constantes

- Ce sont des variables dont la valeur ne peut affectée qu'une fois
- Elles ne peuvent plus être modifiées
- Elles sont définies avec le mot clé **final**

```
final int n = 5;  
final int t;  
...  
t = 8;  
n = 10; // erreur : n est déclaré final
```

Structure de contrôle

➤ Choix

- Si alors sinon : « **if** *condition* { ... } **else** { ... } »

Il n'y a pas de mot-clé « then » dans la structure Choix



➤ Itérations

- Boucle : « **for** (*initialisation ; condition ; modification*) { ... } »

- Boucle (*for each*) : « **for** (*Type var : Collection*) { ... } »

- Tant que : « **while** (*condition*) { ... } »

Nouveauté Java 5

- Faire jusqu'à : « **do** { ... } **while** (*condition*) »

➤ Sélection bornée

- Selon faire : « **switch** *ident* { **case** valeur0 : ... **case** valeur1 : ... **default**: ... } »

Penser à vérifier si **break** est nécessaire dans chaque case



- Le mot clé **break** demande à sortir du bloc

Structure de contrôle

➤ Exemple : structure de contrôle

```
public class SwitchBreak {  
  
    public static void main (String[] argv) {  
        int n = ...;  
        System.out.println("Valeur de n :" + n);  
        switch(n) {  
            case 0 : System.out.println("nul");  
                    break;  
  
            case 1 :  
            case 2 : System.out.println("petit");  
            case 3 :  
            case 4 :  
            case 5 : System.out.println("moyen");  
                    break;  
            default : System.out.println("grand");  
        }  
        System.out.println("Adios...");  
    }  
}
```

➤ Faisons varier n

Valeur de n : 0
nul
Adios...

Valeur de n : 1
petit
moyen
Adios...

Valeur de n : 6
grand
Adios...



**Se demander si
break est nécessaire**

Opérateurs sur les types primitifs

➤ Opérateurs arithmétiques

- Unaires : « +a, -b »
- Binaires : « a+b, a-b, a*b, a%b »
- Incrémentation et décrémentation : « a++, b-- »
- Affectation élargie : « +=, -=, *=, /= »

➤ Opérateurs comparaisons

- « a==b, a!=b, a>b, a<b, a>=b, a<=b »

➤ Opérateurs logiques

- Et : « a && b », « a & b »
- Ou : « a || b », « a | b »

➤ Conversion de type explicite (cast)

- « (NouveauType)variable »



Attention : erreur

```
boolean t = true;  
if (t == true) {...}
```

Préférer :

```
boolean t = true;  
if (t) {...}
```

Opérateurs sur les types primitifs

- Exemple : simulation du Loto
 - Pas optimisé mais montre l'utilisation des concepts précédents

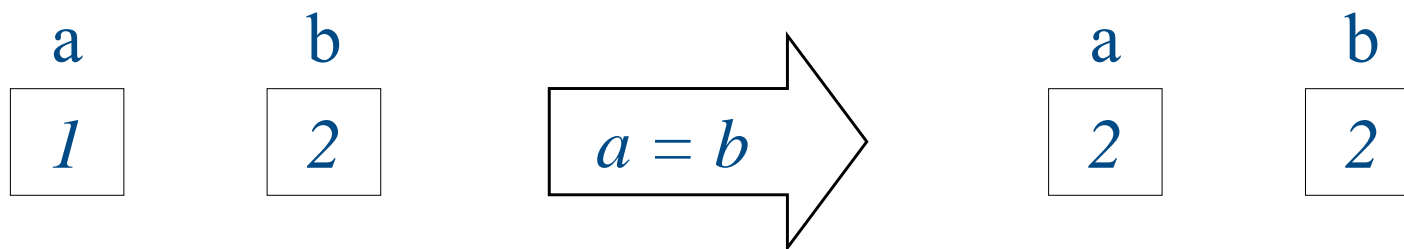
```
public class ExempleTypesPrimitifs {  
  
    public static void main (String[] argv) {  
        int compteur = 0;  
  
        while(compteur != 100) {  
            // Prend un nombre aléatoire  
            double nbreAleatoire = Math.random() * 1000;  
  
            // Etablie un index de 0 à 10  
            int index = compteur % 10;  
  
            // Construction de l'affichage  
            System.out.println("Index:" + index +  
                "Nbre Aléatoire:" + (int)nbreAleatoire);  
  
            // Incrémentation de la boucle  
            compteur+= 1;  
        }  
    }  
}
```

A voir plus tard...

```
Console [<arrêté> C... (24/07/04 15:57) x  
Index:0 Nbre Aléatoire:281  
Index:1 Nbre Aléatoire:369  
Index:2 Nbre Aléatoire:960  
Index:3 Nbre Aléatoire:824  
Console Tâches
```

Affectation, copie et comparaison

- Affecter et recopier un type primitif
 - « $a=b$ » signifie a prend la valeur de b
 - a et b sont distincts
 - Toute modification de a n'entraîne pas celle de b
- Comparer un type primitif
 - « $a == b$ » retourne « true » si les valeurs de a et b sont identiques



Les tableaux en Java

- Les tableaux sont considérés comme des **objets**
- Fournissent des collections ordonnées d'éléments
- Les éléments d'un tableau peuvent être
 - Des variables d'un type primitif (*int, boolean, double, char, ...*)
 - Des références sur des objets (à voir dans la partie Classes et Objets)
- Création d'un tableau
 - ① Déclaration = déterminer le type du tableau
 - ② Dimensionnement = déterminer la taille du tableau
 - ③ Initialisation = initialiser chaque case du tableau

Les tableaux en Java : Déclaration

① Déclaration

- La déclaration précise simplement le type des éléments du tableau

```
int[] monTableau;
```

```
monTableau null
```

- Peut s'écrire également

```
int monTableau[];
```



Attention : une déclaration de tableau ne doit pas préciser de dimensions

```
int monTableau[5]; // Erreur
```

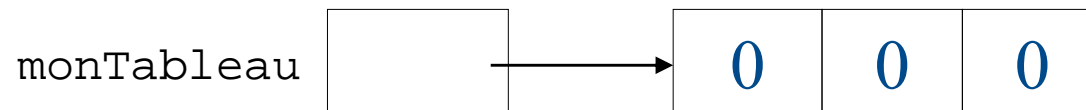
Les tableaux en Java : Dimensionnement

② Dimensionnement

- Le nombre d'éléments du tableau sera déterminé quand l'objet tableau sera effectivement créé en utilisant le mot clé **new**
- La taille déterminée à la création du tableau est fixe, elle ne pourra plus être modifiée par la suite
- Longueur d'un tableau : « `monTableau.length` »

```
int[] monTableau; // Déclaration  
monTableau = new int[3]; // Dimensionnement
```

- La création d'un tableau par **new**
 - Alloue la mémoire en fonction du type de tableau et de la taille
 - Initialise le contenu du tableau à 0 pour les types simples



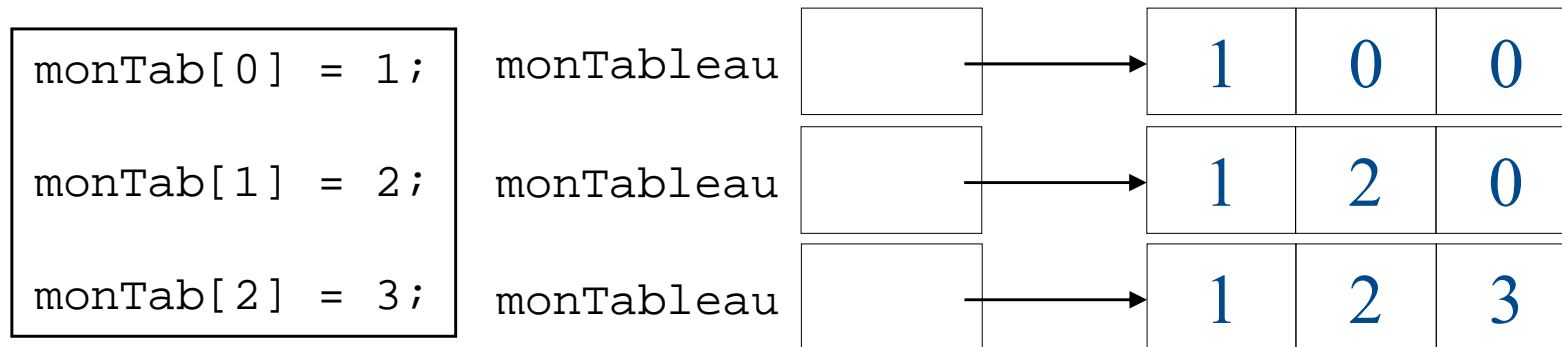
Les tableaux en Java : Initialisation

③ Initialisation

- comme en C/C++ les indices commencent à zéro
- l'accès à un élément d'un tableau s'effectue suivant cette forme

```
monTab[varInt]; // varInt >= 0 et <monTab.length
```

- Java vérifie automatiquement l'indice lors de l'accès (exception levée)



- Autre méthode : en donnant explicitement la liste de ses éléments entre {...}

```
int[] monTab = {1, 2, 3}
```

- est équivalent à

```
monTab = new int[3];  
monTab[0] = 1; monTab[1] = 2; monTab[2] = 3;
```

Les tableaux en Java : Synthèse

① Déclaration

```
int[] monTableau;
```

② Dimensionnement

```
monTableau = new int[3];
```

③ Initialisation

```
monTableau[0] = 1;  
monTableau[1] = 2;  
monTableau[2] = 3;
```

Ou ①② et ③

```
int[] monTab = {1, 2,  
3};
```

```
for (int i = 0; i < monTableau.length; i++) {  
    System.out.println(monTableau[i]);  
}
```

```
for (int current : monTableau) {  
    System.out.println(curent);  
}
```

Même chose avec
l'utilisation du *for each*

Les tableaux en Java : Multidimensionnels

➤ Tableaux dont les éléments sont eux mêmes des tableaux

➤ Déclaration

```
type[][] monTableau;
```

➤ Tableaux rectangulaires

➤ Dimensionnement :

```
monTableau = new type[2][3]
```

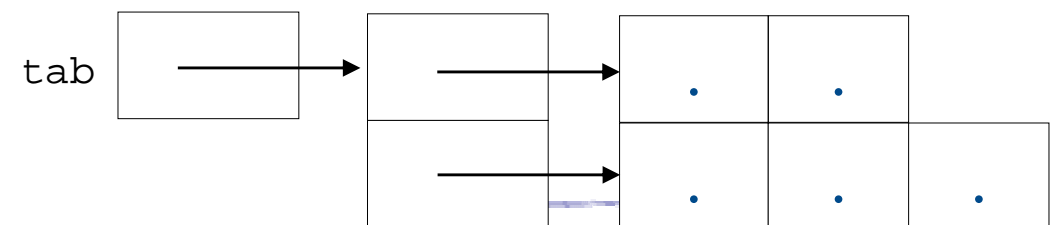
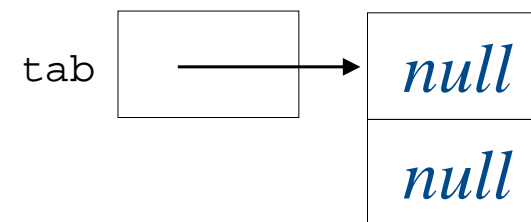
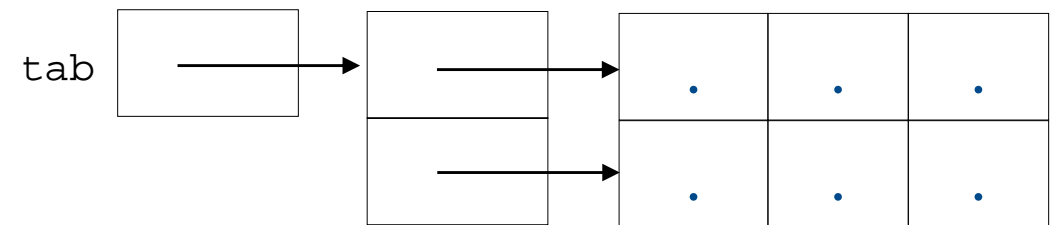
➤ Tableaux non-rectangulaires

➤ Dimensionnement :

```
monTableau = new type[2]
```

```
monTableau[0] = new type[2]
```

```
monTableau[1] = new type[3]
```



Petite précision du « System.out.println(...) »

- Usages : affichage à l'écran
 - « System.out.println(...) » : revient à la ligne
 - « System.out.print(...) » : ne revient pas à la ligne
- Différentes sorties possibles
 - « out » sortie standard
 - « err » sortie en cas d'erreur (non temporisée)
- Tout ce que l'on peut afficher...
 - Objets, nombres, booléens, caractères, ...
- Tout ce que l'on peut faire ...
 - Concaténation sauvage entre types et objets avec le « + »

```
System.out.println("a=" + a + "donc a < 0 est " + a < 0);
```

Commentaires et mise en forme

➤ Documentation des codes sources

➤ Utilisation des commentaires

```
// Commentaire sur une ligne complète  
int b = 34; // Commentaire après du code
```

```
/* Le début du commentaire  
** Je peux continuer à écrire ...  
Jusqu'à ce que le compilateur trouve cela */
```

➤ Utilisation de l'outil Javadoc (à voir dans la partie les Indispensables)

➤ Mise en forme

- Facilite la relecture
- Crédibilité assurée !!!!
- Indentation à chaque niveau de bloc

```
if (b == 3) {  
    if (cv == 5) {  
        if (q) {  
            ...  
        } else {  
            ...  
        }  
        ...  
    }  
    ...  
}
```

Préférer



```
if (b == 3) {  
if (cv == 5) {  
if (q) {  
...  
} else {...}  
...  
}  
...  
}
```

Éviter





Programmation Orientée Objet application au langage Java

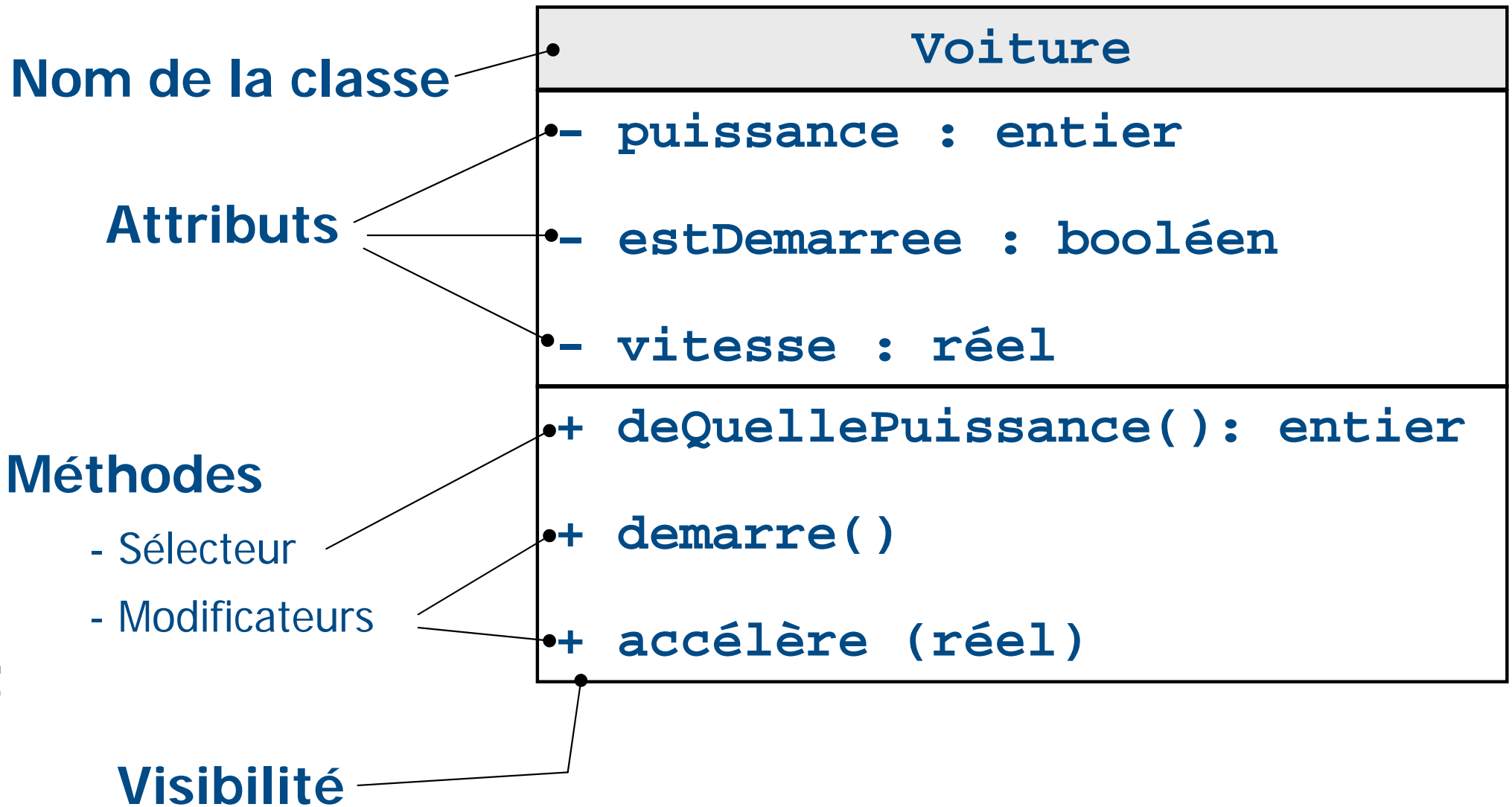
Classes et Objets

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Classe et définition

- Une classe est constituée
 - Données ce qu'on appelle des **attributs**
 - Procédures et/ou des fonctions ce qu'on appelle des **méthodes**
- Une classe est un modèle de définition pour des objets
 - Ayant même structure (même ensemble d'attributs)
 - Ayant même comportement (même méthodes)
 - Ayant une sémantique commune
- Les **objets** sont des représentations dynamiques, du modèle défini pour eux au travers de la classe (**instanciation**)
 - Une classe permet d'**instancier** (créer) plusieurs objets
 - Chaque objet est instance d'une classe et une seule

Classe et notation UML



Codage de la classe « Voiture »

Nom de la classe

Attributs

Sélecteur

Modificateurs

```
public class Voiture {  
    • private int puissance;  
    • private boolean estDemarree;  
    • private double vitesse;  
  
    public int deQuellePuissance() {  
        return puissance;  
    }  
  
    public void demarre() {  
        estDemarree = true;  
    }  
  
    • public void accelere(double v) {  
        if (estDemarree) {  
            vitesse = vitesse + v  
        }  
    }  
}
```

Classe et visibilité des attributs

- Caractéristique d'un attribut
 - Variables « globales » de la classe
 - Accessibles dans toutes les méthodes de la classe

```
public class Voiture {  
  
    private int puissance;  
    private boolean estDemarree;  
    private double vitesse;  
  
    public int deQuellePuissance() {  
        return puissance;  
    }  
  
    public void demarre() {  
        estDemarree = true;  
    }  
  
    public void accelere(double v) {  
        if (estDemarree) {  
            vitesse = vitesse + v  
        }  
    }  
}
```

Attributs visibles
dans les méthodes

Distinction entre attributs et variables

- Caractéristique d'une variable
 - Visible à l'intérieur du bloc qui le définit

```

public class Voiture {

    private int puissance;
    private boolean estDemarree;
    private double vitesse;

    public int deQuellePuissance() {
        return puissance;
    }

    public void demarre() {
        estDemarree = true;
    }

    public void accelere(double v) {
        if (estDemarree) {
            double avecTolerance;
            avecTolerance = v + 25;
            vitesse = vitesse + avecTolerance
        }
    }
}
    
```

Variable visible uniquement dans cette méthode

Variable peut être définie n'importe où dans un bloc

Conventions en Java : de la rigueur et de la classe ...

➤ Conventions de noms

- `CeciEstUneClasse`
- `celaEstUneMethode(...)`
- `jeSuisUneVariable`
- `JE_SUIS_UNE_CONSTANTE`

➤ Un fichier par classe, une classe par fichier

- Classe « *Voiture* » décrite dans le fichier `Voiture.java`
- Il peut exceptionnellement y avoir plusieurs classes par fichier (cas des *Inner classes*)



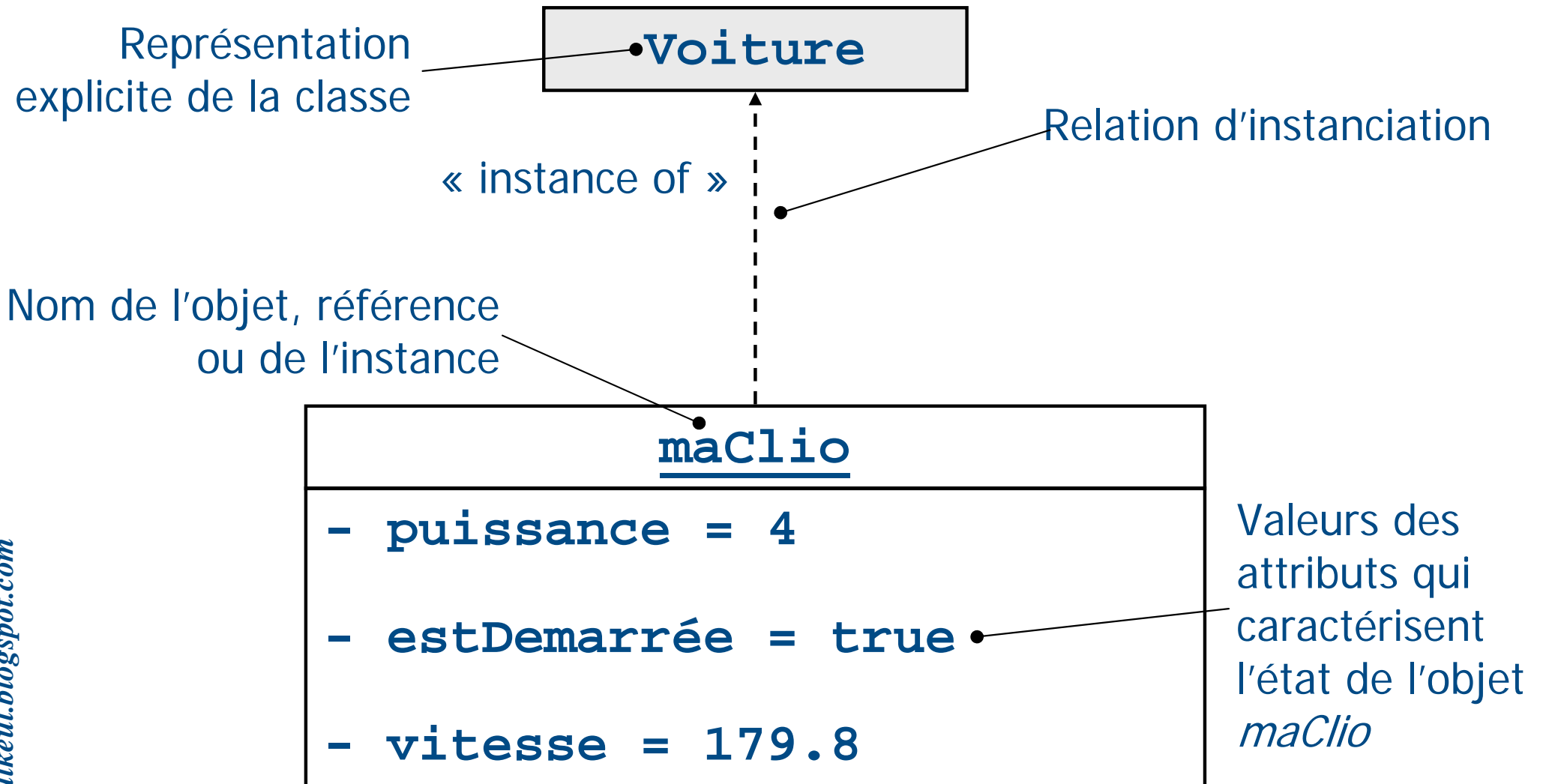
Respecter les minuscules et les majuscules des noms

Objet et définition

- Un objet est **instance** d'une seule classe
 - Se conforme à la description que celle-ci fournit
 - Admet une valeur propre à l'objet pour chaque attribut déclaré dans la classe
 - Les valeurs des attributs caractérisent l'**état** de l'objet
 - Possibilité de lui appliquer toute opération (**méthode**) définie dans la classe
- Tout objet est manipulé et identifié par sa référence
 - Utilisation de pointeur caché (plus accessible que le C++)
 - On parle indifféremment d'**instance**, de **référence** ou d'**objet**

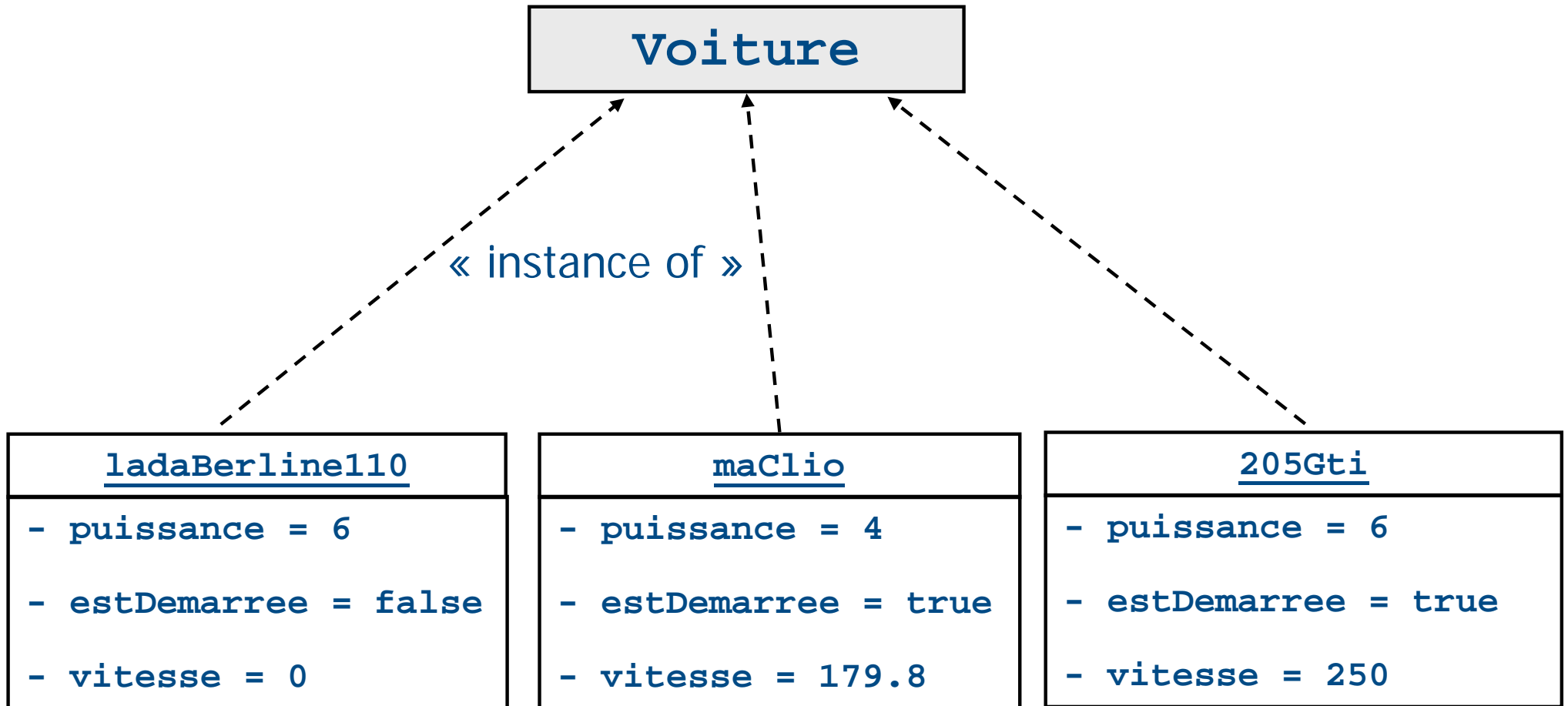
Objet et notation UML

- *maClio* est une instance de la classe *Voiture*



États des objets

- Chaque objet qui est une instance de la classe *Voiture* possède ses propres valeurs d'attributs



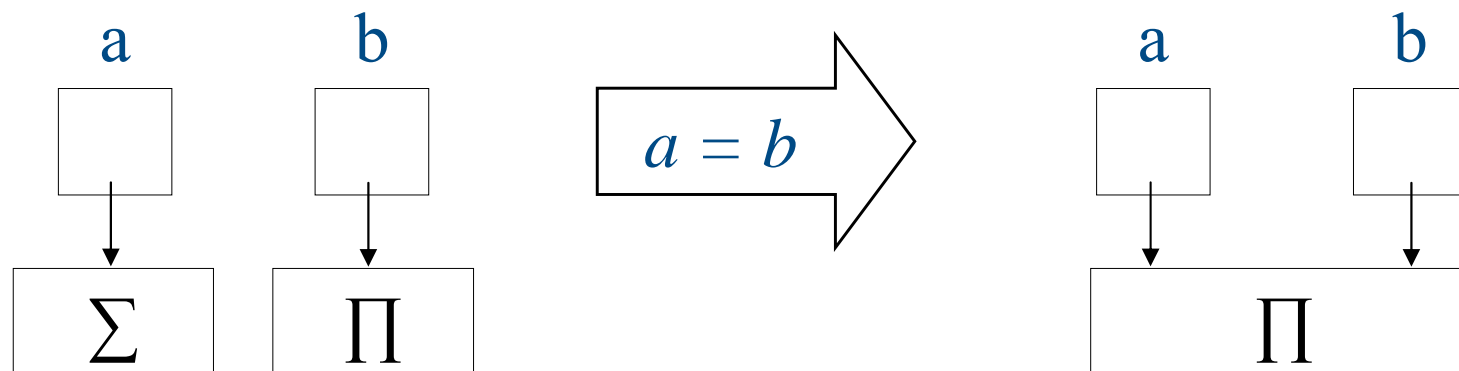
Affectation et comparaison

➤ Affecter un objet

- « $a = b$ » signifie a devient identique à b
- Les deux objets a et b sont identiques et toute modification de a entraîne celle de b

➤ Comparer deux objets

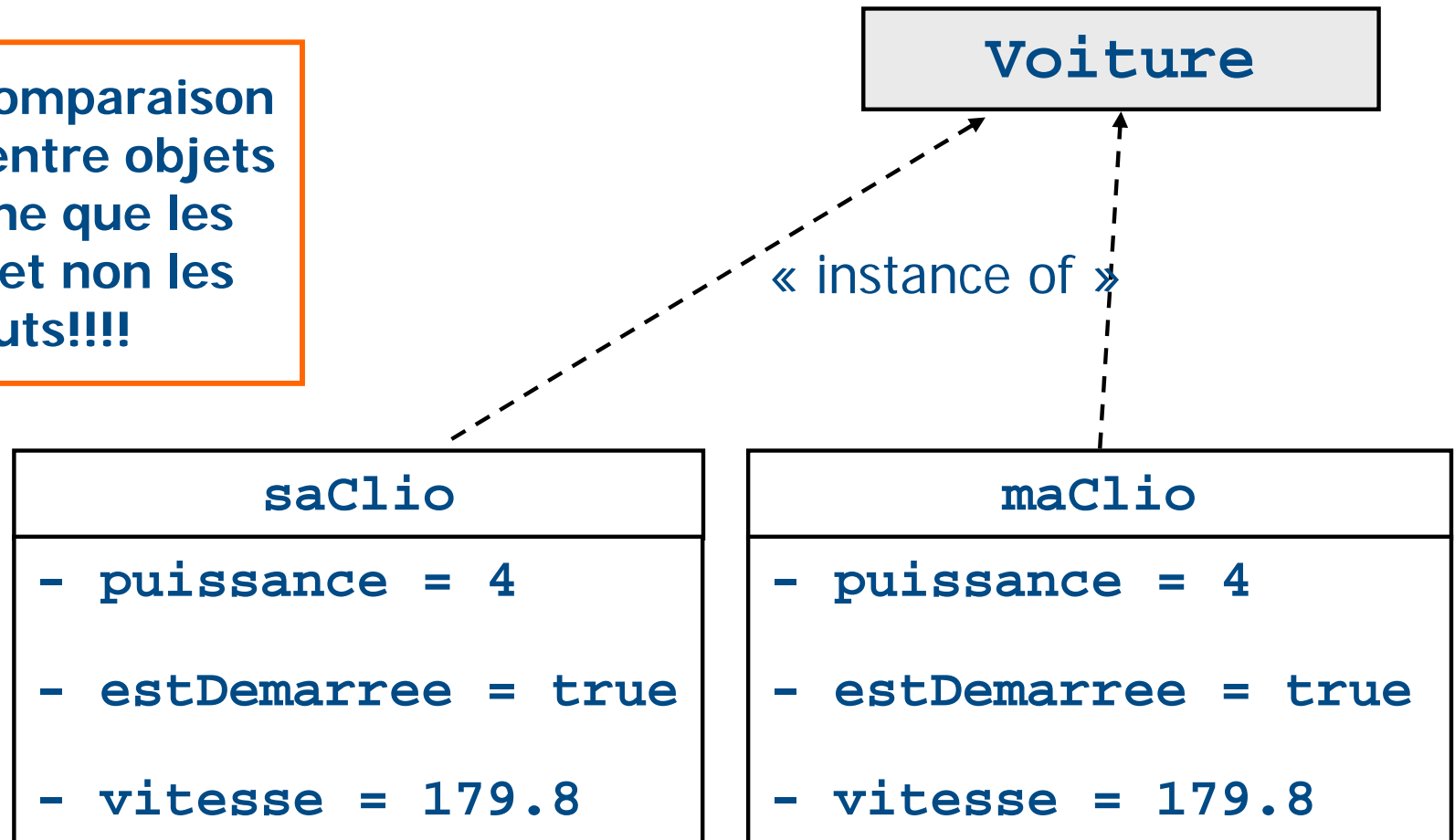
- « $a == b$ » retourne « true » si les deux objets sont identiques
- C'est-à-dire si les références sont les mêmes, cela ne compare pas les attributs



Affectation et comparaison

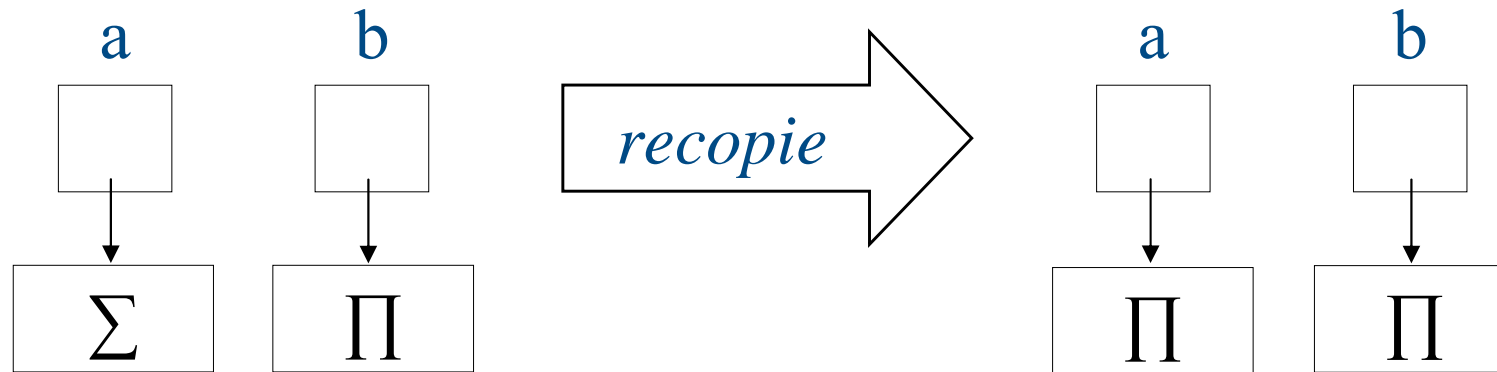
- L'objet **maClio** et **saClio** ont les mêmes attributs (états identiques) mais ont des références différentes
 - **maClio != saClio**

Le test de comparaison (== et !=) entre objets ne concerne que les références et non les attributs!!!!



Affectation et comparaison

- Recopier les attributs d'un objet « clone() »
 - Les deux objets a et b sont distincts
 - Toute modification de a n'entraîne pas celle de b



- Comparer le contenu des objets : « equals(Object o) »
 - Renvoyer « true » si les objets a et b peuvent être considérés comme identiques au vu de leurs attributs



Recopie et comparaison dans les parties suivantes

Structure des objets

- Un objet est constitué d'une partie « **statique** » et d'une partie « **dynamique** »
- Partie « **statique** »
 - Ne varie pas d'une instance de classe à une autre
 - Permet d'activer l'objet
 - Constituée des méthodes de la classe
- Partie « **dynamique** »
 - Varie d'une instance de classe à une autre
 - Varie durant la vie d'un objet
 - Constituée d'un exemplaire de chaque attribut de la classe

Cycle de vie d'un objet

➤ Création

- Usage d'un Constructeur
- L'objet est créé en mémoire et les attributs de l'objet sont initialisés

➤ Utilisation

- Usage des Méthodes et des Attributs (non recommandé)
- Les attributs de l'objet peuvent être modifiés
- Les attributs (ou leurs dérivés) peuvent être consultés

L'utilisation d'un objet non construit provoque une exception de type *NullPointerException*



➤ Destruction et libération de la mémoire lorsque

- Usage (éventuel) d'un *Pseudo-Destructeur*
- L'objet n'est plus référencé, la place mémoire occupée est récupérée

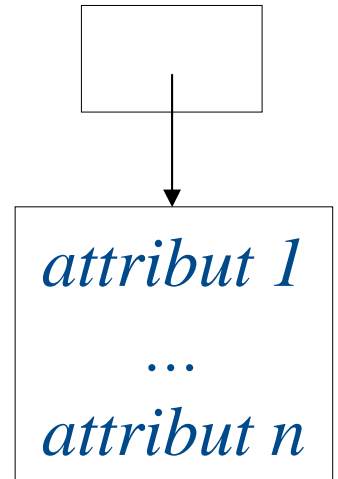
Création d'objets : déroulement

- La création d'un objet à partir d'une classe est appelée une **instanciation**
- L'objet créé est une **instance** de la classe
- Déclaration
 - Définit le nom et le type de l'objet
 - Un objet seulement déclaré vaut « **null** » (mot réservé du langage)
- Création et allocation de la mémoire
 - Appelle de méthodes particulières : les constructeurs
 - La création réserve la mémoire et initialise les attributs
- Renvoi d'une référence sur l'objet maintenant créé
 - `monObjet != null`

monObjet

null

monObjet



Création d'objets : réalisation

- La création d'un nouvel objet est obtenue par l'appel à **new** Constructeur (paramètres)
 - Il existe un constructeur par défaut qui ne possède pas de paramètre (si aucun autre constructeur avec paramètre n'existe)



Les constructeurs portent le même nom que la classe

Déclaration

Création et allocation mémoire

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        • Voiture maVoiture;
        • maVoiture = new Voiture();

        // Déclaration et création en une seule ligne
        Voiture maSecondeVoiture = new Voiture();

    }

}
```

Création d'objets : réalisation

➤ Exemple : construction d'objets

```
public class TestMaVoiture {  
  
    public static void main (String[] argv) {  
  
        // Déclaration puis création  
        Voiture maVoiture;  
        maVoiture = new Voiture();  
  
        // Déclaration d'une deuxième voiture  
        Voiture maVoitureCopie;  
        // Attention!! pour l'instant maVoitureCopie vaut null  
  
        // Test sur les références.  
        if (maVoitureCopie == null) {  
  
            // Création par affectation d'une autre référence  
            maVoitureCopie = maVoiture;  
            // maVoitureCopie possède la même référence que maVoiture  
  
        }  
        ...  
    }  
}
```

Déclaration

Affectation par
référence

Le constructeur de « Voiture »

➤ Actuellement

- On a utilisé le constructeur par défaut sans paramètre
- On ne sait pas comment se construit la « Voiture »
- Les valeurs des attributs au départ sont indéfinies et identique pour chaque objet (puissance, etc.)

➤ Rôle du constructeur en Java

Les constructeurs portent le même nom que la classe et n'ont pas de valeur de retour



- Effectuer certaines initialisations nécessaires pour le nouvel objet créé

➤ Toute classe Java possède au moins un constructeur

- Si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoquée

Le constructeur de « Voiture »

- Le constructeur de *Voiture*
 - Initialise *vitesse* à zéro
 - Initialise *estDemaree* à faux
 - Initialise *puissance* à la valeur passée en paramètre du constructeur

Constructeur
avec un
paramètre

```
public class Voiture {  
  
    private int puissance;  
  
    private boolean estDemarree;  
  
    private double vitesse;  
  
    public Voiture(int p) {  
        puissance = p;  
        estDemaree = false;  
        vitesse = 0;  
    }  
    ...  
}
```

Construire une « Voiture » de 7 CV

➤ Création de la *Voiture*

➤ Déclaration de la variable *maVoiture*

➤ Création de l'objet avec la valeur 7 en paramètre du constructeur

Déclaration

```
public class TestMaVoiture {  
  
    public static void main(String[] argv) {  
  
        // Déclaration puis création  
        ● Voiture maVoiture;  
  
        ● maVoiture = new Voiture(7);  
  
        Voiture maSecVoiture;  
        // Sous entendu qu'il existe  
        // explicitement un constructeur : Voiture(int)  
  
        maSecVoiture = new Voiture(); // Erreur  
  
    }  
}
```

Création et
allocation
mémoire
avec *Voiture(int)*

Constructeur sans arguments

➤ Utilité

- Lorsque l'on doit créer un objet sans pouvoir décider des valeurs de ses attributs au moment de la création
- Il remplace le constructeur par défaut qui est devenu inutilisable dès qu'un constructeur (avec paramètres) a été défini dans la classe

```
public class Voiture {

    private int puissance;
    private boolean estDemarree;
    private double vitesse;

    public Voiture() {
        puissance = 4;
        estDemaree = false;
        vitesse = 0;
    }
    public Voiture(int p) {
        puissance = p;
        estDemaree = false;
        vitesse = 0;
    }...
}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        Voiture maVoiture;
        maVoiture = new Voiture(7);
        Voiture maSecVoiture;
        maSecVoiture = new Voiture(); // OK
    }
}
```

Constructeurs multiples

➤ Intérêts

- Possibilité d'initialiser un objet de plusieurs manières différentes
- On parle alors de **surchage** (overloaded)
- Le compilateur distingue les constructeurs en fonction
 - de la position des arguments
 - du nombre
 - du type

```
public class Voiture {  
    ...  
    public Voiture() {  
        puissance = 4; ...  
    }  
  
    public Voiture(int p) {  
        puissance = p; ...  
    }  
  
    public Voiture(int p, boolean estDemaree) {  
        ...  
    }  
}
```



Chaque constructeur possède le même nom (le nom de la classe)

Accès aux attributs

- Pour accéder aux données d'un objet on utilise une notation pointée

identificationObjet.nomAttribut

```
public class TestMaVoiture {  
  
    public static void main (String[] argv) {  
  
        // Déclaration puis création  
        Voiture v1 = new Voiture();  
        Voiture v2 = new Voiture();  
  
        // Accès aux attributs en écriture  
        v1.puissance = 110;  
  
        // Accès aux attributs en lecture  
        System.out.println("Puissance de v1 = " + v1.puissance);  
    }  
}
```

**Il n'est pas recommandé
d'accéder directement aux
attributs d'un objet**



Envoi de messages : appel de méthodes

- Pour « demander » à un objet d'effectuer un traitement il faut lui **envoyer un message**
- Un message est composé de trois parties
 - Une référence permettant de désigner l'objet à qui le message est envoyé
 - Le nom de la méthode ou de l'attribut à exécuter
 - Les éventuels paramètres de la méthode

`identificationObjet.nomDeMethode(« Paramètres éventuels »)`

- Envoi de message similaire à un appel de fonction
 - Le code défini dans la méthode est exécuté
 - Le contrôle est retourné au programme appelant

Envoi de messages : appel de méthodes



Ne pas oublier les parenthèses pour les appels aux méthodes

Voiture	
-	...
+	deQuellePuissance() : entier
+	demarre()
+	accélère (réel)
+	...

```
public class TestMaVoiture {  
  
    public static void main (String[] argv) {  
  
        // Déclaration puis création  
        Voiture maVoiture = new Voiture();  
  
        // La voiture démarre  
        maVoiture.demarre();  
  
        if (maVoiture.deQuellePuissance() == 4) {  
            System.out.println("Pas très Rapide...");  
        }  
  
        // La voiture accélère  
        maVoiture.accélère(123.5);  
  
    }  
}
```

Envoi d'un message à l'objet *maVoiture*
Appel d'un modificateur

Envoi d'un message à l'objet *maVoiture*
Appel d'un sélecteur

Envoi de messages : passage de paramètres

- Un paramètre d'une méthode peut être
 - Une variable de type simple
 - Une référence d'un objet typée par n'importe quelle classe
- En Java tout est passé par valeur
 - Les paramètres effectifs d'une méthode
 - La valeur de retour d'une méthode (si différente de *void*)
- Les types simples
 - Leur valeur est recopiée
 - Leur modification dans la méthode n'entraîne pas celle de l'original
- Les objets
 - Leur modification dans la méthode entraîne celle de l'original!!!
 - Leur référence est recopiée et non pas les attributs

Envoi de messages : passage de paramètres

► Exemple : passage de paramètres

```
public class TestMaVoiture {  
  
    public static void main (String[] argv) {  
  
        // Déclaration puis création 1  
        Voiture maVoiture = new Voiture();  
  
        // Déclaration puis création 2  
        Voiture maSecondeVoiture = new Voiture();  
  
        // Appel de la méthode compare(voiture) qui  
        // retourne le nombre de différence  
        int p = maVoiture.compare(maSecondeVoiture);  
  
        System.out.println("Nbre différence :" + p);  
  
    }  
}
```

Référence comme
paramètre

Appel d'un sélecteur avec
passage d'une référence

Voiture	
-	...
+	accélère (réel)
+	compare (Voiture) : entier
+	...

L'objet « courant »

- L'objet « courant » est désigné par le mot clé **this**
 - Permet de désigner l'objet dans lequel on se trouve
 - **self** ou **current** dans d'autres langages
 - Désigne une référence particulière qui est un membre caché



Ne pas tenter d'affecter une nouvelle valeur à this !!!!

```
this = ... ; // Ne pas y penser
```

- Utilité de l'objet « courant »
 - Rendre explicite l'accès aux propres attributs et méthodes définies dans la classe
 - Passer en paramètre d'une méthode la référence de l'objet courant

L'objet « courant » : attributs et méthodes

- Désigne des variables ou méthodes définies dans une classe

```
public class Voiture {  
  
    ...  
    private boolean estDemarree; ←  
    private double vitesse; →  
  
    public int deQuellePuissance() {  
        ...  
    }  
  
    public void accelere(double vitesse) {  
        if (estDemarree) ←  
            → this.vitesse = this.vitesse + vitesse;  
    }  
}
```

Désigne la variable *vitesse*

Désigne l'attribut *vitesse*

Désigne l'attribut *demarree*



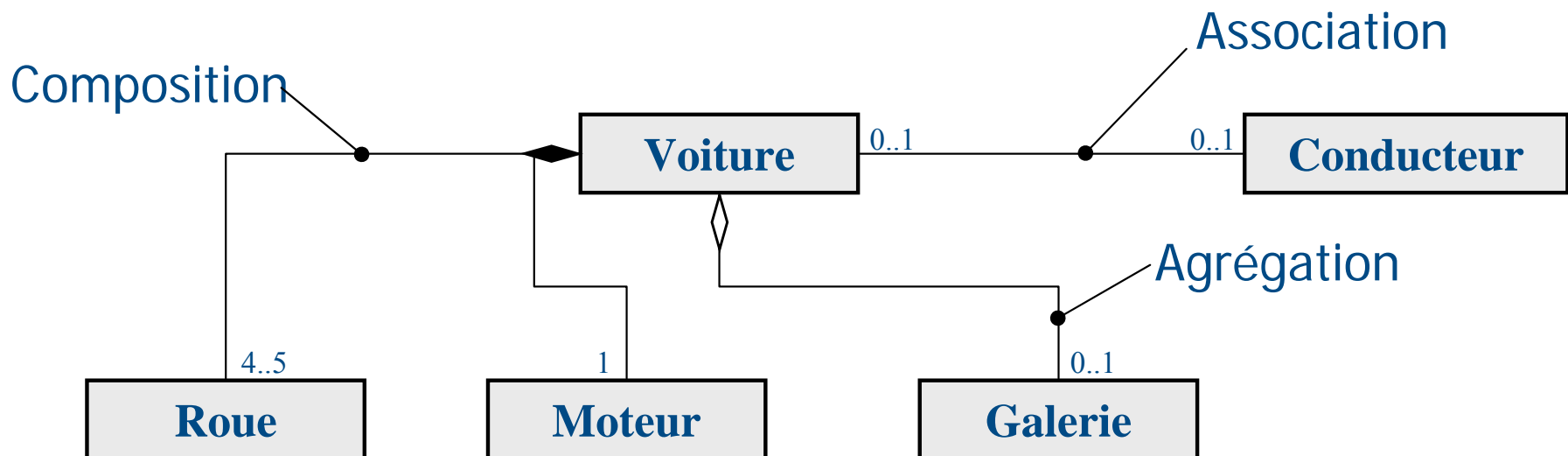
***this* n'est pas nécessaire lorsque les identificateurs de variables ne présentent aucun équivoque**

Le retour d'UML...

- Association : les objets sont sémantiquement liés —
 - Exemple : une Voiture est conduite par un Conducteur
- Agrégation : cycle de vie indépendant ◇
 - Exemple : une Voiture et une Galerie
- Composition : cycle de vie identiques ◆
 - Exemple : voiture possède un moteur qui dure la vie de la voiture

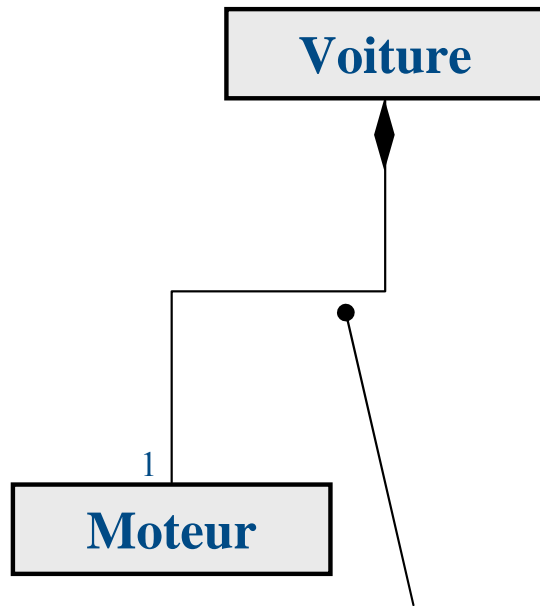


Les losanges sont attachés à la classe qui contient



Codage de l'association : composition

- L'objet de classe *Voiture* peut envoyer des messages à l'objet de classe *Moteur* : Solution 1



A discuter : si le moteur d'une voiture est « mort », il y a la possibilité de le changer

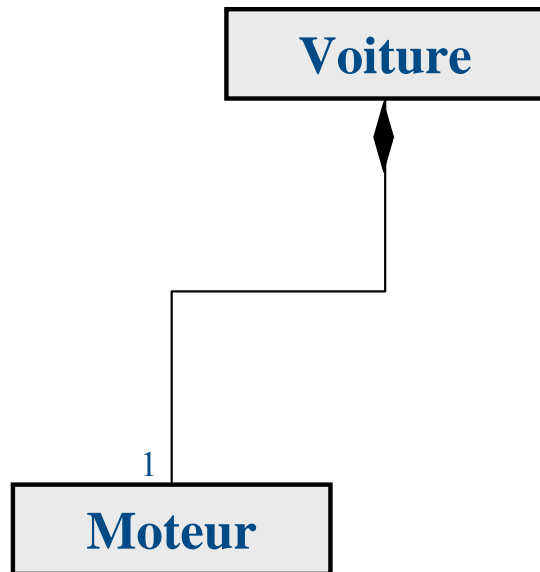
Attribut qui stocke la référence du moteur

```
public class Voiture {
    private Moteur leMoteur;
    ...
    public Voiture(int p) {
        leMoteur = new Moteur(p);
        ...
    }
    ...
}
```

Création de l'objet Moteur

Codage de l'association : composition

- L'objet de classe *Moteur* n'envoie pas de message à l'objet de classe *Voiture* : Solution 1



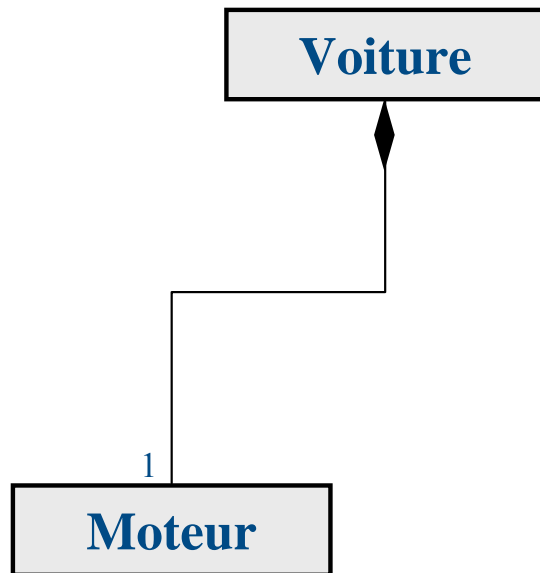
Attribut qui stocke la puissance

```
public class Moteur {
    private int puissance;
    ...
    public Moteur(int p) {
        puissance = p;
        ...
    }
    ...
}
```

La puissance est donnée lors de la construction

Codage de l'association : composition

- Il peut être nécessaire que les deux objets en composition s'échangent des messages : Solution 2
 - L'objet *Voiture* « voit » l'objet *Moteur*



```
public class Voiture {  
    private Moteur leMoteur;  
    ...  
    public Voiture(int p) {  
        leMoteur = new Moteur(p, this);  
        ...  
    }  
    ...  
}
```

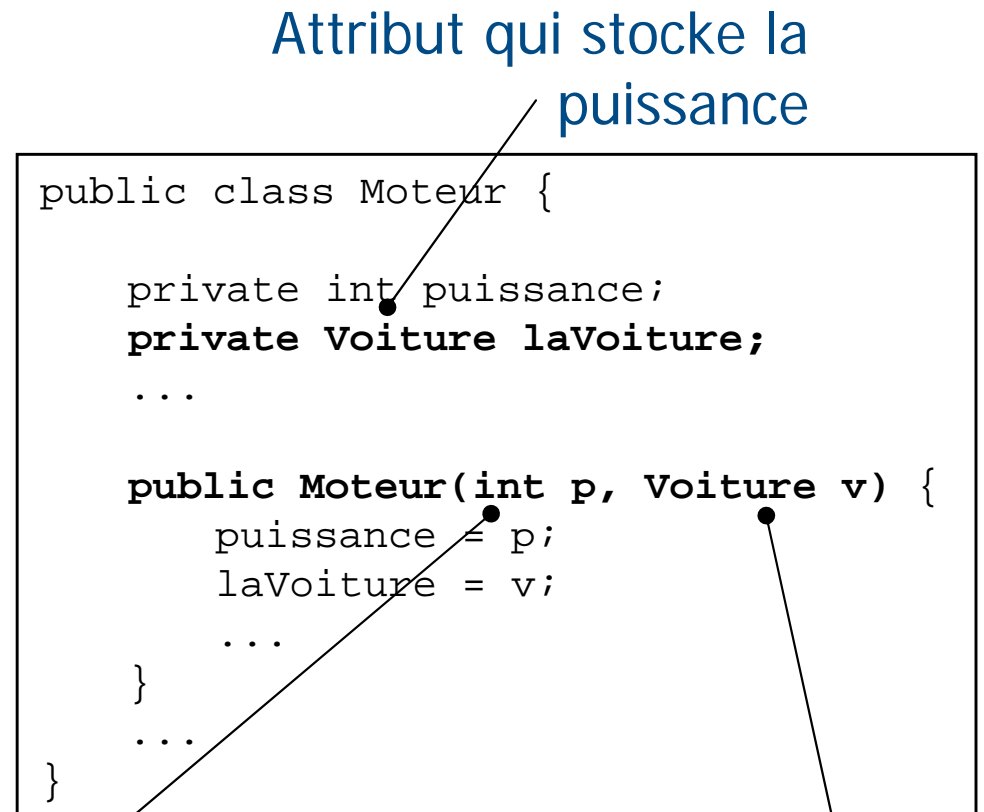
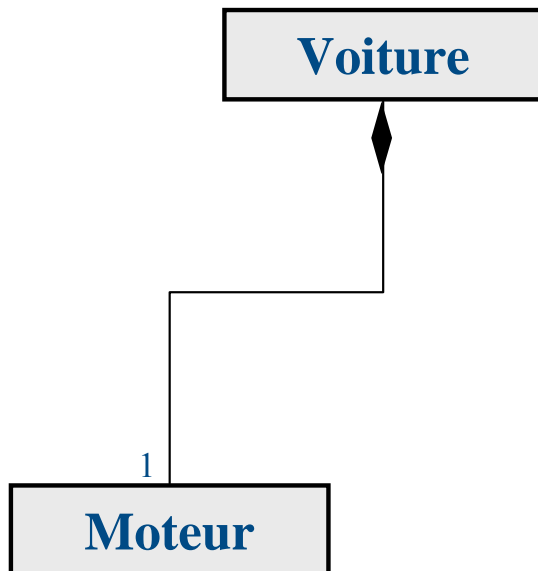
Attribut qui stocke la référence du Moteur

Création de l'objet Moteur

Transmission de la référence de l'objet courant

Codage de l'association : composition

- Il peut être nécessaire que les deux objets en composition s'échangent des messages : Solution 2
 - L'objet *Moteur* « voit » l'objet *Voiture*

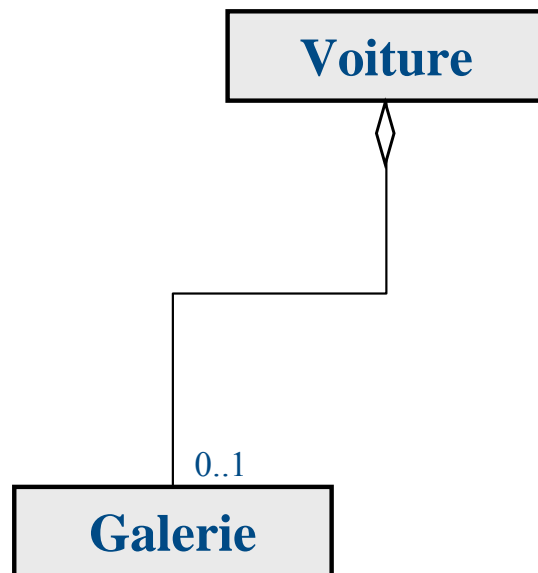


La puissance est donnée lors de la construction

Référence d'un objet Voiture en paramètre

Codage de l'association : agrégation

- L'objet de classe *Galerie* n'envoie pas de message à l'objet de classe *Voiture*



Attribut qui stocke une référence de *Galerie*

```
public class Voiture {
    private Galerie laGalerie;
    ...

    public Voiture(Galerie g) {
        laGalerie = g;
        ...
    }
    ...
}
```

Un objet *Galerie* est transmis au moment de la construction de *Voiture*

Destruction et ramasse-miettes

- La destruction des objets se fait de manière implicite
- Le ramasse-miettes ou Garbage Collector se met en route
 - Automatiquement
 - Si plus aucune variable ne référence l'objet
 - Si le bloc dans lequel il est défini se termine
 - Si l'objet a été affecté à « null »
 - Manuellement :
 - Sur demande explicite par l'instruction « `System.gc()` »
- Un pseudo-destructeur « *protected void finalize()* » peut être défini explicitement par le programmeur
 - Il est appelé juste avant la libération de la mémoire par la machine virtuelle, mais on ne sait pas quand
 - Conclusion : pas très sûr!!!!



Préférer définir une méthode et de l'invoquer avant que tout objet ne soit plus référencé : *détruit()*

Destruction et ramasse-miettes

```
public class Voiture {

    private boolean estDemarree;
    ...

    protected void finalize() {
        estDemarree = false;
        System.out.println("Moteur arrêté");
    }
    ...
}
```



Pour être sûr que finalize s'exécute il faut absolument appeler explicitement *System.gc()*

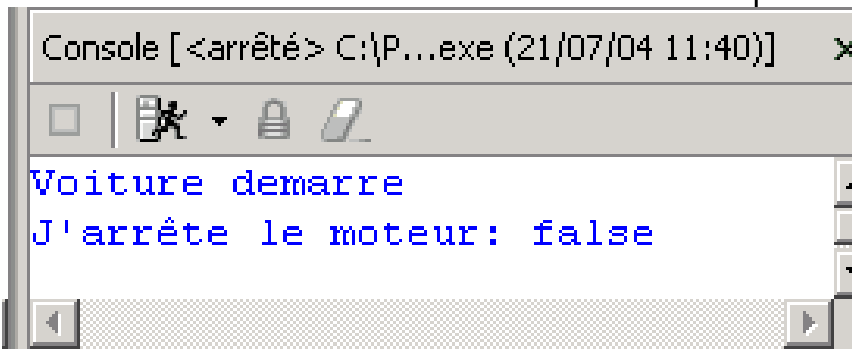
Force le programme à se terminer

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();
        maVoiture.demarre();
        // maVoiture ne sert plus à rien
        maVoiture = null;

        // Appel explicite du garbage collector
        System.gc();

        // Fin du programme
        System.exit(0);
        System.out.println("Message non visible");
    }
}
```



Gestion des objets

- Afficher son type et sa place mémoire : *System.out.println()*

```
System.out.println(maVoiture) // Voiture@119c082
```

- Récupérer son type : méthode « *Class getClass()* »

```
maVoiture.getClass(); // Retourne un objet de type Class  
Class classVoiture = maVoiture.getClass(); // Class est une classe!!!
```

- Tester son type : opérateur « *instanceof* » mot clé « *class* »

```
if (maVoiture instanceof Voiture) {...} // C'est vrai
```

OU

```
if (maVoiture.getClass() == Voiture.class) {...} // C'est vrai  
// également
```

Surcharge

- La surcharge *overloading* n'est pas limitée aux constructeurs, elle est possible également pour n'importe quelle méthode
- Possibilité de définir des méthodes possédant le même nom mais dont les arguments diffèrent
- Quand une méthode surchargée est invoquée le compilateur sélectionne automatiquement la méthode dont le nombre et le type des arguments correspondent au nombre et au type des paramètres passés dans l'appel de la méthode

Des méthodes surchargées peuvent avoir des types de retour différents à condition qu'elles aient des arguments différents



Surcharge

► Exemple : une voiture surchargée

```
public class Voiture {
    private double vitesse;
    ...

    public void accelere(double vitesse) { ←
        if (estDemarree) {
            this.vitesse = this.vitesse + vitesse;
        }
    }

    → public void accelere(int vitesse) {
        if (estDemaree) {
            this.vitesse = this.vitesse +
                (double)vitesse;
        }
    }
    ...
}

public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();

        // Accélération 1 avec un double
        maVoiture.accelere(123.5); ←
        // Accélération 2 avec un entier
        → maVoiture.accelere(124);
    }
}
```

Constructeurs multiples : le retour

- Appel explicite d'un constructeur de la classe à l'intérieur d'un autre constructeur
 - Doit se faire comme première instruction du constructeur
 - Utilise le mot-clé *this(paramètres effectifs)*

➤ Exemple

- Implantation du constructeur sans paramètre de *Voiture* à partir du constructeur avec paramètres

```
public class Voiture {
    ...
    public Voiture() {
        ← this(7, new Galerie());
    }
    ← public Voiture(int p) {
        this(p, new Galerie());
    }
    → public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

Encapsulation

- Possibilité d'accéder aux attributs d'une classe Java mais pas recommandé car contraire au principe d'encapsulation
 - Les données (attributs) doivent être protégés et accessibles pour l'extérieur par des sélecteurs
- Possibilité d'agir sur la visibilité des membres (attributs et méthodes) d'une classe vis à vis des autres classes
- Plusieurs niveaux de visibilité peuvent être définis en précédant d'un modificateur la déclaration d'un *attribut*, *méthode* ou *constructeur*
 - private
 - public
 - protected

A revoir dans la partie suivante

Encapsulation : visibilité des membres d'une classe

+ public

- private

classe

La classe peut être utilisée par n'importe quelle classe

Utilisable uniquement par les classes définies à l'intérieur d'une autre classe.

Une classe privée n'est utilisable que par sa classe englobante

attribut

Attribut accessible partout où sa classe est accessible. N'est pas recommandé du point de vue encapsulation

Attribut restreint à la classe où est faite la déclaration

méthode

Méthode accessible partout où sa classe est accessible.

Méthode accessible à l'intérieur de la définition de la classe

Encapsulation

➤ Exemple : encapsulation

```
public class Voiture {  
  
    private int puissance;  
    ...  
  
    public void demarre() {  
        ...  
    }  
  
    private void makeCombustion() {  
        ...  
    }  
  
}
```

Une méthode privée ne peut plus être invoquée en dehors du code de la classe où elle est définie



```
public class TestMaVoiture {  
  
    public static void main (String[] argv) {  
        // Déclaration puis création de maVoiture  
        Voiture maVoiture = new Voiture();  
  
        // Démarrage de maVoiture  
        maVoiture.demarre();  
  
        maVoiture.makeCombustion(); // Erreur  
    }  
  
}
```

Les chaînes de caractères « String »

➤ Ce sont des objets traités comme des types simples ...

➤ Initialisation

```
String maChaine = "Bonjour!"; // Cela ressemble à un type simple
```

➤ Longueur

```
maChaine.length(); // Avec les parenthèses car c'est une méthode
```

➤ Comparaison

```
maChaine.equals("Bonjour!"); // Renvoie vrai
```

➤ Concaténation

```
String essai = "ess" + "ai";  
String essai = "ess".concat("ai");
```



**Faites attention à la comparaison
de chaînes de caractères.**

```
maChaine == "toto";
```

Comparaison sur les références !!

Les Chaînes modifiables « StringBuffer »

- Elles sont modifiables par insertion, ajouts, conversions, etc
- On obtient une « StringBuffer » avec ses constructeurs

```
StringBuffer mCM = new StringBuffer(int length);  
StringBuffer mCM = new StringBuffer(String str);
```

- On peut les transformer en chaînes normales *String*

```
String s = mCM.toString();
```

- On peut leur ajouter n'importe (surcharge) quoi

```
mCM.append(...); // String, int, long, float, double, boolean,  
char
```

- On peut leur insérer n'importe (surcharge) quoi

```
mCM.insert(int offset, ...); // String, int, long, float, double,  
boolean, char
```

Les chaînes décomposables « StringTokenizer »

- Elles permettent la décomposition en mots ou éléments suivant un délimiteur

```

this is a test => this
                  is
                  a
                  test
    
```

- On obtient une « StringTokenizer » avec ses constructeurs

```

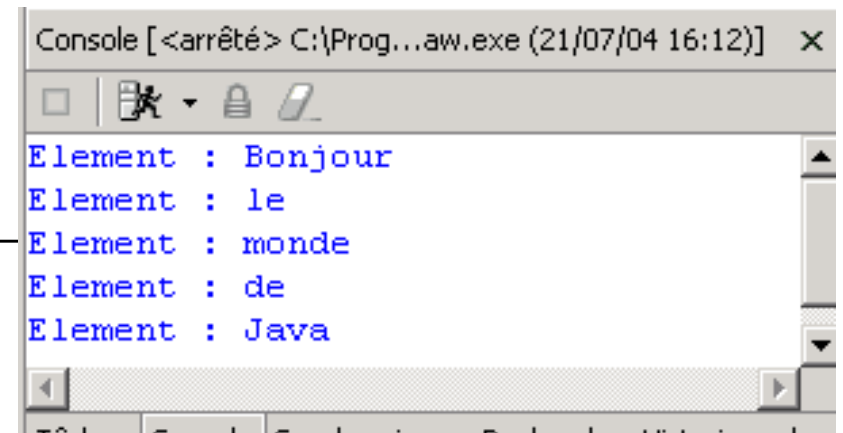
StringTokenizer mCM = new StringTokenize(String str); // Délimiteur = blanc
StringTokenizer rMCM = new StringTokenizer(String str, String delim);
    
```

- Un exemple

```

StringTokenizer st =
    new StringTokenizer("Bonjour,
        le monde|de|Java", "|");

while(st.hasMoreElements())
    System.out.println("Element : " + st.nextElement());
    
```



Variables de classe

- Il peut être utile de définir pour une classe des attributs indépendamment des instances : nombre de Voitures créées
- Utilisation des Variables de classe comparables aux « variables globales »
- Usage des **variables de classe**
 - Variables dont il n'existe qu'un seul exemplaire associé à sa classe de définition
 - Variables existent indépendamment du nombre d'instances de la classe qui ont été créés
 - Variables utilisables même si aucune instance de la classe n'existe

Variables de classe

- Elles sont définies comme les attributs mais avec le mot-clé **static**

```
public static int nbVoitureCreees;
```

Attention à l'encapsulation. Il est dangereux de laisser cette variable de classe en public.



- Pour y accéder, il faut utiliser non pas un identificateur mais le nom de la classe

```
Voiture.nbVoitureCreees = 3;
```

Il n'est pas interdit d'utiliser une variable de classe comme un attribut (au moyen d'un identificateur) mais fortement déconseillé



Constantes de classe

➤ Usage

- Ce sont des constantes liées à une classe
- Elles sont écrites en MAJUSCULES

Une constante de classe est généralement toujours visible 



➤ Elles sont définies (en plus) avec le mot-clé final

```
public class Galerie {  
    public static final int MASSE_MAX = 150;  
}
```

➤ Pour y accéder, il faut utiliser non pas un identificateur d'objet mais le nom de la classe (idem variables de classe)

```
if (maVoiture.getWeightLimite() <= Galerie.MASSE_MAX) {...}
```

VARIABLES ET CONSTANTES DE CLASSE

➤ Exemple : constantes de classe

```
public class Voiture {

    public static final int PTAC_MAX = 3500;
    private int poids;
    public static int nbVoitureCrees;
    ...

    public Voiture(int poids, ...) {
        this.poids = poids;
        nbVoitureCrees++;
        ...
    }
}
```

Dangereux car
possibilité de
modification
extérieure...

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture(2500);
        ...
        System.out.println("Poids maxi:" +
            Voiture.PTAC_MAX);
        System.out.println(Voiture.nbVoitureCrees);
        ...
    }
}
```

Utilisation de
Variables et
Constantes de classe
par le nom de la
classe Voiture

Méthodes de classe

- Usage
 - Ce sont des méthodes qui ne s'intéressent pas à un objet particulier
 - Utiles pour des calculs intermédiaires internes à une classe
 - Utiles également pour retourner la valeur d'une variable de classe en visibilité *private*
- Elles sont définies comme les méthodes d'instances, mais avec le mot clé **static**

```
public static double vitesseMaxToleree() {  
    return vitesseMaxAutorisee*1.10;  
}
```

- Pour y accéder, il faut utiliser non pas un identificateur d'objet mais le nom de la classe (idem variables de classe)

```
Voiture.vitesseMaxToleree()
```

Méthodes de classe

➤ Exemple : méthode de classe

```
public class Voiture {  
  
    private static int nbVoitureCreees;  
    ...  
  
    public static int getNbVoitureCreees(){  
        return Voiture.nbVoitureCreees;  
    }  
}
```

Déclaration d'une variable de classe privée. Respect des principes d'encapsulation.

Déclaration d'une méthode de classe pour accéder à la valeur de la variable de classe.

```
public class TestMaVoiture {  
  
    public static void main (String[] argv) {  
        // Déclaration puis création de maVoiture  
        Voiture maVoiture = new Voiture(2500);  
        ...  
  
        System.out.println("Nbre Instance :" +  
            Voiture.getNbVoitureCreees());  
    }  
}
```

Méthodes de classe : erreur classique

➤ Exemple (suite) : méthode de classe

```
public class Voiture {  
  
    private Galerie laGalerie;  
    ...  
  
    public Voiture(Galerie g) {  
        laGalerie = g;  
        ...  
    }  
  
    public static boolean isGalerieInstall() {  
        return (laGalerie != null)  
    }  
}
```

Déclaration d'un objet
Galerie non statique



On ne peut pas utiliser de variables d'instance dans une méthode de classe!!!!

Erreur : Utilisation d'un attribut non statique dans une zone statique

Méthodes de classe

- Rappel : les types simples (int, double, etc.) possède un alter-ego objet disposant de méthodes de conversion
- Par exemple la classe *Integer* « encapsule » le type **int**

- Constructeur à partir d'un int ou d'une chaîne de caractères

```
public Integer(int value);  
public Integer(String s);
```

- Disponibilité de méthodes qui permettent la conversion en type simple

```
Integer valueObjet = new Integer(123);  
int valuePrimitif = valueObjet.intValue();
```

Ou

```
int valuePrimitif = valueObjet; (AutoBoxing)
```

- Des méthodes de classe très utiles qui permettent à partir d'une chaîne de caractères de transformer en type simple ou type object

```
String maValueChaine = new String("12313");  
int maValuePrimitif = Integer.parseInt(maValueChaine);
```

Attention aux erreurs de conversion. Retour d'une exception. Voir dans la dernière partie du cours



Les tableaux en Java : application Objets

① Déclaration

```
Voiture[]  
monTableau;
```

② Dimensionnement

```
monTableau = new Voiture[3];
```

③ Initialisation

```
monTableau[0] = new Voiture(5);  
monTableau[1] = new Voiture(7);  
monTableau[2] = new Voiture(8);
```

Ou ①② et ③

```
Voiture[] monTab = {  
    new Voiture(5),  
    new Voiture(7),  
    new Voiture(8)  
};
```



```
for (int i = 0; i < monTableau.length; i++) {  
    System.out.println(monTableau[i].demarre());  
}
```

Varargs : passage de paramètres en nombre indéfini

- **Varargs** est une nouveauté Java 5 permettant de passer en paramètre un nombre *indéfini* de valeurs de *même type*
- Pour ceux qui ont connu le langage Turbo Pascal, l'équivalent du `System.out.println()` le permettait déjà
- Avant la version Java 5, il fallait passer en paramètre un tableau d'un type donné pour réaliser la même chose

```
public ajouterPassager(String[] tab)
```

- La syntaxe de *varargs* est la suivante : utilisation de « ... »

```
public ajouterPassager(String... tab)
```

Varargs : passage de paramètres en nombre indéfini

- Du côté de la méthode où le *varargs* est défini, les données sont manipulées comme un tableau

```
public ajouterPassager(String... tab) {  
    for (String current : tab) {  
        System.out.println(current)  
    }  
}
```

- Du côté client qui fait un appel à la méthode, les données peuvent être envoyées comme un

- Tableau

```
String passagers = {"Tony", "Luck",  
"John"};  
maVoiture.ajouterPassager(passagers);
```

- Ensemble de paramètres

```
maVoiture.ajouterPassager("Tony", "Luck", "John");
```

Varargs : passage de paramètres en nombre indéfini

- Comme un varargs est considéré comme un tableau le contenu peut être vide

```
public Voiture(int... carac) {  
    ...  
}  
public static void main(String[] argv) {  
    new Voiture();  
}
```

- Si un varargs est accompagné d'un ou plusieurs autres paramètres, le varargs doit obligatoirement être placé en dernier

```
public Voiture(String mod, int... carac) {  
    ...  
}  
  
public Voiture(int... carac, String mod) {  
    ...  
}
```

Varargs : passage de paramètres en nombre indéfini

- Problématiques liées à la surcharge d'une méthode utilisant un varargs
 - Dans le cas de la surcharge d'une méthode la méthode contenant le varargs a la priorité la plus faible

```
public class Voiture {  
    public Voiture(int... carac) {  
    }  
  
    public Voiture(int caract1, int caract2) {  
        ...  
    }  
  
    public static void main(String[] argv) {  
        new Voiture(12, 23);  
        new Voiture(12);  
    }  
}
```



Programmation Orientée Objet application au langage Java

Héritage

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Définition et intérêts

➤ Héritage

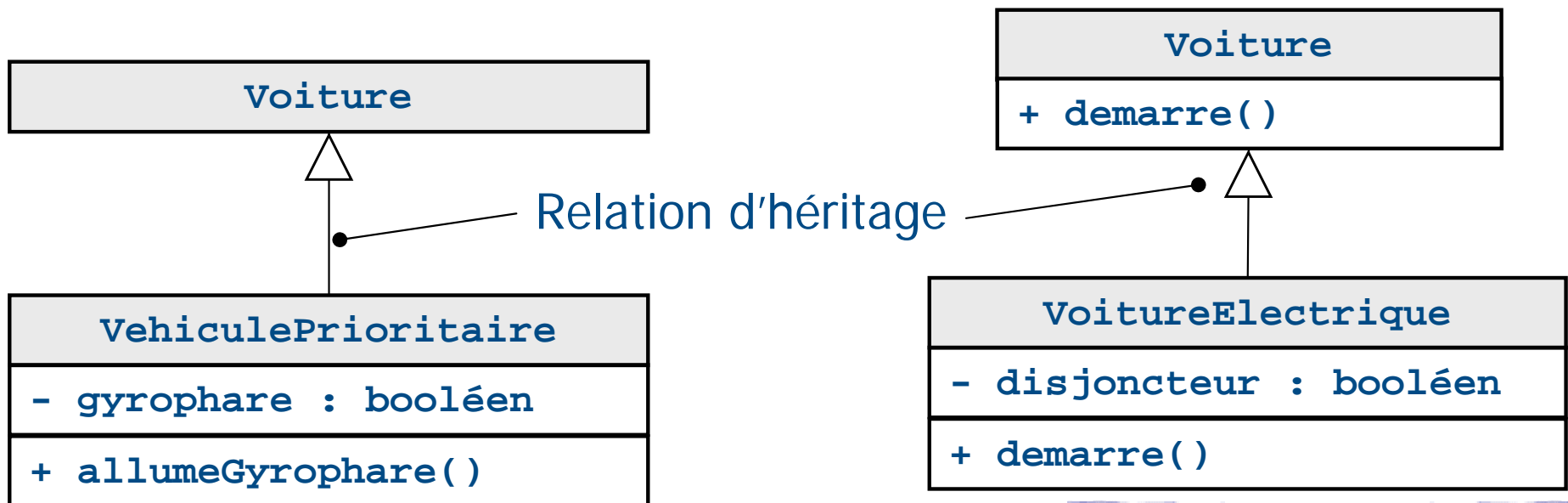
- Technique offerte par les langages de programmation pour construire une classe à partir d'une (ou plusieurs) autre classe en partageant ses attributs et opérations

➤ Intérêts

- **Spécialisation, enrichissement** : une nouvelle classe réutilise les attributs et les opérations d'une classe en y ajoutant et/ou des opérations particulières à la nouvelle classe
- **Redéfinition** : une nouvelle classe redéfinit les attributs et opérations d'une classe de manière à en changer le sens et/ou le comportement pour le cas particulier défini par la nouvelle classe
- **Réutilisation** : évite de réécrire du code existant et parfois on ne possède pas les sources de la classe à hériter

Spécialisation de la classe « Voiture »

- Un véhicule prioritaire est une voiture avec un gyrophare
 - Un véhicule prioritaire répond aux mêmes messages que la Voiture
 - On peut allumer le gyrophare d'un véhicule prioritaire
- Une voiture électrique est une voiture dont l'opération de démarrage est différente
 - Une voiture électrique répond aux même messages que la Voiture
 - On démarre une voiture électrique en activant un disjoncteur



Classes et sous-classes

- Un objet de la classe *VehiculePrioritaire* ou *VoitureElectrique* est aussi un objet de la classe *Voiture* donc il dispose de tous les attributs et opérations de la classe *Voiture*

VehiculePrioritaire	
	- gyrophare : booléen
	+ allumeGyrophare()
Hérité de Voiture	- puissance : entier
	- estDemarree : boolean
	- vitesse : flottant
	+ deQuellePuissance() : entier
	+ demarre() + accelere(flottant)

VoitureElectrique	
	- disjoncteur : booléen
	+ demarre()
Hérité de Voiture	- puissance : entier
	- estDemarree : boolean
	- vitesse : flottant
	+ deQuellePuissance() : entier
	+ demarre() + accelere(flottant)

Classes et sous-classes : terminologie

➤ Définitions

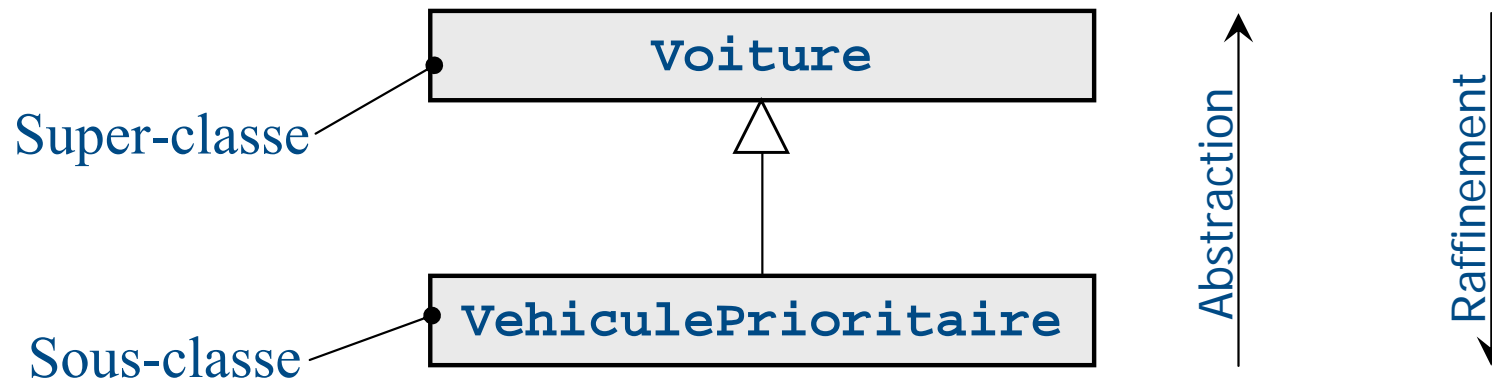
- La classe *VehiculePrioritaire* **hérite** de la classe *Voiture*
- *Voiture* est la **classe mère** et *VehiculePrioritaire* la **classe fille**
- *Voiture* est la **super-classe** de la classe *VehiculePrioritaire*
- *VehiculePrioritaire* est une **sous-classe** de *Voiture*

➤ Attention

- Un objet de la classe *VehiculePrioritaire* ou *VoitureElectrique* est forcément un objet de la classe *Voiture*
- Un objet de la classe *Voiture* n'est pas forcément un objet de la classe *VehiculePrioritaire* ou *VoitureElectrique*

Généralisation et Spécialisation

- La généralisation exprime une relation « **est-un** » entre une classe et sa super-classe

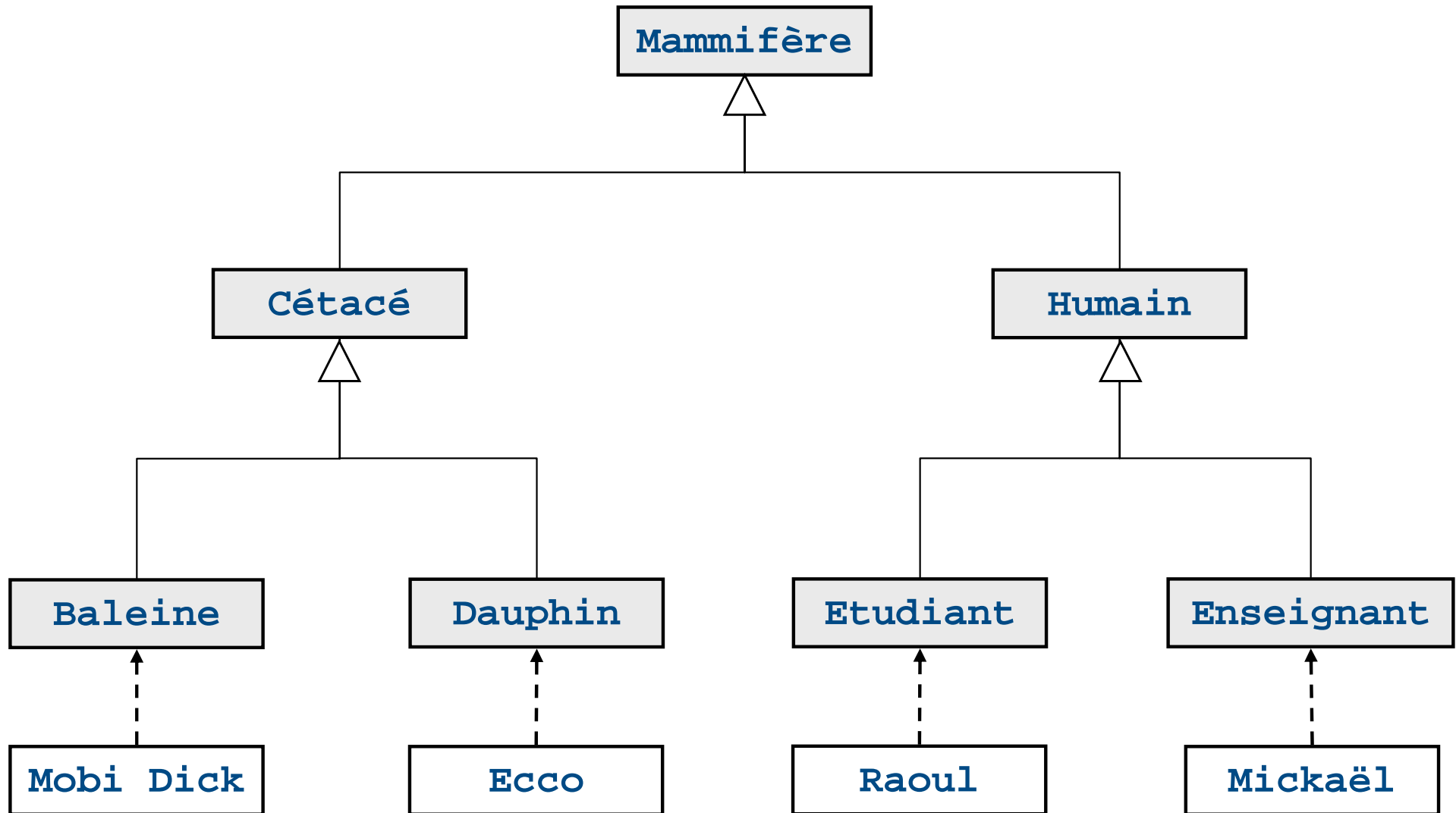


 *VehiculePrioritaire est une Voiture*

- L'héritage permet
 - de **généraliser** dans le sens abstraction
 - de **spécialiser** dans le sens raffinement

Exemple d'héritage

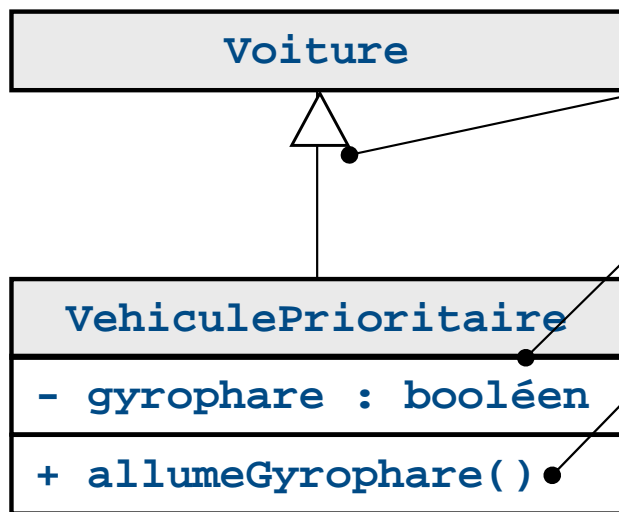
➤ Exemple : espèces



Héritage et Java

➤ Héritage simple

- Une classe ne peut hériter que d'une seule autre classe
- Dans certains autres langages (ex : C++) possibilité d'héritage multiple
- Utilisation du mot-clé **extends** après le nom de la classe

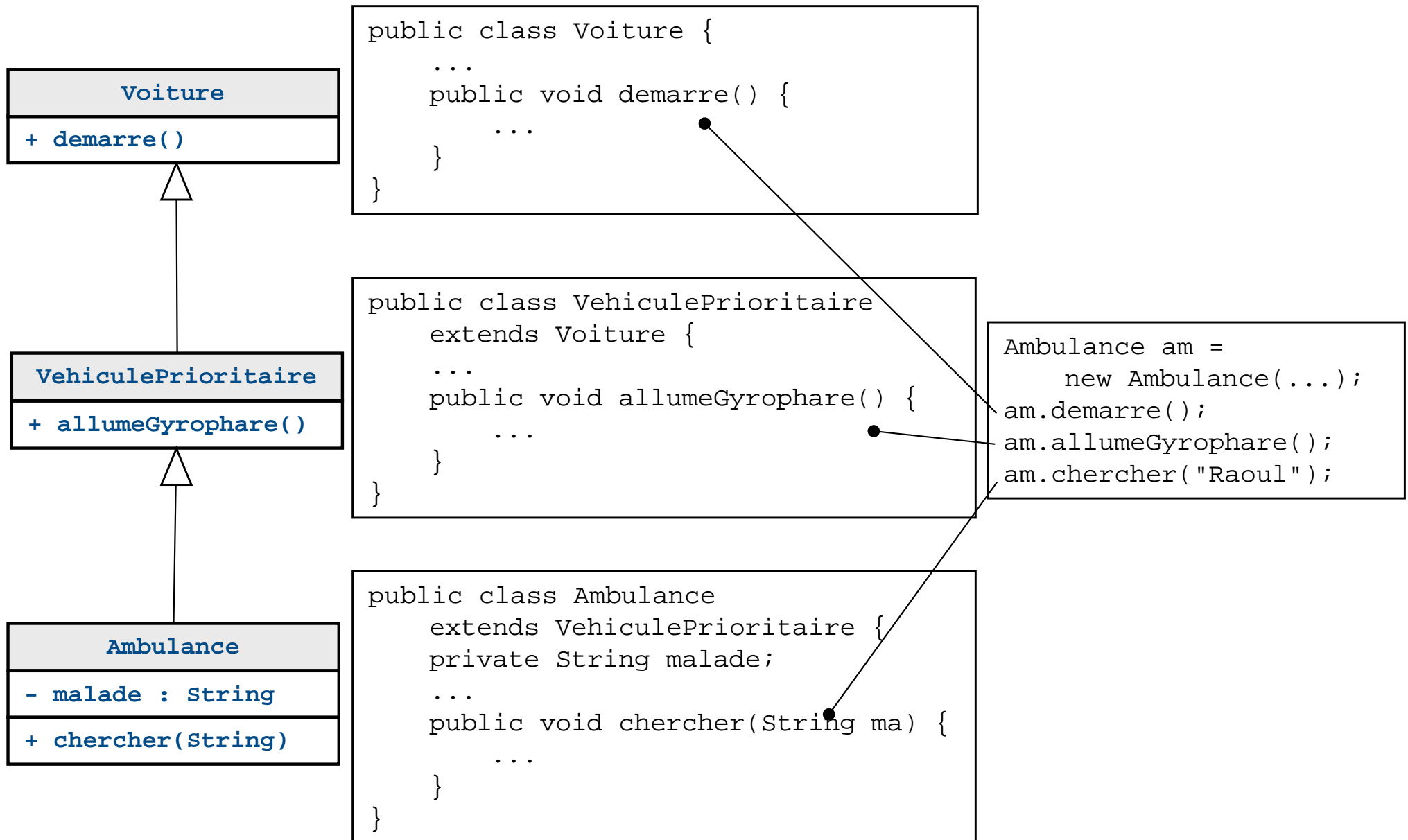


```
public class VehiculePrioritaire extends Voiture {
    private boolean gyrophare;
    ...
    public void allumeGyrophare() {
        gyrophare = true;
    }
    ...
}
```

**N'essayez pas d'hériter de plusieurs classes (extends *Voiture*, *Sante*, ...)
ça ne fonctionne pas**



Héritage à plusieurs niveaux



Surcharge et redéfinition

- L'héritage
 - Une sous-classe peut ajouter des nouveaux attributs et/ou méthodes à ceux qu'elle hérite (surcharge en fait partie)
 - Une sous-classe peut redéfinir (redéfinition) les méthodes à ceux dont elle hérite et fournir des implémentations spécifiques pour celles-ci
- Rappel de la *surcharge* : possibilité de définir des méthodes possédant le même nom mais dont les arguments (paramètres et valeur de retour) diffèrent

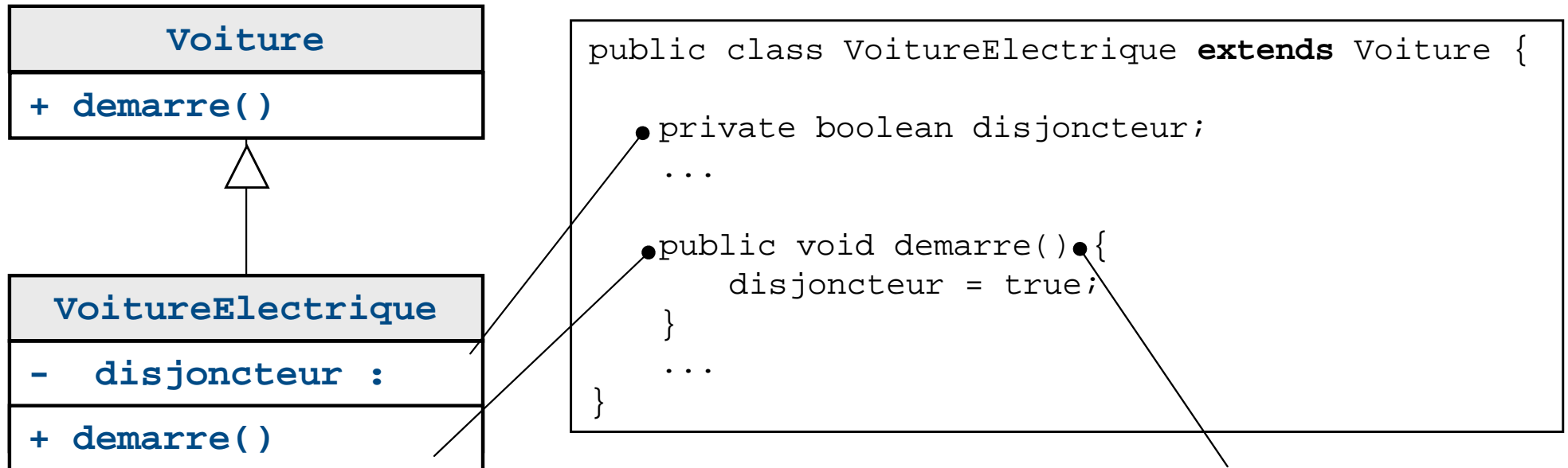
Des méthodes surchargées peuvent avoir des types de retour différents à condition qu'elles aient des arguments différents



- *Redéfinition* (overriding) : lorsque la sous-classe définit une méthode dont le nom, les paramètres et le type de retour sont identiques

Surcharge et redéfinition

- Une voiture électrique est une voiture dont l'opération de démarrage est différente
 - Une voiture électrique répond aux mêmes messages que la *Voiture*
 - On démarre une voiture électrique en activant un disjoncteur



Redéfinition de la méthode

Surcharge et redéfinition

```
public class Voiture {  
    ...  
    public void demarre() {  
        ...  
    }  
}
```

**Ne pas confondre surcharge et redéfinition.
Dans le cas de la surcharge la sous-classe
ajoute des méthodes tandis que la redéfinition
« spécialise » des méthodes existantes**



Redéfinition

Surcharge

```
public class VoitureElectrique  
    extends Voiture {  
    ...  
    public void demarre() {  
        ...  
    }  
}
```

VoitureElectrique possède
« au plus » une méthode de
moins que *VehiculePrioritaire*

```
public class VehiculePrioritaire  
    extends Voiture {  
    ...  
    public void demarre(int code) {  
        ...  
    }  
}
```

VehiculePrioritaire possède
« au plus » une méthode de
plus que *VoitureElectrique*

Redéfinition avec réutilisation

➤ Intérêt

- La redéfinition d'une méthode cache le code de la méthode héritée
- Réutiliser le code de la méthode hérité par le mot-clé **super**
- **super** permet ainsi la désignation explicite de l'instance d'une classe dont le type est celui de la classe mère
- Accès aux attributs et méthodes redéfinies par la classe courante mais que l'on désire utiliser

```
super.nomSuperClasseMethodeAppelee(...);
```

➤ Exemple de la Voiture : les limites à résoudre

- L'appel à la méthode *demarre* de *VoitureElectrique* ne modifie que l'attribut *disjoncteur*

Redéfinition avec réutilisation

➤ Exemple : réutilisation de méthode



La position de **super** n'a ici aucune importance

```
public class Voiture {  
    private boolean estDemarree;  
    ...  
    public void demarre() {  
        estDemarree = true;  
    }  
}
```

Mise à jour de l'attribut *estDemarree*

```
public class VoitureElectrique extends Voiture {  
    private boolean disjoncteur;  
    ...  
    public void demarre() {  
        disjoncteur = true;  
        super.demarre();  
    }  
    ...  
}
```

Envoi d'un message par appel de *demarre*

```
public class TestMaVoiture {  
    public static void main (String[] argv) {  
        // Déclaration puis création  
        VoitureElectrique laRochele =  
            new VoitureElectrique(...);  
        laRochele.demarre();  
    }  
}
```

Usage des constructeurs : suite

➤ Possibilité comme les méthodes de réutiliser le code des constructeurs de la super-classe

➤ Appel explicite d'un constructeur de la classe mère à l'intérieur d'un constructeur de la classe fille

➤ Utilise le mot-clé **super**

L'appel au constructeur de la super-classe doit se faire absolument en première instruction

`super(paramètres du constructeur);`

➤ Appel implicite d'un constructeur de la classe mère est effectué quand il n'existe pas d'appel explicite. Java insère implicitement l'appel **super()**



Usage des constructeurs : suite

➤ Exemple : constructeurs voiture

```
public class Voiture {
    ...

    public Voiture() {
        this(7, new Galerie());
    }

    public Voiture(int p) {
        this(p, new Galerie());
    }

    • public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
}
```



L'appel au constructeur de la super-classe doit se faire absolument en première instruction

Implantation du constructeur de *VoiturePrioritaire* à partir de *Voiture*

```
public class VoiturePrioritaire
    extends Voiture {
    private boolean gyrophare;

    public VoiturePrioritaire(int p, Galerie g) {
        super(p, null);
        this.gyrophare = false;
    }
}
```

Usage des constructeurs : suite

➤ Exemple : chaînage des constructeurs

```
public class A {  
    • public A() {  
        System.out.println("Classe A");  
    }  
}
```

```
public class B extends A {  
    • public B(String message) {  
        super(); // Appel implicite  
        System.out.println("Classe B");  
        System.out.println(message);  
    }  
}
```

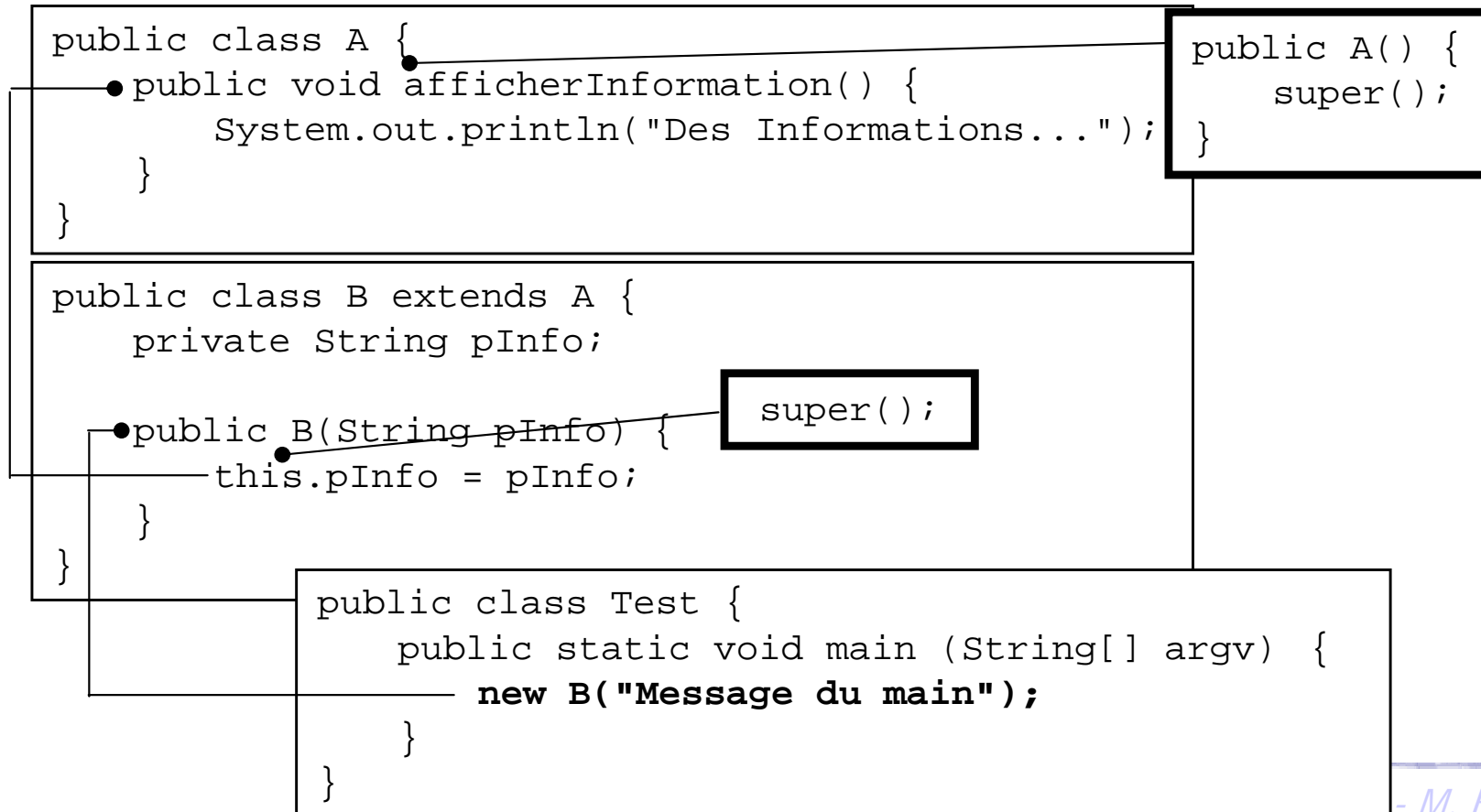
```
public class C extends B {  
    • public C(String debut) {  
        super("Message issu C" + debut);  
        System.out.println("Classe C");  
        System.out.println("Fin");  
    }  
}
```

```
public class Test {  
    public static void main (String[] argv) {  
        new C(" et Message du main");  
    }  
}
```

```
Console [ <arrête> C:\Pr...exe (23/07/04 15:15)]  
Classe A  
Classe B  
Message issu C et Message du main  
Classe C  
Fin
```

Usage des constructeurs : suite

- Rappel : si une classe ne définit pas explicitement de constructeur, elle possède alors un constructeur par défaut
 - Sans paramètre
 - Qui ne fait rien
 - Inutile si un autre constructeur est défini explicitement



Usage des constructeurs : suite

➤ Exemple : constructeur explicite

```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

Constructeurs explicites
désactivation du
constructeur par défaut

**Erreur : il n'existe pas dans
Voiture de constructeur sans
paramètre**

```
public class VoiturePrioritaire
    extends Voiture {

    private boolean gyrophare; super();

    public VoiturePrioritaire(int p, Galerie g) {
        this.gyrophare = false;
    }
}
```


La classe **Object** : le mystère résolu

- La classe **Object** est la classe de plus haut niveau dans la hiérarchie d'héritage
 - Toute classe autre que **Object** possède une super-classe
 - Toute classe hérite directement ou indirectement de la classe **Object**
 - Une classe qui ne définit pas de clause **extends** hérite de la classe **Object**

```
public class Voiture extends Object {  
    ...  
  
    public Voiture(int p, Galerie g) {  
        puissance = p;  
        moteur = new Moteur(puissance);  
        galerie = g;  
        ...  
    }  
    ...  
}
```

Object
+ Class getClass() + String toString() + boolean equals(Object) + int hashCode() ...



Il n'est pas nécessaire
d'écrire explicitement
extends Object

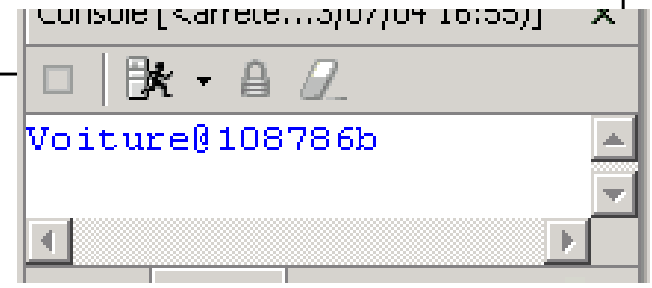
La classe Object : le mystère résolu

Avant redéfinition

```
public class Voiture {  
    ...  
    public Voiture(int p) {  
        this(p, new Galerie());  
    }  
}
```

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture maVoiture = new Voiture(5);  
        System.out.println(maVoiture);  
    }  
}
```

```
public String toString() {  
    return (this.getClass().getName() +  
            "@" + this.hashCode());  
}
```

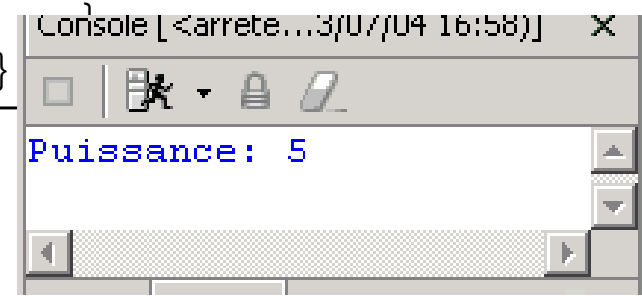


Après redéfinition

```
public class Voiture {  
    ...  
    public Voiture(int p) {  
        this(p, new Galerie());  
    }  
  
    public String toString() {  
        return("Puissance:" + p)  
    }  
}
```

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture maVoiture = new Voiture(5);  
        System.out.println(maVoiture);  
    }  
}
```

```
.ln(maVoiture.toString());
```



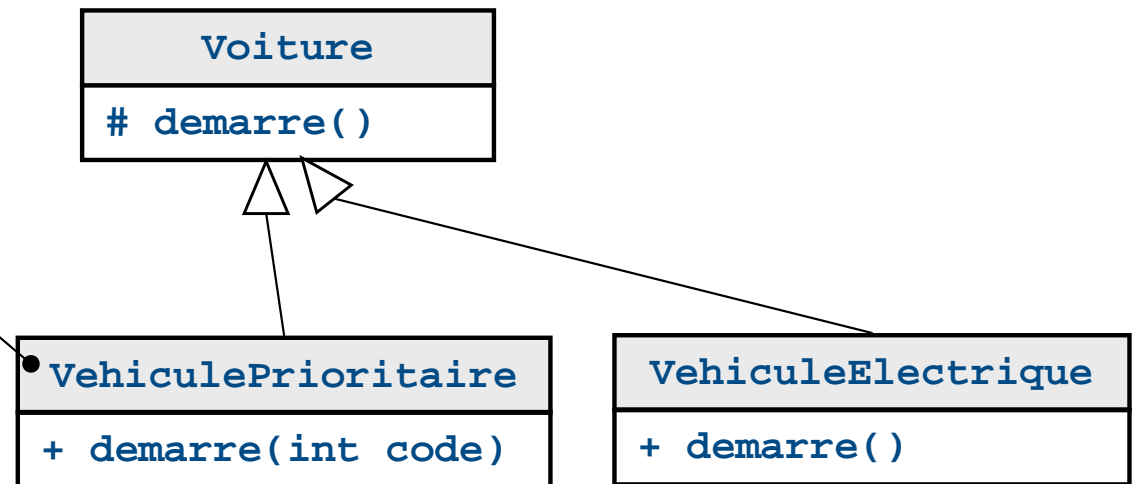
Redéfinition de la méthode
String toString()

Droits d'accès aux attributs et méthodes

- Exemple de la Voiture : les limites à résoudre
 - *demarre()* est disponible dans la classe *VehiculePrioritaire*
C'est-à-dire que l'on peut démarrer sans donner le code !!!
 - Solution : protéger la méthode *demarre()* de la classe Voiture

➤ Réalisation

- Utilisation du mot-clé **protected** devant la définition des méthodes et/ou attributs
- Les membres sont accessibles dans la classe où ils sont définis, dans toutes ses sous-classes



demarre() n'est pas accessible « publiquement » dans un objet *VehiculePrioritaire*

Droits d'accès aux attributs et méthodes

➤ Exemple : accès aux méthodes

```
public class Voiture {
    private boolean estDemarree;
    ...
    protected void demarre() {
        estDemarree = true;
    }
}
```

```
public class VoiturePrioritaire
    extends Voiture {
    private int codeVoiture;

    public void demarre(int code) {
        if (codeVoiture == code) {
            super.demarre();
        }
    }
}
```

```
public class TestMaVoiture {
    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        VehiculeElectrique laRochele = new VehiculeElectrique(...);
        laRochele.demarre(); // Appel le demarre de VehiculeElectrique

        VehiculePrioritaire pompier = new VehiculePrioritaire(...);
        pompier.demarre(1234); // Appel le demarre VoiturePrioritaire
        pompier.demarre(); // Erreur puisque demarre n'est pas public
    }
}
```

Méthodes et classes finales

➤ Définition

- Utilisation du mot-clé **final**
- Méthode : interdire une éventuelle redéfinition d'une méthode

```
public final void demarre();
```

- Classe : interdire toute spécialisation ou héritage de la classe concernée

```
public final class VoitureElectrique extends Voiture {  
    ...  
}
```



La classe *String* par exemple est finale



Programmation Orientée Objet application au langage Java

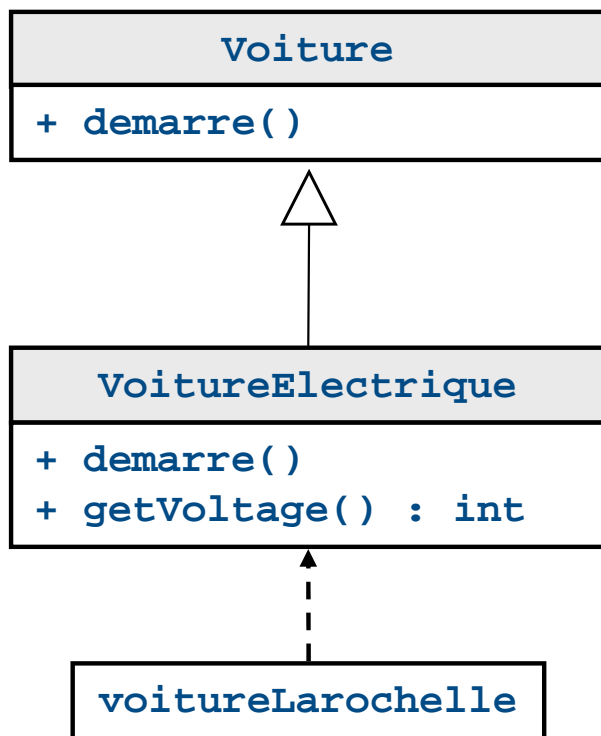
Héritage et Polymorphisme

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Définition du polymorphisme

➤ Définition

- Un langage orienté objet est dit polymorphique, s'il offre la possibilité de pouvoir percevoir un objet en tant qu'instance de classes variées, selon les besoins
- **Une classe B qui hérite de la classe A peut être vue comme un sous-type du type défini par la classe A**



➤ Rappel

- *voitureLarochelle* est une instance de la classe *VoitureElectrique*

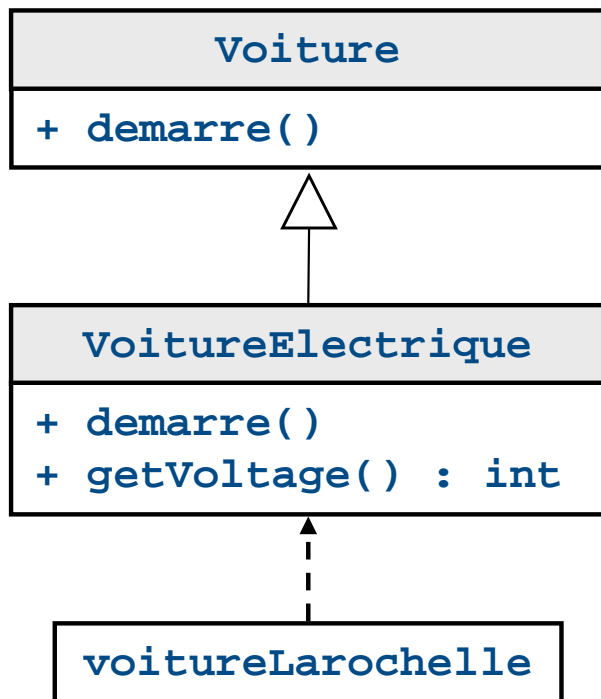
➤ Mais aussi

- *voitureLarochelle* est une instance de la classe *Voiture*

Polymorphisme et Java : surclassement

➤ Java est polymorphique

- A une référence de la classe *Voiture*, possible d'affecter une valeur qui est une référence vers un objet de la classe *VoitureElectrique*
- On parle de **surclassement** ou **upcasting**
- A une référence d'un type donné, soit A, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous classe directe ou indirecte de A



Objet de type sous-classe directe de Voiture

```

public class Test {
    public static void main (String[] argv) {

        Voiture voitureLarochelle =
            new VoitureElectrique(...);

    }
}
  
```


Polymorphisme et Java : surclassement

➤ A la compilation

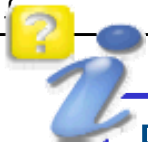
- Lorsqu'un objet est « surclassé », il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
- Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence

```
public class Test {
    public static void main (String[] argv) {

        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

        // Utilisation d'une méthode de la classe Voiture
        voitureLarochelle.demarre();

        // Utilisation d'une méthode de la classe VoitureElectrique
        System.out.println(voitureLarochelle.getVoltage()); // Erreur
    }
}
```



Examiner le type de la référence

La méthode *getVoltage()* n'est pas disponible dans la classe Voiture!!!

Polymorphisme et Java : surclassement

➤ Exemple : surclassement

```

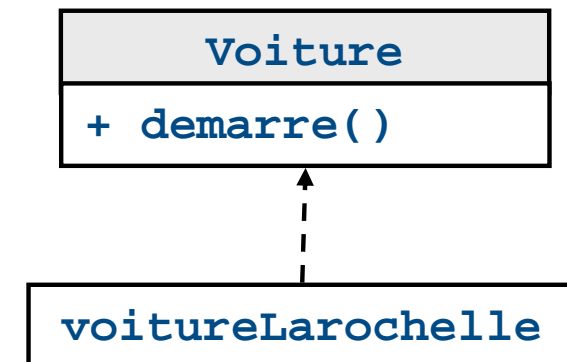
public class Test {
    public static void main (String[] argv) {
        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

        // Utilisation d'une méthode de la classe Voiture
        voitureLarochelle.demarre(); ✓

        // Utilisation d'une méthode de la classe VoitureElectrique
        System.out.println(voitureLarochelle.getVoltage()); ✗
    }
}

```

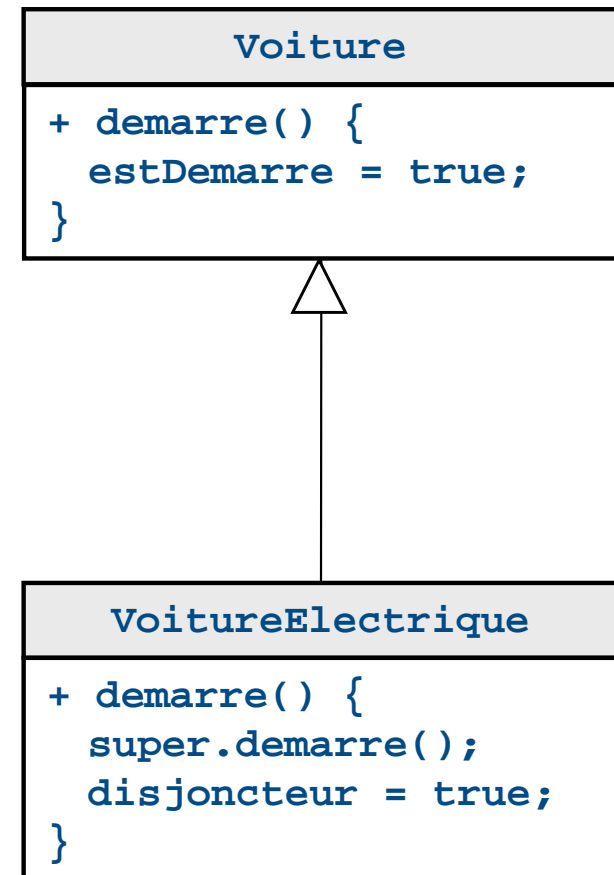
Remarque : Quel code va être effectivement exécuté lorsque le message `demarre()` est envoyé à `voitureLarochelle` ??



Polymorphisme et Java : lien dynamique

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture voitureLarochelle =  
            new VoitureElectrique(...);  
  
        voitureLarochelle.demarre();  
    }  
}
```

L'objet `voitureLarochelle` initialise les attributs de la classe *VoitureElectrique*



`voitureLarochelle.demarre()`

Constat : C'est la méthode *demarre()* de *VoitureElectrique* qui est appelée. Puis elle appelle (par `super...`) la méthode de la super-classe

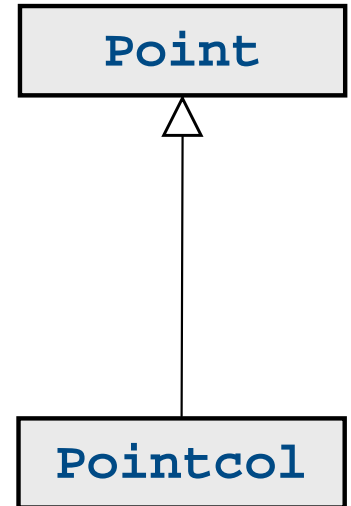
Polymorphisme et Java : lien dynamique

► Exemple : lien dynamique

```
public class Point {
    private int x,y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void deplace(int dx, int dy) { x += dx; y+=dy; }
    public void affiche() { System.out.println("Je suis en "+ x + " " +
y);}
}
```

```
public class Pointcol extends Point {
    private byte couleur;
    public Pointcol(int x, int y, byte couleur) {
        super(x,y);
        this.couleur = couleur;
    }
    public void affiche() {
        super.affiche();
        System.out.println("et ma couleur est : " + couleur);
    }
}
```

```
public class Test {
    public static void main (String[] argv) {
        Point p = new Point(23,45);
        p.affiche();
        Pointcol pc = new Pointcol(5,5,(byte)12);
        p = pc;
        p.affiche();
        p = new Point(12,45);
        p.affiche();
    }
}
```



```

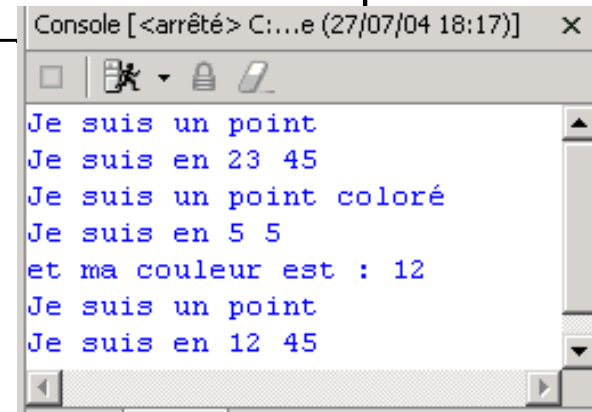
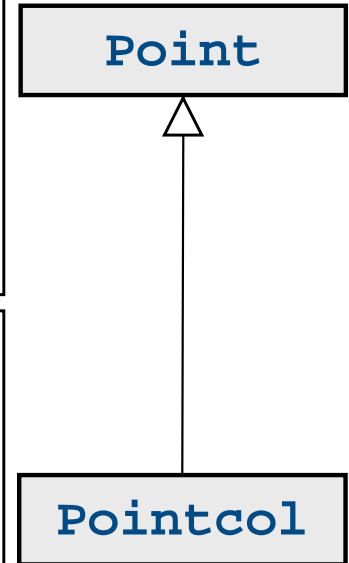
Je suis en 23 45
Je suis en 5 5
et ma couleur est : 12
Je suis en 12 45
    
```

Polymorphisme et Java : lien dynamique

```
public class Point {
    private int x,y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void deplace(int dx, int dy) { x += dx; y+=dy; }
    public void affiche() {
        this.identifie();
        System.out.println("Je suis en " + x + " " + y);
    }
    public void identifie() {System.out.println("Je suis un point");}
}
```

```
public class Pointcol extends Point {
    private byte couleur;
    public Pointcol(int x, int y, byte couleur) {...}
    public void affiche() {
        super.affiche();
        System.out.println("et ma couleur est : " + couleur);
    }
    public void identifie() {System.out.println("Je suis un point coloré");}
}
```

```
public class Test {
    public static void main (String[] argv) {
        Point p = new Point(23,45);
        p.affiche();
        Pointcol pc = new Pointcol(5,5,(byte)12);
        p = pc;
        p.affiche();
        p = new Point(12,45);
        p.affiche();
    }
}
```



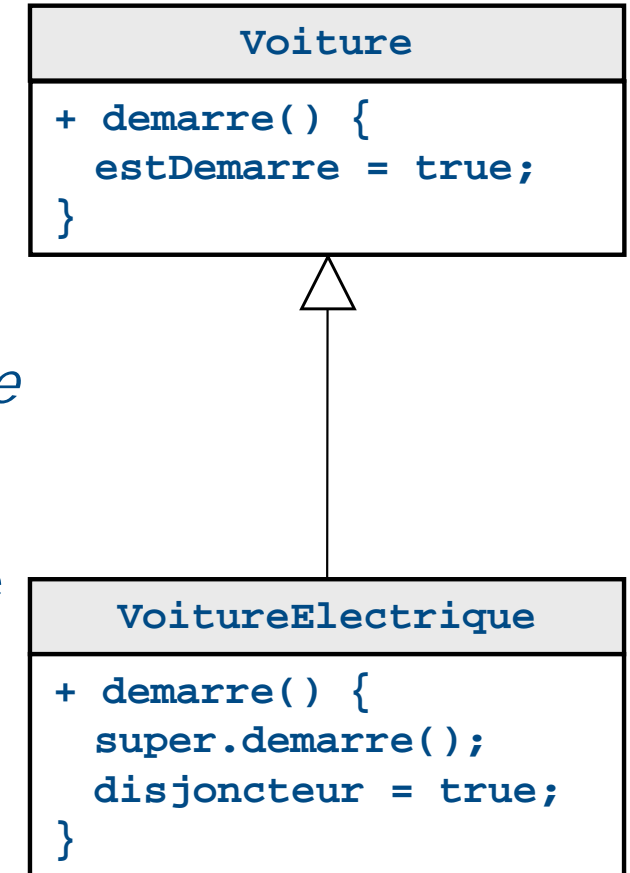
Polymorphisme et Java : lien dynamique

➤ A l'exécution

- Lorsqu'une méthode d'un objet est accédée au travers d'une référence « surclassée », c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est invoquée et exécutée
- La méthode à **exécuter** est déterminée à l'exécution et non pas à la **compilation**
- On parle de **liaison tardive**, **lien dynamique**, **dynamic binding**, **latebinding** ou **run-time binding**

Polymorphisme et Java : bilan

```
public class Test {
    public static void main (String[] argv) {
        Voiture maVoit = new VoitureElectrique(...);
        maVoit.demarre();
    }
}
```



➤ Surclassement (compilation)

- Une variable *maVoit* est déclarée comme étant une référence vers un objet de la classe *Voiture*
- Un objet de la classe *VoitureElectrique* est créé
- Pour le compilateur *maVoit* reste une référence d'un objet de la classe *Voiture*, et il empêche d'accéder aux méthodes spécifiques à *VoitureElectrique*

➤ Liaison dynamique (exécution)

- Une variable *maVoit* est bel et bien une référence vers un objet de la classe *VoitureElectrique*

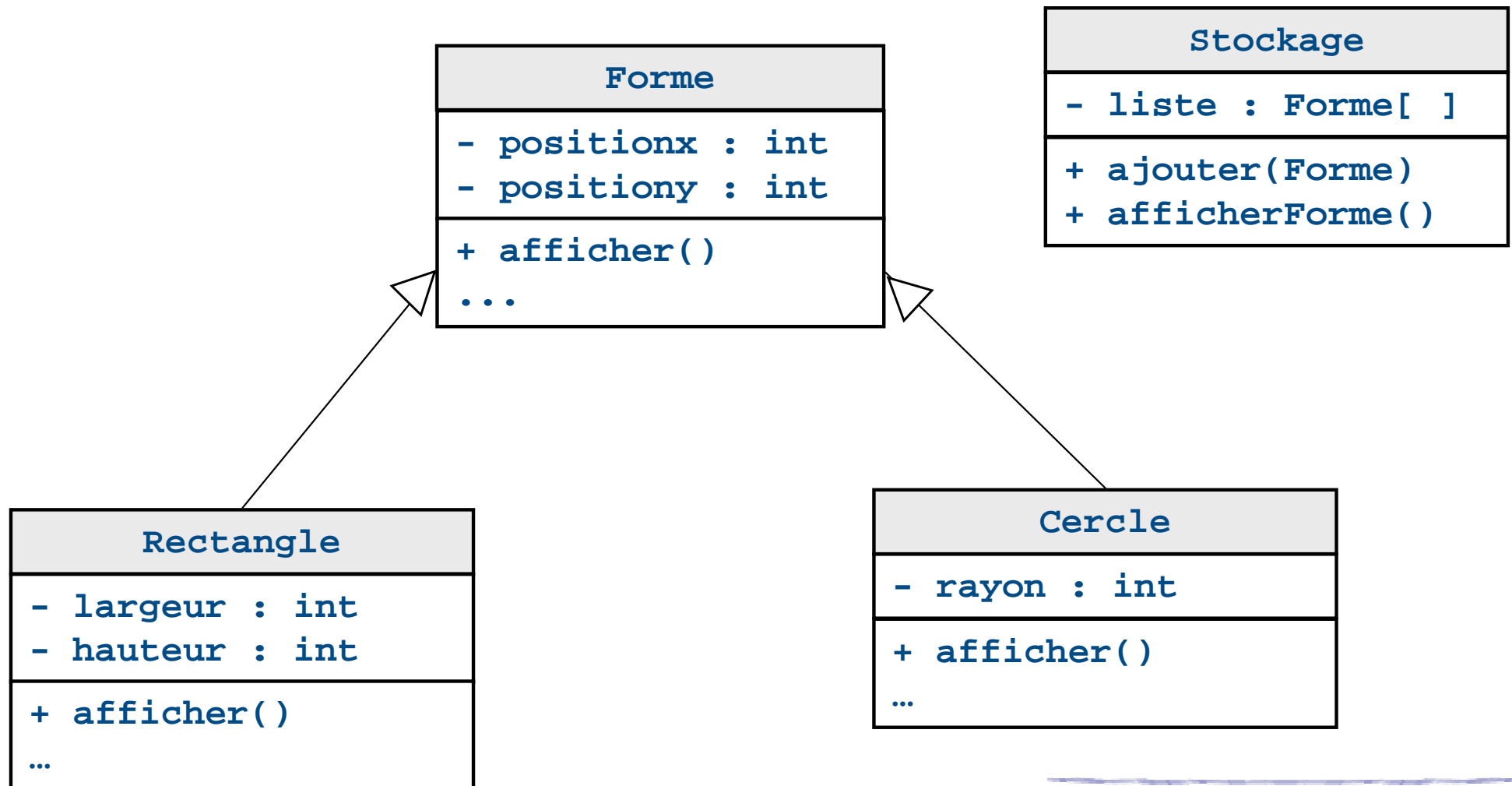
Polymorphisme : ok, mais pourquoi faire ?

- Que des avantages ...
 - Plus besoin de distinguer différents cas en fonction de la classe des objets
 - Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage
 - Une plus grande facilité d'évolution du code. Possibilité de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule la classe de base
 - Développement **plus rapide**
 - Plus grande **simplicité** et **meilleure organisation** du code
 - Programmes plus facilement **extensibles**
 - Maintenance du code **plus aisée**

Polymorphisme : un exemple typique

➤ Exemple : la géométrie

- Stocker des objets *Forme* de n'importe quel type (*Rectangle* ou *Cercle*) puis les afficher



Polymorphisme : un exemple typique

➤ Exemple (suite) : la géométrie

```
public class Stockage {
    private Forme[] liste;
    private int taille;
    private int i;

    public Stockage(int taille) {
        this.taille = taille;
        liste = new Forme[this.taille];
        i = 0;
    }

    public void ajouter(Forme f) {
        if (i < taille) {
            liste[i] = f;
            i++;
        }
    }

    public void afficherForme() {
        for (int i = 0; i < taille; i++) {
            liste[i].afficher();
        }
    }
}
```



Si un nouveau type de Forme est défini, le code de la classe *Stockage* n'est pas modifié

```
public class Test {
    public static void main (String[] argv) {
        Stockage monStock = new Stockage(10);
        monStock.ajouter(new Cercle(...));
        monStock.ajouter(new Rectangle(...));

        Rectangle monRect = new Rectangle(...);
        Forme tonRect = new Rectangle(...);
        monStock.ajouter(monRect);
        monStock.ajouter(tonRect);
    }
}
```

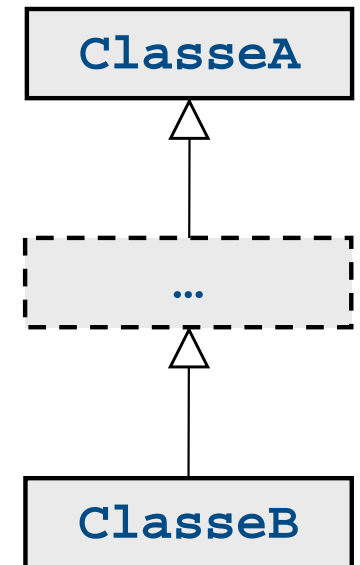
Polymorphisme : downcasting

➤ Intérêt

- Force un objet à « libérer » les fonctionnalités cachées par le surclassement
- Conversion de type explicite (cast). Déjà vu pour les types primitifs

```
ClasseA monObj = ...  
ClasseB b = (ClasseB) monObj
```

- Pour que le « cast » fonctionne, il faut qu'à l'exécution le type effectif de *monObj* soit « compatible » avec le type *ClasseB*



- Compatible : on peut tester la compatibilité par le mot clé **instanceof**

```
obj instanceof ClasseB
```

Retourne vrai ou faux

Polymorphisme : downcasting

➤ Exemple : downcasting

```
public class Test {
    public static void main (String[] argv) {
        Forme maForme = new Rectangle();
        // Je ne peux pas utiliser les méthodes de la classe Rectangle

        // Déclaration d'un objet de type Rectangle
        Rectangle monRectangle;
        if (maForme instanceof Rectangle) {
            monRectangle = (Rectangle)maForme;
            // Utilisation possible des méthodes spécifiques de Rectangle
        }
    }
}
```

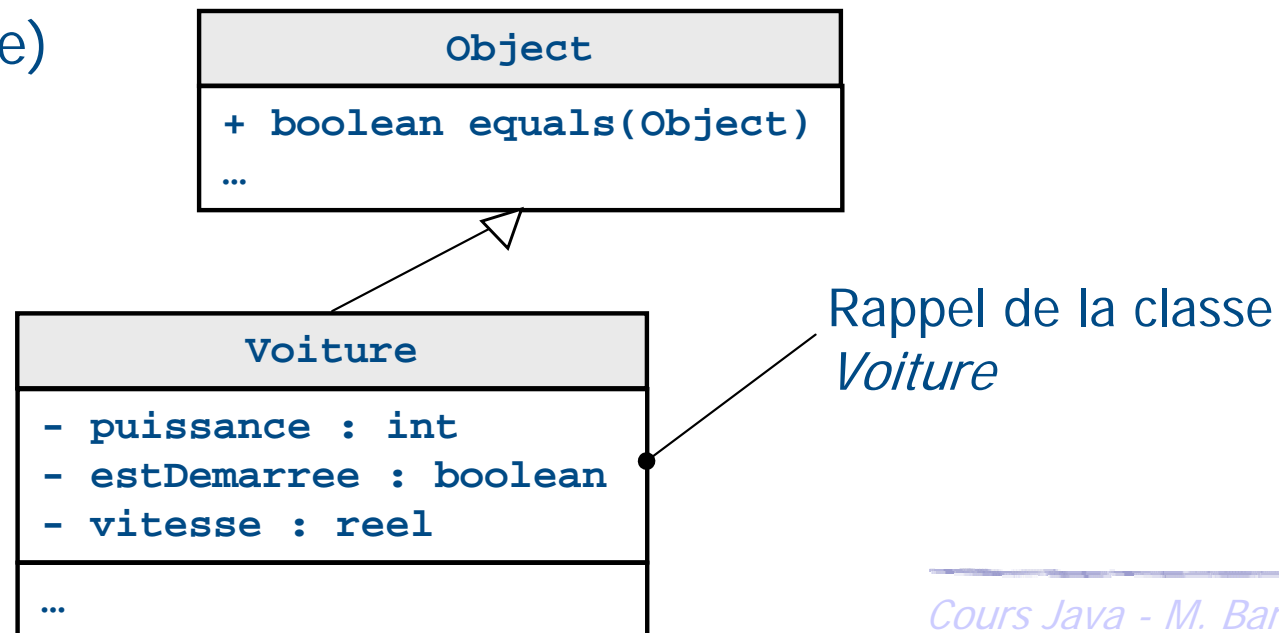
Réalisation de la conversion de l'objet de type Forme en objet de type Rectangle

Attention si la compatibilité est fautive et si le cast est effectué une exception de type *ClassCastException* est levée



La méthode « equals() »

- Deux possibilités pour comparer des objets d'une classe
 - Créer une méthode ad-hoc « *boolean comparer(MaClasse c) {...}* » qui compare les attributs
 - Redéfinir la méthode « *boolean equals(Object o)* » pour garder la compatibilité avec les autres classes de Java
 - Re-implémenter la méthode « *boolean equals(Object o)* » en comparant les attributs (en utilisant une conversion de type explicite)



La méthode « equals() »

► Exemple : redéfinition de la méthode *equals*

```
public class Voiture extends Object {  
    public boolean equals(Object o) {  
        if (!o instanceof Voiture) {  
            return false;  
        }  
  
        Voiture maVoit = (Voiture)o;  
        return this.puissance == maVoit.puissance && this.estDemarree ==  
            maVoit.estDemarree && this.vitesse == maVoit.vitesse;  
    }  
    ...  
}
```

Redéfinition de la méthode *equals* de la classe *Object*

Même valeurs d'arguments

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture maVoit = new Voiture(...);  
        VoitureElectrique maVoitele = new VoitureElectrique(...);  
  
        maVoit.equals(maVoitele); --> TRUE  
    }  
}
```

Attention : l'égalité de référence == vérifie si les références sont les mêmes, cela ne compare pas les attributs



Classes abstraites : intérêts

- On ne connaît pas toujours le comportement par défaut d'une opération commune à plusieurs sous-classes
 - Exemple : toit d'une voiture décapotable. On sait que toutes les décapotables peuvent ranger leur toit, mais le mécanisme est différent d'une décapotable à l'autre
 - Solution : on peut déclarer la méthode « abstraite » dans la classe mère et ne Pas lui donner d'implantation par défaut
- Méthode abstraite et conséquences : 3 règles à retenir
 - Si une seule des méthodes d'une classe est abstraite, alors la classe devient aussi abstraite
 - On ne peut pas instancier une classe abstraite car au moins une de ses méthodes n'a pas d'implémentation
 - Toutes les classes filles héritant de la classe mère abstraite doivent implémenter toutes ses méthodes abstraites ou sinon elles sont aussi abstraites

Classes abstraites et Java

- Le mot clé **abstract** est utilisé pour spécifier qu'une classe est abstraite
- Une classe abstraite se déclare ainsi

```
public abstract class NomMaClasse {  
    ...  
}
```

- Une méthode abstraite se déclare ainsi

```
public abstract void maMethode(...);
```

Pour créer une méthode abstraite, la signature (nom et paramètres) est déclarée sans spécifier le corps et en ajoutant le mot clé **abstract**

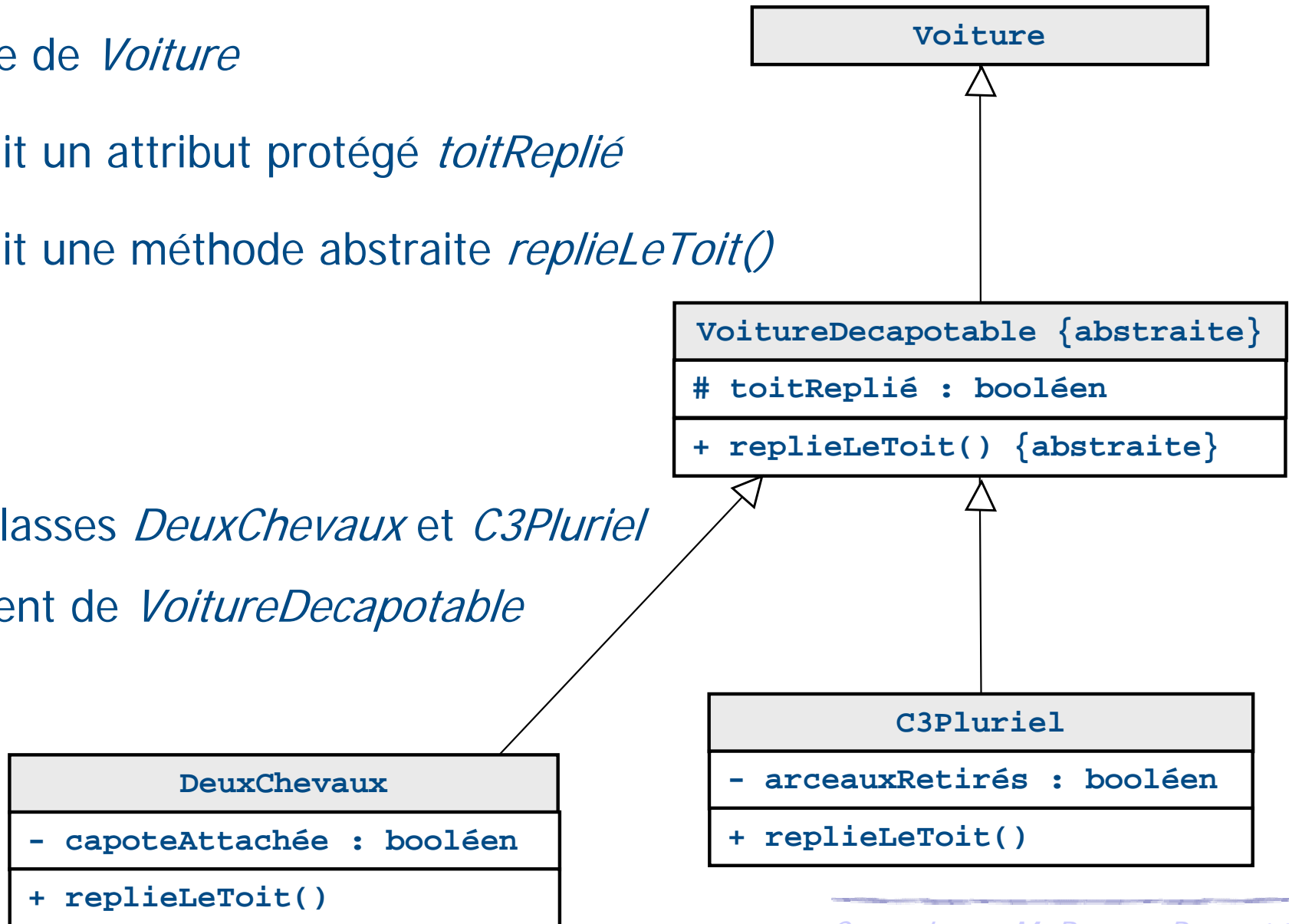


Classes abstraites : exemple VoitureDecapotable

➤ La classe *VoitureDecapotable*

- Hérite de *Voiture*
- Définit un attribut protégé *toitReplié*
- Définit une méthode abstraite *replieLeToit()*

- Les classes *DeuxChevaux* et *C3Pluriel* héritent de *VoitureDecapotable*



Classes abstraites : exemple VoitureDecapotable

➤ Exemple : une voiture décapotable

Classe
abstraite

```
public abstract class VoitureDecapotable
    extends Voiture {
    protected boolean toitReplié;

    public abstract void replieLeToit();
}
```

Méthode
abstraite

```
public class DeuxChevaux extends VoitureDecapotable {
    private boolean capoteAttachee;

    public void replieLeToit() {
        this.toitReplie = true;
        this.capoteAttachee = true;
    }
}
```

```
public class C3Pluriel extends VoitureDecapotable {
    private boolean arceauxRetirés;

    public void replieLeToit() {
        this.toitReplie = true;
        this.arceauxRetirés = true;
    }
}
```

Attention : ce n'est pas de la redéfinition. On parle d'implémentation de méthode abstraite



Classes abstraites : exemple VoitureDecapotable

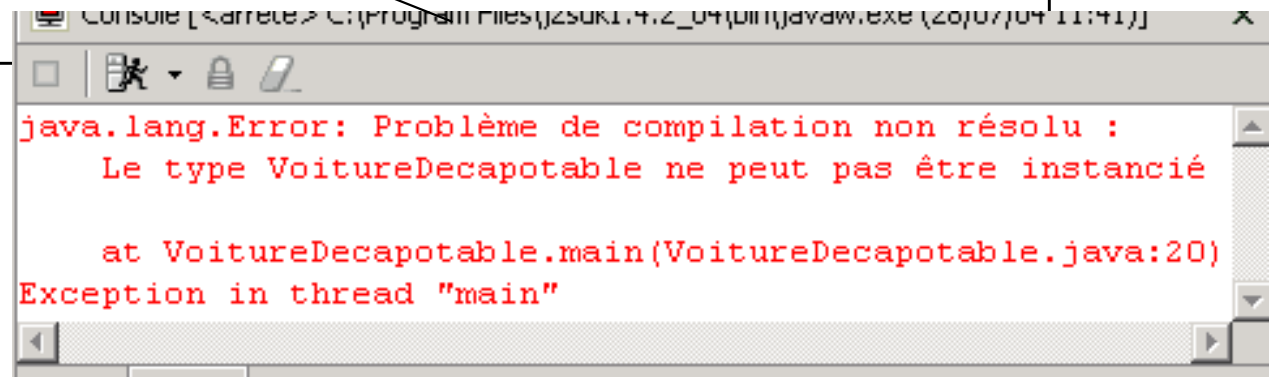
➤ Exemple (suite) : une voiture décapotable

```
public class Test {
    public static void main (String[] argv) {
        // Déclaration et création d'une DeuxCheveaux
        VoitureDecapotable voitureAncienne = new DeuxCheveaux(...);
        // Envoi de message
        voitureAncienne.replieLeToit();

        // Déclaration et création d'une C3Pluriel
        VoitureDecapotable voitureRecente = new C3Pluriel(...);
        // Envoi de message
        voitureRecente.replieLeToit();

        // Déclaration et création d'une VoitureDecapotable
        VoitureDecapotable voitureDecapotable =
            new VoitureDecapotable(...); // Erreur
    }
}
```

Attention : La classe *VoitureDecapotable* ne peut être instanciée puisqu'elle est abstraite



Classes abstraites : exemple Forme

➤ Exemple : la classe *Forme*

- Les méthodes *surface()* et *périmètre()* sont abstraites
- Ces méthodes n'ont de « sens » que pour les sous-classes *Cercle* et *Rectangle*

Cercle
- rayon : int
+ surface() : double + périmètre() : double

Forme {abstraite}
- positionx, positiony : int
+ deplace(x,y) + surface() : double {abstraite} + périmètre() : double {abstraite}

```
public abstract class Forme {
    private int positionx, positiony;

    public void deplacer(double dx, double dy){
        x += dx; y += dy;
    }

    public abstract double périmètre();
    public abstract double surface();
}
```

Rectangle
- larg, haut : int
+ surface() : double + périmètre() : double

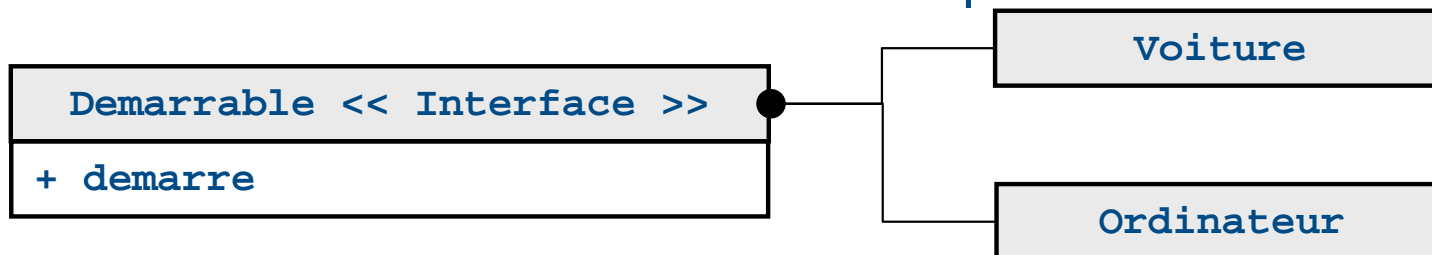
Pas d'implémentation !!

Notion d'interface

- Une interface est un modèle pour une classe
 - Quand toutes les méthodes d'une classe sont abstraites et qu'il n'y a aucun attribut nous aboutissons à la notion d'interface
 - Elle définit la signature des méthodes qui doivent être implémentées dans les classes qui respectent ce modèle
 - Toute classe qui implémente l'interface doit implémenter toutes les méthodes définies par l'interface
 - Tout objet instance d'une classe qui implémente l'interface peut être déclaré comme étant du type de cette interface
 - Les interfaces pourront se dériver

➤ Exemple

- Les choses *Demarrable* doivent posséder une méthode *demarre()*



Notion d'interface et Java

➤ Mise en œuvre d'une interface

- La définition d'une interface se présente comme celle d'une classe. Le mot clé **interface** est utilisé à la place de **class**

```
public interface NomInterface {  
    ...  
}
```

Interface : ne pas confondre avec les interfaces graphiques



- Lorsqu'on définit une classe, on peut préciser qu'elle implémente une ou plusieurs interface(s) donnée(s) en utilisant une fois le mot clé **implements**

```
public class NomClasse implements Interface1, Interface3, ... {  
    ...  
}
```

- Si une classe hérite d'une autre classe elle peut également implémenter une ou plusieurs interfaces

```
public class NomClasse extends SuperClasse implements Inter1, ... {  
    ...  
}
```

Notion d'interface et Java

- Mise en œuvre d'une interface
 - Une interface ne possède pas d'attribut
 - Une interface peut posséder des constantes

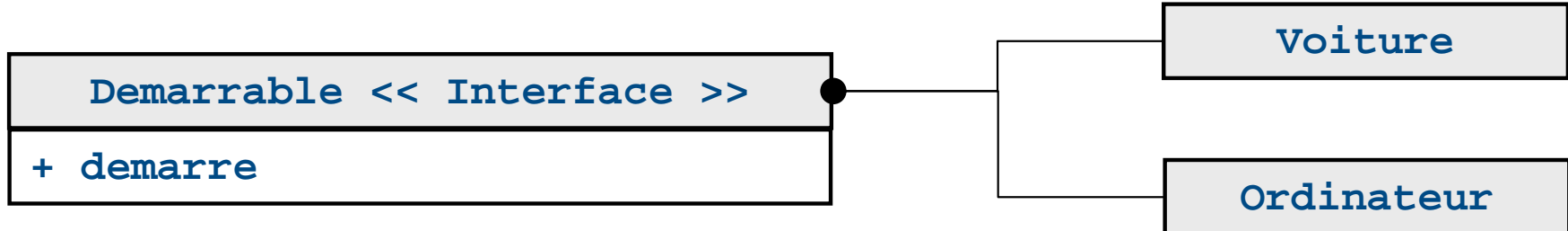
```
public interface NomInterface {  
    public static final int CONST = 2;  
}
```

- Une interface ne possède pas de mot clé **abstract**
- Les interfaces ne sont pas instanciables (Même raisonnement avec les classes abstraites)

```
NomInterface jeTente = new NomInterface(); // Erreur!!
```

Notion d'interface et Java

- Toute classe qui implémente l'interface doit implémenter toutes les méthodes définies par l'interface



```
public interface Demarrable {
    public void demarre();
}
```

```
public class Voiture implements Demarrable {
    ...
    public void demarre() {
        moteurAllumé = true;
    }
}
```

```
public class Ordinateur implements Demarrable {
    ...
    public void demarre() {
        estDemarree = true;
    }
}
```

Doit implémenter toutes les méthodes des interfaces

Une « Voiture » est « Demarrable »

Un « Ordinateur » est « Demarrable »

Notion d'interface et Java

- Tout objet instance d'une classe qui implémente l'interface peut être déclaré comme étant du type de cette interface

```

public class Test {
    public static void main (String[] argv) {
        // Déclaration d'un objet de type Demarrable
        Demarrable dem1;
        // Création d'un objet Voiture
        dem1 = new Voiture();

        // Déclaration et création d'un objet Personne
        Personne pers1 = new Personne(dem1);
        pers1.mettreEnRoute();

        // Déclaration d'un objet de type Demarrable
        Demarrable dem2;
        // Création d'un objet Ordinateur
        dem2 = new Ordinateur();

        // Déclaration et création d'un objet Personne
        Personne pers2 = new Personne(dem2);
        pers2.mettreEnRoute();
    }
}

```

Une personne peut
demarrer tous les
objets *Demarrable*

Notion d'interface et Java

- Exemple : une *Voiture* et un *Ordinateur* sont des objets *Demarrable*

```
public class Person {  
  
    private Demarrable objetDemarrable;  
  
    public Person(Demarrable dem) {  
        objetDemarrable = dem;  
    }  
  
    public void mettreEnRoute() {  
        objetDemarrable.demarre();  
    }  
}
```

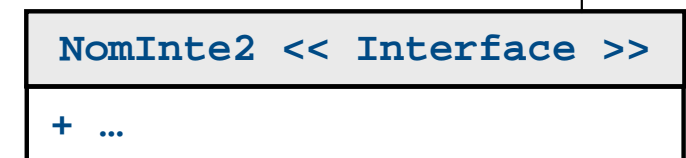
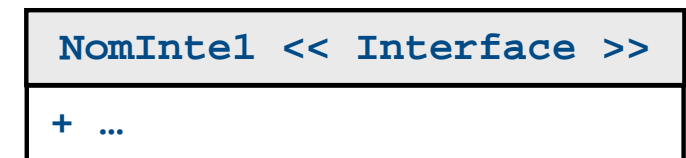
Une personne peut démarrer *Voiture* et *Ordinateur* sans connaître leur nature exacte

Notion d'interface et Java

- Les interfaces pourront se dériver
 - Une interface peut hériter d'une autre interface : « extends »

- Conséquences

- La définition de méthodes de l'interface mère *NomInte1* sont reprises dans l'interface fille *NomInte2*. Toute classe qui implémente l'interface fille doit donner une implémentation à toutes les méthodes mêmes celle héritées



- Utilisation

- Lorsqu'un modèle peut se définir en plusieurs sous-modèles complémentaires

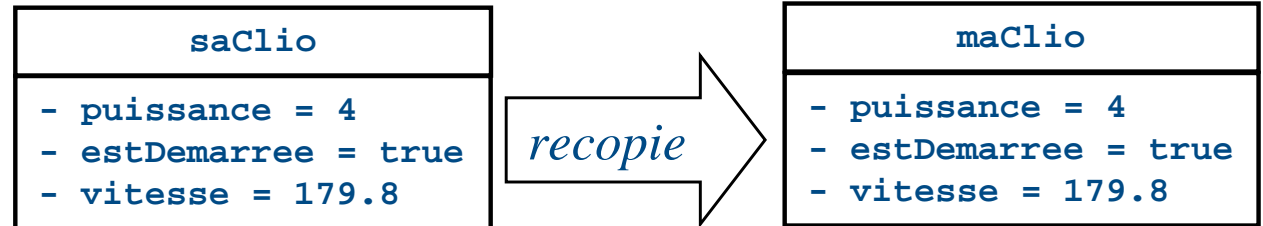
Classes abstraites versus interfaces

- Les classes
 - Elles sont complètement implémentées
 - Une autre classe peut en hériter
- Les classes abstraites
 - Sont partiellement implémentées
 - Une autre classe non abstraite peut en hériter mais doit donner une implémentation aux méthodes abstraites
 - Une autre classe abstraite peut en hériter sans forcément donner une implémentation à toutes les méthodes abstraites
 - Ne peuvent pas être instanciées mais peuvent fournir un constructeur
- Les interfaces
 - Elles ne sont pas implémentées
 - Toute classe qui implémente une ou plusieurs interfaces doit implémenter toutes leurs méthodes (abstraites)

L'interface « Cloneable »

- Deux possibilités pour dupliquer un objet
 - Créer une méthode ad-hoc *public MaClasse dupliquer()* qui retourne une copie de l'objet en ayant créé une nouvelle instance et initialisé les attributs (en utilisant le constructeur)

saClio != maClio mais
le contenu est identique



- Utiliser l'interface « Cloneable » pour garder la compatibilité avec les autres classes de Java
 - Implémenter la méthode *protected Object clone()* de l'interface *Cloneable*

```

public class Voiture implements Demarrable, Cloneable {
    protected Object clone() {
        Voiture copie;
        copie = new Voiture(this.puissance, (Galerie)laGalerie.clone());
        return copie;
    }
}
    
```

Les classes internes « Inner Classes »

➤ Règle de base en Java

- Une classe par fichier et un fichier par classe

➤ Classes locales ou internes

- Définies à l'intérieur d'autres classes (Moteur dans Voiture)

```
public class Voiture {  
    ...  
    class Moteur {  
        ...  
    }  
}
```

➤ Classes anonymes

- Sont des instanciations de classes et des implémentations d'une classe abstraite ou d'une interface
- La ou les méthodes abstraites doivent être implémentées au moment de l'instanciation

```
Demarrable uneInstance =  
    new Demarrable(){  
        public void demarre() {  
            // Code ici  
        }  
    };
```



Les classes anonymes sont très utilisées pour le développement d'IHM avec Java/Swing

Les classes internes « Inner Classes »

➤ Code source : 1 fichier

➤ classe

➤ classe anonyme

➤ classe interne

➤ Génération de byte-code : 3 fichiers

➤ classe *Voiture.class*

➤ anonyme *Voiture\$1.class*

➤ interne *Voiture\$Moteur.class*

Classe anonyme, implémente l'interface *Init*

```
public class Voiture {  
    public Voiture(...) {  
        monMoteur = new Moteur(...);  
        Init monInit = new Init() {  
            public void initialisation() {  
                ...  
            }  
        };  
    }  
    class Moteur {  
        ...  
        public Moteur(...) {  
            ...  
        }  
    }  
}
```

Classe interne

Nom	Taille	Type
Voiture\$1.class	1 Ko	Fichier CLASS
Voiture\$Moteur.class	1 Ko	Fichier CLASS
Voiture.class	1 Ko	Fichier CLASS
Voiture.java	1 Ko	Java Source File

Les fichiers .class qui possèdent dans leur nom un \$ ne sont pas des fichiers temporaires!!!





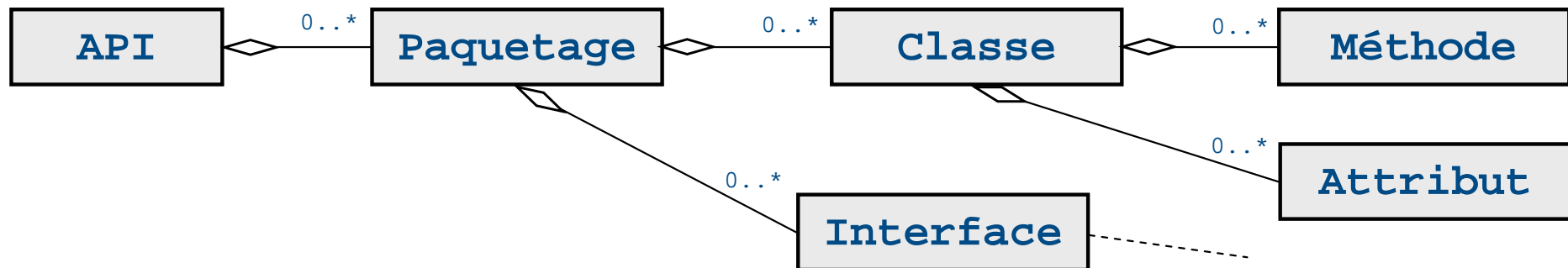
Programmation Orientée Objet application au langage Java

Les indispensables

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Les packages

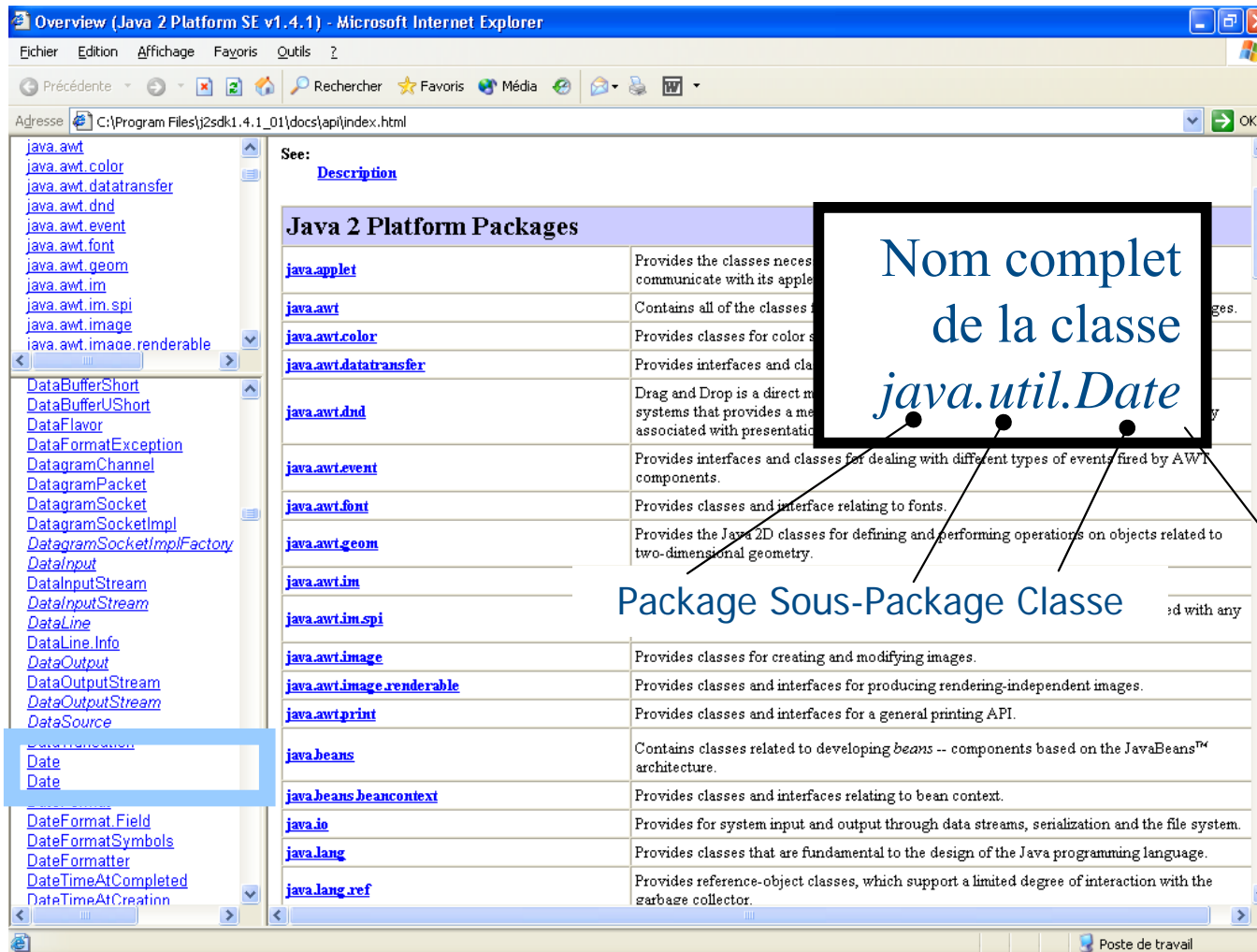
- Le langage Java propose une définition très claire du mécanisme d'emballage qui permet de classer et de gérer les API externes
- Les API sont constituées



- Un package est donc un groupe de classes associées à une fonctionnalité
- Exemples de packages
 - *java.lang* : rassemble les classes de base Java (*Object*, *String*, ...)
 - *java.util* : rassemble les classes utilitaires (*Collections*, *Date*, ...)
 - *java.io* : lecture et écriture

Les packages : ok, mais pourquoi faire?

- L'utilisation des packages permet de regrouper les classes afin d'organiser des libraires de classes Java
- Exemple : la classe *Date* est définie deux fois



java.sql
Class Date

```
java.lang.Object
|
+--java.util.Date
|
+--java.sql.Date
```

All Implemented Interfaces:
[Cloneable](#), [Comparable](#), [Serializable](#)

java.util
Class Date

```
java.lang.Object
|
+--java.util.Date
```

All Implemented Interfaces:
[Cloneable](#), [Comparable](#), [Serializable](#)

Direct Known Subclasses:
[Date](#), [Time](#), [Timestamp](#)

Les packages : utilisation des classes

- Lorsque, dans un programme, il y a une référence à une classe, le compilateur la recherche dans le package par défaut (*java.lang*)
- Pour les autres, il est nécessaire de fournir explicitement l'information pour savoir où se trouve la classe :
 - Utilisation d'**import** (classe ou paquetage)

```
import mesclasses.Point;
import java.lang.String; // Ne sert à rien puisque par défaut
import java.io.ObjectOutput;
```

OU

```
import mesclasses.*;
import java.lang.*; // Ne sert à rien puisque par défaut
import java.io.*;
```

- Nom du paquetage avec le nom de la classe

```
java.io.ObjectOutput toto = new java.io.ObjectOutput(...)
```

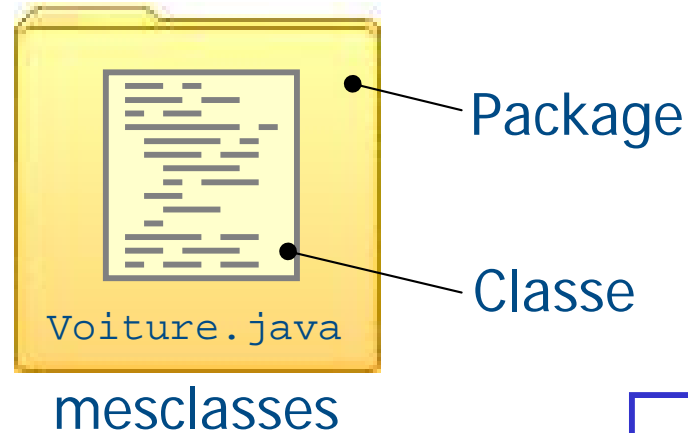


Écriture très lourde préférer la solution avec le mot clé *import*


Les packages : leur « existence » physique

- A chaque classe Java correspond un fichier
- A chaque package (sous-package) correspond un répertoire

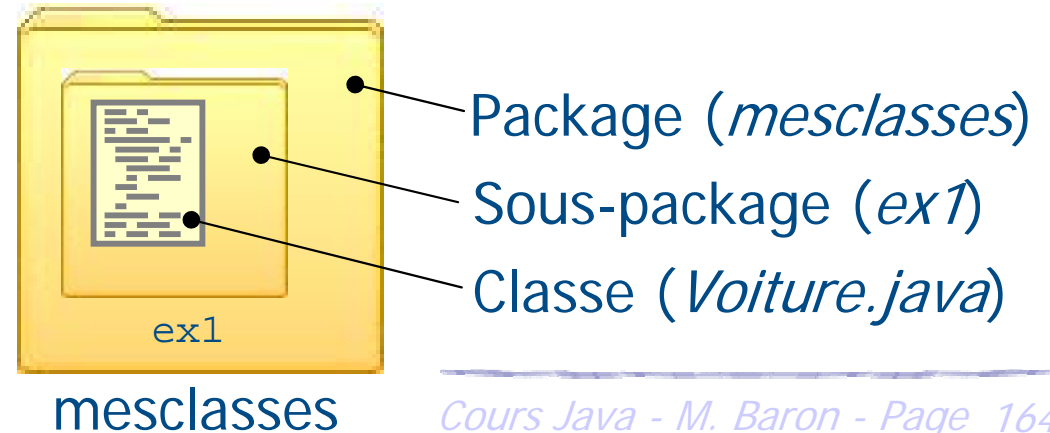
Exemple : *mesclasses.Voiture*



- Un package peut contenir
 - Des classes ou des interfaces
 - Un autre package (sous-package)

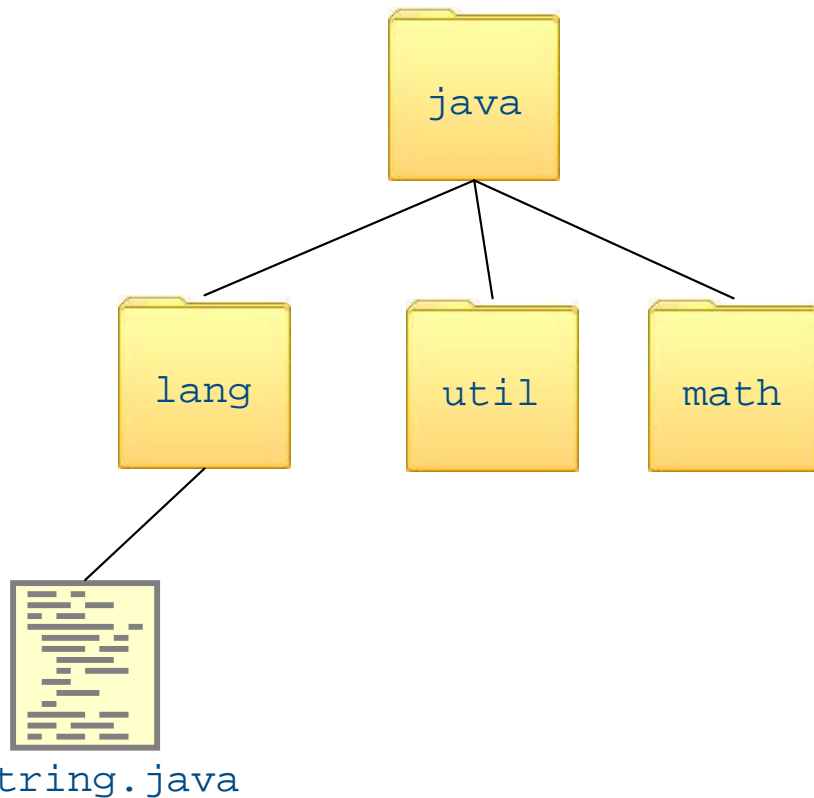
 Le nom des packages est toujours écrit en minuscules

Exemple : *mesclasses.ex1.Voiture*



Les packages : hiérarchie de packages

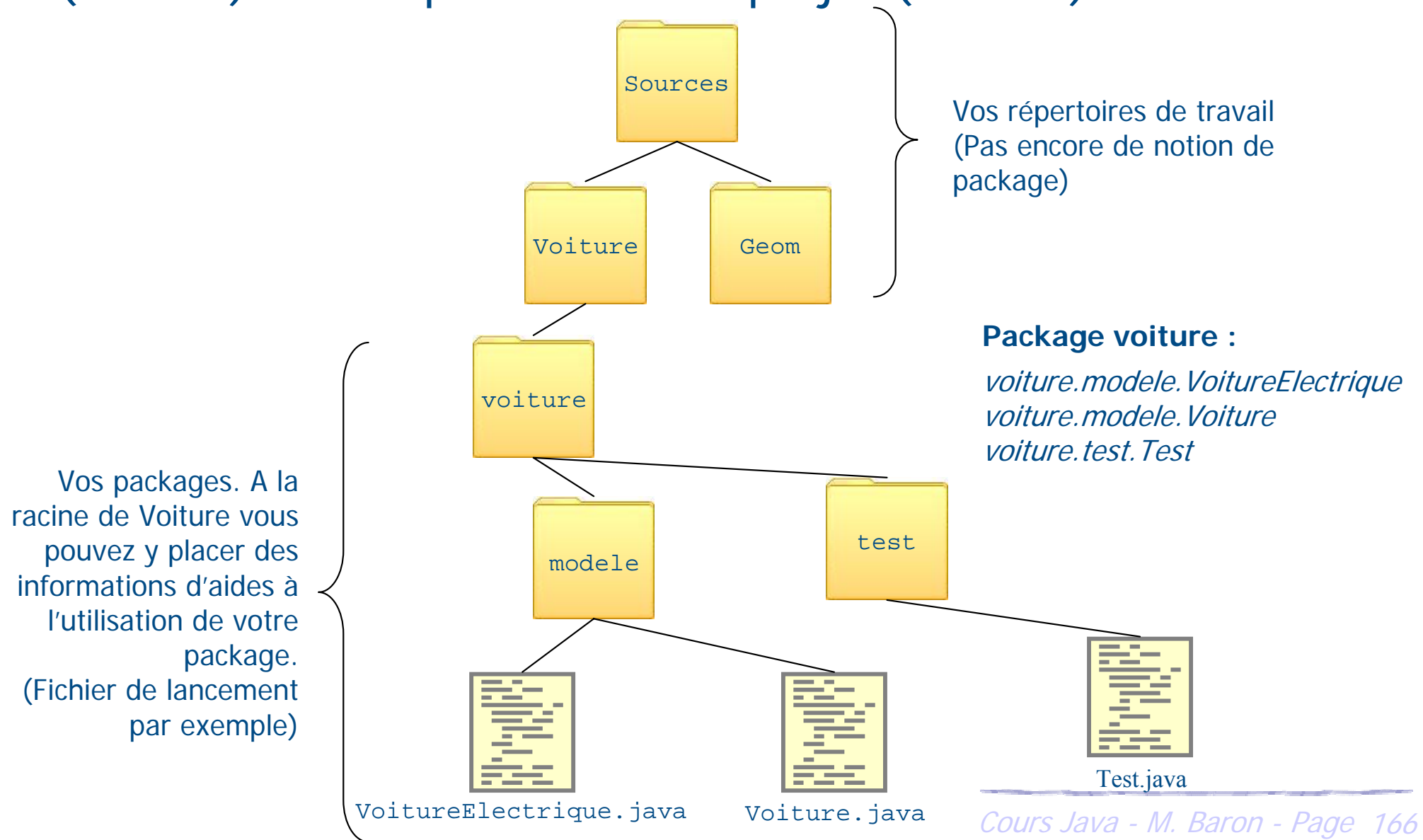
- A une hiérarchie de packages correspond une hiérarchie de répertoires dont les noms coïncident avec les composants des noms de package
- Exemple : la classe *String*



- Bibliothèque pure Java
- Les sources (*.java) se trouvent dans le répertoire *src* du répertoire Java
- Les bytecodes (*.class) se trouvent dans l'archive *rt.jar* du répertoire Java

Les packages : création et conseils

- Quand vous créer un projet nommez le package de plus haut (*voiture*) niveau par le nom du projet (*Voiture*)



Les packages : création et conseils

- Pour spécifier à une classe qu'elle appartient à un package utiliser le mot clé **package**

```
package voiture.modele;
public class VoitureElectrique {
    ...
}
```

```
package voiture.modele;
public class Voiture {
    ...
}
```

```
package voiture.test;
import voiture.modele.VoitureElectrique;
import voiture.modele.Voiture;
import ...

public class Test1 {
    public static void main(String[] argv) {
        ...
    }
}
```

Le mot clé package est toujours placé en première instruction d'une classe



Ne confondez pas héritage et package. Pas la même chose. *VoitureElectrique* est dans le même package que *Voiture*



Les packages : compilation et exécution

- Être placé dans la racine du répertoire Voiture



- La compilation doit prendre en compte les chemins des packages

```
javac voiture\modele\*.java voiture\test\*.java
```

- L'exécution se fait en indiquant la classe principale avec son chemin

```
java voiture.test.Test
```

La séparation entre package, sous-packages et classes se fait à l'aide de point « . » et non de anti-slash « \ »



Les packages : visibilité

- L'instruction `import nomPackage.*` ne concerne que les classes du package indiqué.
Elle ne s'applique pas aux classes des sous-packages

Packages différents

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

public class Essai {
    ...

    public Essai() {
        Date myDate = new Date(...);
        ZipFile myZip = new ZipFile(...);
        ...
    }
    ...
}
```

Essai utilise les classes *Date*
du package *java.util* et
ZipFile du package
java.util.zip

Javadoc et les commentaires

- Deux types de commentaires
 - Commentaires de traitements : précision sur le code lui-même
 - Commentaires de documentation (outil **javadoc** de la JDK : génération automatique de pages html)
- Classes, constructeurs, méthodes et champs
 - Compris entre `/**` et `*/`
 - Première ligne : uniquement `/**`
 - Suivantes : un espace suivi d'une étoile
 - Dernière ligne : uniquement `*/` précédé d'un espace

```
/**  
 * Description de la méthode  
 * Autres caractéristiques  
 */  
    public Voiture(...) {  
        ...  
    }  
}
```

Ajouter du sens et des précisions à vos codes. Expliquer n'est pas traduire!!



Javadoc et les commentaires

➤ Javadoc et intérêts

- **Javadoc** est aux classes ce que les pages de manuel (**man**) sont à **Unix** ou ce que **Windows Help** est aux applications **MS Windows**
- Rédaction de la documentation technique des classes au fur et à mesure du développement de ces mêmes classes puis génération finale du html

➤ Utilisation

- L'entité documentée est précédée par son commentaire
- Suivre la description des méthodes, classes, ...
- Utilisation de tags définis par **javadoc** permettant de typer certaines informations (utilisation possible de balise html)

@author		Nom du ou des auteurs
@version		Identifiant de version
@param		Nom et signification de l'argument (méthodes uniquement)
@since		Version du JDK où c'est apparu (utilisé par SUN)
@return		Valeur de retour
@throws		Classe de l'exception et conditions de lancement
@deprecated		Provoque les avertissements de désapprobation
@see		Référence croisée

Javadoc et les commentaires

➤ Exemple : source de la classe *Object*

```

package java.lang;
/**
 * Class Object is the root of the class hierarchy.
 * Every class has Object as a superclass. All objects,
 * including arrays, implement the methods of this class.
 *
 * @author unascribed
 * @version 1.58, 12/03/01
 * @see java.lang.Class
 * @since JDK1.0
 */
public class Object {
    /**
     * Returns the runtime class of an object. That Class
     * object is the object that is locked by static synchronized
     * methods of the represented class.
     *
     * @return the object of type Class that represents the
     *         runtime class of the object.
     */
    public final native Class getClass();
    ...

```

➤ Génération du code html à partir de l'outil **javadoc**



Pour obtenir les informations de javadoc
javadoc -help

javadoc [options] nomDesClassesJava.java

Javadoc et les commentaires

➤ Exemple : aperçu html de la description de la classe *Object* générée avec javadoc

java.lang
Class Object

java.lang.Object

public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since: JDK1.0
See Also: [Class](#)

Constructor Summary

[Object](#) ()

Method Summary

protected Object	clone ()	Creates and returns a copy of this object.
boolean	equals (Object obj)	Indicates whether some other object is "equal to" this one.
protected void	finalize ()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	getClass ()	Returns the runtime class of an object.
int	hashCode ()	

Method Detail

getClass

```
public final Class getClass ()
```

Returns the runtime class of an object. That `Class` object is the object that is locked by `static synchronized` methods of the represented class.

Returns:

the object of type `Class` that represents the runtime class of the object.

hashCode

```
public int hashCode ()
```

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by `java.util.Hashtable`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an

➤ Jar et intérêts

- L'archiveur **jar** est l'outil standard pour construire les archives qui ont le même objectif que les bibliothèques de programmes utilisées par certains langages de programmation (lib par exemple)

```
java -verbose HelloWorld
```

```
Console [C:\Program Files\Java\j2re1.4.1_01\bin\java.exe (02/09/07 13:26)]
[Loaded java.nio.Buffer from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded sun.misc.AtomicLong from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded sun.misc.AtomicLongCSImpl from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Boolean from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Character from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Number from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Float from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
```

Montre les archives utilisées pour exécuter le programme **HelloWorld**

➤ Utilisation pour la création

- Utilisation de l'outil **jar**
- Pour créer un fichier **.jar** contenant les fichiers du répertoire courant

```
jar cvf hello.jar .
```

Création Verbose Nom archive

Le **.** indique le répertoire courant

Jar

➤ Utilisation pour la création (suite)

- Utilisation d'un fichier manifeste (*MANIFEST.MF*) pour préciser un ensemble d'attributs pour exécuter
- L'attribut *Main-class* par exemple permet de connaître la classe principale à exécuter

```
Manifest-Version: 1.0  
Created-By: 1.4.1_01 (Sun Microsystems Inc.)  
Main-class: HelloWorld
```



MANIFEST.MF


- Création du jar avec un fichier manifeste :

```
jar cvfm hello.jar MANIFESTE.MF .
```

➤ Utilisation pour l'exécution

```
java -jar hello.jar
```

Cette option permet d'exécuter à partir d'une archive du code java



La classe *HelloWorld* est chargée par l'intermédiaire du fichier MANIFESTE.MF

Exception

➤ Définition

- Une exception est un signal indiquant que quelque chose d'exceptionnelle (comme une erreur) s'est produit.
- Elle interrompt le flot d'exécution normal du programme

➤ A quoi ça sert

- Gérer les erreurs est indispensable : mauvaise gestion peut avoir des conséquences catastrophiques (Ariane 5)
- Mécanisme simple et lisible
 - Regroupement du code réservé au traitement des erreurs
 - Possibilité de « récupérer » une erreur à plusieurs niveaux d'une application (propagation dans la pile des appels de méthodes)

➤ Vocabulaire

- Lancer ou déclencher (**throw**) une exception consiste à signaler les erreurs
- Capturer ou attraper (**catch**) une exception permet de traiter les erreurs

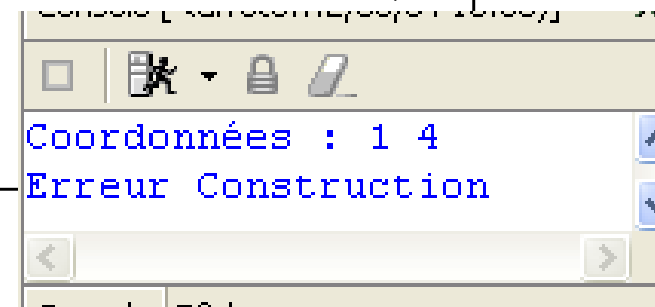
Exception

► Exemple : lancer et capturer une exception

```
public class Point {  
    ... // Déclaration des attributs  
  
    ... // Autre méthodes et constructeurs  
  
    public Point(int x, int y) throws ErrConst {  
        if ((x < 0) || (y < 0)) throw new ErrConst();  
        this.x = x ; this.y = y;  
    }  
  
    public void affiche() {  
        System.out.println("Coordonnées : " + x + " " + y);  
    }  
}
```

```
public class Test {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(1,4);  
            a.affiche();  
            a = new Point(-2, 4);  
            a.affiche();  
        } catch (ErrConst e) {  
            System.out.println("Erreur Construction");  
            System.exit(-1);  
        }  
    }  
}
```

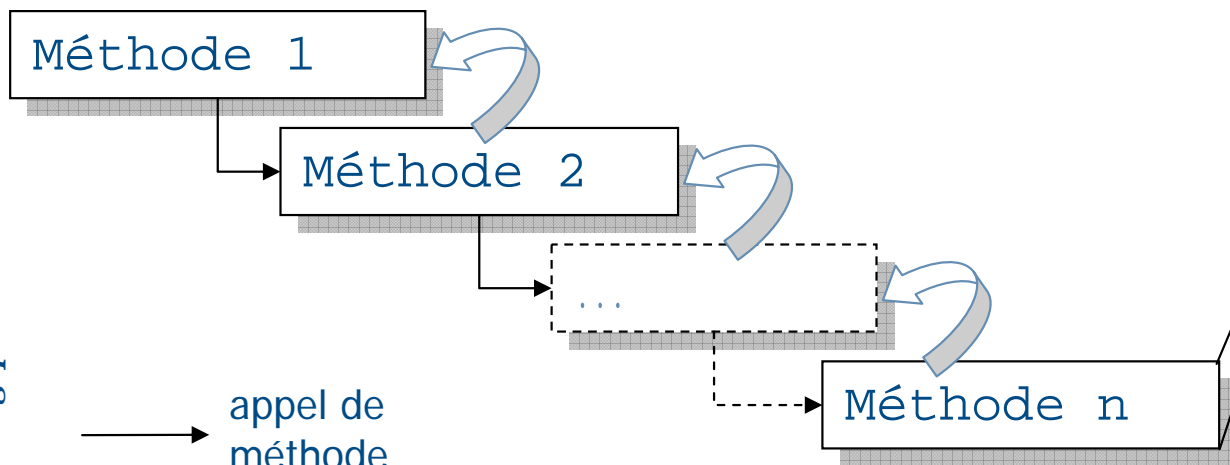
La classe *ErrConst* n'est pas encore définie. A voir plus tard



Exception : mécanisme

➤ Explication

- Lorsqu'une situation exceptionnelle est rencontrée, une exception est lancée
- Si elle n'est pas traitée, elle est transmise au bloc englobant, ..., jusqu'à ce qu'elle soit traitée ou parvienne en haut de la pile d'appel. Elle stoppe alors l'application

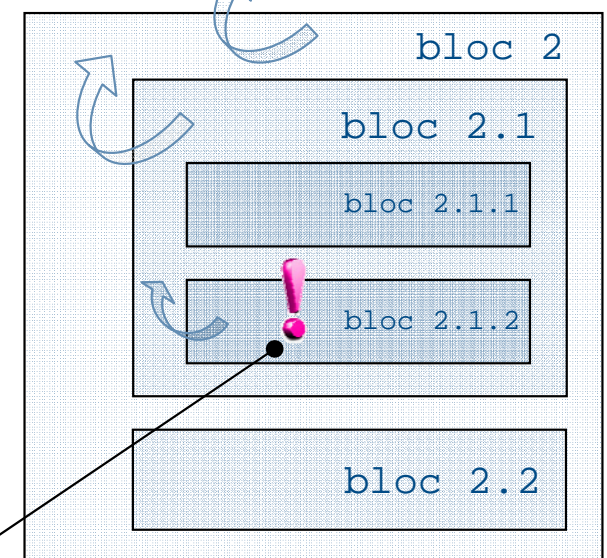


→ appel de méthode

↪ transmission d'exception

Une exception vient de se produire

Méthode n



Exception : lancer ou déclencher

- Une méthode déclare qu'elle peut lancer une exception par le mot clé **throws**

```
public Point(int x, int y) throws ErrConst {  
    ...  
}
```

Permet au constructeur *Point* de lancer une exception *ErrConst*

- Soit la méthode lance une exception, en créant une nouvelle valeur (un objet) d'exception en utilisant le mot clé **throw**

```
public Point(int x, int y) throws ErrConst {  
    if ((x < 0) || (y < 0)) throw new ErrConst();  
    this.x = x ; this.y = y;  
}
```

Création d'une nouvelle valeur d'exception

- Soit la méthode appelle du code qui lance une exception

```
public Point(int x, int y) throws ErrConst {  
    checkXYValue(x,y);  
    this.x = x ; this.y = y;  
}
```

```
private void checkXYValue(in x, int y)  
throws ErrConst {  
    if ((x < 0) || (y < 0))  
        throw new ErrConst();  
}
```

Exception : capturer ou attraper

- On parle ici de gestionnaire d'exception. Il s'agit de traiter par des actions la situation exceptionnelle
- On délimite un ensemble d'instructions susceptibles de déclencher une exception par des blocs **try {...}**

```
try {
    Point a = new Point(1,4);
    a.affiche();
    a = new Point(-2, 4);
    a.affiche();
}
```

Méthodes à risques. Elles sont « surveillées »

- La gestion des risques est obtenue par des blocs **catch (TypeException e) {...}**

```
} catch (ErrConst e) {
    System.out.println("Erreur Construction");
    System.exit(-1);
}
```

- Ces blocs permettent de capturer les exceptions dont le type est spécifié et d'exécuter des actions adéquates

Exception : capturer ou attraper

➤ Compréhension du mécanisme de capture

```
public class Test {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(1,4);  
            a.affiche();  
            a = new Point(-2, 4);  
            a.affiche();  
        } catch (ErrConst e) {  
            System.out.println("Erreur Construction");  
            System.exit(-1);  
        }  
        ...  
    }  
}
```

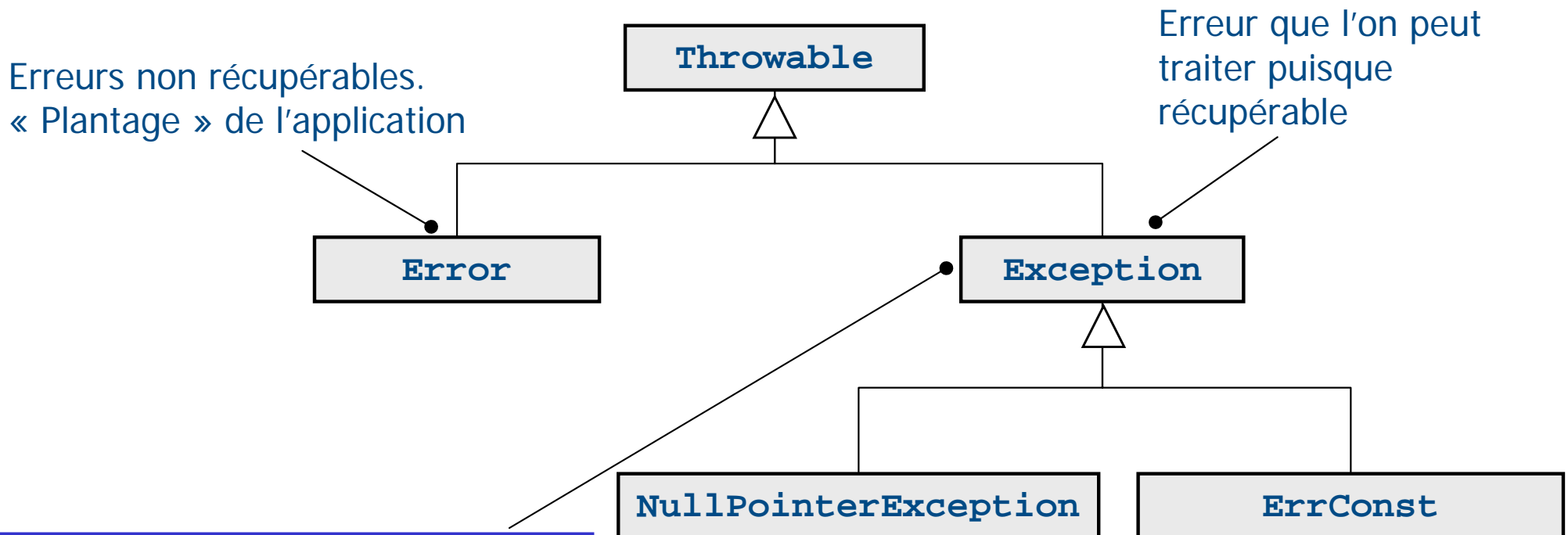
L'erreur exceptionnelle est traitée par le bloc **catch**

Puis, il y a poursuite de l'exécution en dehors du bloc **try catch**

Remarque : si erreur le programme s'arrête (System.exit(-1))

Exception : modélisation

- Les exceptions en Java sont considérées comme des objets
- Toute exception doit être une instance d'une sous-classe de la classe *java.lang.Throwable*

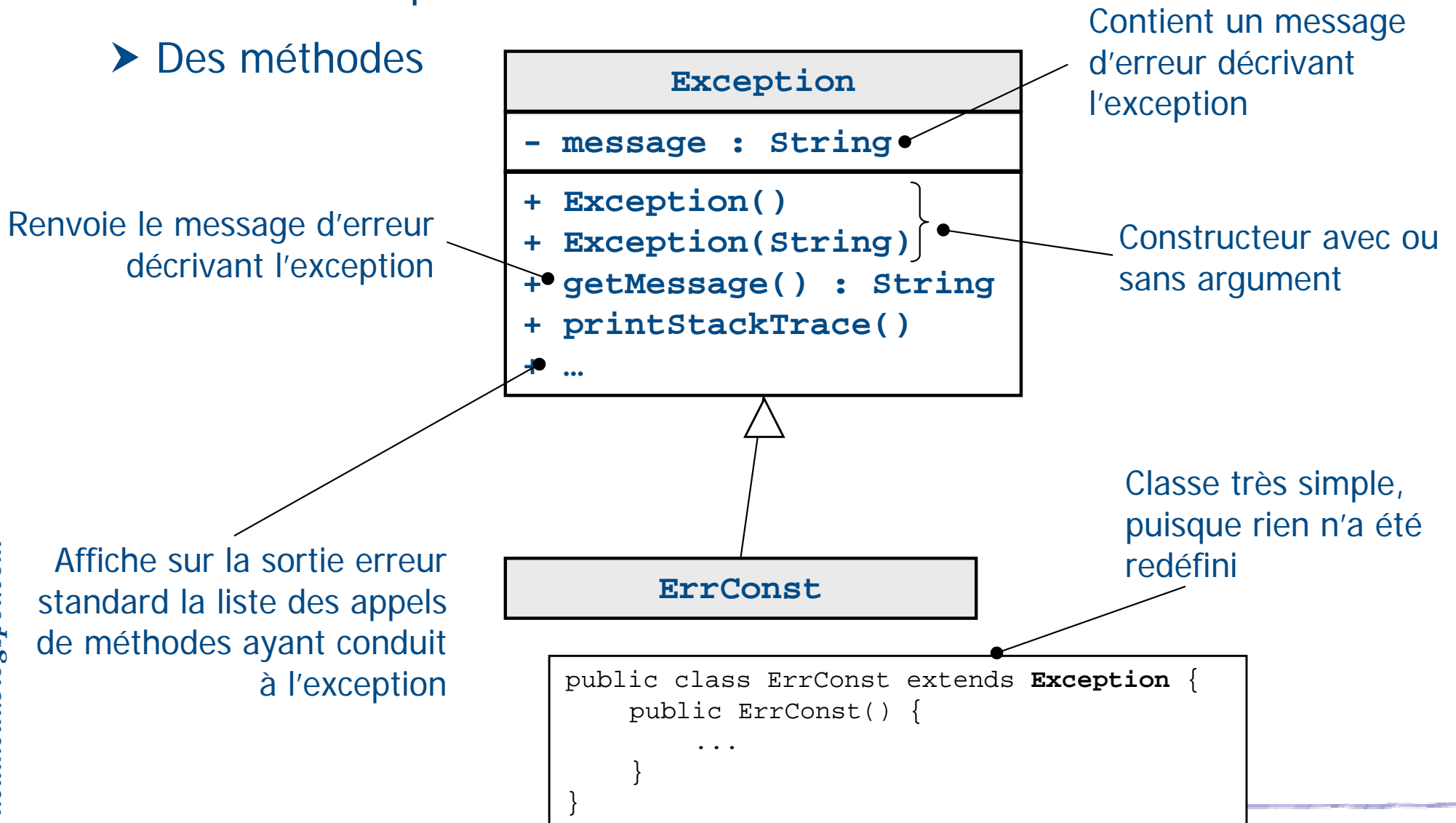


Pour définir de nouveaux types d'exception, on sous-classera la classe *Exception*



Exception : modélisation

- Les exceptions sont des objets nous pouvons donc définir
 - Des attributs particuliers
 - Des méthodes



Exception : modélisation

➤ Exemple : utilisation de l'objet *ErrConst*

Erreur de type *ErrConst*
qui hérite de Exception

```
public class Test {  
    public static void main(String[] argv) {  
        try {  
            ...  
        } catch (ErrConst e) {  
            System.out.println("Erreur Construction");  
  
            System.out.println(e.getMessage());  
  
            e.printStackTrace();  
  
            System.exit(-1);  
        }  
    }  
}
```

Affichage de l'erreur

Affichage de la liste des
méthodes

```
Console [ <arrêté> C:\Program F...n\javaw.exe (02/08/04 22:01) x  
Coordonnées : 1 4  
Erreur Construction  
ErrConst  
    at Point.<init> (Point.java:18)  
    at Test.main (Test.java:20)
```


Exception : attraper les tous ...

- Il est possible de capturer plus d'une exception. Un bloc **try** et plusieurs blocs **catch**

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) < 0)) throw new ErrDepl();
        x += dx ; y +=dy;
    }

    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst();
        this.x = x ; this.y = y;
    }
}
```

Définition d'une nouvelle méthode qui lance une exception

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Bloc dans lequel on souhaite détecter
                les exceptions ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Erreur Déplacement");
            System.exit(-1);
        }
    }
}
```

Attrape la nouvelle exception de type *ErrDepl*

Exception : attrapez les tous ...

- Toute méthode susceptible de lever une exception doit
 - Soit l'attraper (bloc *try catch*)
 - Soit déclarer explicitement qu'elle peut lancer une exception (mot clé *throws*)
- Les exceptions déclarées dans la clause *throws* d'une méthode sont ...

Les exceptions levées dans la méthode (*Point*) et non attrapées par celle-ci

```
public Point(int x, int y) throws ErrConst {
    if ((x < 0) || (y < 0)) throw new ErrConst();
    this.x = x ; this.y = y;
}
```

Les exceptions levées dans des méthodes (*checkXYValue*) appelées par la méthode (*Point*) et non attrapées par celle-ci

```
public Point(int x, int y) throws ErrConst {
    checkXYValue(x,y);
    this.x = x ; this.y = y;
}
```

```
private void checkXYValue(in x, int y) throws ErrConst {
    if ((x < 0) || (y < 0))
        throw new ErrConst();
}
```

Exception : attrapez les tous ...

- Il faut s'assurer que les exceptions soient sous contrôle

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) < 0)) throw new ErrDepl();
        x += dx ; y +=dy;
    }

    public void transformer() {
        ...
        this.deplace(...);
    }
}
```

```
public class ErrDepl extends Exception {
    public ErrDepl() {
        ...
    }
}
```

```
C:\ Défilement C:\WINDOWS\System32\cmd.exe

D:\Documents Mickey\My eBooks\Java\Package>javac Test.java
.\Point.java:28: unreported exception ErrDepl; must be caught or declared to be
thrown
        this.deplace(12,43);
        ^
1 error

D:\Documents Mickey\My eBooks\Java\Package>
```

Ne pas oublier de traiter une exception sinon le compilateur ne vous loupe pas!!!!

Exception : attrapez les tous ...

► Pour garantir une bonne compilation deux solutions

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) < 0)) throw new ErrDepl();
        x += dx ; y +=dy;
    }

    public void transformer() {
        ...
        this.deplace(...);
    }
}
```

Soit en ajoutant explicitement l'instruction **throws** à la méthode *transformer* de façon à rediriger l'erreur

```
public void transformer()
    throws ErrDepl {
    ...
    this.deplace(...);
}
```

Soit en entourant d'un bloc **try ... catch** la méthode qui peut poser problème

```
public void transformer() {
    try {
        ...
        this.deplace(...);
    } catch (ErrDepl e) {
        e.printStackTrace();
    }
}
```

Exception : transmission d'information

- Possibilité d'enrichir la classe *ErrConst* en ajoutant des attributs et des méthodes de façon à communiquer

```
public class Point {  
    public Point(int x, int y) throws ErrConst {  
        if ((x < 0) || (y < 0)) throw new ErrConst(x,y);  
        this.x = x ; this.y = y;  
    }  
    ...  
}
```

```
public class ErrConst extends Exception {  
    private int abs, ord;  
  
    public ErrConst(int x, int y) {  
        this.abs = x;  
        this.ord = y;  
    }  
  
    public int getAbs() { return this.abs; }  
    public int getOrd() { return this.ord; }  
}
```

```
public class Test {  
    public static void main(String[] argv) {  
        try {  
            ...  
            a = new Point(-2, 4);  
        } catch (ErrConst e) {  
            System.out.println("Erreur Construction point");  
            System.out.println("Coordonnées souhaitées : "  
                + e.getAbs() + " " + e.getOrd());  
            System.exit(-1);  
        } ...  
    }  
}
```

ErrConst
- abs, ord : int
+ ErrConst(x,y)
+ getAbs : int
+ getOrd : int

ErrConst permet de connaître les valeurs qui ont fait échouer la construction de *Point*

Exception : finally

- Bloc *finally* : c'est une instruction optionnelle qui peut servir de « nettoyage »
- Elle est exécutée quel que soit le résultat du bloc *try* (c'est-à-dire qu'il ait déclenché ou non une exception)
- Permet de spécifier du code dont l'exécution est garantie quoi qu'il arrive
- L'intérêt est double
 - Rassembler dans un seul bloc un ensemble d'instructions qui autrement auraient du être dupliquées
 - Effectuer des traitements après le bloc *try*, même si une exception a été levée et non attrapée par les blocs *catch*

Exception : finally

➤ Exemple : terminer correctement avec *finally*

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Bloc dans lequel on souhaite détecter
                les exceptions ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
            System.out.println("Fin du programme");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Erreur Déplacement");
            System.out.println("Fin du programme");
            System.exit(-1);
        }
    }
}
```

Ces instructions sont
rappelées plusieurs
fois

Au moyen du mot
clé *finally*, il est
possible de
factoriser

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Bloc dans lequel on souhaite détecter
                les exceptions ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
        } catch (ErrDepl e) {
            System.out.println("Erreur Déplacement");
        }
        finally {
            System.out.println("Fin du programme");
            System.exit(-1);
        }
    }
}
```

Exception : pour ou contre

➤ Exemple : gérer les erreurs sans les exceptions

```
erreurType lireFichier() {
    int codeErreur = 0;
    // Ouvrir le fichier
    if (isFileIsOpen()) {
        // Détermine la longueur du fichier
        if (getFileSize()) {
            // Vérification de l'allocation de la mémoire
            if (getEnoughMemory()) {
                // Lire le fichier en mémoire
                if (readFailed()) {
                    codeErreur = -1;
                }
            } else {
                codeErreur = -2;
            }
        } else {
            codeErreur = -3;
        }

        // Fermeture du fichier
        if (closeTheFileFailed()) {
            codeErreur = - 4;
        }
    } else {
        codeErreur = - 5;
    }
}
```

La gestion des erreurs devient très difficile

Difficile de gérer les retours de fonctions

Le code devient de plus en plus conséquent

Exception : pour ou contre

➤ Le mécanisme d'exception permet

- La concision
- La lisibilité

```
void lireFichier() {  
    try {  
        // Ouvrir le fichier  
        // Détermine la longueur du fichier  
        // Vérification de l'allocation de la mémoire  
        // Lire le fichier en mémoire  
        // Fermer le fichier  
    } catch (FileOpenFailed) {  
        ...  
    } catch (FileSizeFailed) {  
        ...  
    } catch (MemoryAllocFailed) {  
        ...  
    } catch (FileReadFailed) {  
        ...  
    } catch (FileCloseFailed) {  
        ...  
    }  
}
```

Préférer cette solution à la précédente. Programmation propre et professionnelle



Exception : les exceptions courantes

- Java fournit de nombreuses classes prédéfinies dérivées de la classe *Exception*
- Ces exceptions standards se classent en deux catégories
 - Les exceptions explicites (celles que nous avons étudiées), mentionnées par le mot clé ***throws***
 - Les exceptions implicites qui ne sont pas mentionnées par le mot clé ***throws***
- Liste de quelques exceptions
 - *ArithmeticException* (division par zéro)
 - *NullPointerException* (référence non construite)
 - *ClassCastException* (problème de cast)
 - *IndexOutOfBoundsException* (problème de dépassement d'index dans tableau)

Les flux

- Pour obtenir des données, un programme ouvre un flux de données sur une source de données (fichier, clavier, mémoire, etc.)
- De la même façon pour écrire des données dans un fichier, un programme ouvre un flux de données
- Java fournit un paquetage *java.io* qui permet de gérer les flux de données en entrée et en sortie, sous forme de caractères (exemple fichiers textes) ou sous forme binaire (octets, byte)

Les flux

- En Java, le nombre de classes intervenant dans la manipulation des flux est important (plus de 50)
- Java fournit quatre hiérarchies de classes pour gérer les flux de données
 - Pour les flux binaires
 - La classe *InputStream* et ses sous-classes pour lire des octets (*FileInputStream*)
 - La classe *OutputStream* et ses sous-classes pour écrire des octets (*FileOutputStream*)
 - Pour les flux de caractères
 - La classe *Reader* et ses sous-classes pour lire des caractères (*BufferedReader*, *FileReader*)
 - La classe *Writer* et ses sous-classes (*BufferedWriter*, *FileWriter*)

Les flux de caractères

► Exemple : écrire du texte dans un fichier

FileWriter hérite de *Writer* et permet de manipuler un flux texte associé à un fichier

```
public class TestIO {  
    public static void main(String[] argv) {  
        FileWriter myFile = new FileWriter("a_ecrire.txt");  
  
        myFile.write("Voilà ma première ligne dans un fichier");  
  
        myFile.close();  
    }  
}
```

Fermeture du flux
myFile vers le fichier
a_ecrire.txt

Ecriture d'une ligne de
texte dans le fichier
« a_ecrire.txt »

Les flux de caractères

► Exemple : lire l'entrée standard : enfin !!!

« Convertit » un objet de type *InputStream* en *Reader*

```
public class TestIO {
    public static void main(String[] argv) {
        System.out.print("Veuillez saisir votre nom :");

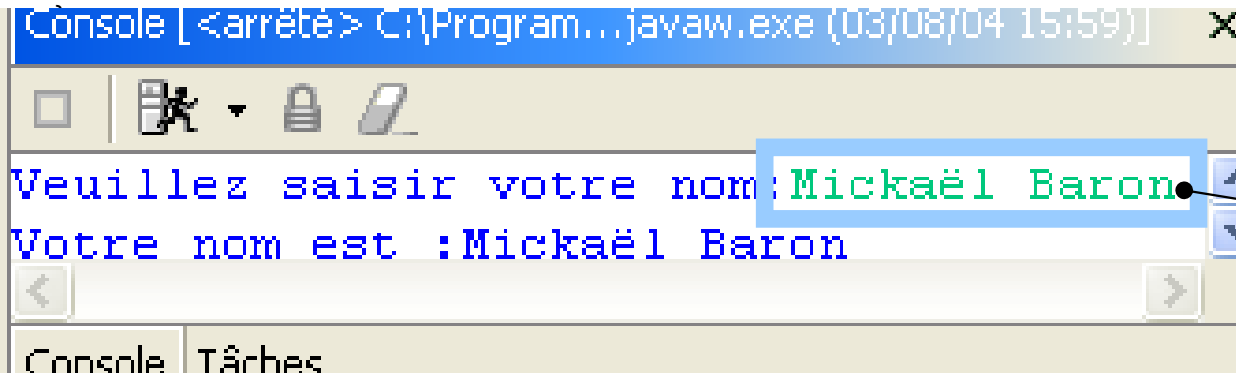
        String inputLine = " ";
        try {
            BufferedReader is = new BufferedReader(new InputStreamReader(System.in));

            inputLine = is.readLine();

            is.close();
        } catch (Exception e) {
            System.out.println("Intercepté : " + e);
        }

        if (inputLine != null)
            System.out.println("Votre nom est :" + inputLine);
    }
}
```

Lit la ligne jusqu'au prochain retour chariot



Chaîne saisie

Les flux de caractères

➤ Exemple : copie de fichier en utilisant les caractères

FileReader et *FileWriter* héritent de *Reader* et *Writer* et permettent de manipuler un flux texte associé à un fichier texte

```
public class TestIO {
    public static void main(String[] argv) {
        • FileReader in = new FileReader("a_lire.txt");
        FileWriter out = new FileWriter("a_ecrire.txt");
        int c;

        while ((c = in.read()) != -1) {
            • out.write(c);
        }

        in.close();
        out.close();
    }
}
```

Transfert de données jusqu'à ce que *in* ne fournisse plus rien

Fermeture des flux et par conséquent des fichiers respectifs

Les flux binaires

➤ Exemple : copie de fichier en utilisant les binaires

Même raisonnement
que pour les
caractères sauf ...


```
public class TestIO {
    public static void main(String[] argv) {
        FileInputStream in = new FileInputStream("a_lire.txt");
        FileOutputStream out = new FileOutputStream("a_ecrire.txt");
        int c;

        while ((c = in.read()) != -1) {
            out.write(c);
        }

        in.close();
        out.close();
    }
}
```


La classe File

- Java dispose d'une classe *File* qui offre des fonctionnalités de gestion de fichiers
- La création d'un objet de type *File*



Attention : ne pas confondre la création de l'objet avec la création du fichier physique

```
File monFichier = new File("truc.dat");
```

File
- name : String
+ File(String nf)
+ createNewFile()
+ delete() : booléen
+ exists() : booléen
+ getName() : String
+ isFile() : booléen
+ ...

Création du fichier portant le nom de *name*

Vérifie si le fichier existe physiquement

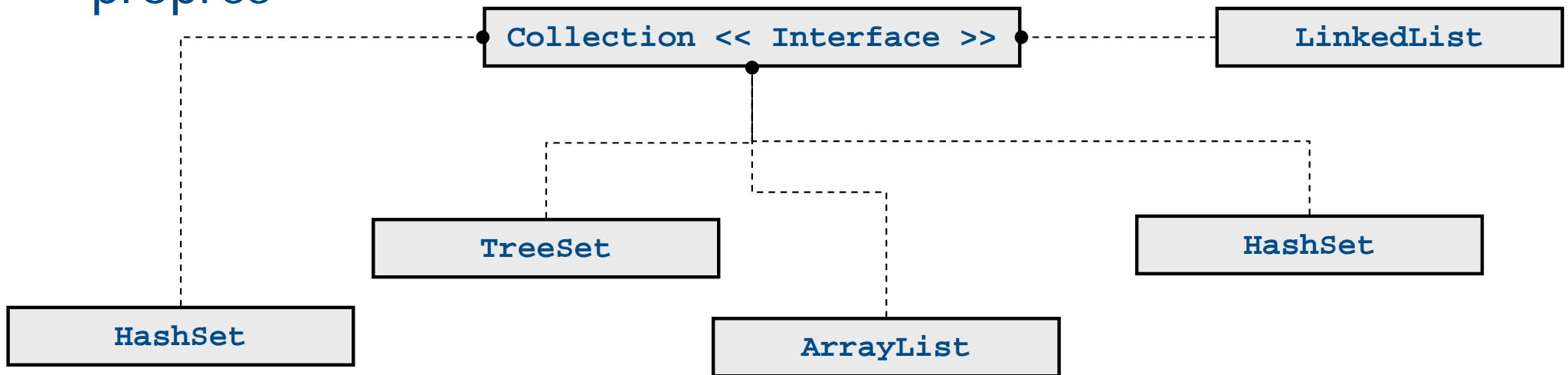
```
File monFichier = new File("c:\\toto.txt");  
if (monFichier.exists()) {  
    monFichier.delete();  
} else {  
    monFichier.createNewFile();  
}
```

Les collections

- Pour l'instant nous avons étudié le tableau pour structurer les données
 - Taille statique
 - Lent pour la recherche d'éléments particuliers
 - Impossibilité d'utiliser un pattern de déplacement dans les éléments
- Java propose depuis la version 2 des classes permettant de manipuler les principales structures de données
 - Les tableaux dynamiques implémentées par *ArrayList* et *Vector*
 - Les listes implémentées par *LinkedList*
 - Les ensembles implémentées par *HashSet* et *TreeSet*

Les collections

- Ces classes implémentent toutes indirectement une même interface *Collection* qu'elles complètent de fonctionnalités propres



- Depuis la version 5 de Java, possibilité d'utiliser les génériques pour *typer* le contenu des Collections
 - Avant : *Voiture maVoiture = (Voiture)myList.get(2)*
 - Maintenant : *Voiture maVoiture = myList.get(2)*

Plus de problème de conversion explicite



Les collections

- L'interface Collection permet
 - *La généricité et références* : possibilité de stocker des éléments de type quelconque, pour peu qu'il s'agisse d'objets. Un nouvel élément introduit dans une collection Java est une référence à l'objet et non une copie
 - *Les itérateurs* : ils permettent de parcourir un par un les différents éléments d'une collection
 - Efficacité des opérations sur des collections
 - Opérations communes à toutes les collections : les collections que nous allons étudier implémentent toutes au minimum l'interface *Collection*, de sorte qu'elles disposent de fonctionnalités communes



Les collections : les génériques Java

- Avec la version 5 de Java possibilité d'exploiter les génériques dans les collections et pour d'autres aspects du langage
- Une syntaxe particulière a été rajoutée de manière à prendre en considération les génériques
 - `< ? >` : signale qu'il faut préciser le type de la classe
 - `< ? , ? >` : signale qu'il faut préciser deux types
- Avec les génériques il va être possible de fixer lors de la construction de la collection le type du contenu stocké dans les collections
- Avantages
 - Toutes les méthodes accesseurs et modifieurs qui manipulent les éléments d'une collection sont *signés* suivant le type défini à la construction de la collection
 - Vérification des types pendant le développement (avant problème de *CastClassException*)

Les collections : Iterator

- Les itérateurs permettent de parcourir les éléments d'une collection sans connaissance précise du type de la collection :

Polymorphisme

- Il existe deux familles d'itérateurs

- *monodirectionnels*

Le parcours de la collection se fait d'un début vers une fin; l'accès à un élément ne se fait qu'une seule fois

- *bidirectionnels*

Le parcours de la collection peut se faire dans les deux sens ; possibilité d'avancer et reculer à sa guise dans la collection

La notion d'Iterator fait partie de l'ensemble des Design Patterns



Les collections : Iterator

➤ Itérateur monodirectionnel : interface *Iterator*

- Par défaut, toutes collections possèdent un attribut de type *Iterator*

Vérifie s'il y a un prochain

Supprime le dernier objet renvoyé par *next()*

```
Iterator << Interface >>  
+ hasNext() : booléen  
+ next() : < ? >  
+ remove()
```

Permet d'obtenir l'objet courant et passe au suivant

c est une collection et on récupère son *Iterator*

On vérifie s'il y a possibilité de récupérer un objet

```
Iterator iter = c.iterator();  
  
while (iter.hasNext()) {  
    ??? o = iter.next();  
    ...  
}
```

On récupère l'objet courant puis on passe au suivant

Les collections : Iterator

➤ Itérateur bidirectionnel : interface *ListIterator*

- Cela concerne les listes et les tableaux dynamiques
- Permet d'ajouter et de supprimer des objets

Iterator << Interface >>

Vérifie s'il y a un précédent

ListIterator << Interface >>

```
+ previous() : < ? >  
+ hasPrevious() : booléen  
+ add(< ? >)  
+ set(< ? >)  
+ ...
```

Ajoute ou modifie à la position courante un élément de la collection

On vérifie s'il y a possibilité de récupérer un objet précédemment

```
Iterator iter = c.listIterator();  
while (iter.hasPrevious()) {  
    ??? o = iter.previous();  
    ...  
}
```

Récupère l'objet précédemment puis on passe au précédent

c est une collection et on récupère son *ListIterator*
Initialise en début de liste

Les collections : LinkedList

- Cette classe permet de manipuler des listes dites doublement chaînées
- A chaque élément de collection, on associe implicitement deux informations qui sont les références à l'élément précédent et suivant



```
LinkedList<String> l1 = new LinkedList<String>();  
ListIterator iter = l1.listIterator();  
  
iter.add("Bonjour");  
iter.add("Coucou");  
  
while(iter.hasPrevious()) {  
    String o = iter.previous();  
    System.out.println(o);  
}
```

Plus rien après ses éléments, on fait un retour en arrière

Ajout des éléments au travers de l'itérateur
L'utilisation de la *LinkedList* est transparente

Les collections : LinkedList

- Possibilité d'utiliser les collections (ici *LinkedList* est un exemple) sans les itérateurs mais moins performant !!!

```
LinkedList<String> l1 = new LinkedList<String>();  
  
l1.add("Bonjour");  
l1.add("Coucou");  
  
for (int i = 0; i < l1.size(); i++) {  
    String o = l1.get(i);  
    System.out.println(o);  
}
```

L'utilisation de la *LinkedList* n'est pas transparente. Connaissance obligatoire de ces méthodes

Utilisation de la méthode *add* de la classe *LinkedList*

Ne pas modifier la collection (*add* de *LinkedList*) pendant qu'on utilise l'itérateur (*next()*)



Les collections : ArrayList

- La classe *ArrayList* est une encapsulation du tableau avec la possibilité de le rendre dynamique en taille
- Possibilité d'utiliser des *ListIterator* mais on préfère son utilisation à un élément de rang donné

```
ArrayList<Object> myArrayList = new ArrayList<Object>();  
  
myArrayList.add("Coucou");  
myArrayList.add(34);  
  
for (int i = 0; i < myArrayList.size(); i++) {  
    Object myObject = myArrayList.get(i);  
    if (myObject instanceof String) {  
        System.out.println("Chaîne:" + ((String)myObject));  
    }  
  
    if (my_object instanceof Integer) {  
        System.out.println("Integer:" + ((Integer)myObject));  
    }  
}
```

Préférer l'utilisation de
la classe *ArrayList* au
lieu de la classe *Vector*



Les collections : HashSet

- La classe *HashSet* permet de gérer les ensembles
- Deux éléments ne peuvent être identiques
- Prévoir deux choses dans vos classes
 - La redéfinition de la méthode *hashCode()* qui est utilisée pour ordonnancer les éléments d'un ensemble (calcul la table de hachage d'un objet)
 - La redéfinition de la méthode *equals(Object)* qui compare des objets de même classe pour connaître l'appartenance d'un élément à l'ensemble

Les collections : HashSet

► Exemple : gestion de points avec *HashSet*

```
public class TestHashSet {
    public static void main(String[] argv) {
        Point p1 = new Point(1,3); Point p2 = new Point(2,2);
        Point p3 = new Point(4,5); Point p4 = new Point(1,8);
        Point p[] = {p1, p2, p1, p3, p4, p3}

        HashSet<Point> ens = new HashSet<Point>();
        for (int i = 0; i < p.length; i++) {
            System.out.println("Le Point "); p[i].affiche();
            boolean ajoute = ens.add(p[i]);
            if (ajoute)
                System.out.println(" a été ajouté");
            else
                System.out.println("est déjà présent");
            System.out.print("Ensemble = "); affiche(ens);
        }
    }

    public static void affiche(HashSet ens) {
        Iterator iter = ens.iterator();
        while(iter.hasNext()) {
            Point p = iter.next();
            p.affiche();
        }
        System.out.println();
    }
}
```

Les collections : HashSet

➤ Exemple : gestion de points avec *HashSet*

```
public class Point {
    private int x,y;

    Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public int hashCode() {
        return x+y;
    }
    public boolean equals(Object pp) {
        Point p = (Point)pp;
        return ((this.x == p.x) &
            (this.y == p.y));
    }
    public void affiche() {
        System.out.print("[ " + x + " "
            + y + " ] ");
    }
}
```

Redéfinition des
méthodes *hashCode()*
et *equals(Object)*

```
Console [<arrêté> C:\Progr...vaw.exe (04/08/04 15:31)]
Le Point [1 3] a été ajouté
Ensemble = [1 3]
Le Point [2 2] a été ajouté
Ensemble = [2 2] [1 3]
Le Point [1 3] est déjà présent
Ensemble = [2 2] [1 3]
Le Point [4 5] a été ajouté
Ensemble = [2 2] [1 3] [4 5]
Le Point [1 8] a été ajouté
Ensemble = [2 2] [1 3] [1 8] [4 5]
Le Point [4 5] est déjà présent
Ensemble = [2 2] [1 3] [1 8] [4 5]
```



Programmation Orientée Objet application au langage Java

Bilan

Mickaël BARON - 2008 (Rév. Août 2009)
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

Bilan

- Ce que nous avons appris
 - Se familiariser avec la programmation orientée objet avec Java
 - Les concepts principaux de l'orientée objet (encapsulation, héritage et polymorphisme)
 - Aperçu des API les plus importantes de la plateforme Java
- Perspectives futures
 - Le langage Java est en constante évolution. Chaque nouvelle version apporte son lot de nouvelles fonctionnalités
 - Structurer les classes en appliquant les patterns de conception (Design Pattern)
 - Le langage Java est un point de départ pour concevoir des applications serveur avec Java Entreprise Edition