



Formations à l'informatique

Découvrez la différence ENI

# Les fondamentaux de la programmation Java (Java SE)

Support de cours

Cours T4ACF-1B1

[www.eni-service.fr](http://www.eni-service.fr)

# Les fondamentaux de la programmation Java (Java SE)

## Module 0

### A propos de ce cours

# Votre formateur

- Son nom
- Ses activités
- Ses domaines de compétence
- Ses qualifications et expériences pour ce cours

# Votre formation - Présentation

- Description
  - Dans cette formation, vous apprendrez la syntaxe du langage Java ainsi que la mise en œuvre de tous les concepts de la programmation orientée objet. Vous apprendrez également à utiliser les principales classes du JDK Java SE et à utiliser IDE Eclipse pour le développement et la mise au point de vos applications.
- Profil des stagiaires
  - Développeurs, analystes programmeurs, chefs de projets.
- Connaissances préalables
  - Avoir une expérience de programmation, quel que soit le langage d'origine ;
  - Avoir développé et livré une application en autonomie ou en équipe ;
  - Idéalement, avoir suivi la formation «T4ACF-0 - La conception orientée objet » ou posséder les connaissances et compétences équivalentes.
- Objectifs à atteindre
  - Utiliser l'IDE Eclipse pour vos projets Java ;
  - Ecrire, compiler, exécuter et déboguer des programmes Java ;
  - Appliquer les concepts de programmation orientée objet au langage Java ;
  - Utiliser la bibliothèque de classes Java ;
  - Gérer les erreurs et mettre en œuvre la journalisation de vos applications ;
  - Réaliser des tests unitaires à l'aide de JUnit.

# Votre formation - Programme

- Module 1 : Introduction
  - Historique de Java
  - Java et le Web
  - Principes et caractéristiques de Java
  - Le développement Java
- Module 2 : Utilisation de l'IDE Eclipse
  - Présentation de la structure d'Eclipse
  - Programmer avec Eclipse
- Module 3 : Les principes de base du langage
  - Les règles syntaxiques
  - Les opérateurs et expressions
  - Les variables et les constantes
  - Les types de données primitifs et les types wrappers
  - Les chaînes de caractères et la classe String
  - Création et utilisation de types de données énumérés : enum
  - La gestion des dates et du temps
  - Les tableaux
  - Conversion de types de données primitifs
  - Les structures de contrôle
  - Affichage sur la sortie standard avec `System.out.println()`
  - Affichage formaté sur la sortie standard avec la méthode `printf()`

# Votre formation – Programme (suite)

- Module 4 : La programmation orientée objet en Java
  - Les principes de la programmation orientée objet
  - Les concepts de programmation objet appliqués à Java
  - Relation entre les classes et les objets
  - Déclaration d'une classe
  - Création d'objets avec l'opérateur new et notion de référence
  - Déclaration des constructeurs et règles de mise en œuvre
  - Finalisation d'objet et le garbage collector
  - Déclaration et manipulation de membres (variables et méthodes) de classes (static) et d'instances
  - Les méthodes et le passage de paramètres par valeur ou référence
  - Bonnes pratiques pour la mise en œuvre des accesseurs
  - Mise en œuvre de l'héritage simple en Java
  - Effectuer des conversions d'objets
  - Les modificateurs d'accès et l'accès aux membres des classes
  - Mise en œuvre de la surcharge de méthodes
  - Simplification de la surcharge de méthodes par la réalisation de méthodes à arguments variables
  - Mise en œuvre de la redéfinition de méthodes
  - La classe Object et ses méthodes utilitaires
  - Utilisation des mots clés this, this() et super, super() et patterns de mises en œuvres
  - Tester le type d'un objet avec l'opérateur instanceof et pattern de mise en œuvre

# Votre formation – Programme (suite)

- Module 5 : Concepts avancés de programmation Java
  - Les packages
  - Les classes abstraites et les interfaces
  - Les expressions Lambdas (nouveau Java 8)
  - Principe de fonctionnement et traitement des erreurs avec les exceptions
- Module 6 : L'API Java
  - Documentation
  - Les collections
  - Les flux
- Module 7 : Les bibliothèques de journalisation Java
  - L'intérêt de la journalisation dans les applications logicielles
  - Les différentes approches Java
  - Implémentation d'un système de journalisation
- Module 8 : Mise en œuvre de tests unitaires avec JUnit
  - Nécessité des jeux de tests unitaires
  - Intégration de JUnit dans Eclipse IDE
  - Les cas de tests
  - Les suites de tests
  - Bonnes pratiques pour la conception des tests

# Votre formation – Ressources à votre disposition

- Le présent support de cours
- Le support de référence
  - Java 8, les fondamentaux du langage Java – Collection Ressources Informatiques – Editions ENI



# Tour de table – Présentez-vous

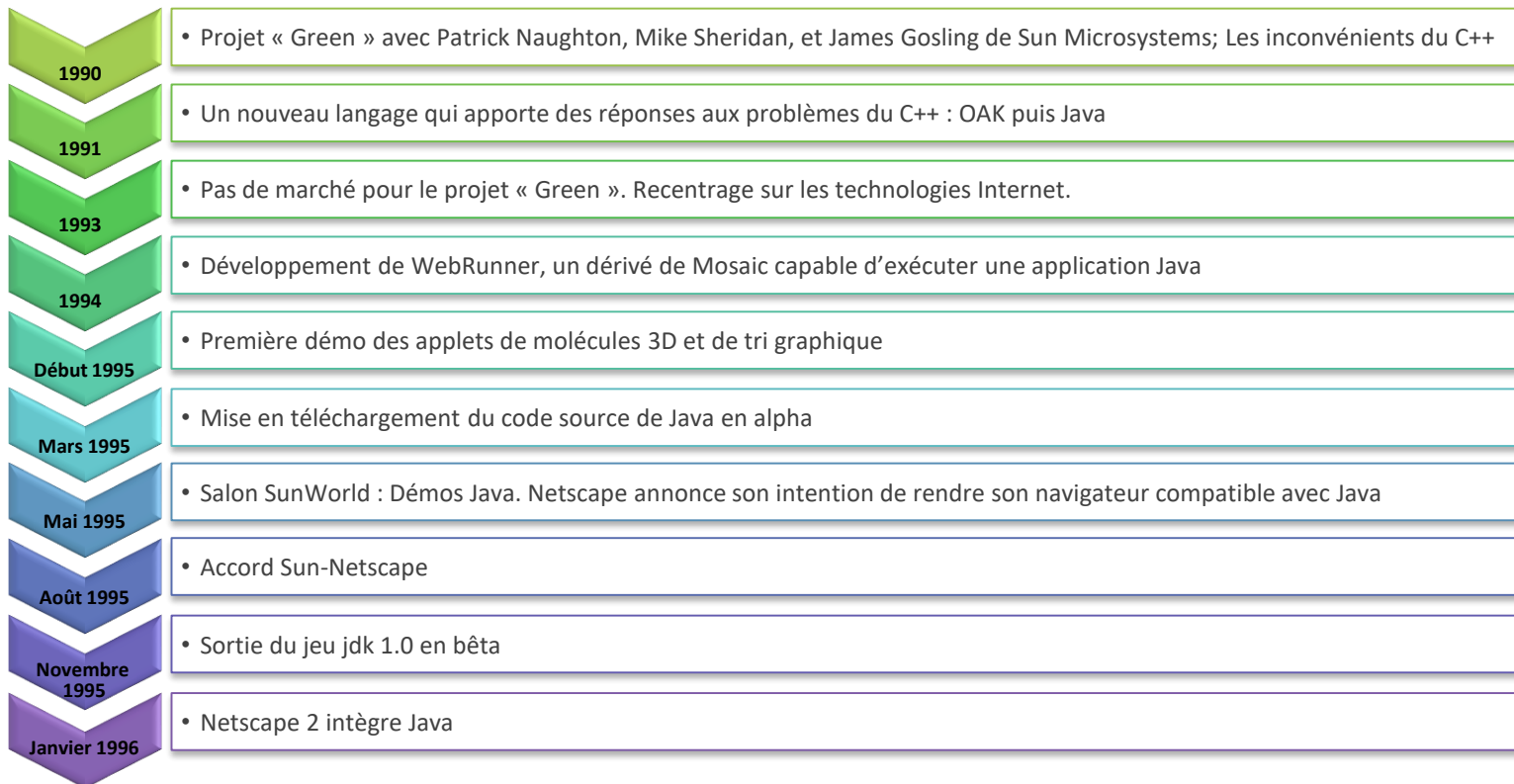
- Votre nom
- Votre société
- Votre métier
- Vos compétences dans des domaines en rapport avec cette formation
- Les objectifs et vos attentes vis-à-vis de cette formation

- Horaires de la formation
  - 9h00-12h30
  - 14h00-17h30
- Pauses
- **Merci d'éteindre vos téléphones portables**

# Les fondamentaux de la programmation Java (Java SE)

## Module 1 Introduction

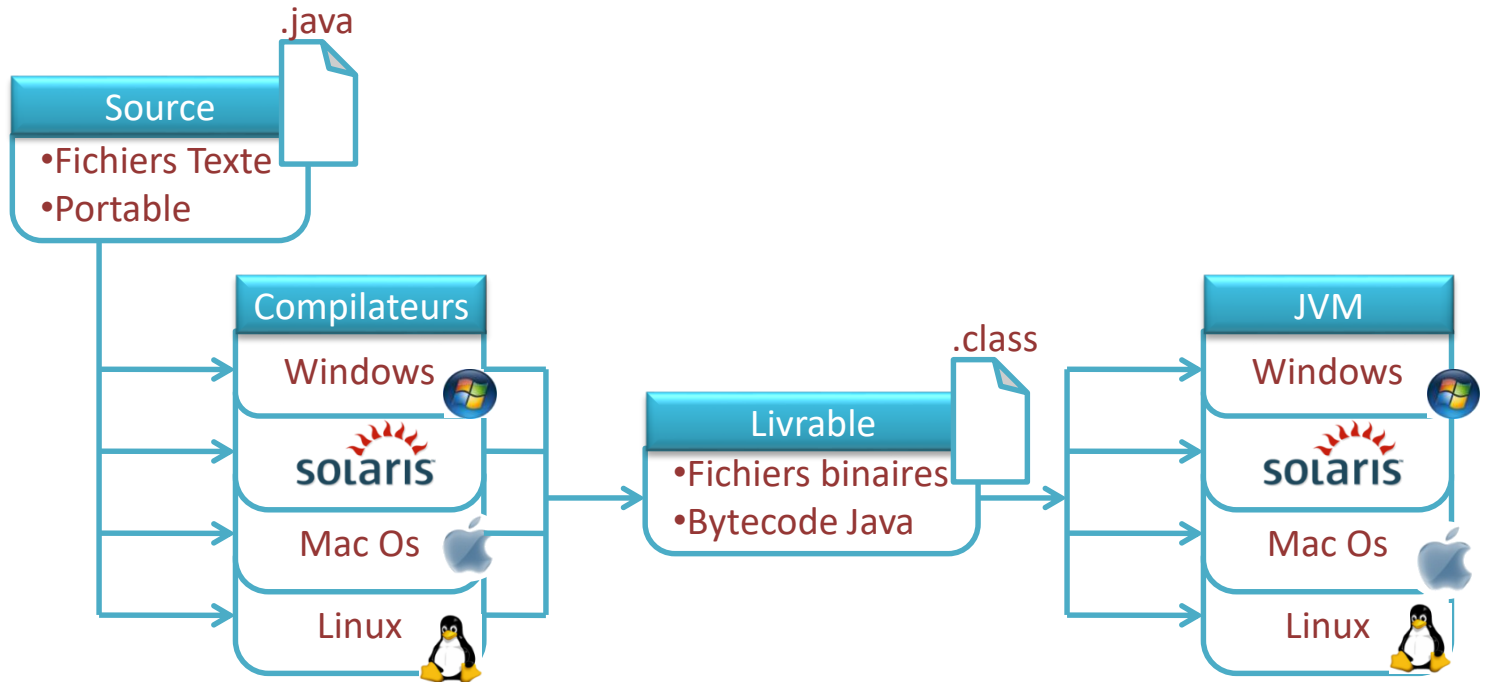
# Historique



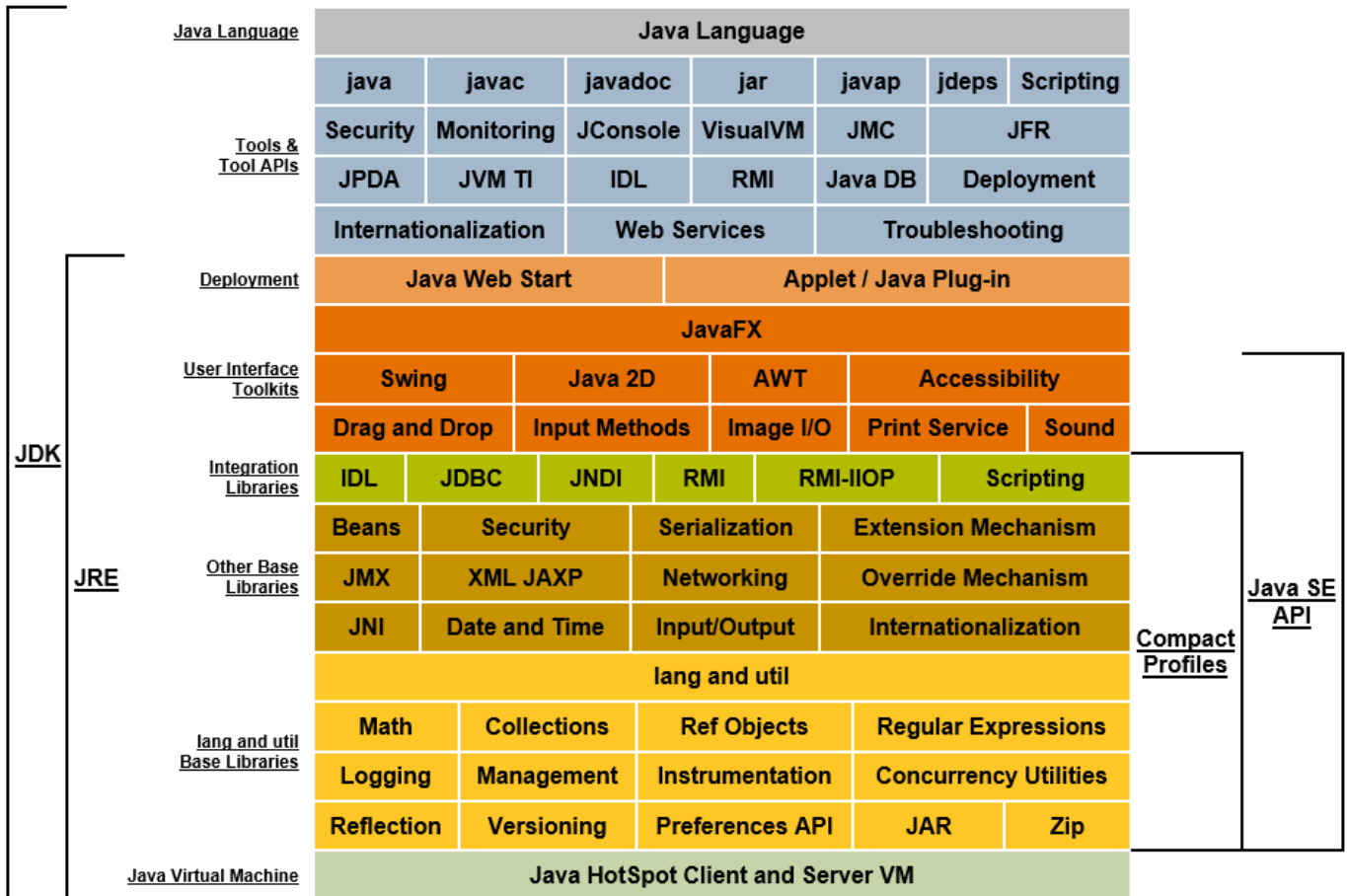
# Caractéristiques

- Orienté objet
  - Meilleure maîtrise de la complexité
  - Réutilisabilité des composants, facilitation de l'évolution
- Interprété
  - Portable
  - Indépendant des architectures
- Compilé
  - Robuste
  - Performant
- Sécurisé
  - Vérification du code : à la compilation et à l'exécution
- Dynamique
  - Chargement paresseux
- Multitâches
- Distribué

# Distribution d'une application



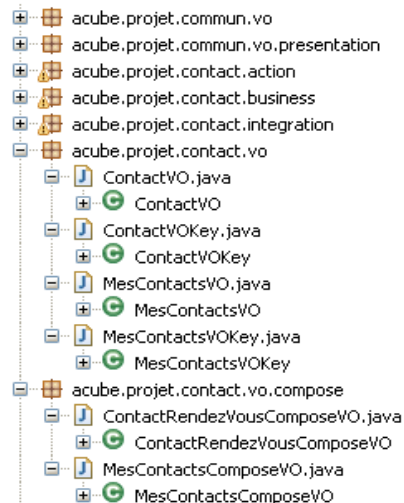
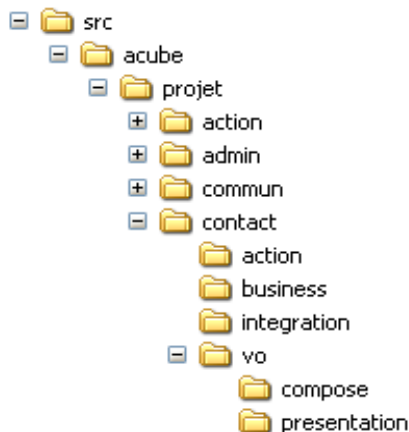
# Plateforme Java SE 8



Source : Site Internet d'Oracle sur la technologie Java  
<http://docs.oracle.com/javase/8/docs/>

# Les fichiers

- Extension « .java »
- Contiennent une « classe » (unité organisationnelle) de même nom que le fichier, à la casse près  $\Rightarrow$  Fichiers sources nombreux
- La compilation génère des fichiers d'extension « .class »
- Équivalence répertoires - packages java





# Les fondamentaux de la programmation Java (Java SE)

## Module 2

### Utilisation de l'IDE Eclipse

# Les outils de développement

- IDE : Integrated Development Environment
  - Éditeurs de texte améliorés
  - Couplage avec les outils de développement
    - Compilateur
    - Débogueur
  - Assistants
- Produits existants
  - Eclipse
  - Netbeans
  - JBuilder
  - Rational Application Developer
  - IntelliJ

# Historique d'Eclipse



- Issu du travail d'IBM 1998-2004
  - But : simplifier la galaxie d'outils nécessaires à la réalisation d'applications professionnelles
  - Moyens : intégration d'outils venant d'acteurs tiers
  - Pour rassurer les partenaires, publication en open-source en 2001
- Confié à l'Eclipse Software Foundation depuis 2004
  - Jusqu'alors, produit trop estampillé IBM
  - Création d'une association à but non lucratif indépendante
  - Première version : 3.0
  - Dernière version en juin 2014 : Luna (4.4)
  - [www.eclipse.org](http://www.eclipse.org)

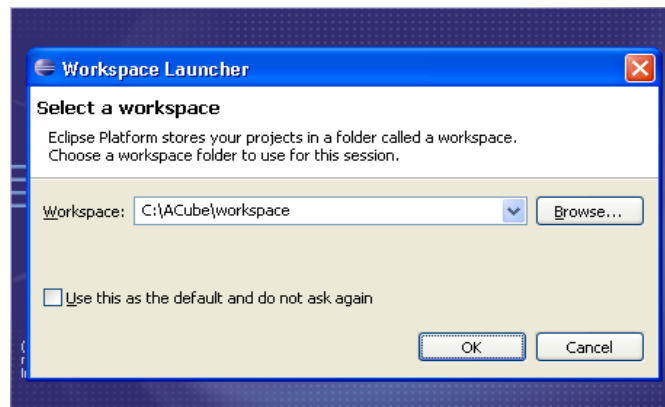
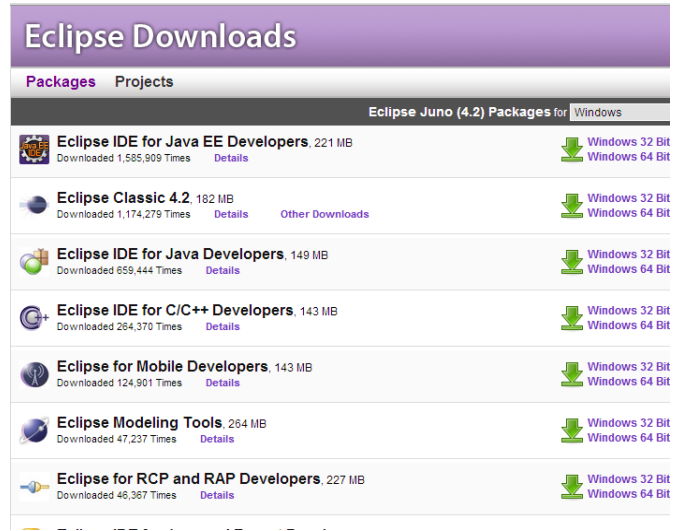
# Présentation Eclipse

## ■ Installation

- Différentes distributions
  - Développement Java
  - Développement JEE
- Distribution zippée

## ■ Workspace

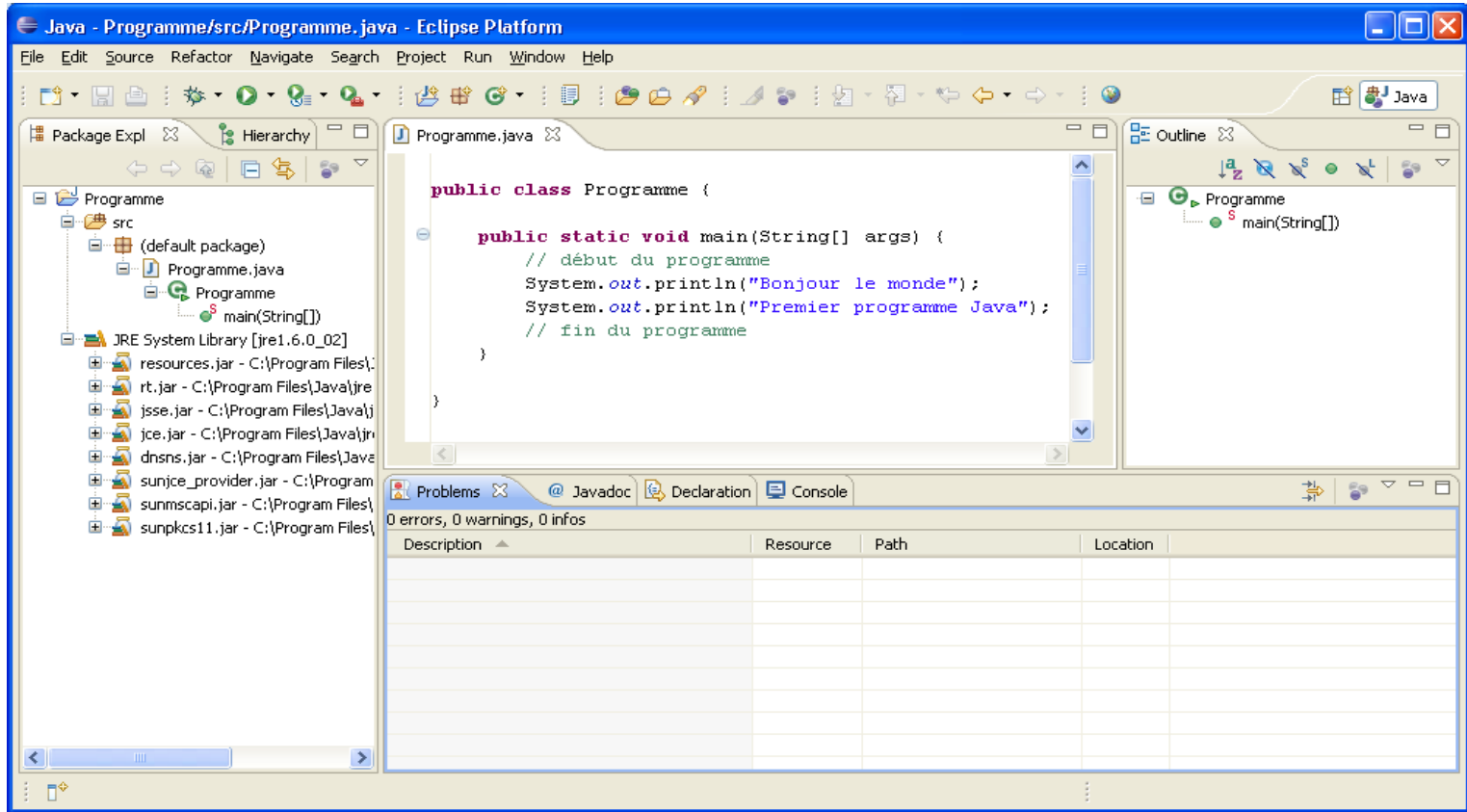
- Répertoire de travail
- Ensemble de projets
- Unité de configuration



# Eclipse : structure

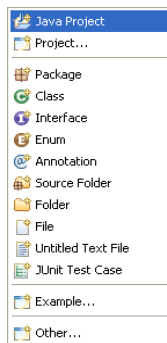
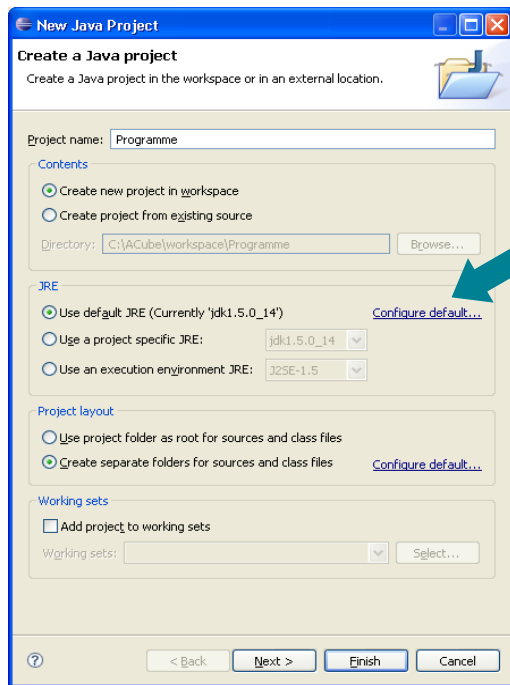
- Le workbench
  - Environnement partagé en fenêtres à onglets
    - Éditeurs
    - Vues
  - Organisation orientée tâche : les perspectives
  - Les assistants
  - Mécanisme de préférences
  - Capable d'héberger des extensions : les plugins
- Les plugins
  - Exploitation du workbench
  - Les fonctionnalités d'Eclipse

# Eclipse pour Java

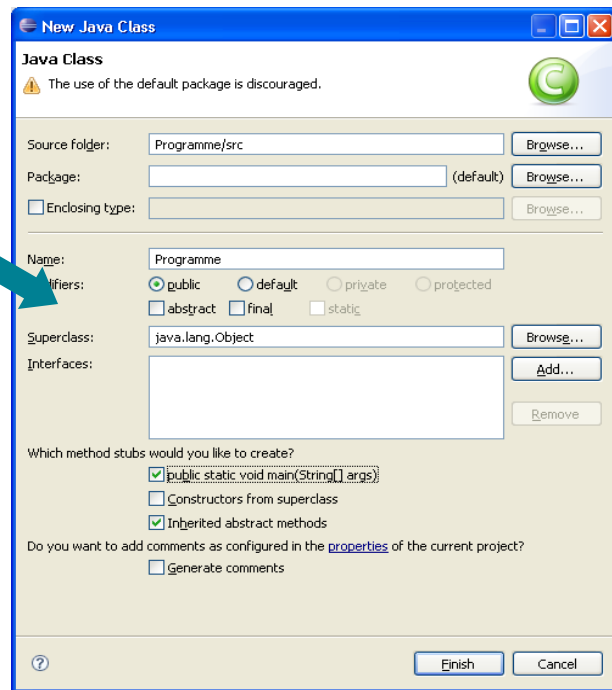


# Premier projet sous Eclipse

## Nouveau projet

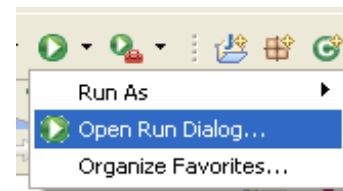
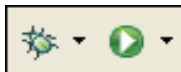


## Nouvelle classe



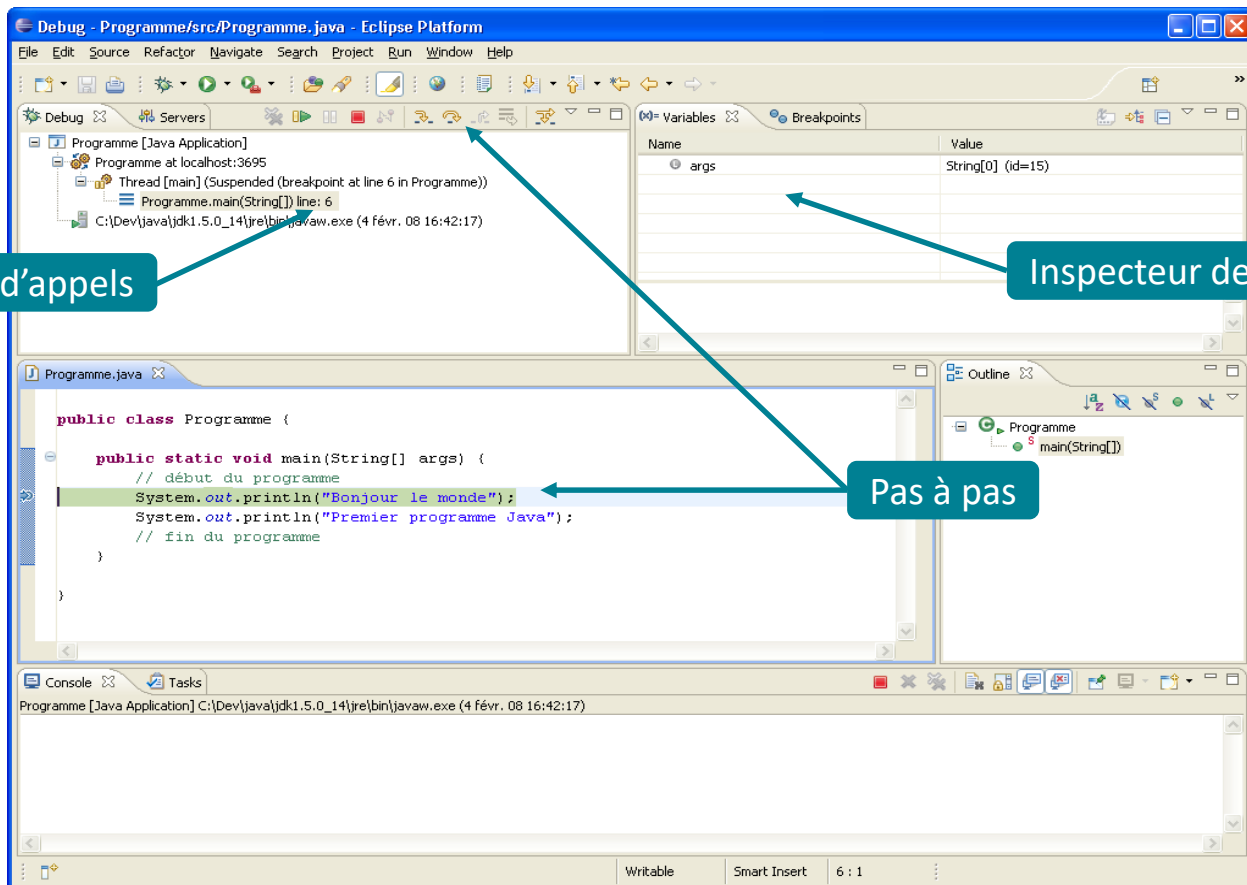
# Développement sous Eclipse

- L'assistant prend en charge de nombreuses tâches
  - Nom du fichier
  - Écriture du point d'entrée du programme
  - Compilation automatique au cours de la frappe
  - Génération de bytecode à la sauvegarde (par défaut)
  - Vue d'erreurs
  - Assistants (complétion, refactoring, templates, corrections, suggestions, présentation)
- Exécution
  - Profil d'exécution à configurer une fois par projet :
  - Les fois suivantes :
  - Entrées/Sorties standard redirigées dans la console d'Eclipse





# Mise au point des programmes



# Outils Eclipse

- Menu "Source"
  - Toggle comment
  - Add/Remove block comment
  - Generate element comment
  - Correct indentation
  - Format
  - Organize imports
  
- Menu "Refactor"
  - Rename
  - Move (+ drag'n drop)
  - Change method signature
  - Extract local variable/method/constant
  - Inline

# Les fondamentaux de la programmation Java (Java SE)

## Module 3

### Les principes de base du langage

# Blocs et instructions

- Exécution séquentielle des instructions
- Instructions simples

```
System.out.println("Bonjour");
```

Ou

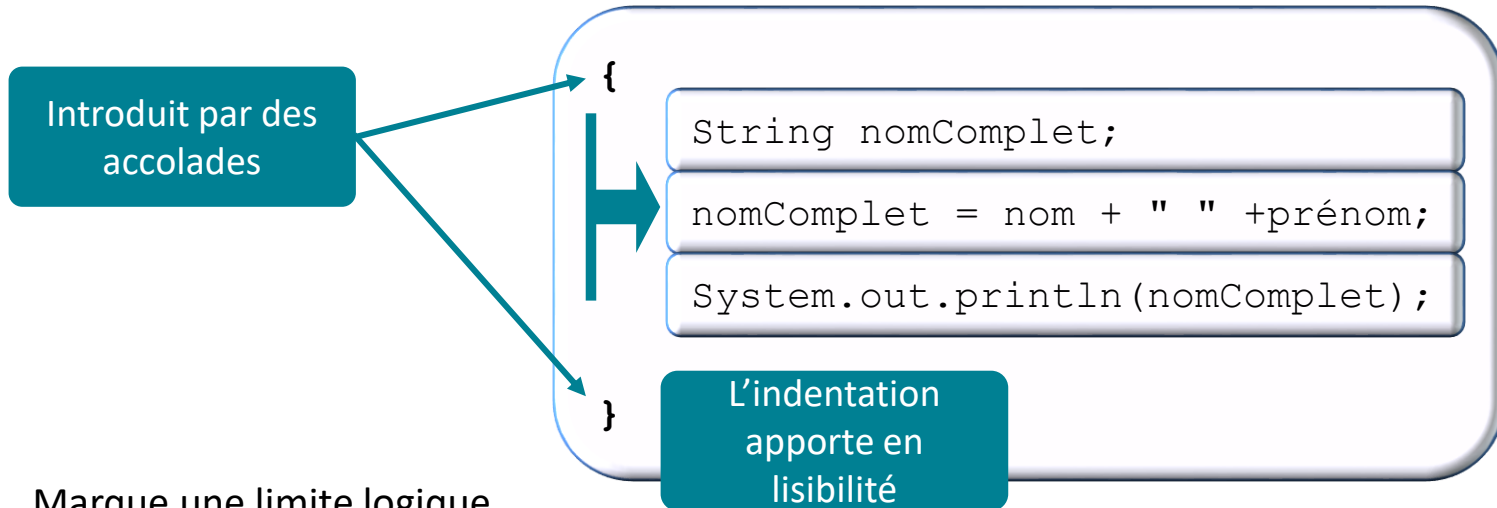
```
System.␣  
    out.␣  
        println("Bonjour")␣  
;
```

Les sauts de ligne  
sont ignorés...

...les espaces  
aussi

# Blocs et instructions

- Instruction « Bloc »



- Marque une limite logique
- Imbrication possible
- Les seuls séparateurs d'instructions sont donc { } et ;

# Les commentaires

- Les commentaires d'implémentation

```
/*  
 * ceci est un bloc de commentaires  
 */  
  
a = b; // ceci est un commentaire de fin de ligne
```

- Les commentaires de documentation

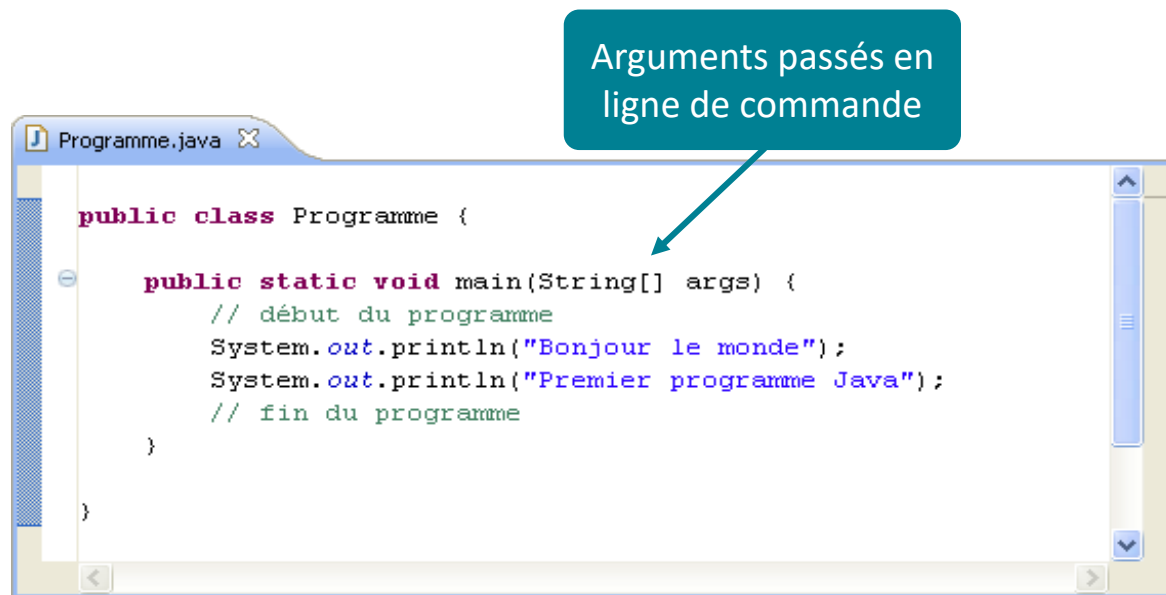
```
/**  
 * Ceci est une documentation de l'attribut  
 */  
String attribut;
```

# Les commentaires

- Les commentaires de documentation
  - Description du contenu du code
  - Requis par les utilisateurs de ce code
  - Sont exploités par des outils de génération de documentation
    - Javadoc
    - DocFlex
  - Marqueurs pour les éléments du langage
    - `@param` : description de l'usage de paramètres
    - `@return` : description des résultats possibles
    - `@throws` : description des types d'erreur en fonction des causes
    - `@author` : l'auteur du code concerné
    - `@since` : version depuis laquelle ce code est en vigueur
    - `@deprecated` : information d'obsolescence
    - `@date` : date du code

# Le point d'entrée du programme

- La procédure principale





# Mise en œuvre

- Vérification de l'environnement java

```
C:\temp>java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
C:\temp>javac -version
javac 1.8.0
```

- Compilation et exécution

Pas de nouvelle,  
bonne nouvelle

C:\temp>java **c** Programme.java

Appel au compilateur

C:\temp>java **Programme**

Bonjour le monde

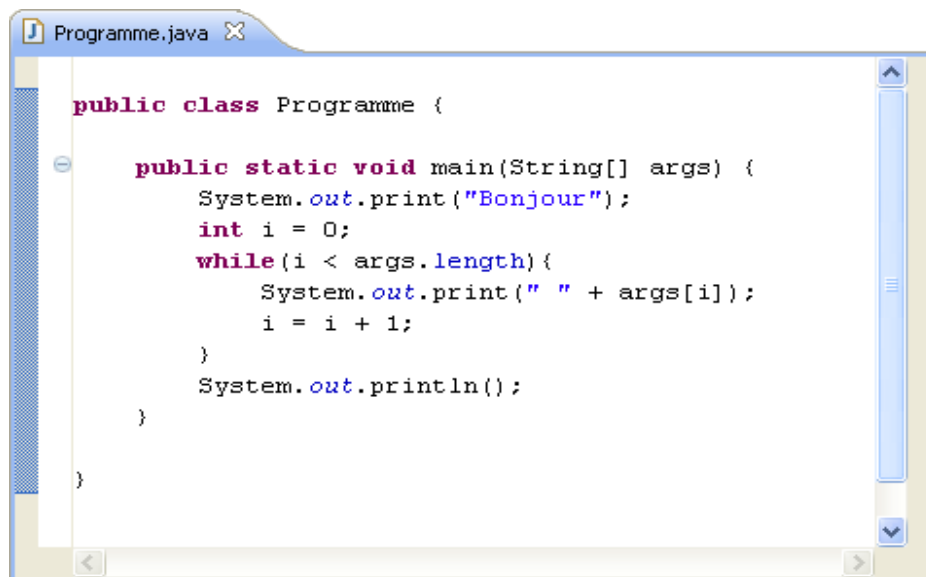
Premier programme Java

C:\temp>

Invocation de la JVM

# Arguments du programme

- la procédure principale : les paramètres



```
public class Programme {  
  
    public static void main(String[] args) {  
        System.out.print("Bonjour");  
        int i = 0;  
        while(i < args.length){  
            System.out.print(" " + args[i]);  
            i = i + 1;  
        }  
        System.out.println();  
    }  
}
```

# Arguments du programme

- Passage des paramètres

```
C:\temp>javac Programme.java  
  
C:\temp>java Programme Pierre Durand  
Bonjour Pierre Durand  
C:\temp>
```

Arguments au  
programme

- Séparés par un espace de la classe contenant le `main`
- Séparés les uns des autres par des espaces
- L'indice du premier paramètre est 0
- Le premier paramètre n'est pas la classe

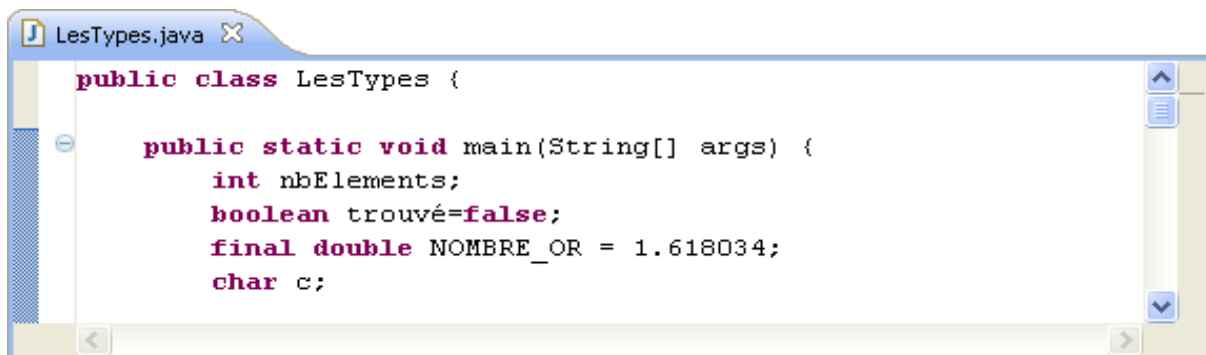
# Les types simples

- Les catégories
  - Entiers : byte, short, int, long
  - Réels : float, double
  - Caractère : char
  - Booléen : boolean
- Déclaration de variables et de constantes

**final** type nomVariable = valeurInit ;

seulement pour les constantes

obligatoire pour les constantes



```
LesTypes.java X
public class LesTypes {

    public static void main(String[] args) {
        int nbElements;
        boolean trouvé=false;
        final double NOMBRE_OR = 1.618034;
        char c;
    }
}
```

# Règles et conventions de nommage

- Règles sur les identificateurs
  - Sensible à la casse
  - Commence par une lettre, mais peut contenir lettres et chiffres
  - N'est pas un mot clé
  - Ne contient pas d'opérateur, ni certains symboles (\$, #, ~, °, etc.)
  - Ne contient pas de séparateur (espace, tabulation, saut de ligne, guillemet ou apostrophe)
- Conventions sur les variables
  - Première lettre en minuscule
  - Camel case : les mots suivants commencent par une majuscule
- Conventions sur les constantes
  - Écrites en majuscules
  - Les mots suivants sont séparés par le soulignement

# Les types simples : les entiers

## ■ Les noms de types entiers

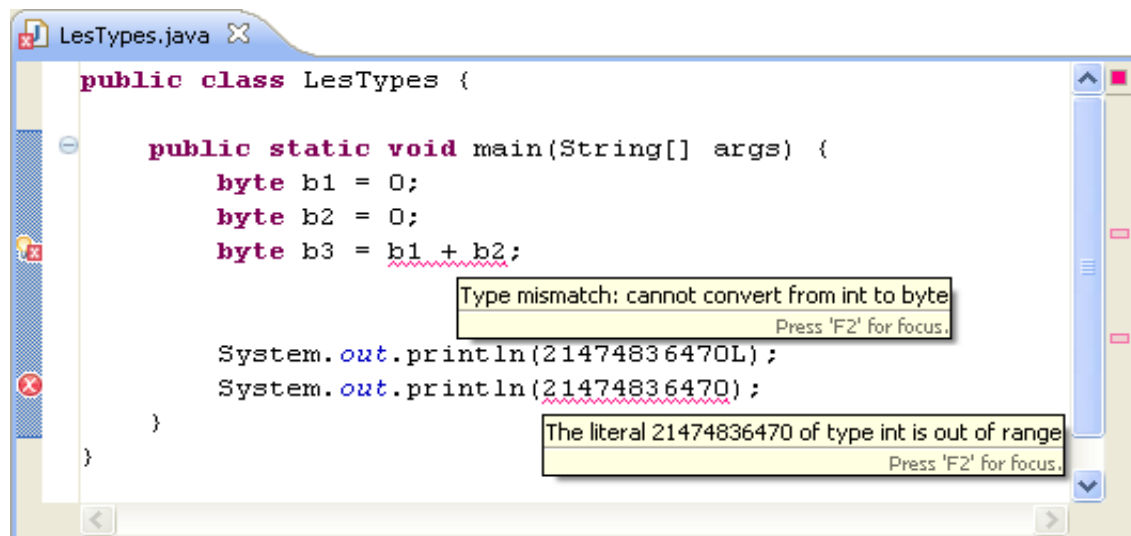
Type	Stockage	Représentation
<b>byte</b>	8 bits	-128 .. 127
<b>short</b>	16 bits	-32 768 .. 32 767
<b>int</b>	32 bits	-2 147 483 648 .. 2 147 483 647
<b>long</b>	64 bits	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807

## ■ Les constantes littérales entières

- Exprimables en octal, décimal ou hexadécimal : le nombre « vingt » peut s'écrire indifféremment :
  - 20
  - 024
  - 0x14
- Le type par défaut est `int`
  - Les constantes littérales sont des `int` sauf à suffixer avec un « `l` » ou « `L` »
  - Les opérations sur des types plus « petits » donnent un résultat en `int`
  - Attention aux affectations

# Les types simples : les entiers

- Exemples d'utilisations problématiques



# Les littéraux binaires

- Dans certains cas, il est nécessaire/pratique d'utiliser la notation binaire
  - Permissions sous Unix
  - Multiplications, divisions par 2 optimisées
  - Masques graphiques
- Jusqu'alors, il fallait exprimer ces valeurs soit :
  - Sous forme entière
  - Sous forme hexadécimale
- Inconvénients
  - Nécessité de faire des conversions
  - Manque de lisibilité



# Les littéraux binaires

- Nouvelle déclaration, exemples :

```
final byte READ= (byte)0b000000100;  
final byte WRITE=(byte)0b000000010;  
int deux = 0b10;  
final int MAX=0b11111111111111111111111111111111;  
long unLong=0b1010L;
```

- Nouveaux usages :

```
if((permission & READ)==READ{  
    short[] cross={  
        (short) 0b10000001,  
        (short) 0b01000010,  
        (short) 0b00100100,  
        (short) 0b00011000,  
        (short) 0b00011000,  
        (short) 0b00100100,  
        (short) 0b01000010,  
        (short) 0b10000001  
    };  
}
```

# Les types simples : les réels

- Les nom de types réels

Type	Stockage	Représentation
<b>float</b>	32 bits	-3.4028e+38 .. -1.4013e-45 .. 1.4013e-45 .. 3.4028e+38
<b>double</b>	64 bits	-1.7977e+308 .. -4.9000e-324 .. 4.9000e-324 .. 1.7977e+308

- Les constantes littérales réelles

- Notations

- Décimale pointée 3.14
    - Scientifique 6.022e23

- Le type par défaut

- Les constantes littérales sont des `double`, sauf à suffixer avec « f » ou « F »
    - Les opérations sur les réels donnent des `double`
    - Attention aux affectations

# Littéraux numériques avec underscores

- But :
  - Augmenter la lisibilité de certains nombres
  - Se rapprocher de leur présentation dans la vie courante
- Moyens :
  - Inclusion du caractère « \_ »
  - Autant de fois que nécessaire
- Limites :
  - Pas accolé au « . » séparateur des décimales
  - Ni en début, ni en fin de littéral, même s'il y a un « F » ou un « L »

# Littéraux numériques avec underscores

- Exemples

```
long milleMilliards= 1_000_000_000_000L;  
int telephone = 01_02_03_04_05;
```

- Erreurs

```
float pi1 = 3_.1415F;  
float pi2 = 3._1415F;  
float pi3 = 3.1415_F;  
float pi4 = 314_e-2F;  
int i1 = 0_xAA;  
int i2 = 0x_AA;
```

- Attention, « \_12 » commence par un caractère valide (l'underscore) et il est donc un identificateur (de variable, de classe, de méthode...)

```
String _42 = "La réponse";
```

# Les types simples : les booléens

- Les booléens

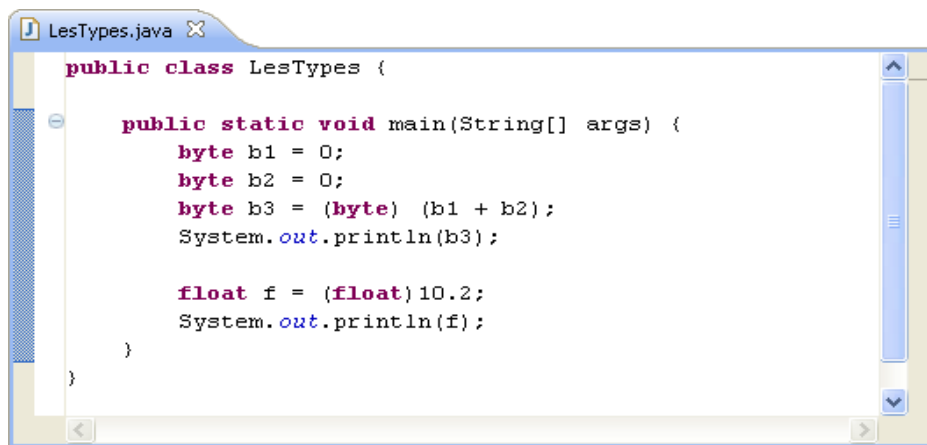
- Un nom de type : `boolean`
- Codage sur 1 bit
- 2 valeurs possibles
  - `true`
  - `false`

- Les caractères

- Un nom de type : `char`
- Codage sur 16 bits : Unicode utf-16
- Permet de stocker 1 seul caractère
- Délimiteur de constante littérale : l'apostrophe ( ' )

# Conversion de types simples

- L'opérateur de transtypage (appelé aussi "cast")  
`(nouveau_type) expression_a_convertir;`
- Opérateur préfixé
- Associatif à droite
- Attention à la priorité
- Qu'entre types compatibles
  - entier  $\Rightarrow$  réel
  - réel  $\Rightarrow$  entier (pertes)
  - entier  $\Rightarrow$  caractère
  - caractère  $\Rightarrow$  entier



```
public class LesTypes {  
  
    public static void main(String[] args) {  
        byte b1 = 0;  
        byte b2 = 0;  
        byte b3 = (byte) (b1 + b2);  
        System.out.println(b3);  
  
        float f = (float) 10.2;  
        System.out.println(f);  
    }  
}
```

# Les types complexes : les tableaux

- Deux déclarations possibles

```
Type[] nomTableau;
```

```
Type nomTableau[];
```

- Initialisation avec l'opérateur `new`

```
nomTableau = new Type[dimension];
```

⇒ L'initialisation n'est pas le remplissage

- Initialisation par liste de valeurs

```
nomTableau = {valeur1, valeur2, valeur3};
```

- La taille du tableau est immuable

- Accès aux éléments du tableau : l'opérateur `[]`

```
premierPrénom = tableauPrénoms[0];
```

- Propriété `length`

```
int longueur = nomTableau.length;
```

# Utilitaires sur les tableaux

- Copie de tableaux

```
System.arraycopy(tabSource, indiceSource, tabDest, indiceDest, nb);
```

- Autres manipulations

```
position = Arrays.binarySearch(tab, valeur);  
identiques = Arrays.equals(tab1, tab2);  
Arrays.fill(tab, valeur);  
Arrays.fill(tab, indiceDébut, indiceFin, valeur);  
Arrays.sort(tab);  
Arrays.sort(tab, indiceDébut, indiceFin);
```

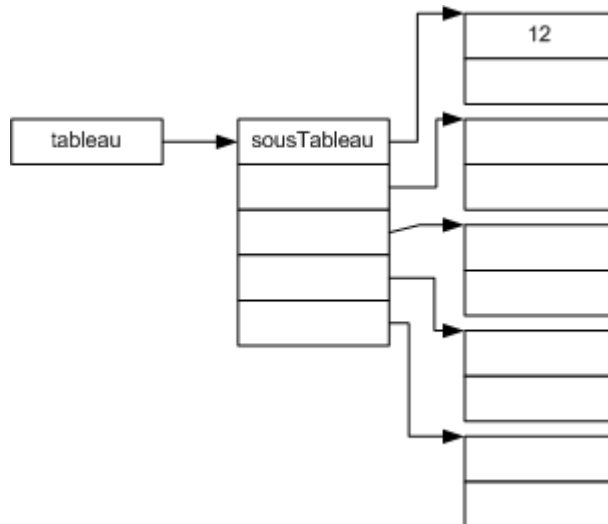


# Les tableaux à dimensions multiples

- Préalable
  - La mémoire est linéaire
  - Une seule dimension
  - La notion de ligne ou de colonne peu pertinente
  - Un tableau est seulement une référence sur ses enregistrements
  - Attention aux affectations sur le type tableau : copie de référence
- Tableaux de tableaux
  - Imbrication
  - La profondeur d'imbrication est la dimension du tableau
- Déclaration

```
LesTypes.java X
public class LesTypes {

    public static void main(String[] args) {
        int[][] tableau = new int[5][2];
        int[] sousTableau = tableau[0];
        sousTableau[0] = 12;
    }
}
```



# Chaînes de caractères

- Le type java des chaînes est `String` (avec une majuscule : C'est une classe !)
- Les chaînes sont délimitées par les guillemets ("")
- L'opérateur « + » peut être utilisé pour des concaténations simples
- Les objets de type `String` possèdent tout un ensemble de méthodes permettant de réaliser des opérations sur les chaînes.

# Les opérateurs

- Priorité des opérateurs

Forte priorité													Faible Priorité	
.	++x	new (type)	*	+	<<	<	==	&	^		&&		?:	= *=
[]	--x		/	-	>>	<=	!=						/= %=	
(args)	+x		%		>>>	>							+= -=	
x++	-x					>=								<<= >>=
x--	~													>>>= &=
	!													

# Les opérateurs

## ■ Les opérateurs unaires

.	Accès aux membres
[ ]	Accès indexé aux tableau
(args)	Appel de fonction et passage de paramètres ; associatif à gauche
x++	Incrémentation de x d'une unité en notation post-fixée ; l'expression x++ a la valeur de x avant son incrémentation
x--	Décrémentation de x d'une unité en notation post-fixée ; l'expression x-- a la valeur de x avant sa décrémentation
++x	Incrémentation de x d'une unité en notation préfixée ; l'expression ++x a la valeur de x après son incrémentation ; de ce fait l'opérateur est plus rapide
--x	Décrémentation de x d'une unité en notation préfixée ; l'expression --x a la valeur de x après sa décrémentation ; de ce fait l'opérateur est plus rapide
+x	Equivaut à x
-x	Opposé de x
~	Complément binaire
!	Négation booléenne
new	Création d'une instance
(type)	Transtypage vers type ; opérateur associatif à droite

# Les opérateurs

- Les opérateurs binaires

Arithmétique	$x*y$	L'expression vaut le produit de x par y
	$x/y$	L'expression vaut la division entière de x par y si les opérandes sont entiers, la division réelle sinon
	$x\%y$	L'expression vaut le modulo : reste de la division entière de x par y
	$x+y$	L'expression vaut la somme de x et y
	$x-y$	L'expression vaut la différence entre x et y
Binaire	$x<<y$	L'expression vaut le résultat du décalage des bits de x de y positions vers la gauche
	$x>>y$	L'expression vaut le résultat du décalage des bits de x de y positions vers la droite; remplissage par des 0 pour les positifs et des 1 pour les négatifs (signe inchangé)
	$x>>>y$	L'expression vaut le résultat du décalage des bits de x de y positions vers la droite; le remplissage est fait par des zéros

# Les opérateurs

## ■ Les opérateurs binaires (suite)

Comparaison	<b>x&lt;y</b>	L'expression vaut true si x est inférieur à y, false sinon
	<b>x&lt;=y</b>	L'expression vaut true si x est inférieur ou égal à y, false sinon
	<b>x&gt;y</b>	L'expression vaut true si x est supérieur à y, false sinon
	<b>x&gt;=y</b>	L'expression vaut true si x est supérieur ou égal à y, false sinon
	<b>x==y</b>	L'expression vaut true si x est égal à y, false sinon
	<b>x!=y</b>	L'expression vaut true si x est différent de y, false sinon
Logique	<b>x&amp;y</b>	L'expression vaut true si x ET y valent true, false sinon ; y est évalué même si x vaut false
	<b>x^y</b>	L'expression vaut true si les booléens x et y sont opposés, false sinon
	<b>x y</b>	L'expression vaut true si x OU y valent true, false sinon ; y est évalué même si x vaut true
	<b>x&amp;&amp; y</b>	L'expression vaut true si x ET y valent true, false sinon ; y n'est pas évalué si x vaut false (dans ce cas, si y est une fonction, elle n'est pas appelée)
	<b>x   y</b>	L'expression vaut true si x OU y valent true, false sinon ; y n'est pas évalué si x vaut true (dans ce cas, si y est une fonction, elle n'est pas appelée)

# Les opérateurs

- L'opérateur ternaire
  - $x ? y : z$
  - Si  $x$  vaut `true` alors l'expression vaut  $y$  sinon elle vaut  $z$
- L'affectation
  - $x = y$
  - $x$  est la copie de la valeur de  $y$

# Les opérateurs

## ■ Les opérateurs d'affectation combinés

Arithmétique	<b>x*=y</b>	x vaut le produit de x par y
	<b>x/=y</b>	x vaut la division entière de x par y si les opérandes sont des entiers, la division réelle sinon
	<b>x%=y</b>	x vaut le modulo : reste de la division entière de x par y
	<b>x+=y</b>	x vaut la somme de x et y
	<b>x-=y</b>	x vaut la différence entre x et y
Binaire	<b>x&lt;&lt;= y</b>	x vaut le résultat du décalage des bits de x de y positions vers la gauche
	<b>x&gt;&gt;= y</b>	x vaut le résultat du décalage des bits de x de y positions vers la droite ; remplissage par des 0 pour les positifs et des 1 pour les négatifs (signe inchangé)
	<b>x&gt;&gt;&gt;= y</b>	x vaut le résultat du décalage des bits de x de y positions vers la droite ; le remplissage est fait par des zéros
Logique	<b>x&amp;y</b>	x vaut true si x ET y valent true, false sinon ; y est évalué même si x vaut false
	<b>x^y</b>	x vaut true si les booléens x et y sont opposés, false sinon
	<b>x y</b>	x vaut true si x OU y valent true, false sinon ; y est évalué même si x vaut true

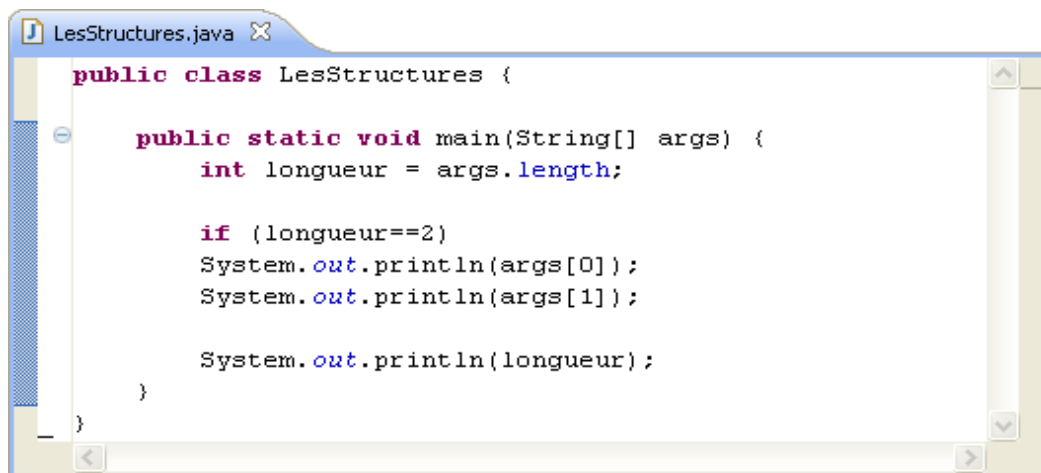


# Structures de contrôle

- La structure séquentielle
  - C'est la structure élémentaire
  - Les instructions sont exécutées les unes après les autres
- La structure conditionnelle
  - Permet l'alternative entre différentes instructions
  - Différentes instructions (si, si-sinon, structure à entrées multiples)
- La structure itérative/répétitive
  - Permet d'exécuter une suite d'instructions de multiples fois
  - Traitement en boucle
  - Différentes boucles (tant que, pour, pour chaque)

# Structures de contrôle

- Les termes ne s'appliquent qu'à une seule instruction
  - Instruction simple
  - Bloc
- Exercice : quelles instructions s'exécutent (longueur vaut 1)?



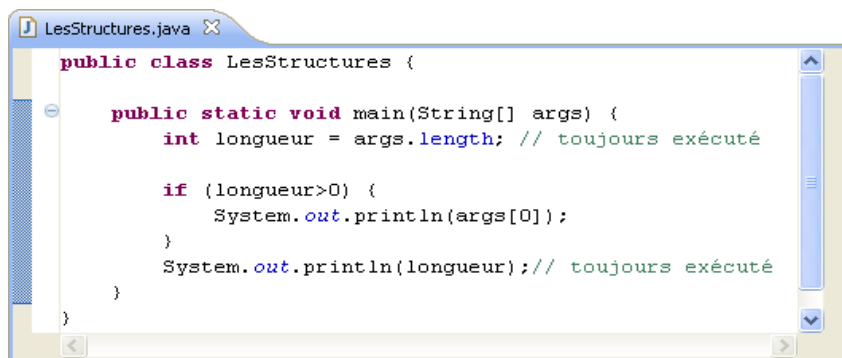
```
public class LesStructures {  
  
    public static void main(String[] args) {  
        int longueur = args.length;  
  
        if (longueur==2)  
            System.out.println(args[0]);  
            System.out.println(args[1]);  
  
        System.out.println(longueur);  
    }  
}
```

# Structures conditionnelles

## ■ Structure alternative

### ■ If

- Mot clé `if`
- Condition d'entrée entre parenthèses



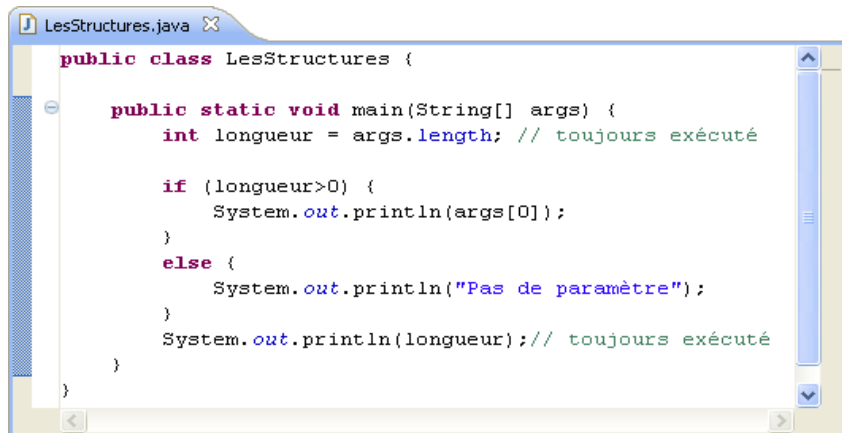
```
LesStructures.java X
public class LesStructures {

    public static void main(String[] args) {
        int longueur = args.length; // toujours exécuté

        if (longueur>0) {
            System.out.println(args[0]);
        }
        System.out.println(longueur); // toujours exécuté
    }
}
```

### ■ If - else

- Bloc `else`
- Suit immédiatement le `if`



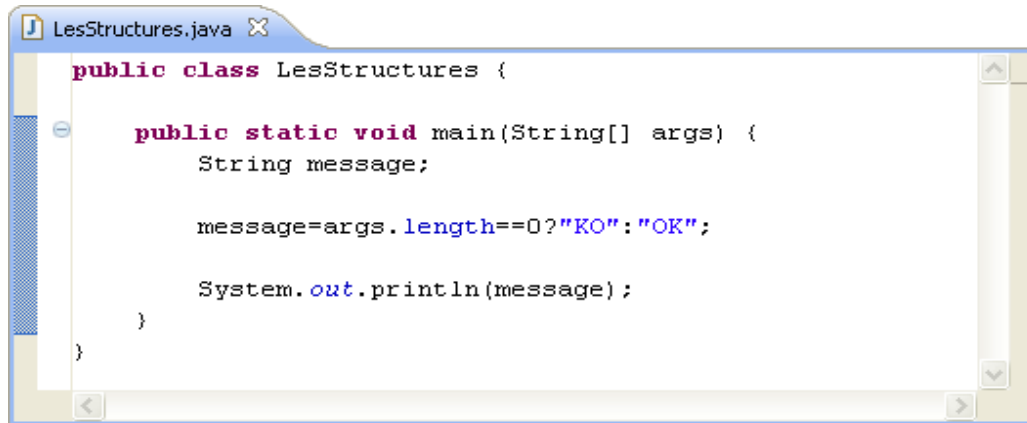
```
LesStructures.java X
public class LesStructures {

    public static void main(String[] args) {
        int longueur = args.length; // toujours exécuté

        if (longueur>0) {
            System.out.println(args[0]);
        }
        else {
            System.out.println("Pas de paramètre");
        }
        System.out.println(longueur); // toujours exécuté
    }
}
```

# Structures conditionnelles

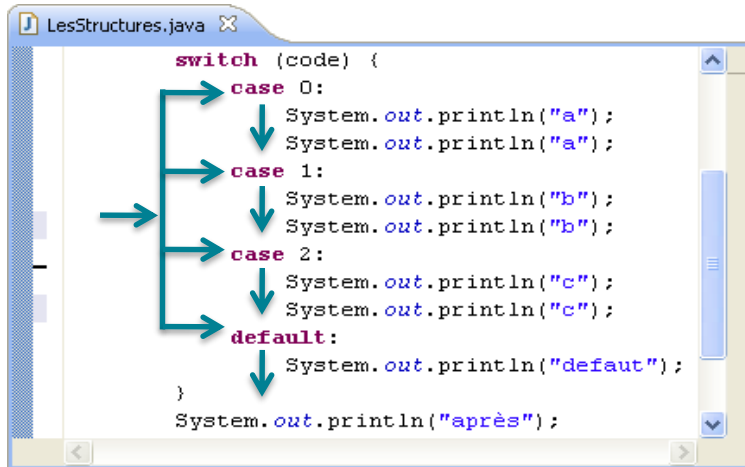
- Opérateur ternaire, l'opérateur d'alternative
  - Le seul opérateur ternaire
  - Permet d'écrire un if-else élémentaire de façon concise
  - Peut devenir difficile à lire s'il y a de l'imbrication
  - Syntaxe :  
expressionATester ? valeurSiTestVrai : valeurSiTestFaux
  - Les deux valeurs doivent être de même type
  - Si le type des valeurs est booléen, cet opérateur est inutile



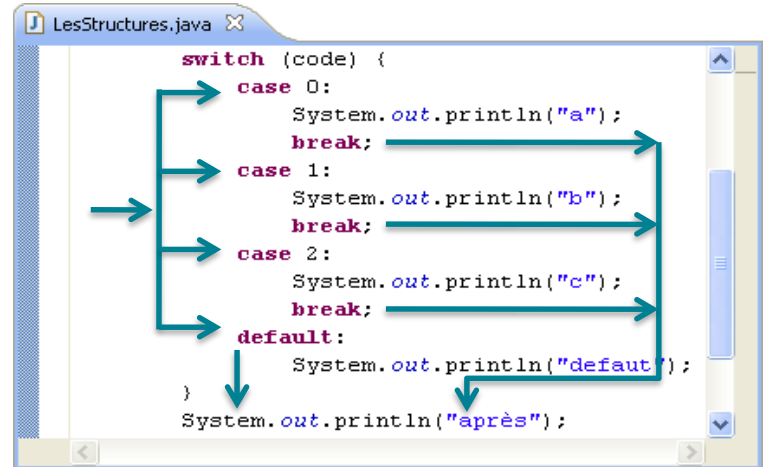
```
public class LesStructures {  
  
    public static void main(String[] args) {  
        String message;  
  
        message=args.length==0?"KO":"OK";  
  
        System.out.println(message);  
    }  
}
```

# Structures conditionnelles

- Structure à entrées multiples (switch - case)
  - Fonctionne comme un aiguillage
  - Par défaut, enchaînement des cas
    - Utile pour les procédures de reprise de traitement
    - Souvent utilisé en remplacement de `else+if` ⇒ Ne pas oublier le `break` !
    - Plutôt pour des traitements simples ⇒ soucis de lisibilité
  - Disponible seulement sur `byte`, `char`, `int`, `short`
  - La clause `default` est facultative et peut être placée à n'importe quelle place.



```
switch (code) {  
    case 0:  
        System.out.println("a");  
        System.out.println("a");  
    case 1:  
        System.out.println("b");  
        System.out.println("b");  
    case 2:  
        System.out.println("c");  
        System.out.println("c");  
    default:  
        System.out.println("default");  
}  
System.out.println("après");
```



```
switch (code) {  
    case 0:  
        System.out.println("a");  
        System.out.println("a");  
        break;  
    case 1:  
        System.out.println("b");  
        System.out.println("b");  
        break;  
    case 2:  
        System.out.println("c");  
        System.out.println("c");  
        break;  
    default:  
        System.out.println("default");  
}  
System.out.println("après");
```

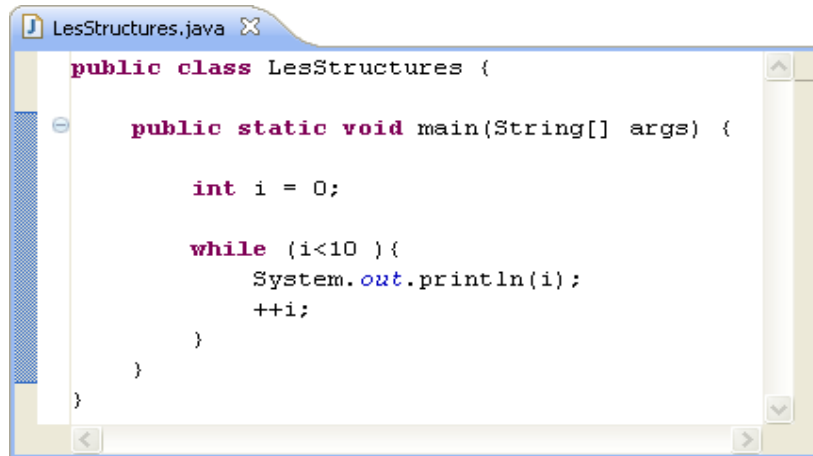
# Le switch avec les chaînes

- Depuis le JDK 1.7, il est possible de comparer une variable de type `String` avec des littéraux chaînes dans la structure `switch`
- De façon masquée, le compilateur fait appel à la méthode `equals` de la classe
- Il s'agit donc d'une comparaison sensible à la casse

```
switch (message) {  
    case "OK": traiter(); break;  
    case "KO": afficherErreur();  
}
```

# Structures itératives

- Structure "tant que" (`while`)
  - Forme de boucle la plus flexible
  - La paire de parenthèses contient la condition d'entrée
  - Elle est évaluée au début de chaque itération
  - Ne pas oublier de faire évoluer la condition d'entrée



```
LesStructures.java
public class LesStructures {

    public static void main(String[] args) {

        int i = 0;

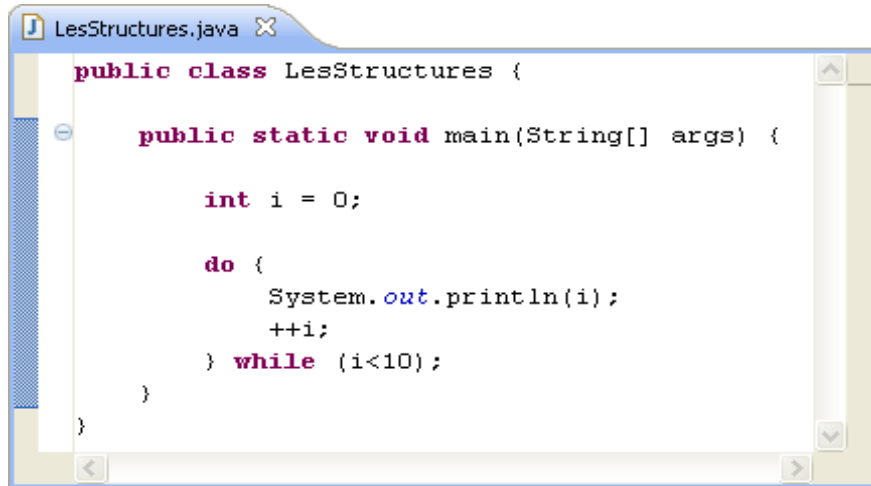
        while (i<10 ){
            System.out.println(i);
            ++i;
        }

    }

}
```

# Structures itératives

- Variante faire-tant que (do - while)
  - C'est toujours une condition d'entrée qu'on exprime
  - Elle est évaluée à la **fin** de chaque itération



```
LesStructures.java X
public class LesStructures {

    public static void main(String[] args) {

        int i = 0;

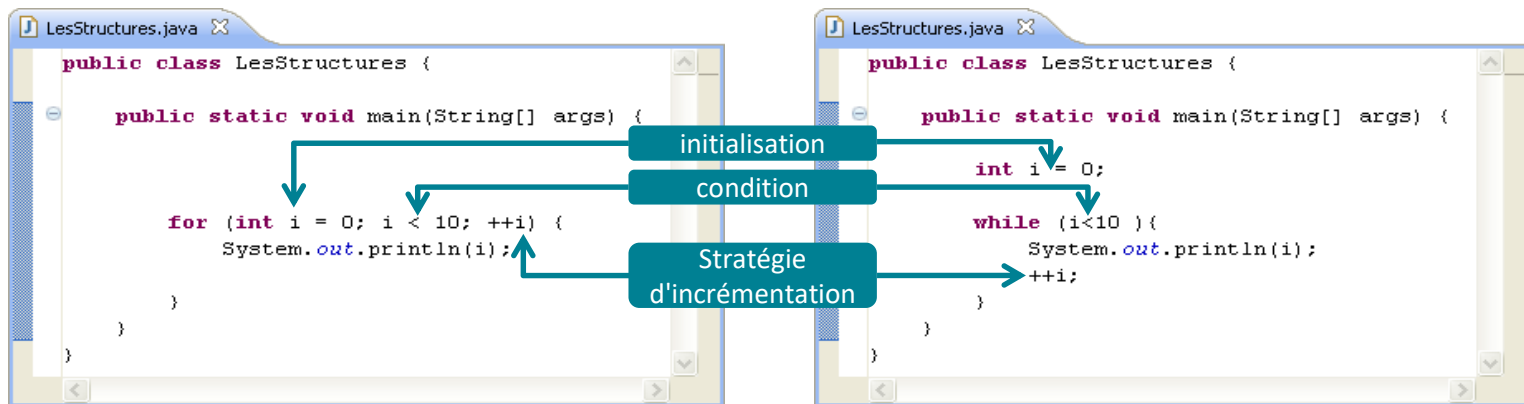
        do {
            System.out.println(i);
            ++i;
        } while (i<10);

    }
}
```



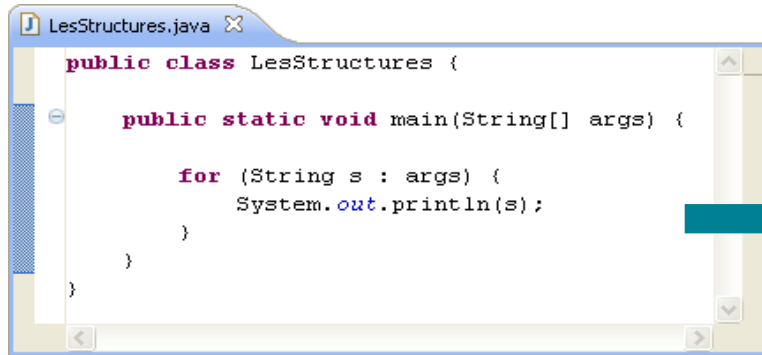
# Structures itératives

- Boucle "pour" (for)
  - Écriture condensée d'un while
  - Attention aux ";"
  - Expression d'une boucle infinie : for ( ; ; )



# Structures itératives

- La nouvelle boucle "pour chaque"
  - Depuis le jdk 1.5
  - Permet de parcourir une collection ou un tableau
  - Évite l'utilisation d'un compteur
  - Inconvénient : on ne sait pas quel est le numéro de l'itération courante

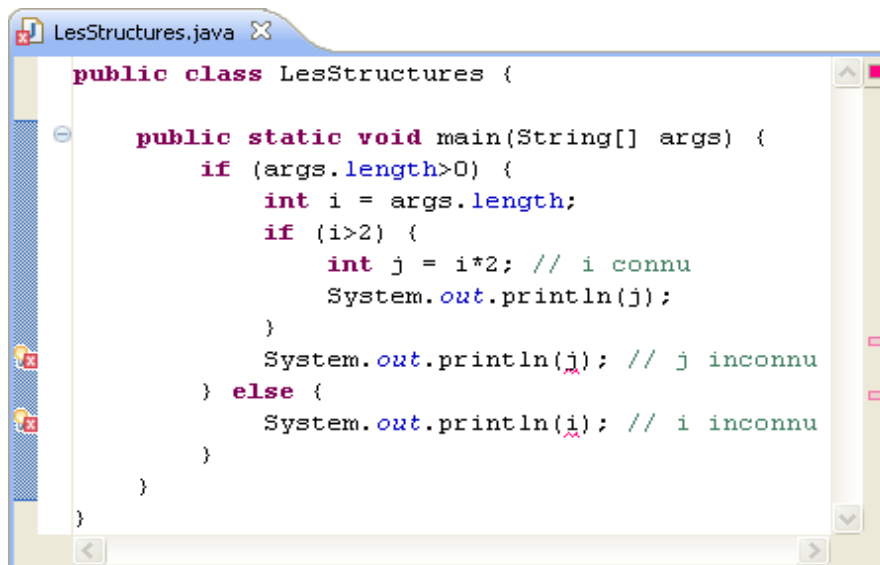


```
public class LesStructures {  
  
    public static void main(String[] args) {  
  
        for (String s : args) {  
            System.out.println(s);  
        }  
    }  
}
```

Se lit : pour chaque "s" de type String contenu dans args, faire...

# Portée des variables

- Visibilité dans le bloc de déclaration
- Invisibilité des variables des sous blocs
- Destruction des variables en fin de bloc
- Les arguments des fonctions font partie du bloc suivant
- Les clauses du `for` font partie du bloc suivant



```
public class LesStructures {  
  
    public static void main(String[] args) {  
        if (args.length>0) {  
            int i = args.length;  
            if (i>2) {  
                int j = i*2; // i connu  
                System.out.println(j);  
            }  
            System.out.println(j); // j inconnu  
        } else {  
            System.out.println(i); // i inconnu  
        }  
    }  
}
```

# Bonnes pratiques

- Affectations :

Éviter les affectations multiples en ligne et préférer la multiplication des lignes :

```
a = b = c = 0; // A éviter
a = 0; //OK
b = 0;
c = 0;
```

- Parenthèses :

La priorité des opérateurs est complexe. Une paire de parenthèse "superflue" clarifie souvent l'expression.

```
if(a && b || c && d) {...} // A éviter
if((a && b) || (c && d)) {...} // plus clair
```

- Constantes :

Pas de constantes littérales numériques dans le code (mis à part l'initialisation des compteurs)

```
for(int i = 0; i < 12; i++){...}
for(int i = 0; i < I_MAX; i++){...}
```

# Les classes d'enveloppe (wrapper)

- Les collections ne gèrent que des objets
- Le type simple doit être encapsulé dans une collection
- Les grands types

- Package `java.lang`

- Boxing et Unboxing

```
Integer monInt = new Integer(12);  
int i = monInt.intValue();
```

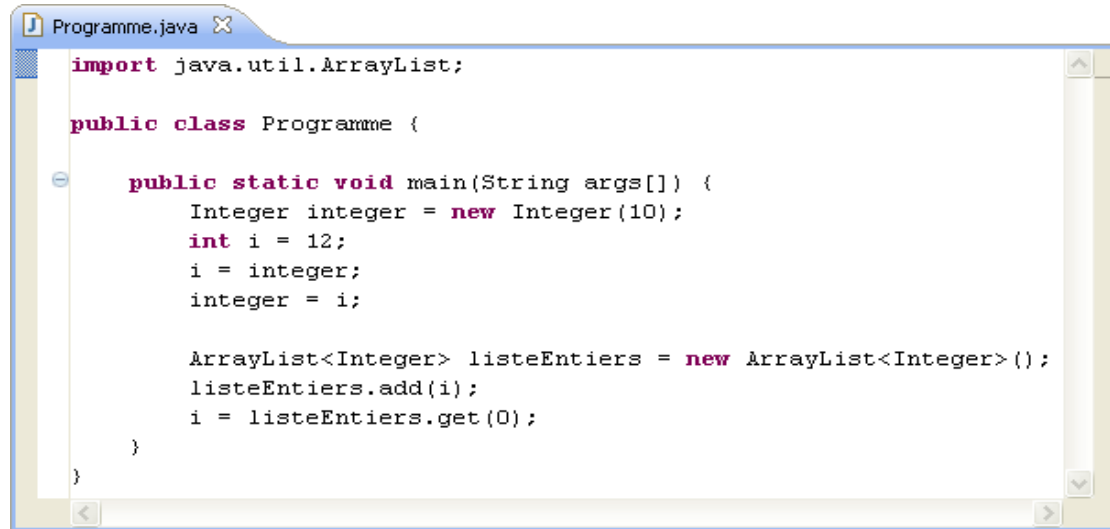
- Conversion de et vers `String`

- `String.valueOf()`
  - `Integer.parseInt(maChaine)`,  
`Double.parseDouble(maChaine)...`

Type simple	Classe d'enveloppe
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

# Les classes wrapper : autoboxing

- Nouveauté de Java 5
- Raccourci de syntaxe pour le boxing et l'unboxing
- Transparence de la gestion des instances dans les collections génériques



```
Programme.java X
import java.util.ArrayList;

public class Programme {

    public static void main(String args[]) {
        Integer integer = new Integer(10);
        int i = 12;
        i = integer;
        integer = i;

        ArrayList<Integer> listeEntiers = new ArrayList<Integer>();
        listeEntiers.add(i);
        i = listeEntiers.get(0);
    }
}
```

# Les énumérations

- Fonctionnalité ajoutée en Java 5
- Permet de restreindre les valeurs d'une variable

```
public enum Couleur{PIQUE, COEUR, CARREAU, TREFLE}  
Couleur c = Couleur.PIQUE;
```

- C'est une classe spéciale
  - Le compilateur ajoute automatiquement certaines méthodes

- de classe : `values()`, `valueOf(String name)`

```
for (Couleur c : Couleur.values()) {  
    System.out.println(c);  
}
```

- d'instance : `name()`, `compareTo(Couleur c)`, `toString()`

- Constructeur par défaut et surchargé
- Variables d'instances additionnelles et getters

# Les fondamentaux de la programmation Java (Java SE)

## Module 4

# La programmation orientée objet en Java



# Historique

- Monde de la simulation
- Grammaires procédurales impropres à la modélisation des problèmes
- Besoin d'exprimer un environnement de façon naturelle
- Décomposition d'un sujet à comprendre, analogue au fonctionnement de l'esprit humain :
  - Le "monde" est composé de "choses", "d'objets"
  - Ces objets interagissent entre eux
- Conception orientée objet : une approche descriptive
- Premiers langages objet : SIMULA, Smalltalk, C++, Eiffel

# Les 3 concepts fondateurs

- Encapsulation
  - L'objet forme un tout
  - L'objet appartient à une nature (type) qui ne peut changer
  - L'objet est garant de son état
- Héritage
  - L'objet peut être une évolution d'un autre, plus général
- Polymorphisme
  - Des objets de natures différentes peuvent réagir au même message

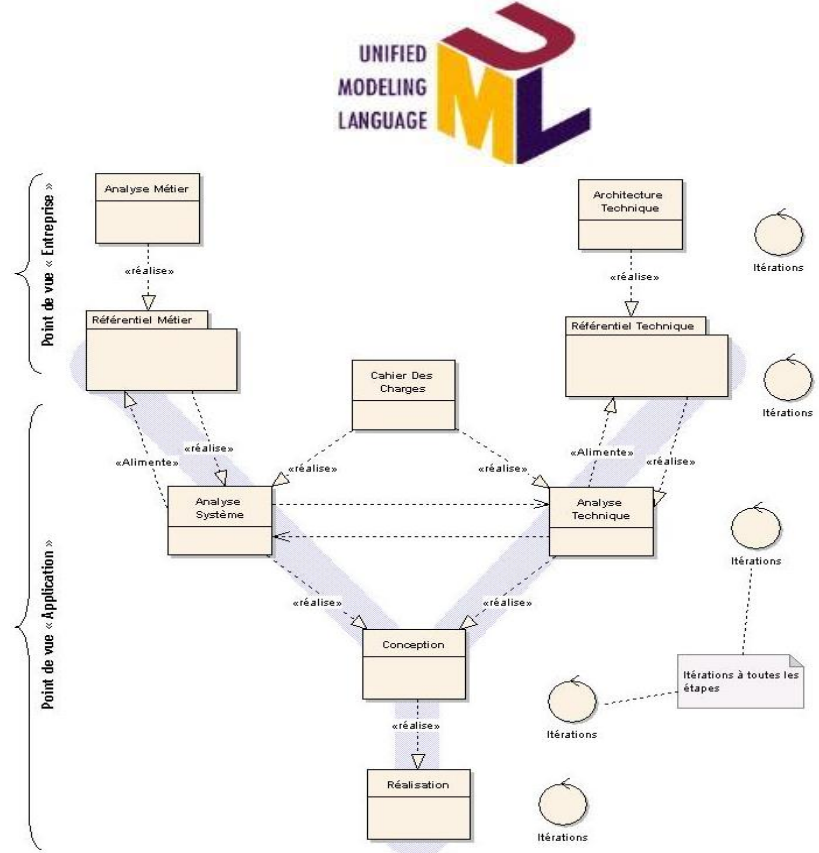
# Encapsulation : un tout

- Approche descriptive
  - Énumération des propriétés/attributs pertinents pour l'étude  
ex : Une voiture a comme caractéristiques un kilométrage, une quantité de carburant et une immatriculation
  - ⇒ L'ensemble des propriétés de l'objet à un instant donné constitue son état
  - ⇒ Modifier la valeur d'un des attributs revient à modifier l'état de l'objet
  - Énumération des capacités de l'objet étudié : les services qu'il peut réaliser  
ex : Une voiture peut rouler une certaine distance
  - Traitements implémentés sous forme procédurale : les méthodes

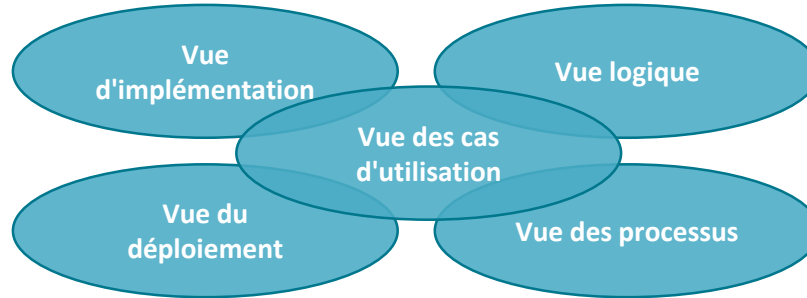
# Encapsulation : un tout

- Attributs + méthodes = Objet
  - Un nom permet de résumer les attributs et les méthodes : la classe  
ex : Voiture
  - Émergence d'un concept
    - Réflexion sur les propriétés communes à toutes les voitures  
⇒ Différence entre le concept et l'exemplaire
  - Principe d'abstraction :
    - Un nom résume un tout complexe
    - Permet d'appréhender la réalité sans s'attacher au superflu
    - Cela explique que l'approche est naturelle  
ex : un voyageur peut prendre l'avion sans tout connaître de l'avion

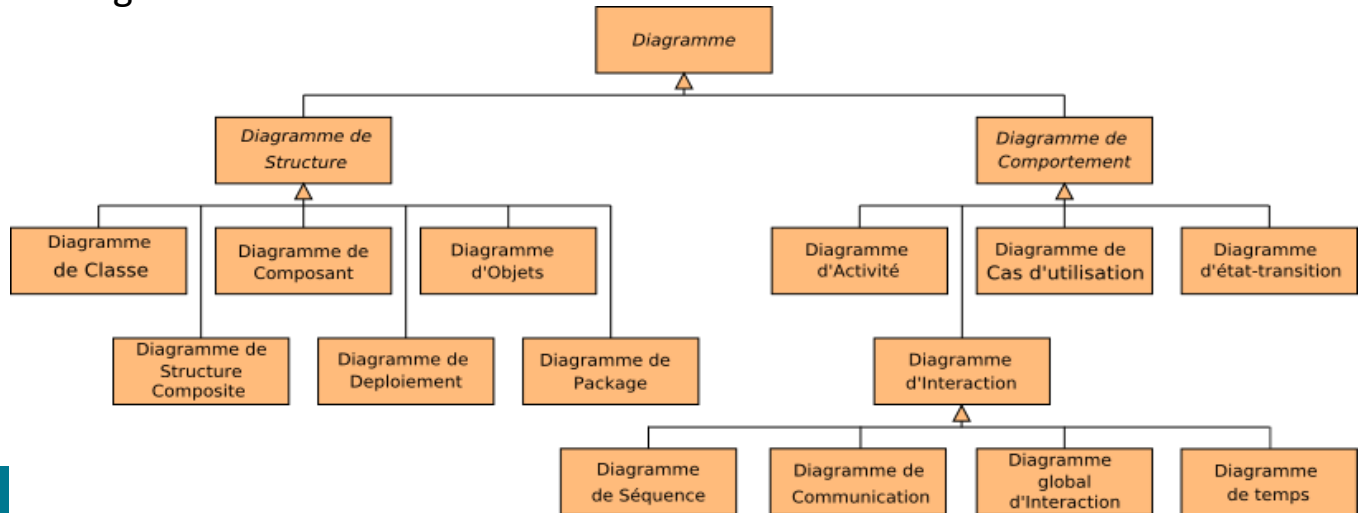
- Unified Modelling Language
- Langage graphique ≠ méthode
- Boîte à outils
  - Vues
  - Diagrammes
  - Modèles d'éléments
- Reprend, étend plusieurs autres travaux de modélisation : Booch, Rumbaugh et Ivar Jacobson
- Association avec une méthode
  - OMT
  - RUP
  - 2TUP



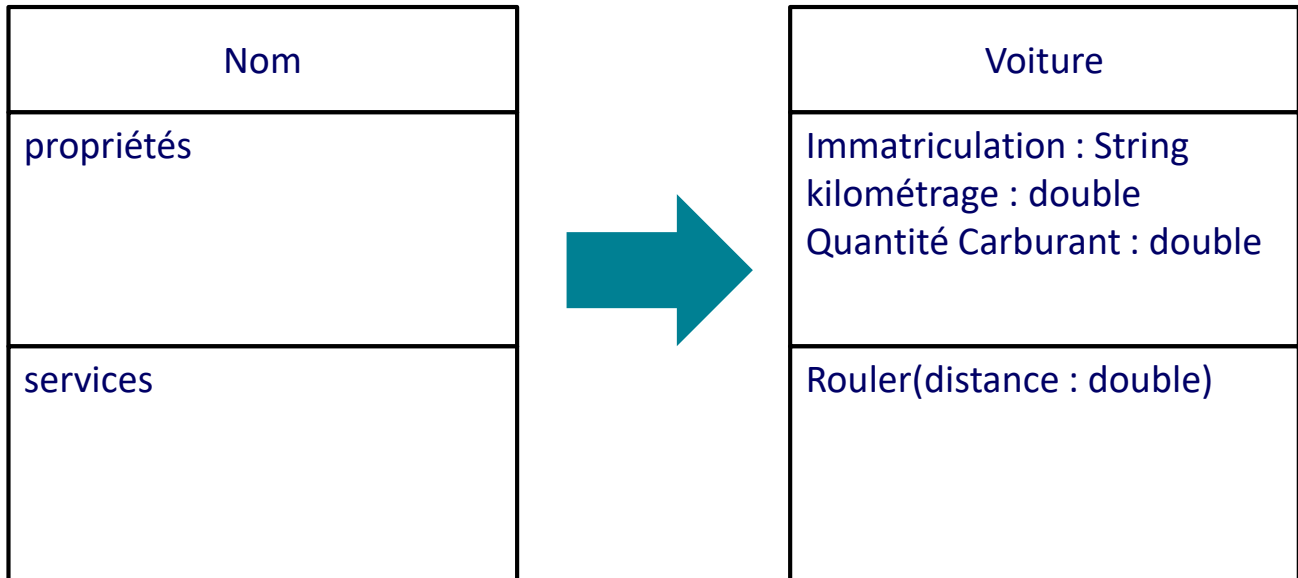
- Les vues



- Les diagrammes

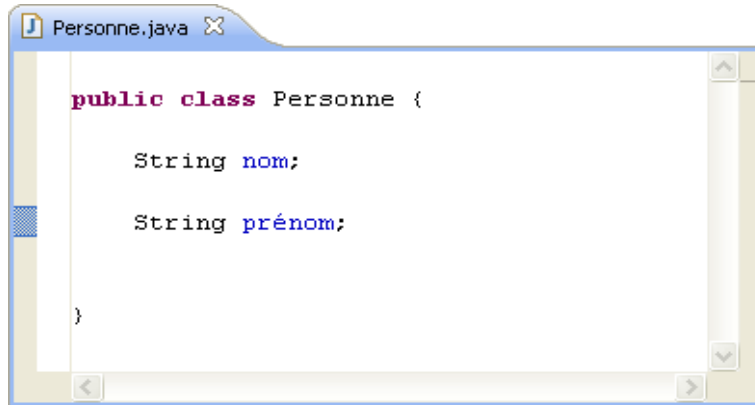


# UML : l'élément classe



# La classe Java

- Mot-clé `class`
- Marque la nature des objets issus de cette classe
- Un objet ne peut changer de nature au cours de sa vie
- Une classe publique par fichier
- Le fichier a le même nom que la classe publique
- Tout ce qui est dans l'accolade participe à la description du concept
- Il n'y a pas d'ordre dans la description



```
Personne.java X
public class Personne {
    String nom;
    String prénom;
}
```



# Attributs et méthodes

- Membres d'instance
- Pas d'ordre dans les déclarations
- Attributs
  - Structure de donnée
  - Type simple ou complexe
- Méthodes
  - Équivalent d'une fonction ou d'une procédure dans un contexte objet
  - En Java, tout est en objet : que des méthodes

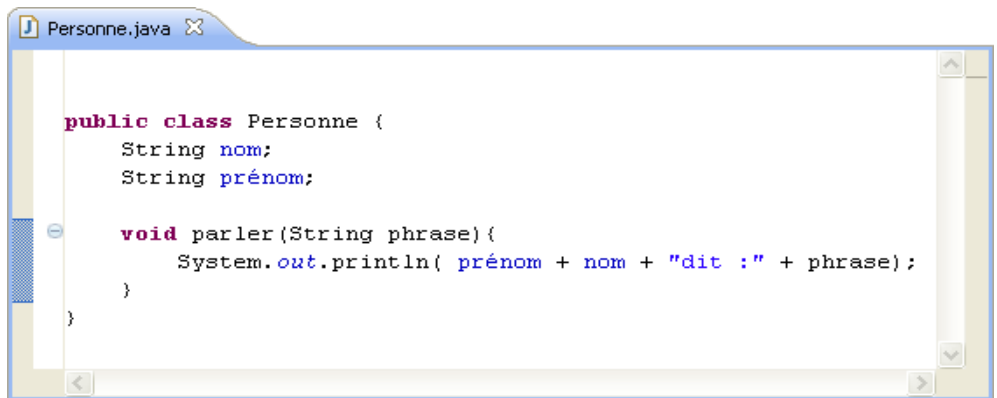
# Classe : exemple Java

Personne

nom : String

prénom : double

parler(phrase : String)



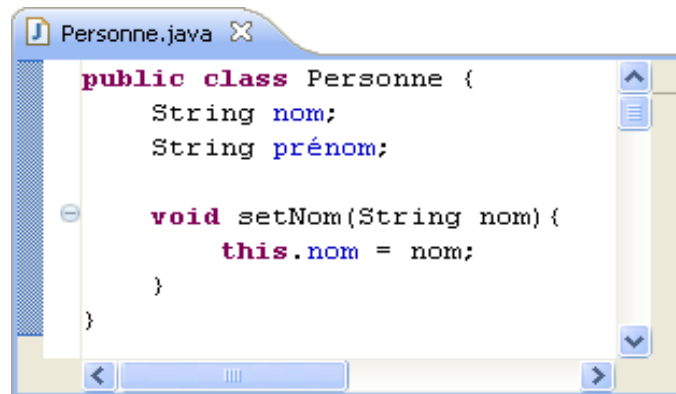
```
Personne.java X

public class Personne {
    String nom;
    String prénom;

    void parler(String phrase){
        System.out.println( prénom + nom + "dit :" + phrase);
    }
}
```

# Attributs

- Attributs
  - Déclaration directement dans la classe
  - Variable dont la durée de vie est l'instance
  - Global à l'instance
- `this`
  - Mot-clé de résolution d'espace de nommage
  - Parfois, il y a des ambiguïtés entre paramètre de méthode et attribut d'instance
  - Règle :
    - La variable désignée : celle déclarée dans le bloc le plus interne



```
public class Personne {  
    String nom;  
    String prénom;  
  
    void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

# Les méthodes

- Factorisation, réutilisation du code
- Modularité
- Différences procédures/fonctions
  - Suite à son exécution, la fonction a une valeur  
ex : `y = carré(4); // carré(4) à la valeur 16`  
La fonction peut donc apparaître à droite d'une affectation.  
C'est comme les fonctions mathématiques.
  - Suite à son exécution, une procédure n'a pas de valeur. Elle est plus proche d'un sous-programme.
- En Java, la différence syntaxique est légère : le type de retour `void` (vide) marque la définition d'une procédure
- Le mot méthode désigne indifféremment une fonction ou une procédure en programmation orientée objet (et donc en Java)

# Les méthodes

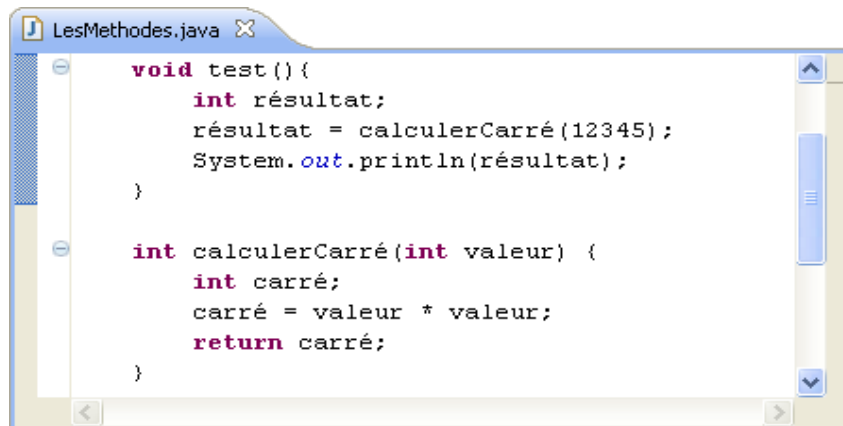
## ■ Déclaration

### ■ Fonction

```
type maFonction(type paramètre1, type paramètre2){  
    // code source de la fonction  
    return valeur;  
}
```

### ■ Procédure

```
void maProcédure(type paramètre1, type paramètre2){  
    // code source de la procédure  
}
```



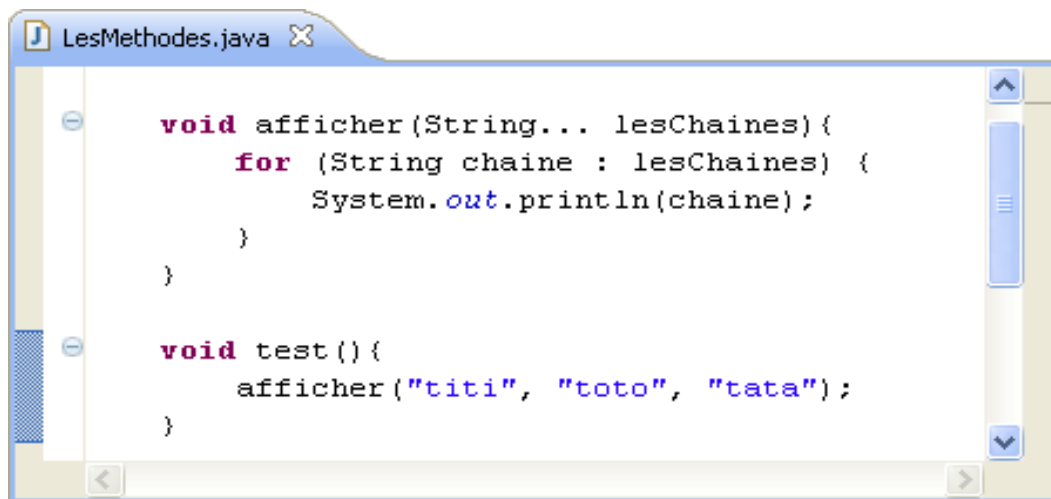
```
LesMethodes.java X  
  
void test(){  
    int résultat;  
    résultat = calculerCarré(12345);  
    System.out.println(résultat);  
}  
  
int calculerCarré(int valeur) {  
    int carré;  
    carré = valeur * valeur;  
    return carré;  
}
```

# Les méthodes : les paramètres

- Les paramètres formels et effectifs
  - Aspect statique : valeurs potentielles
  - Aspect dynamique : valeurs effectives
- Passage par valeur et par référence
  - Passage par valeur des types simples : copie
    - Les valeurs dans et hors de la méthode sont indépendantes
  - Passage par référence des types complexes : copie de la référence
    - Toute la structure de donnée n'est pas copiée
    - Réaffecter la copie de la référence n'influence pas la structure d'origine

# Les méthodes : l'ellipse

- Permet d'avoir un nombre variable de paramètres
- Est un raccourci de syntaxe pour l'utilisation d'un tableau
- Expression : `Type... paramètre`



```
LesMethodes.java X

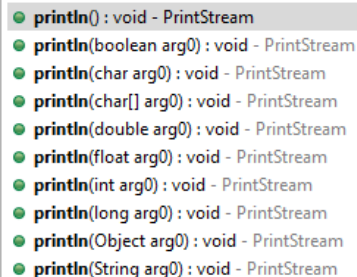
void afficher(String... lesChaines){
    for (String chaine : lesChaines) {
        System.out.println(chaine);
    }
}

void test(){
    afficher("titi", "toto", "tata");
}
```

# Les méthodes : Surcharge

- Plusieurs méthodes de même identifiant
- Signatures différentes (en nombre et/ou en types de paramètres)
  - Attention : le type de retour de la méthode ne fait pas partie de la signature
- Unité sémantique
- Souvent, réutilisation entre les méthodes surchargées
- Parfois appelé "polymorphisme paramétrique"

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println  
  
    }  
  
}
```

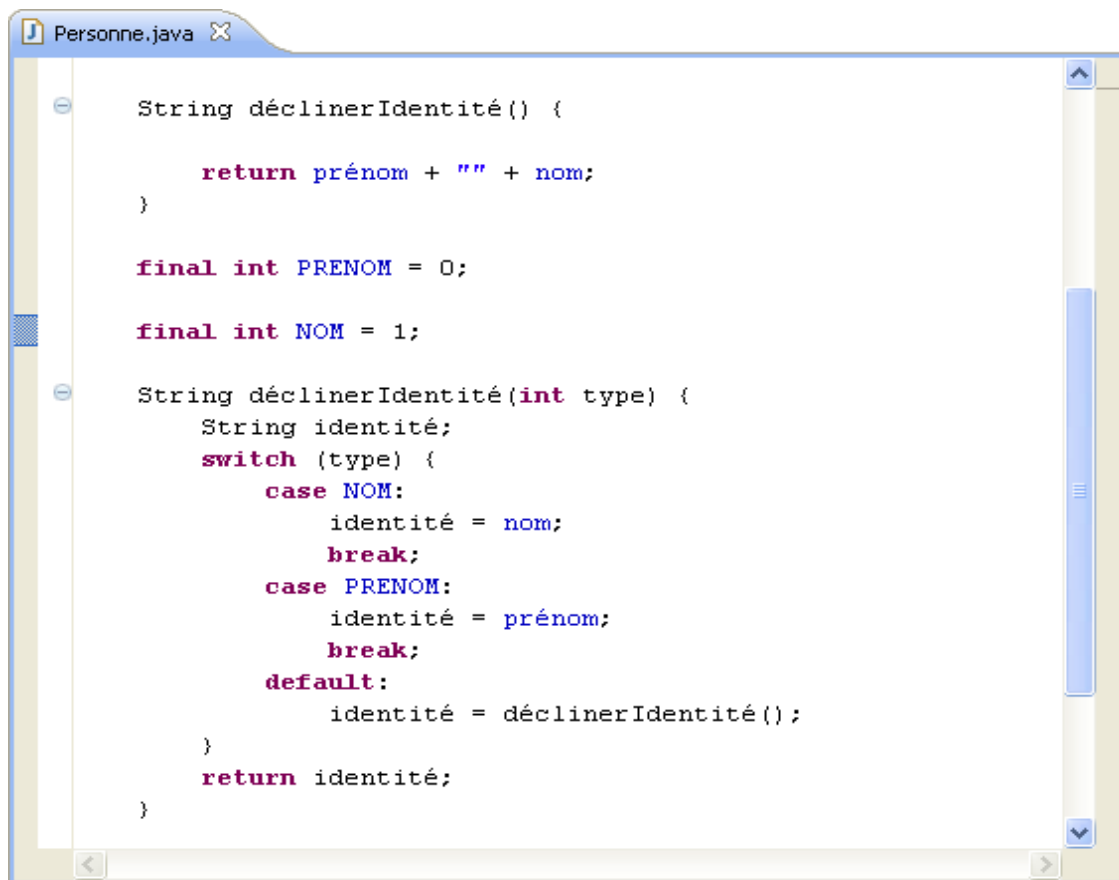


A screenshot of an IDE's auto-completion menu for the `println` method. The menu lists ten overloads, each with a green circular icon to its left. The overloads are: `println() : void - PrintStream`, `println(boolean arg0) : void - PrintStream`, `println(char arg0) : void - PrintStream`, `println(char[] arg0) : void - PrintStream`, `println(double arg0) : void - PrintStream`, `println(float arg0) : void - PrintStream`, `println(int arg0) : void - PrintStream`, `println(long arg0) : void - PrintStream`, `println(Object arg0) : void - PrintStream`, and `println(String arg0) : void - PrintStream`. The first option, `println() : void - PrintStream`, is highlighted with a light gray background.

Press 'Ctrl+Space' to show Template Proposals



# Les méthodes : implémentation



```
Personne.java X

String declinerIdentité() {

    return prénom + " " + nom;
}

final int PRENOM = 0;

final int NOM = 1;

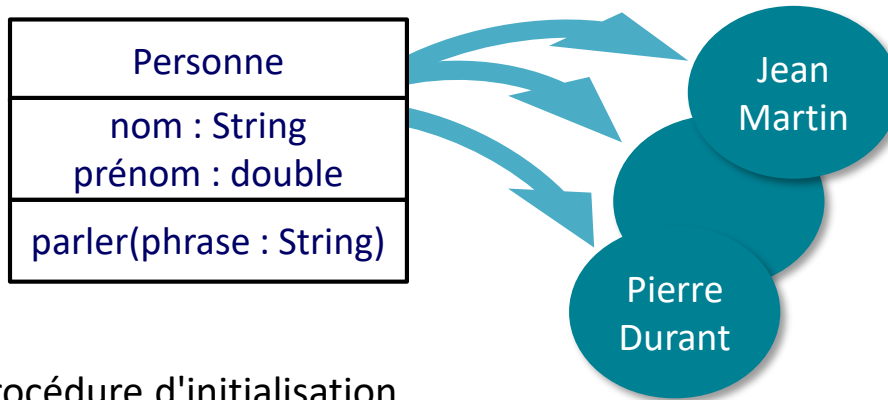
String declinerIdentité(int type) {
    String identité;
    switch (type) {
        case NOM:
            identité = nom;
            break;
        case PRENOM:
            identité = prénom;
            break;
        default:
            identité = declinerIdentité();
    }
    return identité;
}
```

# Classe et instance

- La classe
  - Concept
  - Vision statique
  - Ne "fait" rien
    - ⇒ Nécessité de rendre actif tout ce qui est décrit
    - ⇒ Utiliser le point d'entrée du programme qui réalise des traitements
- L'instance
  - Instance = objet = exemplaire  
ex : La voiture immatriculée "123 ABC 45"
  - Vision dynamique
  - Indépendance des "vies" des instances

# Le constructeur

- Mécanisme qui utilise la classe comme une fabrique à objets



- Procédure d'initialisation
  - Même identifiant que la classe
  - Pas de type de retour
  - Construit et initialise des objets (instances)
  - Pont entre le concept et la réalité
  - Opérateur `new`

# Le constructeur (suite)

```
Programme.java X
public class Programme {

    public static void main(String args[]) {
        Personne p = new Personne();
        p.parler("Bonjour");
    }
}
```

Appel sur la classe :  
C'est à la classe qu'on demande un  
nouvel exemplaire

```
Personne.java X
public class Personne {
    String nom;
    String prénom;

    Personne(){
        nom = "Dupont";
        prénom = "Michel";
    }

    void parler(String phrase){
        System.out.println( prénom + nom + "dit :" + phrase);
    }
}
```

Initialisation sur l'instance

# Le constructeur (suite)

- Examen de l'instruction de construction

```
Personne p | = | new Personne ();
```

1                      3                      2

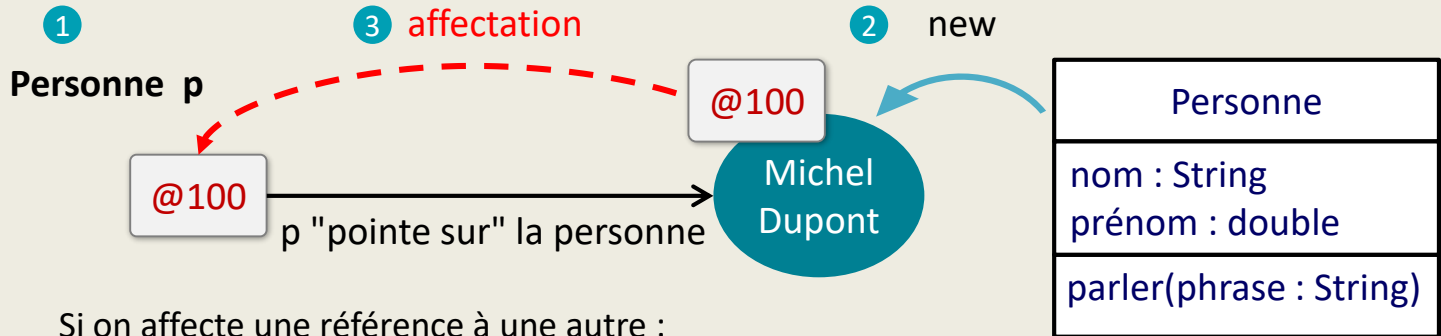
- 1 Création d'une "variable de manipulation d'une Personne"  
Pas de création de `Personne`. Valorisée à `null`
- 2 Création d'une nouvelle instance de `Personne`. Cela se traduit par une réservation de mémoire à une adresse (par ex: @100)
- 3 Affectation de la `Personne` nouvellement créée à la variable `p` par copie de l'adresse



"p" n'est pas l'objet, mais un moyen de le manipuler

# Le constructeur (suite)

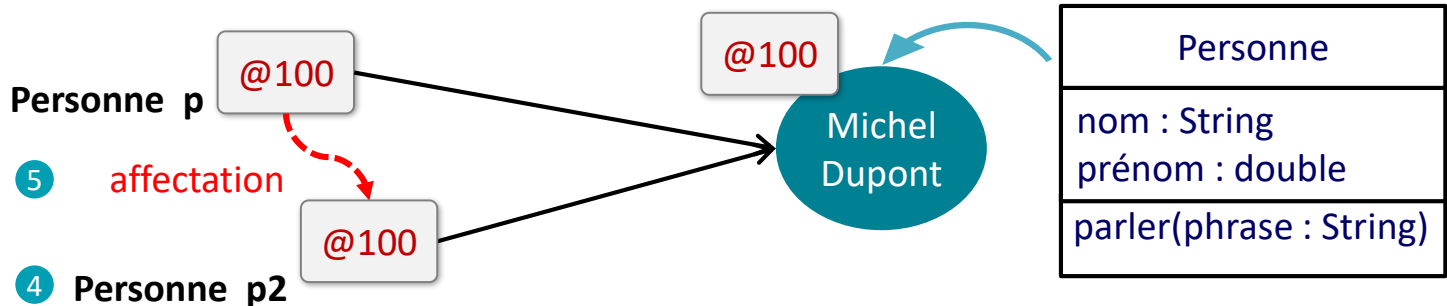
## Référence aux objets



Si on affecte une référence à une autre :

**4** `Personne p2;`

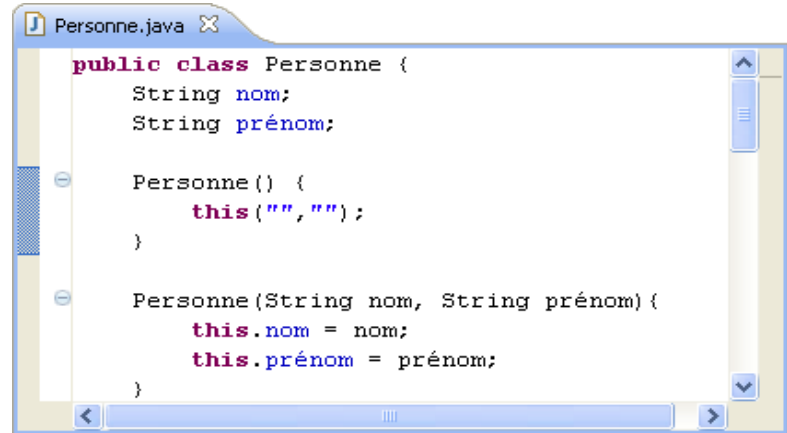
**5** `p2 = p;`



**p et p2 manipulent le même objet !**

# Le constructeur (suite)

- Surcharge des constructeurs
  - Possible comme sur les méthodes
  - Permet de proposer des constructeurs plus simples ou au contraire plus complets
  - Appel d'un constructeur à l'autre possible
    - `this()`
    - Il n'y a qu'un objet créé
    - `this()` réutilise seulement le bloc d'initialisation
    - Réutilisation du code



```
Personne.java
public class Personne {
    String nom;
    String prénom;

    Personne() {
        this("", "");
    }

    Personne(String nom, String prénom) {
        this.nom = nom;
        this.prénom = prénom;
    }
}
```

# Le constructeur (suite)

- Le constructeur par défaut
  - Défini tant qu'un autre constructeur n'est pas exprimé
  - ⇒ Écrire un autre constructeur "efface" le constructeur par défaut
  - C'est un constructeur sans paramètre
  - Il ne fait rien

```
Personne () { }
```

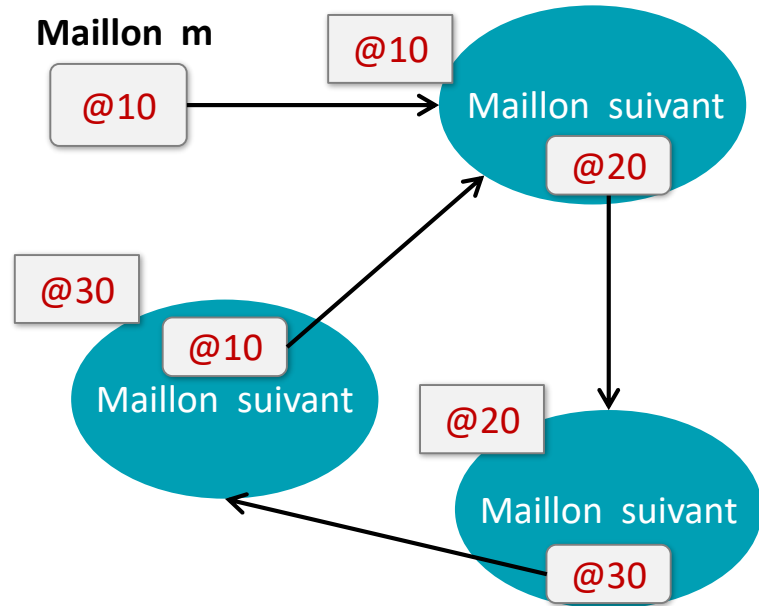


# Le destructeur

- Construction donc destruction
- Préoccupation d'usage mémoire
- En Java : le destructeur ne peut pas être appelé explicitement
- Processus de fond : le garbage collector (ramasse-miettes)
- Système de comptage des références : si le nombre est à 0, l'objet est orphelin donc suppression
- Permet (presque) de s'affranchir de la réflexion : qui est responsable de la destruction de l'objet ?

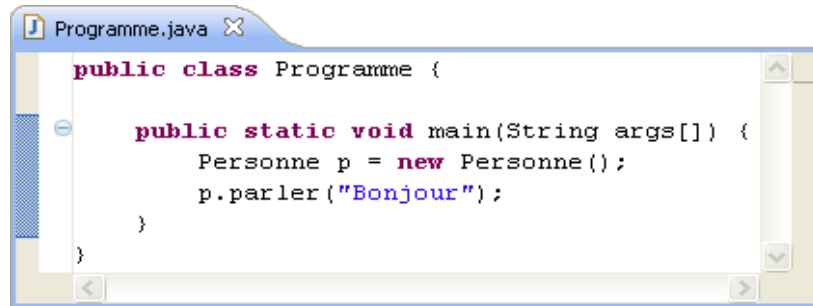
# Le destructeur

- Références circulaires
  - Si "m" disparaît :
    - La chaîne de maillons est orpheline (grappe d'objets)
    - Pris séparément, chaque maillon n'est pas orphelin
  - ⇒ Le garbage collector ne nettoie pas la mémoire des objets d'adresses 10 20 et 30
  - ⇒ Il faut "casser" au moins un lien : valeur `null` dans un des "suivant"

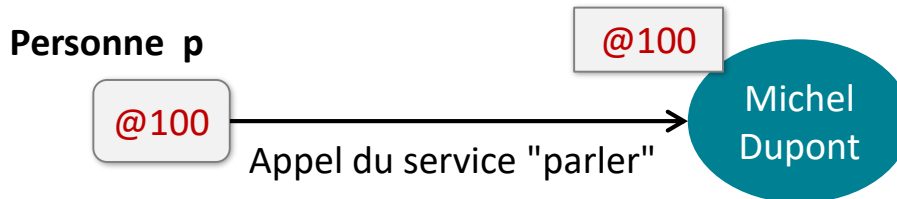


# Accès aux membres

- L'opérateur "."
  - S'applique à la référence sur l'objet
  - N'est possible que si la référence n'est pas nulle
- Envoi d'un message à un objet destinataire
- En réponse, l'objet déclenche un comportement

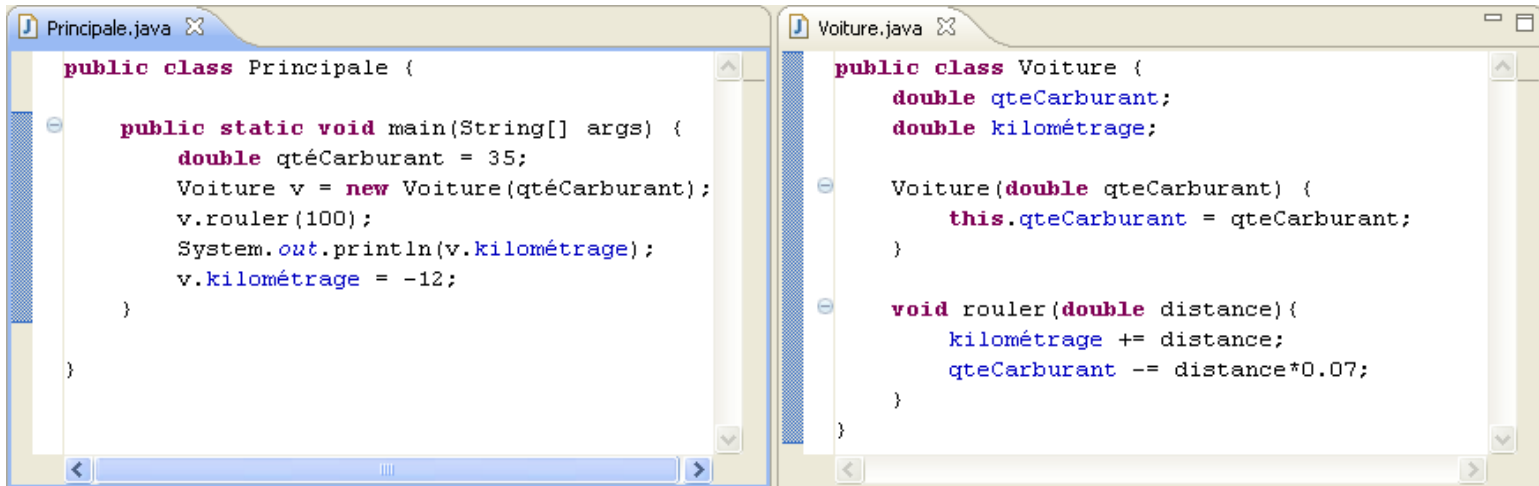


```
public class Programme {  
  
    public static void main(String args[]) {  
        Personne p = new Personne();  
        p.parler("Bonjour");  
    }  
}
```



# Un tout cohérent

- Exemple, que penser de :



```
Principale.java
public class Principale {

    public static void main(String[] args) {
        double qteCarburant = 35;
        Voiture v = new Voiture(qteCarburant);
        v.rouler(100);
        System.out.println(v.kilométrage);
        v.kilométrage = -12;
    }

}

Voiture.java
public class Voiture {

    double qteCarburant;
    double kilométrage;

    Voiture(double qteCarburant) {
        this.qteCarburant = qteCarburant;
    }

    void rouler(double distance){
        kilométrage += distance;
        qteCarburant -= distance*0.07;
    }

}
```

- La voiture doit rouler correctement
  - Modification de l'état de la voiture sans contrôle par celle-ci
- ⇒ Incohérence ⇒ Cas de violation d'encapsulation

# Un tout cohérent : L'encapsulation

⇒ L'objet doit se protéger

- Modificateurs de visibilité

- `private` : invisible à l'extérieur de la classe, même en lecture
- `public` : manipulable de l'extérieur de la classe, même en écriture

- Démarche : sauf bonne raison, on cache tout

- Si on veut autoriser la lecture, il faut passer par un service dédié : le *getter* (accesseur en lecture)

- Idem pour l'écriture : *setter* (accesseur en écriture). But : faire des contrôles de validité des valeurs

# Un tout cohérent : L'encapsulation

```
Principale.java
public class Principale {

    public static void main(String[] args) {
        double qteCarburant = 35;
        Voiture v = new Voiture(qteCarburant);
        v.rouler(100);
        System.out.println(v.getKilométrage());
        v.kilométrage = -12; // impossible
    }
}

Voiture.java
public class Voiture {
    private double qteCarburant;
    private double kilométrage;

    public Voiture(double qteCarburant) {
        this.qteCarburant = qteCarburant;
    }

    public void rouler(double distance) {
        kilométrage += distance;
        qteCarburant -= distance*0.07;
    }

    public double getQteCarburant() {
        return qteCarburant;
    }

    public void setQteCarburant(double qteCarburant) {
        if (qteCarburant > 0 && qteCarburant < 35) {
            this.qteCarburant = qteCarburant;
        }
    }

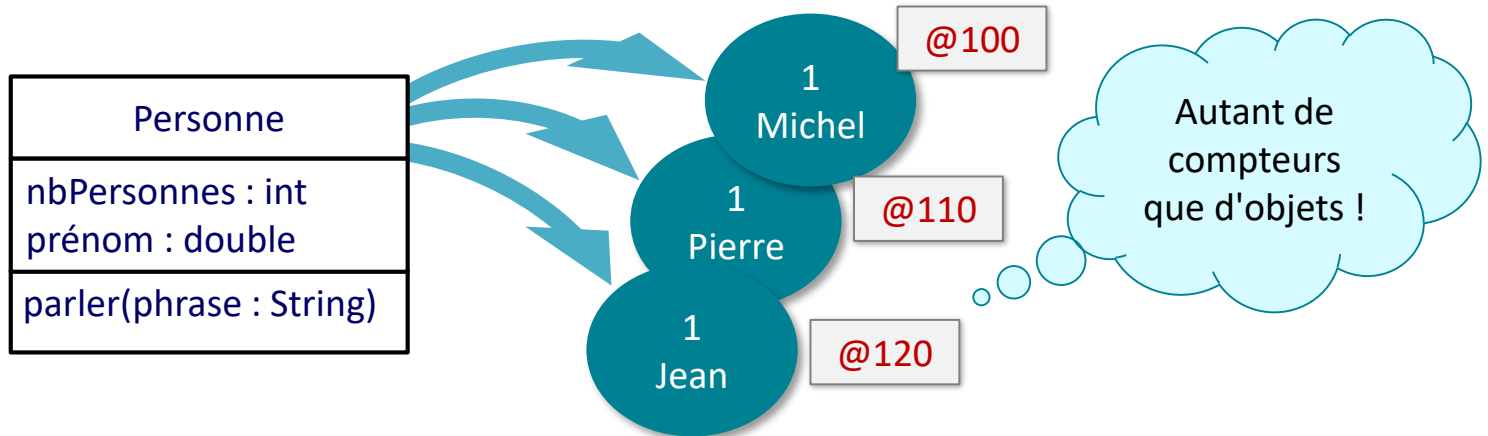
    public double getKilométrage() {
        return kilométrage;
    }
}
```

# Outils Eclipse

- Menu "Source"
  - Generate getters and setters
  - Generate constructor using fields
- Menu "Refactor"
  - Rename (effet sur les getters et setters)
  - Encapsulate field

# Membres de classe

- Question : comment mettre en place un système de comptage des personnes ?
- Quel problème rencontre-t-on ?



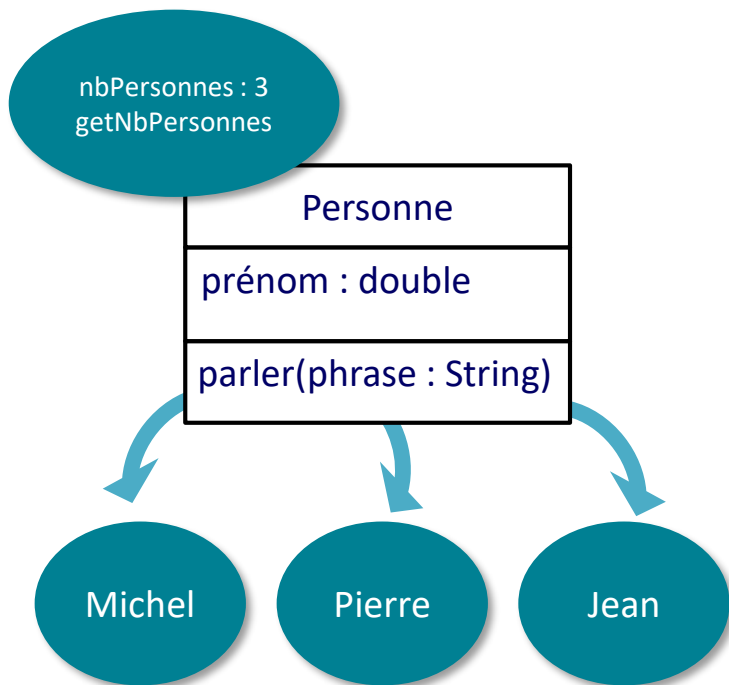
```
Personne(String prénom) {  
    this.prénom = prénom;  
    nbPersonnes ++;  
}
```



# Membres de classe (suite)

- ⇒ Parfois des données ne relèvent pas de l'instance mais de la classe
- Membre de classe  $\neq$  membre d'instance
  - Mot-clé `static` = "de classe"
  - Propriétés de classe
  - Méthodes de classe
- Les méthodes d'instance peuvent accéder aux membres de classe
- Les méthodes de classe ne peuvent pas accéder aux membres d'instance

# Membres de classe (suite)



```
Personne.java X
public class Personne {
    // membres de classe
    private static int nbPersonnes;
    public static int getNbPersonnes() {
        return nbPersonnes;
    }

    // membres d'instance
    private String nom;
    private String prénom;

    Personne(String nom, String prénom) {
        this.nom = nom;
        this.prénom = prénom;
        nbPersonnes++;
    }
}
```

# Membres de classe (suite)

## Accès dans la classe

```
Personne.java X
private static int nbPersonnes;

Personne(String nom, String prénom){
    this.nom = nom;
    this.prénom = prénom;
    // accès au membre de classe depuis l'instance
    // sans conflit de portée
    nbPersonnes++;
}

public static void setNbPersonnes(int nbPersonnes){
    // accès au membre de classe depuis la classe
    // avec conflit de portée
    Personne.nbPersonnes = nbPersonnes;
}
```

## Accès hors de la classe

```
Programme.java X
public class Programme {

    public static void main(String args[]) {
        Personne p = new Personne();
        p.parler("Bonjour");

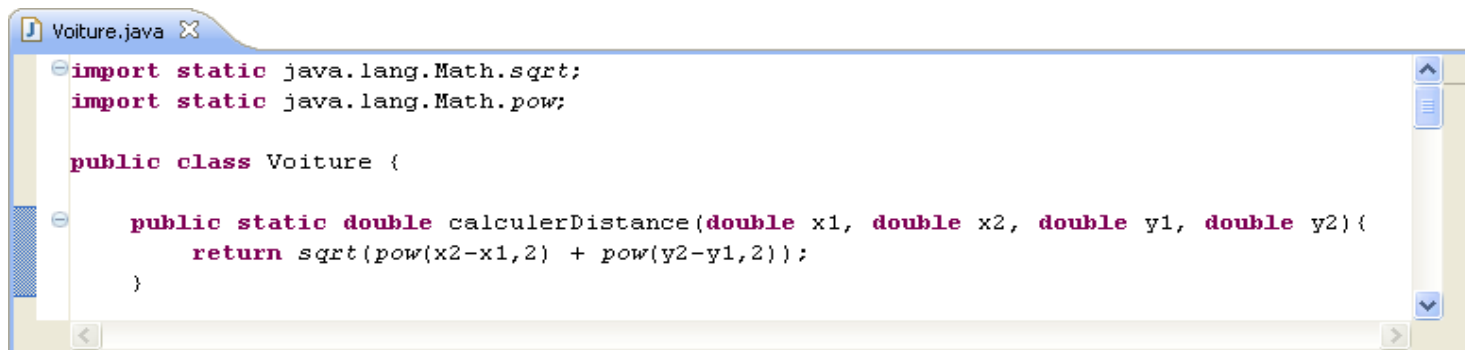
        int nb = Personne.getNbPersonnes();
        System.out.println(nb);
    }
}
```

# Membres de classe (suite)

- Stratégies d'implémentations
    - Membre de classe = mutualisation de code
    - Mettre `static` tout ce qui peut l'être
      - Constantes
      - Méthodes dont le résultat ne dépend pas de l'état de l'instance
    - Transformer les méthodes liées à l'instance en méthodes `static` quand c'est possible
- ex : calcul de la distance entre l'emplacement actuel de la voiture et la destination :  
calcul de la distance entre deux points

# Membres de classe (suite)

- Classes utilitaires
  - Ne contiennent que des membres de classe
  - Exemple : `java.lang.Math`
- Imports statiques
  - Nouveauté du jdk 1.5 : mot-clé `import static`
  - Permet d'éviter des expressions fastidieuses  
ex : `Math.sqrt(Math.pow(x2-x1, 2) + Math.pow(y2-y1, 2))`
  - Portent sur les membres
  - Attention aux conflits de noms



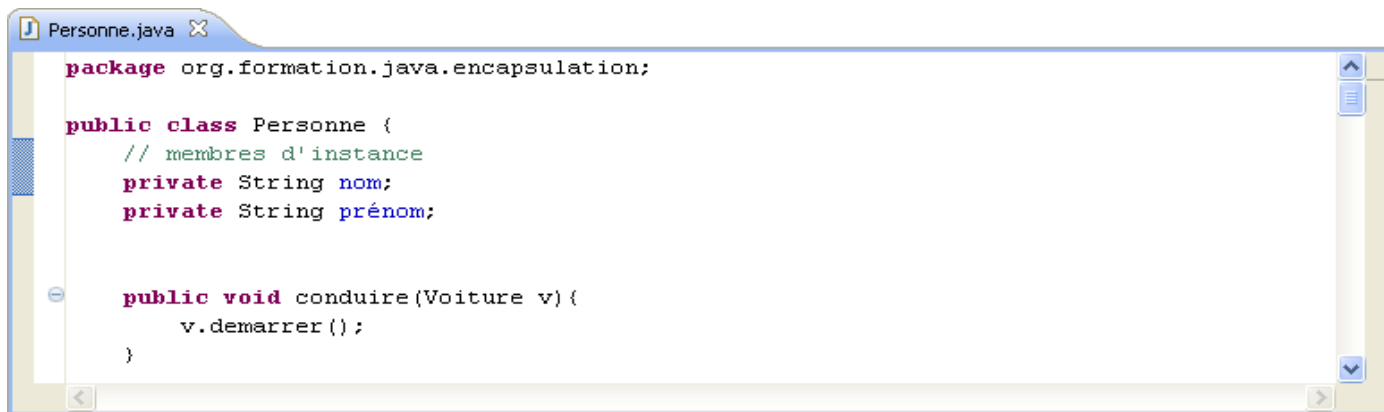
```
Voiture.java X
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

public class Voiture {

    public static double calculerDistance(double x1, double x2, double y1, double y2){
        return sqrt(pow(x2-x1,2) + pow(y2-y1,2));
    }
}
```

# Les collaborations

- Dépendance
  - Forme générale de collaboration
  - Forme de collaboration la plus faible
  - Connaissance via `import`
  - Pas de référence permanente : pas de relation conteneur-contenu



```
Personne.java X
package org.formation.java.encapsulation;

public class Personne {
    // membres d'instance
    private String nom;
    private String prénom;

    public void conduire(Voiture v) {
        v.demarrer();
    }
}
```

# Les collaborations (suite)

- Relation conteneur-contenu
  - Construire des objets complexes à partir d'objets simples
  - Différentes sortes de collaborations  
Ex : quelle différence sémantique entre "une voiture contient des passagers" et "une voiture contient un moteur"
- Composition
  - Relation "forte"
  - Les cycles de vies des objets sont liés
    - La construction du conteneur implique la construction du contenu
    - La destruction du conteneur provoque la destruction du contenu
    - La destruction du contenu dégrade le conteneur
  - Exemple : Voiture - Moteur
  - Notion de "possession"

# Les collaborations (suite)

- Agrégation
  - Relation conteneur-contenu "faible"
  - Les cycles de vies des objets ne sont pas liés
    - La construction du conteneur est indépendante de celle du contenu
    - La destruction du conteneur ne provoque pas la destruction du contenu
    - Le contenu est facultatif dans le conteneur
  - Exemple : Voiture - Passager
- En Java, à la différence du C++, mêmes déclarations
- Précautions relatives à la composition
  - Charge au programmeur de synchroniser les cycles de vie pour la composition, dans le constructeur
  - Le conteneur ne donne pas accès au contenu ⇒ violation d'encapsulation
  - Le problème est caduque pour l'agrégation car le conteneur ne possède pas le contenu



# Les collaborations (suite)

```
Principale.java X
public static void main(String[] args) {
    Personne unePersonne = new Personne();

    Voiture uneVoiture = new Voiture(35.0, 70);

    unePersonne.conduire(uneVoiture);
}
```

```
Personne.java X
package org.formation.java.encapsulation;

public class Personne {
    // membres d'instance
    private String nom;
    private String prénom;

    public void conduire(Voiture v) {
        v.setConducteur(this);
        v.demarrer();
    }
}
```

```
Voiture.java X
public class Voiture {

    // pas de différence de déclaration entre
    // composition
    private Moteur moteur;
    // et agrégation
    private Personne conducteur;

    public Voiture(double qteCarburant, int puissance) {
        moteur = new Moteur(puissance);
        this.qteCarburant = qteCarburant;
    }

    public void setConducteur(Personne conducteur) {
        this.conducteur = conducteur;
    }

    public int getPuissance() {
        return moteur.getPuissance();
    }

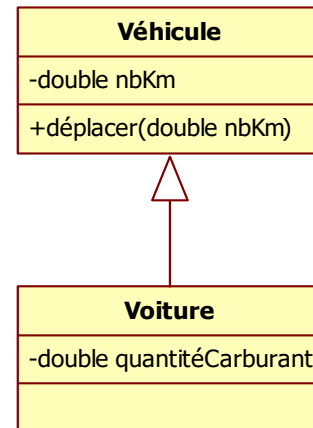
    public void demarrer() {
        if (conducteur != null) {
            setEtat(MARCHE);
        }
    }
}
```

# L'héritage en Java

- Concept issu d'observations du réel
  - L'esprit humain classe naturellement les concepts : classification du règne animal, etc.
  - Besoin de rattacher un concept précis à un autre plus général  
ex : une voiture est un véhicule, une moto est un véhicule
- Avantages
  - Gain de temps en ne rappelant pas les qualités déjà admises  
ex : une moto est un véhicule motorisé à 2 roues
  - Abstraction par l'assimilation de plusieurs concepts à un seul  
ex : Tous les véhicules peuvent se déplacer
  - ⇒ Permet de se focaliser sur l'essentiel, tout en conservant la nature spécifique des objets.

# L'héritage en Java (suite)

- Expression UML :
  - Flèche : sens de généralisation
  - Voiture spécialise Véhicule
  - Véhicule est une généralisation de Voiture
- Héritage au sens patrimonial
  - Ex : une voiture est un véhicule
  - Une instance de Voiture contient tout d'une instance de Véhicule
  - Les membres statiques ne sont pas hérités (ne sont pas dans l'instance)
- Expression Java : mot clé `extends`



The screenshot shows two overlapping Java code editors. The left editor, titled `Véhicule.java`, contains the following code:

```
package org.formation.java.heritage;

public class Véhicule {
    private double nbKm;

    public void déplacer(double distance) {
        nbKm += distance;
    }
}
```

The right editor, titled `Voiture.java`, contains the following code:

```
package org.formation.java.heritage;

public class Voiture extends Véhicule {
    private double quantitéCarburant;
}
```

# L'héritage en Java (suite)

- Permet d'avoir une vision générale d'objets spécifiques
  - Ex : les véhicules se déplacent (peu importe leur nature spécifique)
  - Simplification : on ne s'intéresse qu'au minimum nécessaire (abstraction)

```
Véhicule tab[] = new Véhicule[3];  
tab[0] = new Voiture();  
tab[1] = new Moto();  
tab[2] = new Vélo();  
for(Véhicule v : tab){  
    v.déplacer(10);  
}
```

- Classe mère de toutes les autres : `java.lang.Object`

# Exemple : l'API Java

- Classe de base `Object` : héritage implicite dessus

Overview Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

## Class ArrayList<E>

java.lang.Object  
    java.util.AbstractCollection<E>  
        java.util.AbstractList<E>  
            java.util.ArrayList<E>

**All Implemented Interfaces:**

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

**Direct Known Subclasses:**

AttributeList, RoleList, RoleUnresolvedList

# Le transtypage

- Référencer une instance d'une classe fille par une variable sur une classe mère est toujours possible

```
Véhicule véhicule = new Voiture(); //ok
```

- L'assimilation d'une instance d'une classe mère à une classe fille n'est pas possible

```
Voiture voiture = new Véhicule(); // KO
```

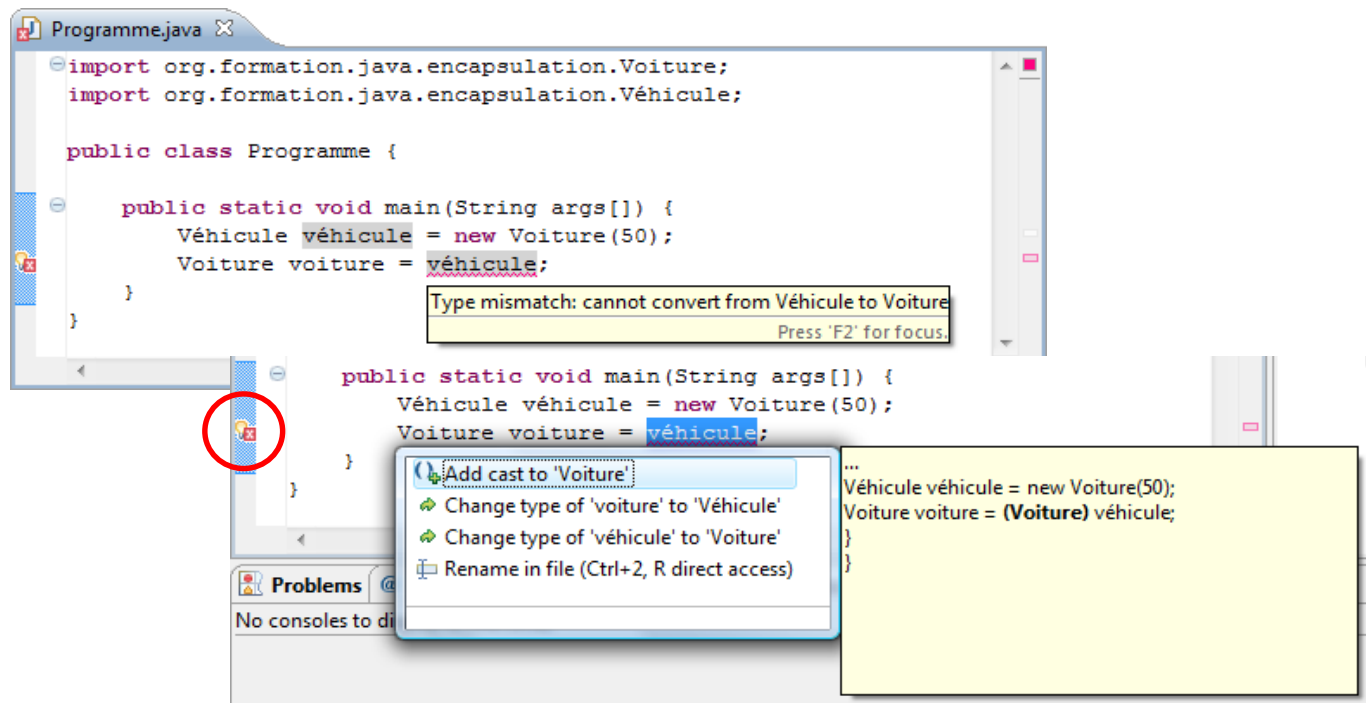
- Une instance ne change pas de nature bien que référencée par une variable d'un type parent

- On peut alors retourner vers un type enfant par une opération de transtypage explicite (cast)

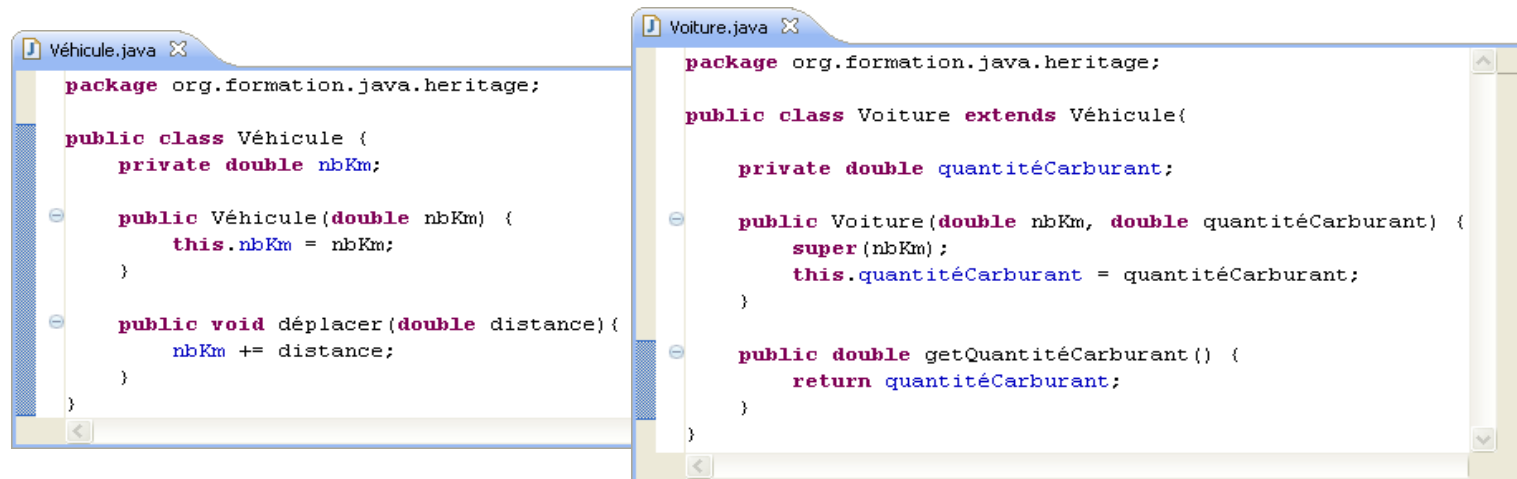
```
Voiture voiture = (Voiture) véhicule; // ok
```

# Eclipse et le transtypage

- Eclipse propose de corriger les problèmes de transtypages :



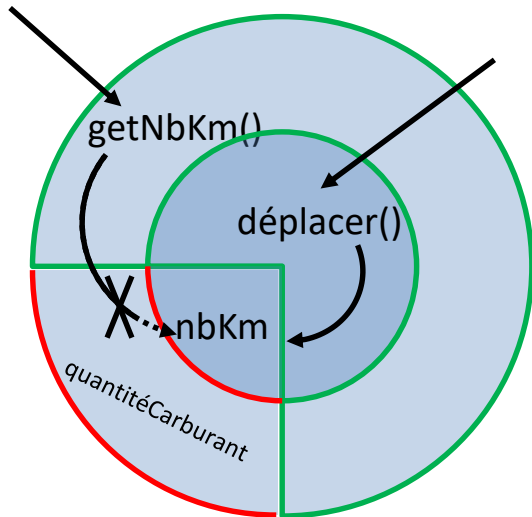
# Réutilisation de code



- Constructeur
  - Appel automatique au super-constructeur par défaut
  - Sinon, appel explicite au super-constructeur voulu : `super()`
  - Réutilisation du bloc d'initialisation, pas de deuxième instance



# Visibilité des membres



```
Voiture.java X
package org.formation.java.heritage;

public class Voiture extends Véhicule{

    private double quantitéCarburant;

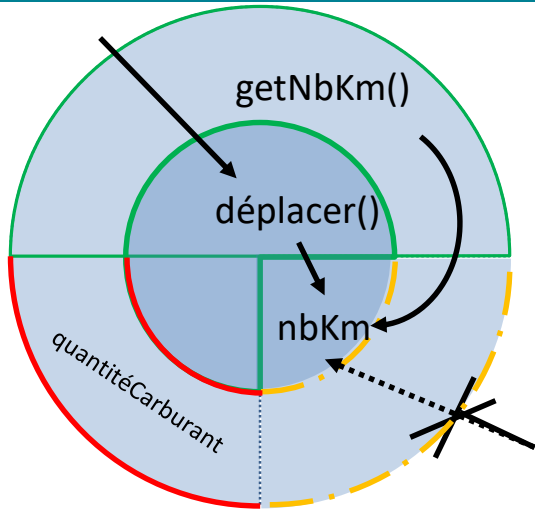
    public Voiture(double nbKm, double quantitéCarburant) {
        super(nbKm);
        this.quantitéCarburant = quantitéCarburant;
    }

    public double getNbKm() {
        return nbKm;
    }
}
```

The field Véhicule.nbKm is not visible  
Press 'F2' for focus.

- Bien qu'elle en soit constituée, la classe Voiture ne peut pas voir nbKm car déclaré privé dans Véhicule
- Il existe un niveau intermédiaire, `protected`, qui permet de n'autoriser l'accès qu'aux classes dérivées

# Visibilité des membres (suite)



```
Véhicule.java
package org.formation.java.heritage;

public class Véhicule {
    protected double nbKm;

    public Véhicule(double nbKm) {
        this.nbKm = nbKm;
    }

    public void déplacer(double distance) {
        nbKm += distance;
    }
}
```

- Les objets extérieurs n'ont pas accès aux attributs protégés
- Prévoir l'utilisation future que les classes dérivées pourront faire des attributs
  - Bonne pratique : mettre les attributs en `protected` par défaut
  - Révéler les attributs (`public`) au coup par coup
  - Cacher (`private`) si on peut affirmer qu'aucune sous-classe ne peut prendre la responsabilité de modifier l'attribut
- En Java, `protected` ouvre la visibilité aussi aux classes non apparentées du même package (équivalent des classes amies du C++). C'est un cas de violation de l'encapsulation.

# Le polymorphisme

- Partager des méthodes communes n'est pas suffisant  
ex : une voiture ne se déplace pas comme un bateau
- ⇒ Classes spécifiques = comportement spécifique
- ⇒ Mais garder une forme d'appel homogène  
ex : tous les Véhicules savent se déplacer sur une distance, mais pas de la même façon

Polymorphisme : un message homogène, pour des comportements différenciés



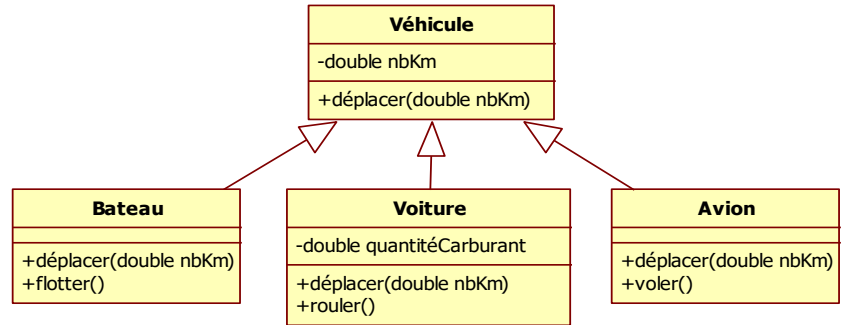
**Polymorphisme**  
Un message homogène, pour des comportements différenciés

# Le polymorphisme (suite)

- On veut manipuler un tableau

```
for (Véhicule v : tab){  
    v.déplacer(10);  
}
```

- Mais que déplacer soit spécifique selon que l'instance est de nature Bateau ou autre



```
package org.formation.java.heritage;

public class Véhicule {

    protected double nbKm;

    public Véhicule(double nbKm) {
        this.nbKm = nbKm;
    }

    public void déplacer(double distance) {
        nbKm += distance;
    }
}

package org.formation.java.heritage;

public class Voiture extends Véhicule{

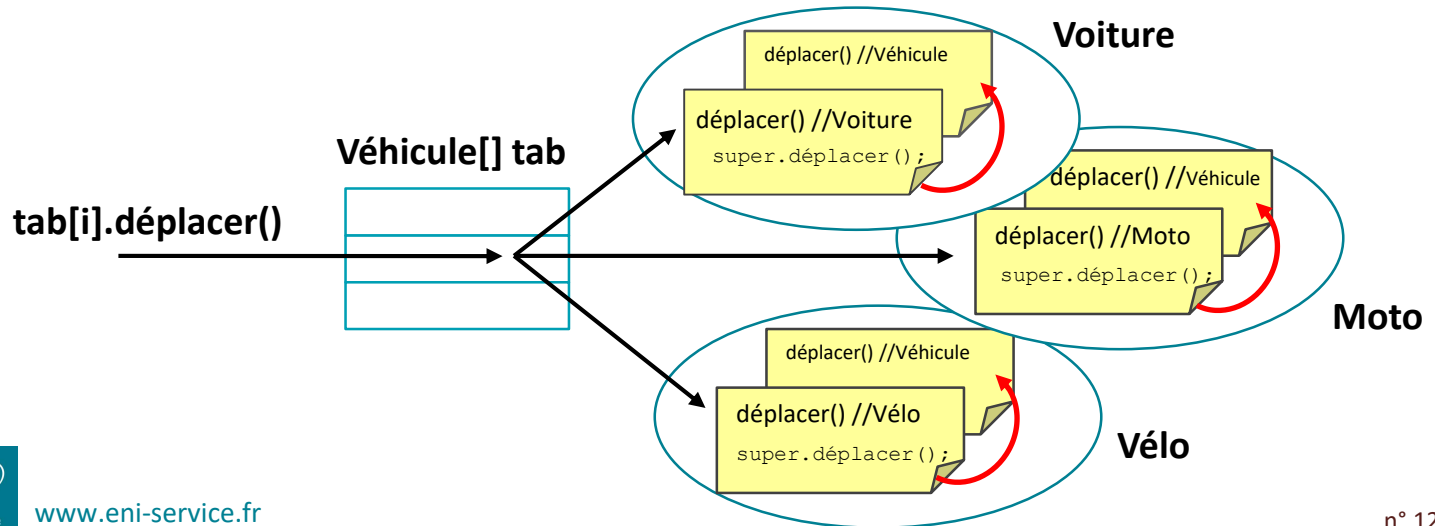
    private double quantitéCarburant;

    public Voiture(double nbKm, double quantitéCarburant) {
        super(nbKm);
        this.quantitéCarburant = quantitéCarburant;
    }

    public void déplacer(double distance) {
        quantitéCarburant -= distance*0.07;
    }
}
```

# Le polymorphisme (suite)

- En Java, réécriture d'une méthode qui possède exactement la même signature que la méthode héritée
  - Les 2 méthodes restent présentes dans l'instance
  - La nouvelle méthode masque l'ancienne
  - Lors de l'appel sur une instance, c'est la dernière définie qui est immédiatement disponible
  - La méthode redéfinie peut appeler la méthode héritée via "super." ⇒ Réutilisation de code
- ex : `super.déplacer()` ;
- Mais elle ne peut pas "sauter une génération"
- Ex : `super.super.déplacer()` ; //incorrect



Les fondamentaux de la programmation Java (Java SE)

## Module 5

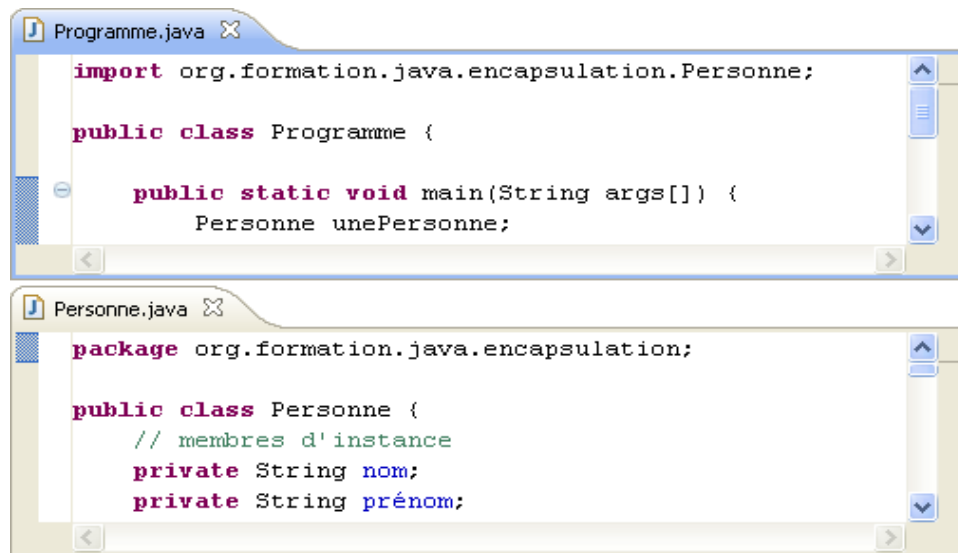
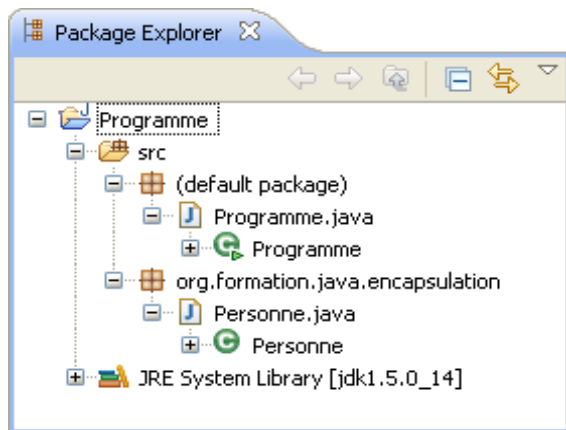
# Concepts avancés de programmation Java

# Les packages

- Multiplicité des classes - fichiers
- Nécessité de rangement
  - Niveau physique : répertoires
  - Niveau logique : packages
- Nomenclature des noms de packages
  - Convention : Le nom de domaine Internet constitue une base
    - com.sun ..., fr.eni ...
- Une classe doit déclarer le package auquel elle appartient
  - Mot clé « package »
- Une classe connaît celles qui sont dans le même package
- Une classe doit spécifier les classes d'autres packages auxquelles elle veut avoir accès
  - Nommage complet (FQCN : Fully-Qualified Class Name) : java.util.ArrayList
  - Importation : Mot clé « import »
  - Pas d'importation des classes du package « java.lang » (String, System, ...)

# Les packages

- La directive d'importation peut porter sur une classe :
  - `import org.formation.java.encapsulation.Personne;`
- Ou sur un package complet :
  - `import org.formation.java.encapsulation.*;`
    - Pas de récursivité !





# Les classes abstraites

- Pas d'exemplaire pour le concept
  - Empêcher l'appel au constructeur par `new`
  - Forcer l'héritage
  - But : factoriser et réutiliser du code
  - Améliorer la conformité à la réalité  
ex : il n'existe pas de véhicule qui ne soit que véhicule mais, ni une voiture, ni un exemplaire d'aucune des sous-classes
  - ⇒ Syntaxe : `public abstract class`
- Le bloc constructeur reste pertinent
  - Appel via `super()` dans les classes filles
  - Sert de bloc d'initialisation factorisé

# Les méthodes abstraites

- Cas de définition incomplète
  - Par ex : un véhicule consomme de l'énergie pour son déplacement
  - Mais les modalités de consommations sont inconnues à ce stade
  - ⇒ Ça dépend du type de véhicule, i.e. de la sous-classe réelle
  - ⇒ `protected abstract void consommer(double distance);`
- Par conséquent, la classe est abstraite
  - pas d'implémentation de consommer dans Véhicule
  - Impossible de créer une instance incomplète
- Soit les classes dérivées donnent une implémentation
- Soit elles sont elles-mêmes abstraites

# Les classes finales

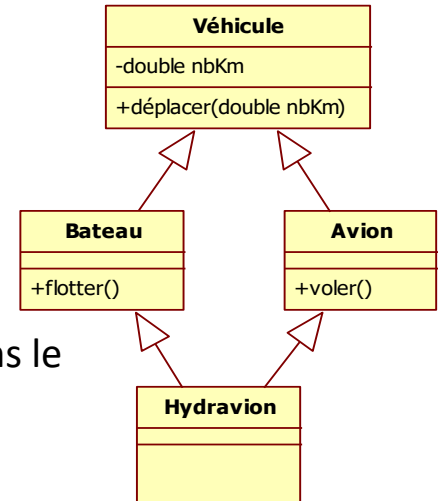
- Logique de propriété/économique/contractuelle :
  - Empêcher de réutiliser nos classes par héritage
  - Garantit que la classe finale ne sera jamais classe mère
    - ⇒ Syntaxe : `public final class`
    - ⇒ Ne peut être utilisé conjointement avec `abstract` (objectifs contraires)
- ⇒ N'empêche pas la réutilisation par composition
- ⇒ Oblige à revenir sur le code pour étendre les capacités par héritage
- Méthodes finales : granularité plus fine
  - Interdire la redéfinition d'une méthode dans les classes filles
  - Syntaxe (exemple) : `public final void f(){...}`

# Outils Eclipse

- Menu "Source"
  - Override/implement methods
  - Generate constructor from superclass
  - Generate constructor using fields
- Menu "Refactor"
  - Pull up
  - Push down
  - Extract superclass
  - Use supertype where possible

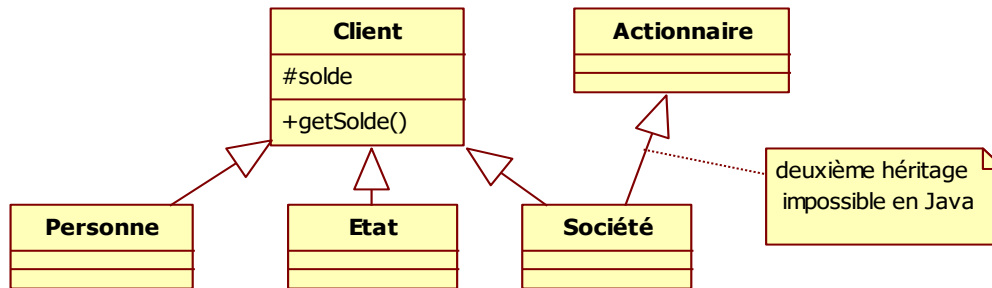
# Héritage multiple

- Conceptuellement possible
  - Ex : un amphibie tient de la voiture et du bateau, etc.
  - C++, SmallTalk, Eiffel, ... permettent de faire cela
- Problème de l'ancêtre commun
  - Toujours un ancêtre commun (au moins `Object`)
  - Méthodes communes héritées 2 fois
  - Laquelle mettre en œuvre
  - Syntaxe pour effectuer le choix
- Constat : source de d'erreurs et d'incompréhension dans le code
- Java choisit de ne pas permettre d'héritage multiple
  - Choix de la branche d'héritage peu aisée parfois
  - Redondance de code pour combler le manque
  - Alternative partielle : les interfaces



# Les interfaces

- Problème : comment gérer des listes d'objets hétérogènes mais qui ont des capacités communes  
ex : une banque fait un traitement sur le solde de ses clients (personnes, sociétés, états).
- Erreur : Rendre Client mère de Personne, Société, Etat
- Les enfants n'ont plus d'autre héritage possible  
ex : Société actionnaire
- Ce qui rapproche les instances dans le traitement : c'est leur capacité à fournir un solde, en non pas telle ou telle nature



# Les interfaces (suite)

- Ressemble à une classe totalement abstraite
- Différence fondamentale :  
La classe exprime la nature de l'objet  
L'interface marque la capacité
- Une interface est un contrat
  - Contrat : liste de signatures de méthodes publiques
  - Une classe peut respecter des contrats :
    - Déclaration de rattachement à l'interface : **implements**
    - Fourniture d'une implémentation pour chaque méthode de l'interface
- Syntaxe  

```
public interface MonInterface{}
```
- Comme pour les classes : une interface par fichier

# Les interfaces (suite)

```
Principale.java X
public class Principale {

    public static void main(String[] args) {
        Banque banque = new Banque();
        banque.ajouterClient(new Personne());
        banque.ajouterClient(new Societe());
        banque.traiter();
    }
}
```

```
Societe.java X
public class Societe implements Client{

    protected double solde=100;

    public double getSolde() {
        return solde;
    }
}
```

```
Client.java X
public interface Client {

    double getSolde();
}
```

```
Personne.java X
public class Personne implements Client{

    protected double solde=10;

    public double getSolde() {
        return solde;
    }
}
```

```
Banque.java X
public class Banque {

    protected ArrayList<Client> clients =
        new ArrayList<Client>();

    public void traiter(){
        for (Client c : clients) {
            System.out.println(c.getSolde());
        }
    }

    public void ajouterClient(Client c){
        clients.add(c);
    }
}
```



# Les interfaces (suite)

- Solution aux problèmes de mutation de types

- Une instance ne peut changer de nature
- Traduit une erreur de design

ex : une Personne qui est prise soit comme client, soit comme actionnaire. L'héritage sur la classe Client interdit un transtypage sur Actionnaire.

⇒ Client et Actionnaire traduisent des ROLES et non des natures

- Solution pour rendre les couches applicatives indépendantes

- Dans l'exemple précédent, la classe Banque est indépendante de la NATURE des instances de Client qu'elle gère
- Indépendance
- Modularité
- ⇒ Évolutivité

# Les interfaces (suite)

- Quand réaliser des interfaces ?
  - Besoin de rendre des modules indépendants
  - Indépendance des couches fonctionnelles par rapport aux couches système
  - Constat de problème de mutation de types
  - Tentative de réalisation d'héritage multiple
- Quand ne pas les réaliser ?
  - Potentiellement, toute méthode peut faire l'objet d'une interface
  - L'intérêt doit être avéré
  - Difficulté de prévoir les futurs besoins d'indépendance
  - Module de refactoring d'Eclipse : Extraction d'interface

# Les interfaces vides

- Quel intérêt si le contrat ne contraint à rien ?
- Sert de marqueur
  - Le concepteur choisit d'abonner sa classe à un rôle ou pas
  - Exemple : tant qu'il n'abonne pas sa classe à l'interface `Cloneable`, le concepteur interdit la duplication des instances
- Exemples
  - `Serializable` : autorise le mécanisme de sérialisation à transformer une instance mémoire en un flux binaire
  - `Cloneable` : autorise le mécanisme de duplication à faire une copie mémoire de l'instance

# L'interface Serializable

- Interface `java.io.Serializable`
  - `public MaClasse implements Serializable{}`
  - La sérialisation concerne tous les membres de l'instance
    - Types simples
    - Types complexes
      - Tableaux
      - Objets eux-mêmes sérialisables (sinon erreur : `NotSerializableException`)
  - ⇒ Sérialisation en cascade
  - Sauf membres marqués `transient`
- Classe `java.io.ObjectOutputStream`
  - Permet de sauvegarder un objet
  - "Écrivain" vers un flux
  - Constructeur `ObjectOutputStream(OutputStream out)` : prend le flux destination en paramètre (ex : `FileOutputStream`)
  - Méthode `writeObject(Object o)` : écrit l'objet (erreur : `NotSerializableException` si l'instance n'implémente pas l'interface)

# L'interface Serializable (suite)

- Classe `java.io.ObjectInputStream`
  - Permet de créer un objet à partir d'une sauvegarde
  - Lecteur de flux
  - Constructeur `ObjectInputStream(InputStream in)` : prend le flux destination en paramètre (ex : `FileInputStream`)
  - Méthode `Object readObject()` : lit l'objet sur le flux. Il faut transtyper la référence retour

# L'interface Cloneable

- Constat : l'affectation d'une référence à une autre ne duplique pas l'objet
- Comment faire une copie
- Méthode `clone()` sur `Object`
- Implémentation de l'interface `Cloneable` pour autoriser la duplication
- Éventuellement, modification de la visibilité de `clone()` en `public`
- Pas de clonage en cascade : redéfinition de `clone()` pour dupliquer les composants

# L'interface Comparable

- Permet de donner la notion d'ordre entre les instances d'une classe

```
public MaClasse implements Comparable<MaClasse>{}
```
- Implémentation de la méthode `int compareTo(T o)`
  - Utilisation :
    - `x.compareTo(y);`
    - `Collections.sort(liste, null);` // liste de comparables
  - Vaut 0 si les objets sont équivalents
  - Est < 0 si x est d'ordre inférieur à y
  - Est > 0 si x est d'ordre supérieur à y

# Les interfaces fonctionnelles

- Nouveauté de Java 8
- SAM Interfaces (Single Abstract Method Interfaces)
  - Une seule méthode abstraite
  - Runnable, Comparable, Comparator, ...
- Identifiables par l'annotation `@FunctionalInterface`
  - Pour le compilateur !

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```



# Interfaces : méthodes par défaut

- Depuis Java 8
- Implémentation « par défaut » de méthodes dans les interfaces
  - Les interfaces peuvent avoir du code d'implémentation !
  - Ces méthodes n'ont pas obligatoirement à être redéfinies
- Mot clé `default`
  - Remplace `abstract` !
- Permettent de faire évoluer les APIs en conservant une compatibilité ascendante
- L'API standard en profite grandement en enrichissant certaines de ses interfaces
  - Dans l'API de Collections dont les interfaces s'enrichissent de plusieurs méthodes

```
interface Person {  
    public abstract void sayGoodBye();  
    public default void sayHello() {  
        System.out.println("Hello there!");  
    }  
}
```

# Les expressions lambdas

- Plus grande nouveauté de Java 8
  - JSR-335
- Ajout de la programmation fonctionnelle en Java
- Peuvent être assimilées à des fonctions anonymes
  - Peuvent être affectées dans des interfaces fonctionnelles
  - Le code de l'expression sera l'implémentation de **LA** méthode
- Syntaxe :
  - (`<paramètres>`) -> `implémentation`
    - S'il n'y a qu'une seule instruction dans l'implémentation
  - (`<paramètres>`) -> { `implémentation` }
    - S'il y a plusieurs instructions

# Les expressions lambdas : exemple

- Prenons la méthode `sort()` de la classe `Collections`
  - `static <T> void sort(List<T> list, Comparator<? super T> c)`
- Sans les expressions lambdas avant Java 8, mise en œuvre possible (via une classe anonyme) :

```
Collections.sort(liste, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

# Les expressions lambdas : exemple (suite)

- Depuis Java 8, avec les lambdas, on peut écrire :

- Forme longue avec bloc et typage :

```
Collections.sort(liste, (String s1, String s2) -> { return s1.length() - s2.length(); } );
```

- Forme courte sans bloc (1 seule instruction) :

- Mot clé `return` implicite !

```
Collections.sort(liste, (String s1, String s2) -> s1.length() - s2.length() );
```

- Forme courte sans bloc et sans types :

- Inférence de type sur les paramètres

```
Collections.sort(liste, (s1, s2) -> s1.length() - s2.length() );
```

# Les expressions lambdas : autres exemples

- Avec Runnable :

```
Runnable r1 = () -> System.out.println("My Runnable");  
Thread t = new Thread(r1);  
t.start();
```

- Avec ActionListener :

```
JButton bouton = new JButton("Click me !");  
  
bouton.addActionListener((e) -> bouton.setText("Thank You !"));
```

# Interfaces fonctionnelles de l'API

- Package `java.util.function`
- Fournit des interfaces fonctionnelles de base
  - Pour une utilisation avec les lambdas
- `Consumer<T>` : méthode qui accepte un unique argument (type T) et ne retourne pas de résultat.
  - `void accept(T);`
- `Function<T, R>` : méthode qui accepte un argument (type T) et retourne un résultat (type R).
  - `R apply(T);`
- `Supplier<T>` : méthode qui ne prend pas d'argument et qui retourne un résultat (type T).
  - `T get();`

# Les références de méthodes

- Utilisées pour définir une méthode en tant qu'implémentation de la méthode abstraite d'une interface fonctionnelle

- Syntaxe :

`Classe::methode`

- Ou bien :

`instance::methode`

- Référence vers une méthode static :

```
Supplier<Double> random = Math::random;  
double result = random.get(); // Math.random();
```

- Référence vers une méthode d'instance spécifique :

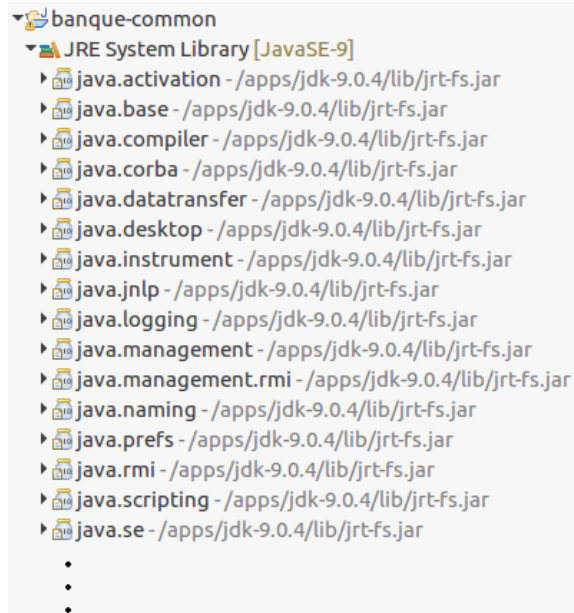
```
Random r = new Random();  
Supplier<Double> random2 = r::nextDouble;  
double result2 = random2.get(); // r.nextDouble();
```

- Référence vers un constructeur

```
Function<String, Thread> factory = Thread::new;  
Thread t = factory.apply("name"); // new Thread("name");
```

# Les modules de Java 9

- Java 9 introduit une nouveauté pour l'organisation du code des projets ainsi que pour la visibilité des packages et classes entre projet
  - Les modules
  - Le JDK 9 a été complètement réorganisé en modules !

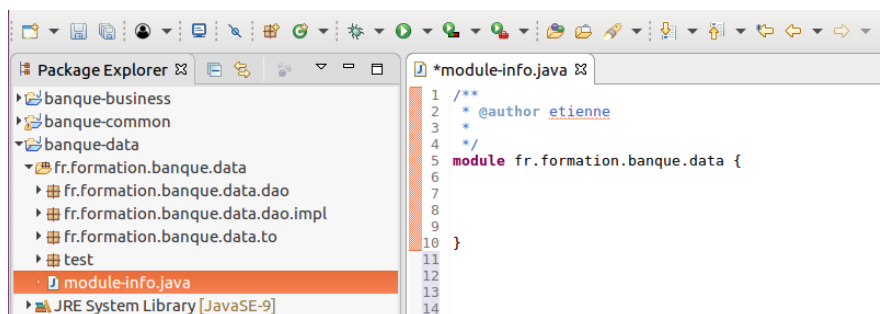


```
banque-common
└─ JRE System Library [JavaSE-9]
   ├── java.activation - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.base - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.compiler - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.corba - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.datatransfer - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.desktop - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.instrument - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.jnlp - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.logging - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.management - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.management.rmi - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.naming - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.prefs - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.rmi - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.scripting - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.se - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── .
   └── .
```



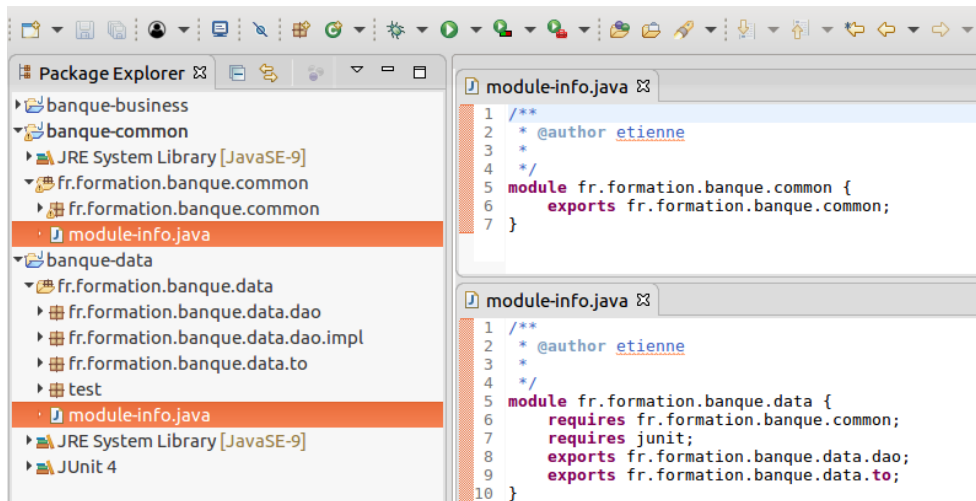
# Usage des modules

- Le principal intérêt des modules réside dans le fait de pouvoir « enfermer » des packages dans le projet et d'en ouvrir d'autres.
  - On pourra ainsi laisser accessible des interfaces et masquer les implémentations !
  - On peut aussi éviter des conflits sur les noms de classes ...
- La déclaration d'un module se fait au sein d'un fichier module-info.java et introduit le nouveau mot clé module
  - La notation des packages s'applique comme convention de nommage
  - On peut considérer que les modules sont un niveau au dessus des packages
  - Par soucis de lisibilité, on conseil également (si l'IDE le permet) de renommer le répertoire des sources du nom du module



# Déclaration et exportation de dépendances

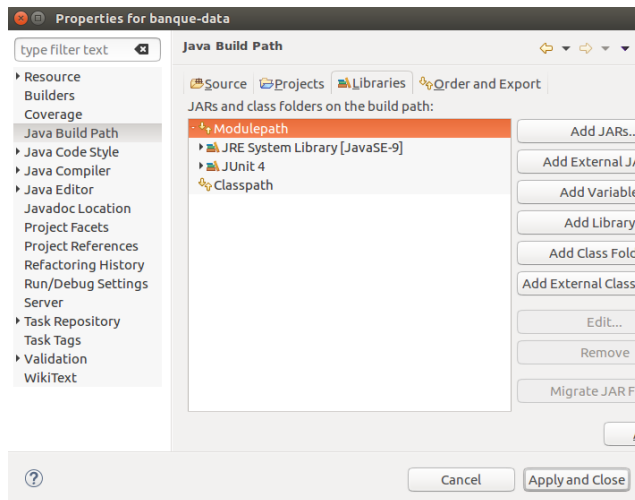
- Lorsqu'un projet déclare des modules, il est important que chaque module déclare ses dépendances nécessaires vis-à-vis des autres modules
  - Le mot clé `requires`
- De plus, si un module est utilisé, par défaut tous ses packages sont inaccessibles en dehors du module. Il faut donc les exporter pour les rendre visibles.
  - Le mot clé `exports`



# La notion de « module path »

- Lorsqu'une bibliothèque Java structurée en module est nécessaire à un projet, elle doit être ajoutée au « module path » si le projet incluant veut pouvoir déclarer une dépendance.
- La commande `java` introduit l'option `--module-path`  

```
$ java --module-path=target --module  
fr.formation.hello/fr.formation.hello.Main  
Hello World
```
- Les IDE prêts pour Java 9 introduisent également cette notion



# Les modules et les librairies Java

- Lors de l'ajout d'une librairie Java à un projet, il est nécessaire de connaître ses modules internes pour pouvoir les déclarer en tant que module.

```
module-info.java
1 /**
2  * @author etienne
3  *
4  */
5 module fr.formation.banque.data {
6     requires fr.formation.banque.common;
7     requires junit;
8     exports fr.formation.banque.data.dao;
9     exports fr.formation.banque.data.to;
10 }
```

- Certaines librairies ne déclarent pas (encore) de modules. Dans ce cas, un nom automatique est déduit

```
$ jar -d --file=lib/junit-4.12.jar
No module descriptor found. Derived automatic module.

module junit@4.12 (automatic)
    requires mandated java.base
    contains junit.extensions
    contains junit.framework
    contains junit.runner
    contains junit.textui
    contains org.junit
```

# Les exceptions

- **Besoin de gérer les erreurs**

```
double diviser(double numerateur, double denominateur){  
    return numerateur/denominateur; //pb possible  
}  
  
q = diviser(n,d);
```

- **Historiquement : code retour d'une fonction**

- **Détournement du fondement mathématique de la syntaxe**

```
int diviser(double numerateur, double denominateur, double *resultat){  
    if(denominateur != 0){  
        *resultat = numerateur/denominateur; //pb écarté  
        return CODE_OK;  
    }  
    else{  
        return CODE_DIV_0;  
    }  
}  
  
code = diviser(n,d,&q);  
if (code==CODE_OK) {...}
```

- **Un mal nécessaire**

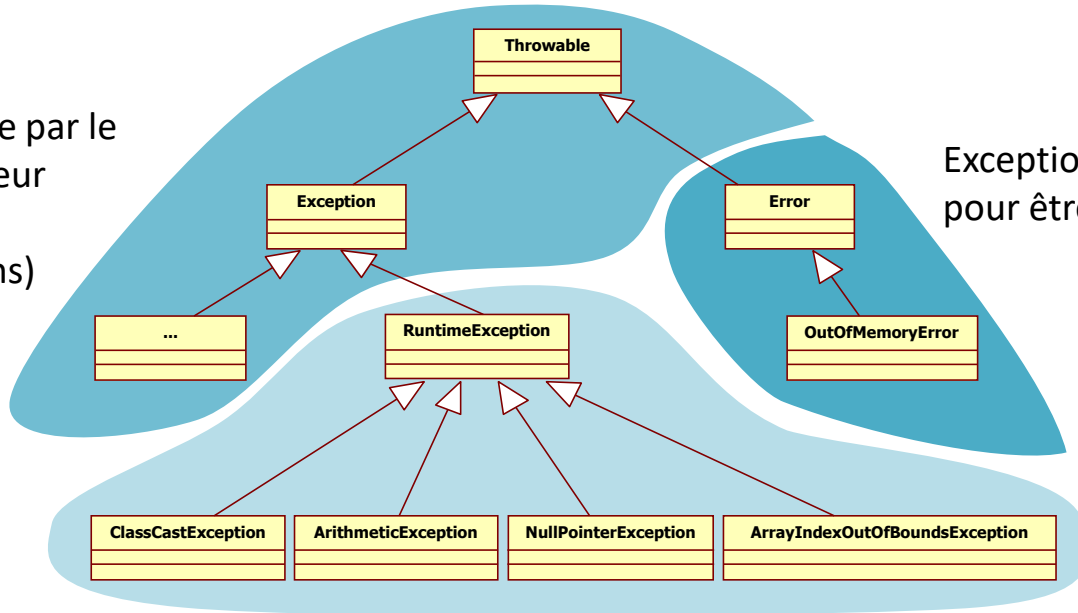
# Les exceptions (suite)

- Besoin d'un autre canal de sortie : les erreurs
  - Séparation du flux de données et d'erreurs
  - La syntaxe d'appel redevient naturelle
  - Syntaxe appropriée en cas d'erreur
- En Java, les erreurs sont des objets :
  - `java.lang.Exception`
  - Il faut les créer
- 2 parties
  - La détection du cas d'erreur = émission de l'erreur
  - Le traitement de l'erreur

# Les exceptions : différents types

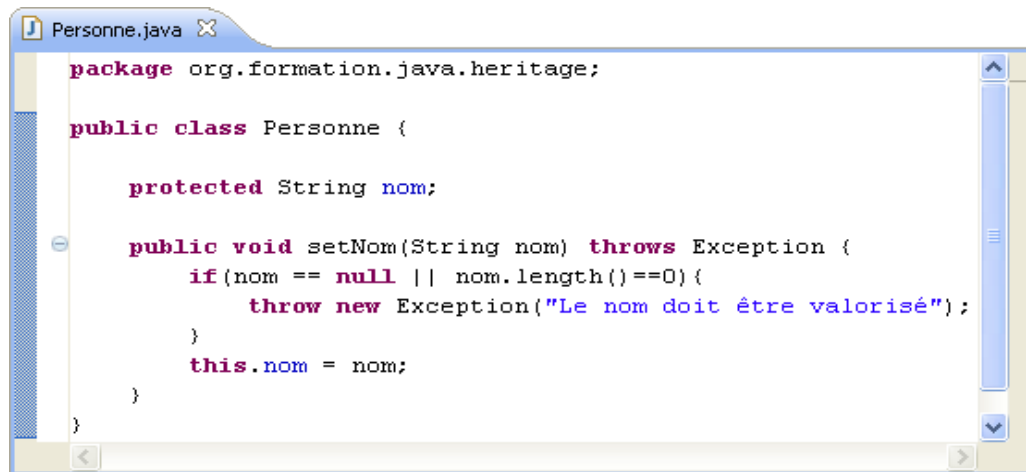
Gestion  
contrainte par le  
compilateur  
(Checked  
Exceptions)

Exceptions trop graves  
pour être gérées (JVM)



Exceptions évitables par de bonnes  
pratiques de programmation  
(Unchecked Exceptions)

# Les exceptions : origine de l'erreur



```
Personne.java X
package org.formation.java.heritage;

public class Personne {

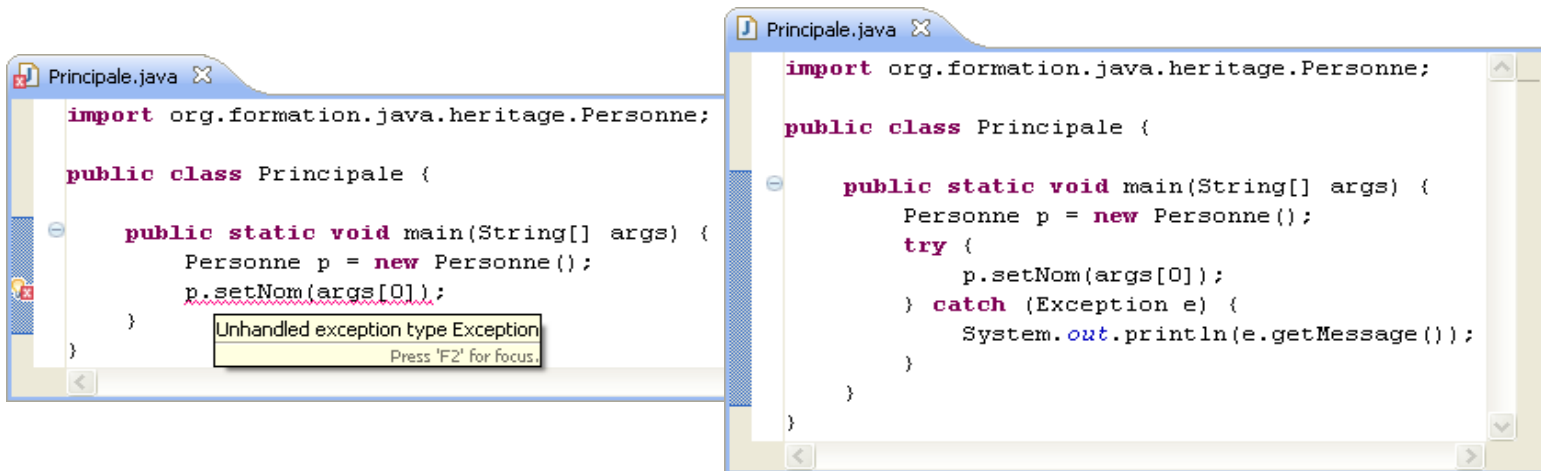
    protected String nom;

    public void setNom(String nom) throws Exception {
        if(nom == null || nom.length()==0){
            throw new Exception("Le nom doit être valorisé");
        }
        this.nom = nom;
    }
}
```

- Test sur les préconditions
- `throw` : envoi d'un objet `Exception` au premier bloc de traitement possible
- Déclaration du/des canal d'erreur `throws Exception`
- Le compilateur vérifie le traitement des exceptions

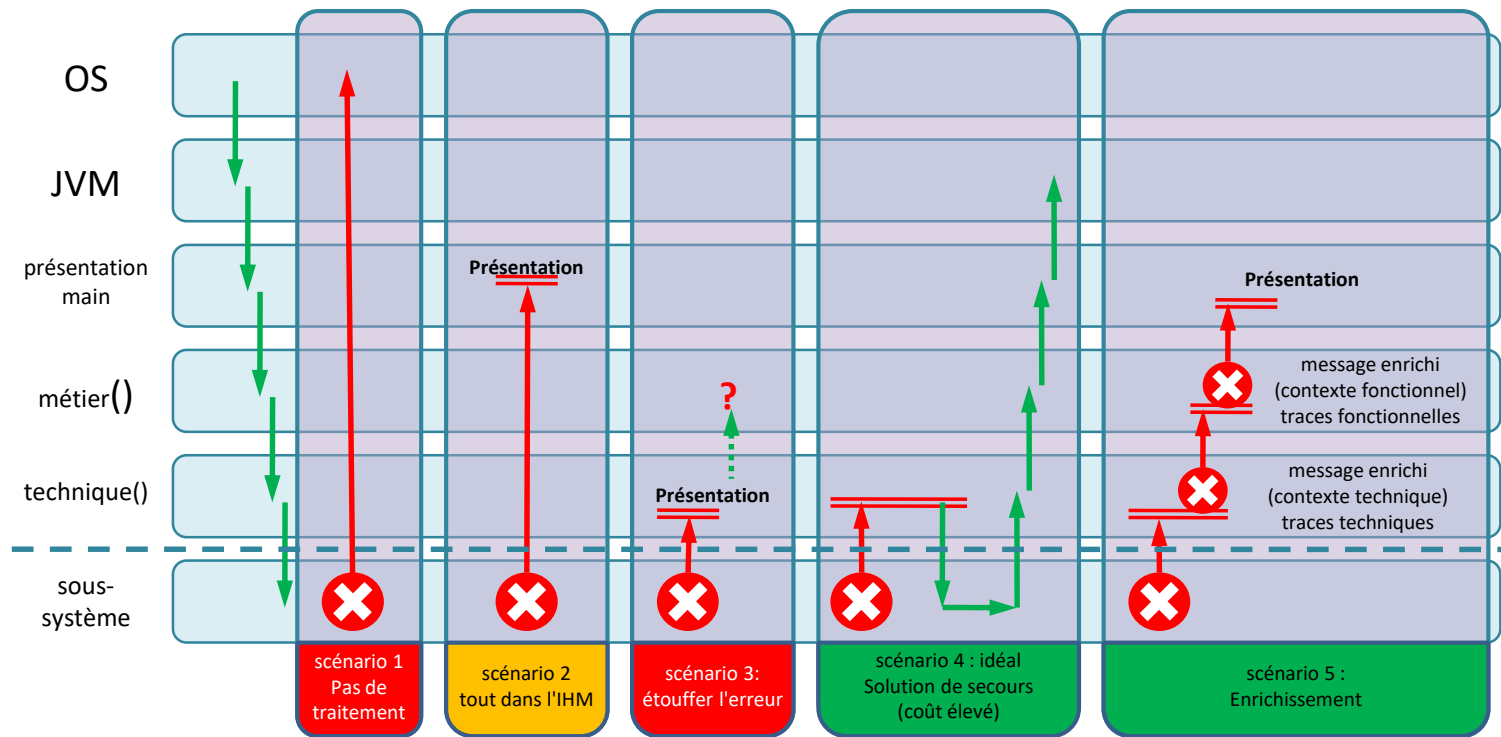


# Les exceptions : traitement de l'erreur



- Gérer les exceptions est presque obligatoire
- Responsabilités du traitement :
  - Réalisation correcte
    - Pas d'erreur
    - Ou interception et traitement alternatif : `try{...} catch (Exception e) {...}`
  - ou alerte via un canal approprié
    - Via un `throws`
    - Via un message à l'utilisateur

# Les exceptions : différents scénarios



# Les exceptions : chaînage

- Constructeurs

- `Exception(Throwable cause);`
- `Exception(String msg, Throwable cause)`

- Usage

```
try{
    unTraitementTechnique();
}
catch(ExceptionTechnique e){
    throw new ExceptionFonctionnelle("message fonctionnel",
    e);
}
```

# Les exceptions : chaînage (suite)

- Méthode `getCause()`
  - Permet de remonter à l'Exception technique en interceptant l'exception fonctionnelle

```
try{  
    unTraitementFonctionnel();  
}  
catch (Exceptionfonctionnelle e) {  
    System.out.println("message technique:" +  
        e.getCause().getMessage());  
}
```

# Les exceptions utilisateur

- Avoir ses propres classes d'exception
  - Exceptions techniques
  - Exceptions métiers
  - Exceptions de présentation
- ⇒ Hériter de `Exception`
- Reprendre les constructeurs
  - Ajouter des propriétés et des méthodes
    - ex : code d'erreur

# Les exceptions : bonnes pratiques

- Qu'est-ce qu'un cas d'erreur ?
  - Les pré-conditions ne sont pas respectées
  - Le sous-système de la fonctionnalité est défaillant
- Qu'est-ce qui n'est pas un cas d'erreur ?
  - Un scénario applicatif qui doit refuser une fonctionnalité  
ex : Interdiction de connexion suite à un mot de passe erroné
  - Le `try-catch` ne doit pas remplacer le `if-else`

# Les exceptions : le finally

- Parfois, il faut avoir la garantie d'exécuter du code même en cas d'exception
  - Exemple : libération de ressources

## ⇒ Le bloc `finally`

```
maRessource = new UneRessource();  
try{  
    maRessource.traiter();  
}  
catch (ExceptionTechnique e){  
    throw new ExceptionFonctionnelle("message fonctionnel",  
    e);  
}  
finally{  
    maRessource.close();  
}
```

- Le `catch` n'est pas obligatoire

# La structure try-with-resource

- JavaSE 7 apporte une amélioration à la structure `try`
- Cette structure simplifie le nettoyage des ressources, ce que l'on fait traditionnellement dans un `finally`

```
try(Connection cnx = DriverManager.getConnection()) {  
    Statement s = cnx.createStatement(...);  
}
```

- La connexion est fermée automatiquement en fin de `try`, quelle que soit la façon dont la sortie du `try` se fait :
  - Fin normale
  - Return
  - Exception
- Il est possible de préciser plusieurs ressources séparées par « ; ».
- Dans ce cas, les `close()` sont appelés dans l'ordre opposé de la création.



# La structure try-with-resource

- Cela est possible parce que `java.sql.Connection` hérite de l'interface `java.lang.AutoCloseable` ou de `java.io.Closeable`
- Cela simplifie la gestion des exceptions aussi. Dans l'exemple cité, le `cnx.createStatement()` et le `cnx.close()` provoquent des `SQLException` qu'il aurait fallu gérer
  - Dans le `try`
  - Dans le `finally`
- Ce nouveau `try` peut être doté des classiques `catch` et/ou `finally`

# Amélioration de la gestion des exceptions

- Java SE 7 améliore aussi la gestion des exceptions
- Constat : du code de gestion d'erreur dupliqué dans les blocs catch. Exemple :

```
try{
    ...
}
catch(ExceptionSpécifique e){
    logger.warn(e);
    throw new ExceptionFonctionnelle("...", e);
}
catch(AutreExceptionSpécifique e1){
    logger.warn(e1);
    throw new ExceptionFonctionnelle("...", e);
}
// pas de gestion pour les autres types d'exception
```

# Amélioration de la gestion des exceptions

- **Moyen** : un bloc `catch` capable d'intercepter plusieurs exceptions. Exemple :

```
try{
    ...
}
catch (ExceptionSpécifique|AutreExceptionSpécifique e) {
    logger.warn(e);
    throw new ExceptionFonctionnelle("...", e);
}
// pas de gestion pour les autres types d'exception
```

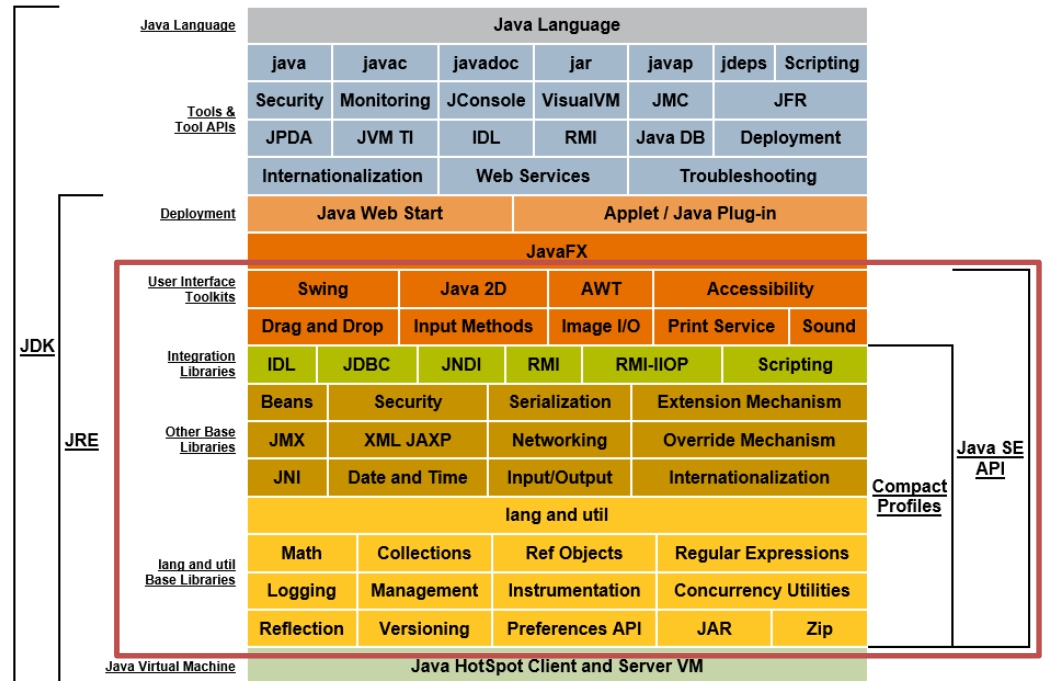
- **Limite** : le message de l'exception fonctionnelle remontée doit être commun, sauf usine à gaz à base de `instanceof` :-/

# Les fondamentaux de la programmation Java (Java SE)

## Module 6 L'API Java

# L'API Java

- Organisée en une multitude de package
- Très largement documentée : La JavaDoc
  - <http://docs.oracle.com/javase/8/docs/api/index.html>



# Les collections

- Différentes formes d'agrégation
  - Les listes
  - Les ensembles
  - Les tables de hachage
  - Les listes chaînées
- Le package `java.util`
  - Les classes synchronisées
  - Les classes non synchronisées

# Les collections : fonctionnement

- Les classes `Vector` et `ArrayList` (non synchronisé)
  - Tableau interne
  - Taille et capacité variables
  - Stratégie d'accroissement
  - `add`, `clear`, `get`, `ensureCapacity`, `trimToSize`, `remove`
- Les classes `Hashtable` et `HashMap` (non synchronisé)
  - Principe des fonctions de hachage
  - `put`, `get`, `remove`
- Les itérateurs
  - Curseurs sur des collections
  - `hasNext`, `next`, `remove`

# Les collections : la généricité

- Problème d'adaptation aux données spécifiques : Collections d'objets

- Soit, il faut adapter les collections pour des types spécifiques ; ex :

`ListePersonnes, ListeVoiture, ...`

- Soit, transtypage systématique de `Object` vers le type voulu

```
Personne p =(Personne) liste.get(10);
```

- Depuis Java 5, les types sont paramétrables

```
ArrayList<Personne> groupe = new ArrayList<Personne>();
```

```
HashMap<String,Voiture> parc = new HashMap<String, Voiture>();
```

```
Personne p = groupe.get(10);
```

```
Voiture v = parc.get("123 ABC 45");
```



# Inférence de type lors de la création d'instances génériques

- Depuis Java 7, il est possible d'avoir une syntaxe plus concise lors de l'instanciation lorsque les types paramétrés sont évidents :

```
Map<String, List<String>> myMap = new HashMap<String,  
List<String>>();
```

- Devient :

```
Map<String, List<String>> myMap = new HashMap<>();
```

- Les « <> » restent obligatoires si l'on ne veut pas instancier le type non générique (raw type)

# Inférence de type sur les constructeurs génériques

- Principe similaire appliqué aux constructeurs génériques de classes :

```
class UneClasse<X>{  
    <T> UneClasse(T t) {  
        ;  
    }  
}
```

- Instanciation :

```
UneClasse<Integer> unObjet = new UneClasse<>("chaine");
```

- Avant Java 7, le compilateur pouvait inférer le type effectif du paramètre.
- Avec Java 7, cette capacité a été étendue pour l'instanciation d'un objet *et de sa classe générique* avec l'utilisation de l'opérateur « <> »

# Initialisation des collections avec Java 9

- Java 9 introduit des méthodes d'initialisation pour les collections
  - Ce sont des méthodes par défaut des interfaces List et Set
  - Attention !
    - Les collections créées de cette façon sont immutables ! Si l'on tente d'ajouter, de supprimer ou de remplacer un élément, une exception de type `UnsupportedOperationException` est levée.
    - Elles n'autorisent pas la présence d'éléments nuls, en cas d'ajout une exception de type `NullPointerException` est levée.

```
List<String> liste = List.of("un", "deux", "trois");
```

# Le Scanner pour la gestion des flux standards

- La classe `java.util.Scanner` permet de lire les flux standards. Elle dispose de méthodes pour lire ces flux de données formatées.
- Un objet `Scanner` se construit à partir d'un flux d'octets ou de caractères (comme par exemple, l'entrée standard : `System.in`) :

```
Scanner scanner = new Scanner(System.in);
```

# Le Scanner pour la gestion des flux standards

- Quelques méthodes de lecture :

- `byte nextByte()`, `short nextShort()`, `int nextInt()`, `long nextLong()`, `float nextFloat()`, `double nextDouble()`, `BigInteger nextBigInteger()`, `BigDecimal nextBigDecimal()`
- Ces méthodes renvoient la prochaine valeur du type indiqué
- `String nextLine()`
- Cette méthode renvoie tous les caractères du flux jusqu'à une marque de fin de ligne

# Le Scanner pour la gestion des flux standards

- Exemple :

```
Scanner scanner = new Scanner(System.in);  
System.out.print("Nom et prénom ? : ");  
String saisie = scanner.nextLine();  
System.out.println("Bonjour " + saisie);
```

# Le Scanner pour la gestion des flux standards

- Utilisation des expressions régulières avec la méthode `next()`.
  - Les expressions régulières ont été introduites avec Java 1.4.
  - Deux classes :
    - `java.util.regex.Pattern`
    - `java.util.regex.Matcher`
- Exemple :

```
Scanner scanner = new Scanner(System.in);
try {
    System.out.print("Aimez-vous Java 5.0 ? ");
    String saisie = scanner.next(Pattern.compile("[Oo]"));
}
catch(InputMismatchException e) {
    System.err.println("Les seules réponses possibles sont O ou o :-)");
}
```

# Le Scanner pour la gestion des flux standards

- Gestion des exceptions

- Les méthodes de lecture du flux sont susceptibles de renvoyer des exceptions dans les cas suivants :

- La saisie ne correspond pas au type de données attendu (Une chaîne saisie sur un `nextInt()` par exemple)
  - `java.util.InputMismatchException`
- Il n'y a pas d'éléments dans le flux
  - `java.util.NoSuchElementException`
- Le scanner est fermé
  - `java.util.IllegalStateException`



# Le Scanner pour la gestion des flux standards

## ■ Tester la présence des données

- La classe `Scanner` possède un ensemble de méthodes permettant de tester la présence et le type de la prochaine donnée (sans la lire !)
- `boolean hasNext()`
  - Y a-t-il une donnée disponible pour la lecture ?
- `boolean hasNextLine()`
  - Y a-t-il une ligne disponible pour la lecture ?
- `boolean hasNextByte()`, `boolean hasNextShort()`,  
`boolean hasNextInt()`, `boolean hasNextLong()`, `boolean hasNextFloat()`, `boolean hasNextDouble()`, `boolean hasNextBigInteger()`, `boolean hasNextBigDecimal()`

Les fondamentaux de la programmation Java (Java SE)

## Module 7

# Les librairies de journalisation Java

# Les bibliothèques de journalisation Java

- Il existe en Java, plusieurs API de journalisation : parmi elles, trois ont su s'imposer.
  - Log4J : Apache Software Foundation
    - <https://logging.apache.org/log4j>
  - Java Logging API : Standard Java
    - Intégrée à Java SE depuis le JDK 1.4
    - Package `java.util.logging`
  - Jakarta Commons Logging : Apache Software Foundation
    - <http://commons.apache.org/logging>
    - Une « méta » API permettant d'encapsuler d'autres API.

- Log4j est un framework de log développé par la communauté Apache.
  - Il est disponible sur le site <http://logging.apache.org/>
  - Une fois la distribution téléchargée, récupérer l'archive log4j-2.x.x.jar
    - Cette librairie, une fois ajoutée au classpath du projet, permettra d'utiliser les fonctionnalités du framework
- Log4j comporte un certain nombre d'avantages par rapport aux autres solutions.
  - Performances
  - Rotation des fichiers de log
  - Patterns de mise en forme personnalisables
  - Possibilité de logger sur plusieurs supports
  - Outil de visualisation et filtrage de logs (Chainsaw)

# Principes de Log4J

- Le Logger vérifie en fonction de sa limite de priorité basse s'il laisse passer le message, dans l'affirmative transfère aux Appenders auxquels il est abonné.
- Chaque Appender fait la même vérification et formate le message avec un Layout avant d'écrire sur le média
- Le Logger est un objet utilisé dans le code pour générer les traces, l'Appender permet de spécifier où écrire et le Layout comment.
- Il existe plusieurs type d'Appender et de Layout

# La configuration de Log4J

- La configuration des logs peut se faire dans un fichier
  - XML
    - log4j2.xml
  - .properties
    - log4j2.properties
- La solution fichier XML est la méthode la plus répandue
  - Associée à une DTD, une validation de la syntaxe est possible
    - Donc l'assistance à la saisie des éléments et attributs dans un IDE !
  - Elle accepte le rechargement à chaud de la configuration

# Configuration XML

```
<?xml version="1.0" encoding="UTF-8"?>

<Configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="Log4j-config.xsd"
    strict="true"
    monitorInterval="60">

    <!-- Définition de propriétés pour réutilisation -->
    <Properties>
        <Property name="filename">target/banque-integration.log</Property>
    </Properties>

    <!-- Définition des 'Appenders', destination et format des messages-->
    <Appenders>
        <Appender type="Console" name="sysout">
            <PatternLayout pattern="%d{DATE} [%t] %5level (%Logger) - %msg %n" />
        </Appender>
        <Appender type="File" name="file" filename="${filename}">
            <PatternLayout pattern="%d{DATE} [%t] %5level (%Logger) - %msg %n" />
        </Appender>
    </Appenders>

    <!-- Définition des 'Loggers', identification des classes Java et de
        leur niveau de journalisation -->
    <Loggers>
        <!-- Un logger spécifique pour la classe DBUtil -->
        <Logger name="fr.formation.banque.integration.dao.DBUtil" level="debug" additivity="false">
            <AppenderRef ref="sysout" />
        </Logger>

        <!-- Il faut au moins un logger 'racine' -->
        <Root level="info">
            <AppenderRef ref="sysout" />
            <AppenderRef ref="file" />
        </Root>
    </Loggers>

</Configuration>
```

# Les Loggers

- Ils reçoivent les messages à journaliser
  - Le processus de log s'effectue sur 5 niveaux de priorité :
    - debug
    - info
    - warn
    - error
    - fatal
  - Le niveau est défini dans la configuration de log4j.
- Pour créer un logger :

```
// Récupérer le logger racine
```

```
Logger logger = LogManager.getRootLogger();
```

```
// Création d'un nouveau logger
```

```
Logger logger = LogManager.getLogger("MonLogger");
```

```
//Création d'un logger statique, basé sur le nom de la classe  
static Logger logger = LogManager.getLogger(test.class);
```



# Les Appenders

- Représentent la destination des messages de log.
  - Il existe des Appenders pour console, fichier, composants graphiques, JMS...
- Principaux Appenders :
  - Async
    - déport de la tâche d'écriture dans un thread; peut dispatcher la trace vers plusieurs appenders "réels".
  - Console
    - vers la sortie standard
  - File
    - vers un fichier
  - RollingFile
    - vers un fichier avec basculement vers sauvegarde une fois une taille atteinte, ou un ordre de bascule reçu sur un port (externally), ou le début d'une nouvelle journée (daily)
  - JDBC
    - vers une base de données
  - SMTP
    - envoi de mail
  - JMS
    - Java messaging services
  - NTEventLog
    - vers la console d'événements NT
  - Syslog : vers un daemon syslog

# Configuration des Appenders

- Paramètres communs à tous les Appenders par héritage
- Threshold : le débit; permet de limiter l'écriture sur le média.
  - `<ThresholdFilter level="DEBUG" onMatch="ACCEPT" onMismatch="DENY"/>`
- Le layout : la disposition des informations à écrire sur le média
  - `<PatternLayout pattern="%d{DATE} [%t] %5level (%logger) - %msg%n" />`
    - SimpleLayout; ex : DEBUG - message de debug
    - PatternLayout; sortie texte configurable
    - HTMLLayout : écriture d'un tableau HTML
    - XMLLayout : écriture d'un fichier XML

# Configuration du PatternLayout

- ConversionPattern : Chaîne contenant des marqueurs  
`%-tailleMin.tailleMaxCHAMP{précision}`
- seuls les caractères % et de champ sont obligatoires
  - % sert de délimiteur de marqueur
  - les caractères de champ sont c,C,d,F,l,L,m,M,n,p,r,t,x,X
- - permet de forcer l'alignement à gauche
- tailleMin : nombre de caractères mini pris par le champ. Si la valeur du champ est plus petite, il y a complétion par des blancs à gauche ou à droite en fonction de l'alignement
- tailleMax : nombre de caractères (introduit par un point) au-delà duquel la valeur du champ est tronquée.
- la précision (entre accolades) : dépend du type de champ

# Configuration du PatternLayout

Champ	Utilité
c	Catégorie ( = Logger). S'il est suivi d'un marqueur de précision (ex : %c{2} ), seuls les <i>n</i> derniers niveaux spécifiés seront affichés : org.formation.logtest → %c{2} = formation.logtest
C (lent)	Nom de classe pleinement qualifié. Possibilité de marqueur de précision Ex : org.formation.logtest.MaClasse → %C{2} = logtest.MaClasse
d	la date; le marqueur de précision: soit 3 constantes : "ABSOLUTE", "DATE" and "ISO8601" qui formatent respectivement en "HH:mm:ss,SSS", "dd MMM yyyy HH:mm:ss,SSS", "yyyy-MM-dd HH:mm:ss,SSS"; soit la syntaxe de java.text.SimpleDateFormat (plus lent)
F (lent)	Nom du fichier de la classe qui a produit la trace
l (L min) (très lent)	Informations de localisation de l'émetteur de la trace: méthode pleinement qualifiée, nom de fichier et numéro de ligne.
L (lent)	Numéro de ligne
m	Le message
M	La méthode qui a émis la trace
n	Le séparateur de ligne en fonction de la plateforme (\r\n ou \r ou \n)
p	Le niveau de gravité de la trace ( = priorité)
r	nombre de millisecondes passées depuis le démarrage de l'application.
t	Nom du thread à l'origine de la trace.
x	Nom du contexte de diagnostic NDC (nested)
X	Nom du contexte de diagnostic MDC (mapped). Obligatoirement utilisé avec la clé qui permet de retrouver la valeur discriminante. Ex: %X{user} ; trouve dans la Map MDC la clé user et affiche la valeur.
%	Permet d'afficher le caractère pourcent.

# Les Loggers - Configuration

- Il est possible de définir des configurations de journalisation spécifiques pour des classes :

```
<Loggers>
  <!-- Un logger spécifique pour la classe DBUtil -->
  <Logger name="fr.formation.banque.integration.dao.DBUtil" level="debug" additivity="false">
    <AppenderRef ref="sysout" />
  </Logger>
  ...
</Loggers>
```

- Les Loggers permettent de diriger la sortie vers un Appender spécifique avec un niveau de journalisation propre à la classe (où a l'ensemble de classes).
- Il faut cependant toujours un Logger « racine »

```
<Loggers>
  <!-- Il faut au moins un logger 'racine' -->
  <Root level="info">
    <AppenderRef ref="sysout" />
    <AppenderRef ref="file" />
  </Root>
</Loggers>
```

- API de Java SE localisée dans le package `java.util.logging`
  - Disponible depuis Java SE 1.4
- Supporte plusieurs niveaux de journalisation
  - (Du plus haut au plus bas)
  - SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST
- La classe `LogManager` sert de fabrique de `Logger` pour les localiser ou bien les créer.
- Cette API permet également
  - de filtrer les messages dans différents fichiers de configuration
  - De formater les traces produites (via les classes `Formatter`)

# Mise en œuvre – java.util.logging

- Définition d'un logger :
  - `private static final Logger log =  
Logger.getLogger("MaClasse");`
- Utilisation dans le code :
  - `log.info("Un message d'information");`
- Avec la configuration par défaut, la sortie produite est la suivante :

```
13 mai 2016 13:45:19 MaClass main  
INFO: Un message d'information
```
- Il est bien sur possible d'agir sur ce format de sortie, ainsi que spécifier le fichier de sortie.

# Principes – java.util.logging

- L'API java.util.logging utilise 3 éléments fondamentaux pour son fonctionnement. Ces éléments peuvent être finement configurés dans un fichier de configuration.
  - Les handlers
  - Les formatters
  - Les loggers
- Les handlers
  - Les handlers peuvent écrire à destination d'un fichier texte ou vers la sortie standard.
    - java.util.logging.FileHandler
    - java.util.logging.ConsoleHandler.
  - Liste des attributs de configuration communs :
    - level : permet de spécifier le niveau de journalisation.
    - formatter : permet d'indiquer une classe Java pour formater le contenu du fichier.
  - Liste des attributs de configuration spécifiques à java.util.logging.FileHandler :
    - pattern : permet de spécifier le nom du fichier.



# Principes – java.util.logging

## ■ Les formatters

- Les formatters sont des classes Java qui permettent de définir la manière dont sont écrits les messages par les handlers.
- Les formatters disponibles sont :

- `java.util.logging.SimpleFormatter` : C'est la valeur par défaut. Les entrées de messages sont formatées sur 2 lignes.

```
dec 16, 2013 10:46:19 AM org.apache.catalina.core.StandardService startInternal  
INFO: Démarrage du service Catalina
```

- `java.util.logging.XMLFormatter` : Permet de générer un journal au format XML pour exploitation avec un outil tel qu'une feuille de style XSL par exemple.

```
<record>  
  <date>2013-16-06T10:48:48</date>  
  <millis>1394099328902</millis>  
  <sequence>6</sequence>  
  <logger>org.apache.catalina.core.StandardService</logger>  
  <level>INFO</level>  
  <class>org.apache.catalina.core.StandardService</class>  
  <method>startInternal</method>  
  <thread>1</thread>  
  <message>Démarrage du service Catalina</message>  
</record>
```

## ■ Les loggers

- Il est possible de définir un logger sur une classe précise :
  - `fr.formation.banque.integration.ClientDAOImpl.level = INFO`
- Ou sur un ensemble de classes :
  - `fr.formation.banque.integration = INFO`

# Configuration – java.util.logging

```
# -----
# La liste des handlers à créer au démarrage.
#
handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
# Niveau de journalisation par défaut
# Les loggers et handlers pourront redéfinir cette valeur
.level=INFO

# Loggers
# -----
# Les packages spécifiés ici utilisent des niveaux de journalisation
# spécifiques, les autres utiliseront le niveau par défaut.
myapp.ui.level=ALL
myapp.business.level=CONFIG
myapp.data.level=SEVERE

# Handlers
# -----
# --- ConsoleHandler ---
java.util.logging.ConsoleHandler.level=SEVERE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
# --- FileHandler ---
java.util.logging.FileHandler.level=ALL
# Spécification du fichier de sortie
java.util.logging.FileHandler.pattern=target/java.log
# Limitation de la taille des fichiers (en octets)
java.util.logging.FileHandler.limit=500000
# Nombre de fichiers d'historique conservés
java.util.logging.FileHandler.count=1
# Formattage des traces
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

- Pour utiliser ce fichier, la JVM du programme Java devra spécifier l'option :
  - -Djava.util.logging.config.file=myLoggingConfigFilePath

Les fondamentaux de la programmation Java (Java SE)

## Module 8

# Mise en œuvre de tests unitaires avec JUnit

# Les tests

- Les logiciels deviennent de plus en plus complexes et importants.
- Pas de programme sans bug!
- Important de tester avant de distribuer son code.

- On peut distinguer les tests en fonction de leur :
  - objectif :
    - scénario : vérifier que le comportement fonctionnel est bien celui attendu
    - non-regression : ce qui fonctionnait dans les version précédentes du code fonctionne toujours dans la nouvelle version
    - performance (bench) ou charge (load) : les temps de réponse à une requête sont conformes aux attentes
    - intégration/fonctionnels : le code s'intègre bien avec les autres éléments du système.
  - cible :
    - scénarios : tester un cas d'utilisation
    - unitaires : tester un composant du système
  - technologie :
    - Web : envoyer des requêtes Web simulant le comportement d'utilisateur(s)

- Exemples d'outils de tests
  - Test Web
    - Load Runner
    - HttpUnit
  - Tests unitaires
    - JUnit
  - Tests fonctionnels/d'intégration
    - JFunc
  - Test de charge/performance
    - JMeter
    - →XP compte parmi ses recommandations les tests unitaires systématiques et les tests d'intégration continue.

- Le test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme.
- JUnit désigne un framework de rédaction et d'exécutions de tests unitaires.
  - La librairie est téléchargeable à l'adresse <http://www.junit.org/>
- Imaginé et développé en Java par Kent Beck et Erich Gamma, auteurs des ouvrages "SmallTalk Best Practice Patterns" et "Design Patterns : Catalogue de modèles de conception réutilisables".
- Avantages de JUnit :
  - Très simple d'utilisation
  - Utilisation graphique ou non
  - Intégré dans presque tous les outils de développement (JBuilder/Eclipse/WSAD)

- But:
  - Offrir au développeur un environnement de développement simple, le plus familier possible, et ne nécessitant qu'un travail minimal pour rédiger de nouveaux tests.
- Idée principale:
  - Représenter chaque test par un objet
  - Un test correspond souvent à une classe du programme
  - Un test pourra être composé de plusieurs tests unitaires dont le rôle sera de valider les différentes méthodes de vos classes

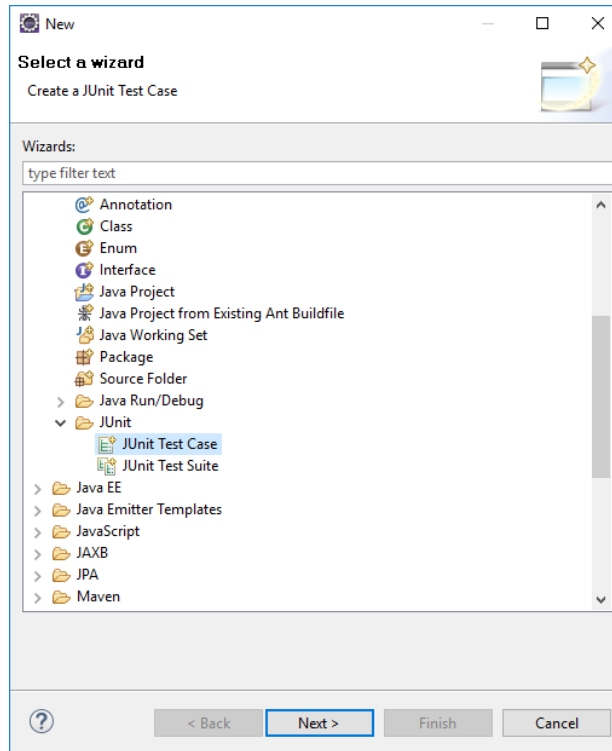


# Les tests JUnit

- Le framework JUnit comporte les concepts suivants :
  - Le test Case ou test unitaire
    - Il teste qu'une classe données répond aux exigences en matière de traitement
  - La suite de test
    - Elle permet de regrouper tous les tests dans une suite afin de lancer la batterie de tests.

# Création d'un test case

- Aller dans le menu File > New > JUnit Test Case
- Si la librairie JUnit n'est pas incluse dans la librairie du projet, Eclipse proposera de l'inclure automatiquement



# Définition du test / Sélection de la classe à tester

New JUnit Test Case

**JUnit Test Case**

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()  
☐ setUp() ☐ tearDown()  
☐ constructor

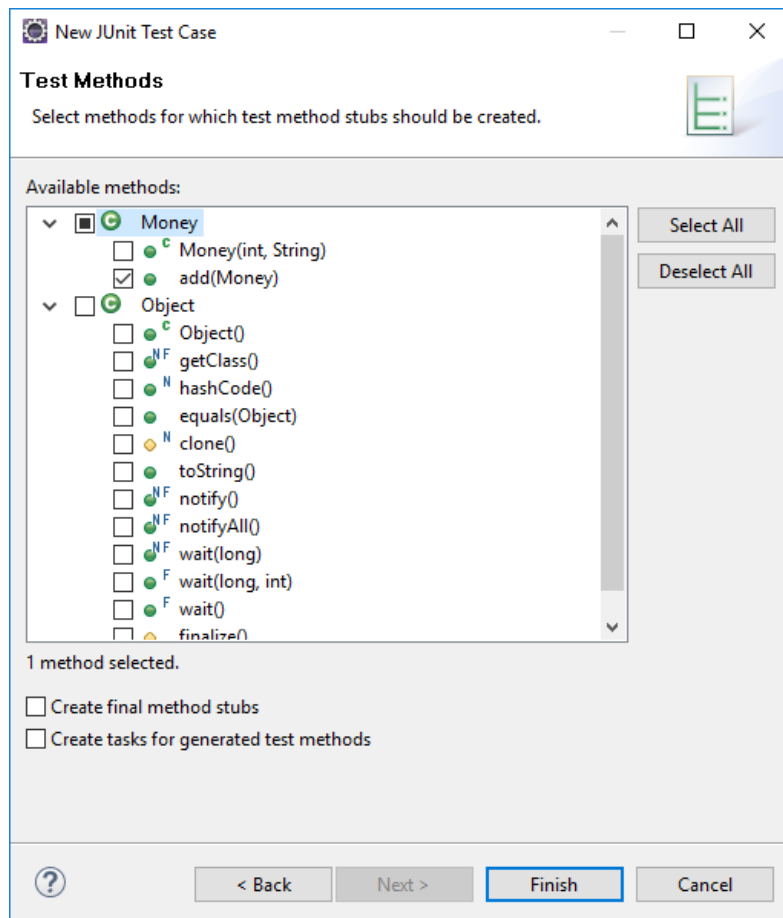
Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

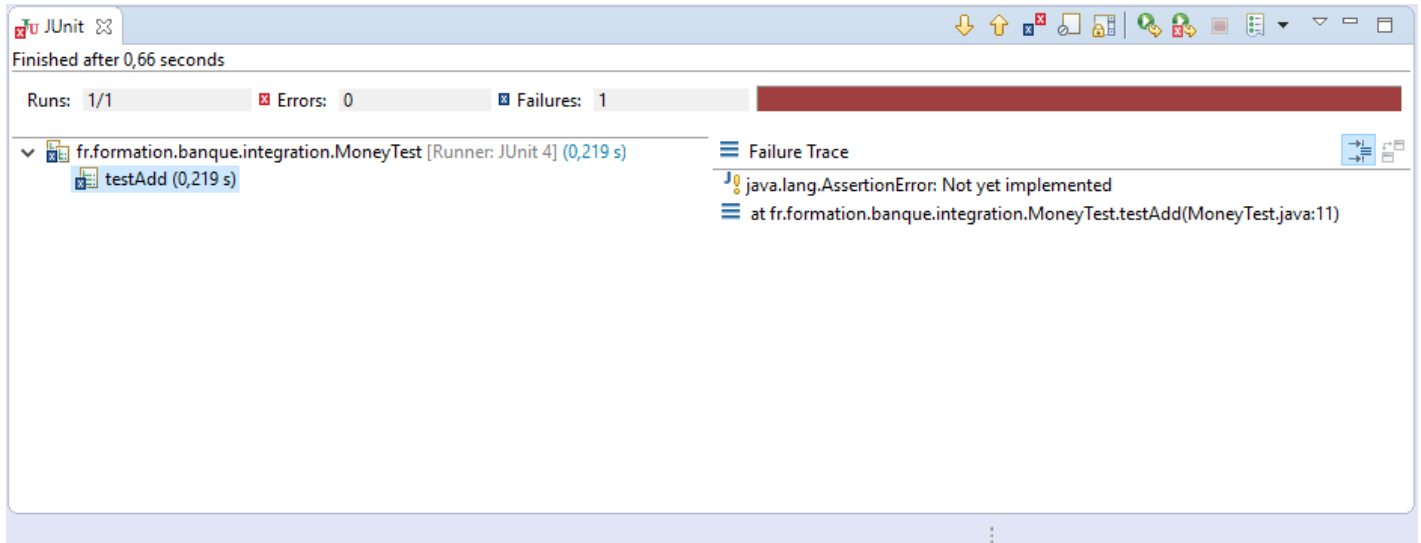
? < Back Next > Finish Cancel

# Sélection des méthodes à tester



# Exécution et résultat des tests

- Run as ... > JUnit Test



# Implémentation du test

- Il suffit ensuite d'implémenter l'appel de méthode de la classe testée
  - Les méthodes assert... du test case permettent de valider les valeurs retournées par la méthode
    - `assertNull()`
    - `assertNotNull()`
    - `assertEqual()`
    - `assertNotEqual()`
    - `assertExist()`
    - ...
  - Les assertions non vérifiées entraînent une remontée d'erreur

# JUnit 4 - annotation @Test

- Les méthodes de test sont annotée avec @Test
  - Beaucoup de simplification par rapport à Junit 3
    - pas besoin de dériver de TestCase
    - les assertions sont des méthodes statiques de la classe org.junit.Assert
    - les méthodes n'ont pas l'obligation d'être nommées en testXxxx
- Paramètres principaux de l'annotation
  - expected : classe Throwable attendue
  - timeout : durée maximale en millisecondes
  - @Ignore permet d'ignorer un test

# JUnit 4 - annotation @Test

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestsUnitaires
{
    @Test
    public void testAddition()
    {
        Calculatrice calc = new Calculatrice();
        assertTrue(5 == calc.additionner(2, 3));
    }
}
```

```
public class TestTimeout
{
    @Test(timeout = 1000)
    public void dureeRespectee()
    {}

    @Test(timeout = 1000)
    public void dureeNonRespectee() throws InterruptedException
    {
        Thread.sleep(10000);
    }
}
```



# Initialisation des méthodes du test

- Méthodes annotées avec @Before ou @After
  - méthodes exécutées avant et après chaque méthode de test
  - méthode publique
  - plusieurs méthodes peuvent être annotées avec la même annotation
    - l'ordre d'invocation des méthodes annotées est indéterminé

# Initialisation du test

- Méthodes annotées avec `@BeforeClass` ou `@AfterClass`
  - méthodes exécutées avant l'invocation du premier test de la classe, et après le dernier test de la classe
  - méthodes publiques et statiques
  - une seule méthode par annotation

# Initialisation : Exemple

```
public class TestFixture
{
    @BeforeClass public static void montageClasse()
    {
        System.out.println(">> Montage avant tous Les tests");
    }

    @AfterClass public static void demontageClasse()
    {
        System.out.println(">> Démontage après tous Les tests");
    }

    @Before public void montage()
    {
        System.out.println("----- AVANT");
    }
    @After public void demontage()
    {
        System.out.println("----- APRES");
    }

    @Test public void test1()
    {
        System.out.println("Test 1");
    }

    @Test public void test2()
    {
        System.out.println("Test 2");
    }
}
```

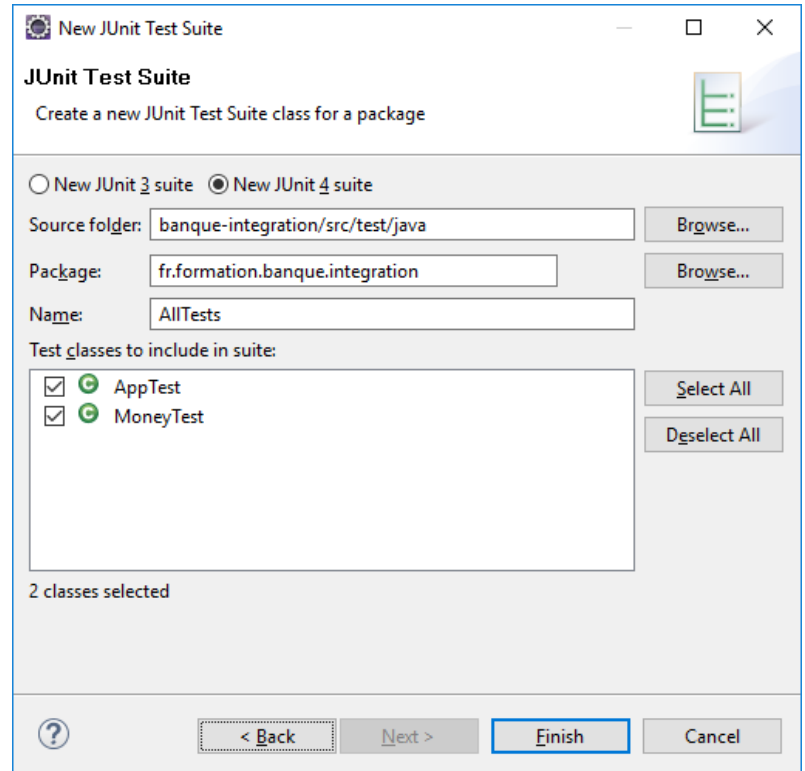
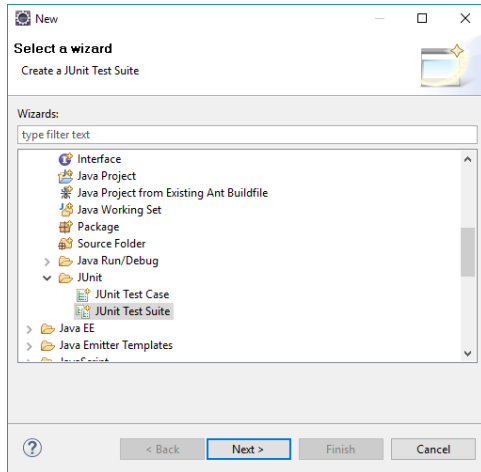
# Les suites de tests

- Classe vide annotée avec
  - `@RunWith`, possibilité de changer la classe d'exécution de la suite
    - `Suite.class` par défaut
  - `@SuiteClasses(Class[])` pour former la suite de tests

```
@RunWith(Suite.class)
@SuiteClasses({ AppTest.class, MoneyTest.class })
public class AllTests {

}
```

# Suite de test avec Eclipse



# Paramétrage des tests

- La classe de test
  - est annotée avec `@RunWith(Parameterized.class)`
    - exécuteur de test acceptant les paramètres
  - contient un constructeur avec les données et le résultat
  - contient une méthode renvoyant une collection des paramètres : données et résultat
    - annotée avec `@Parameters`

# Test paramétré

```
@RunWith(Parameterized.class)
public class TestsParametres
{
    private int a,b, resultat;

    public TestsParametres(int a, int b, int resultat)
    {
        this.a = a; this.b = b; this.resultat = resultat;
    }

    @Parameters
    public static Collection getParametresAddition()
    {
        return Arrays.asList(new Object[][]{{1,2,3}, {3,5,8}, {9,6,15}});
    }

    @Test
    public void additionTest()
    {
        Calculatrice calc = new Calculatrice();
        Assert.assertEquals(resultat, calc.additionner(a, b));
    }
}
```

# Les fondamentaux de la programmation Java (Java SE)

Fin de la formation



# Pour aller plus loin

- ENI Service sur Internet
  - Consultez notre site Web ([www.eni-service.fr](http://www.eni-service.fr))
    - Les actualités
    - Les plans de cours de notre catalogue
    - Les filières thématiques et certifications
  - Abonnez-vous à nos newsletters pour rester informé sur nos nouvelles formations et nos événements en fonction de vos centres d'intérêt
  - Suivez-nous sur les réseaux sociaux
    - Twitter : <http://twitter.com/eniservice>
    - Viadeo : <http://bit.ly/eni-service-viadeo>

# Pour aller plus loin

- Notre accompagnement
  - Tous nos Formateurs sont également Consultants et peuvent :
    - Vous accompagner à l'issue d'une formation sur le démarrage d'un projet.
    - Réaliser un audit de votre système d'information.
    - Vous conseiller, lors de vos phases de réflexion, de migration informatique.
    - Vous guider dans votre veille technologique.
    - Vous assister dans l'intégration d'un logiciel.
    - Réaliser complètement ou partiellement vos projets en assurant un transfert de compétence.

# Votre avis nous intéresse

- Nous espérons que vous êtes satisfait de votre formation. Merci de prendre quelques instants pour nous faire un retour en remplissant le **questionnaire de satisfaction**.
- **Merci pour votre attention, et au plaisir de vous revoir prochainement.**