



INTRODUCTION AU LANGAGE JAVA

Christophe GNAHO
christophe.gnaho@u-paris.fr

PLAN

- Chapitre 1 – Introduction
- Chapitre 2 – Structure d'un programme Java
- Chapitre 3 – Éléments du langage
- Chapitre 4 – Membres de classe
- Chapitre 5 – Organisation des classes
- Chapitre 6 – Les chaînes de caractères
- Chapitre 7 – Les structures d'objets
- Chapitre 8 – Héritage, Polymorphisme, Classes abstraites et Interface
- Chapitre 9 – Encapsulation
- Chapitre 10 – Gestion des erreurs (Les exceptions)
- Chapitre 11 – Les entrées - sorties

Chapitre 1 - Introduction

- Caractéristiques de Java
- Compilation et exécution des programmes
- Environnement de développement

CARACTERISTIQUES DE JAVA

Le langage Java a été conçu à partir des langages objets *SmallTalk* et *C++*. Ses caractéristiques sont résumés ci-dessous :

- **Simplicité** : d'une syntaxe familière aux programmeurs C et C++, il en a été dépouillé de tous les mécanismes complexes (gestion des pointeurs, gestion de la mémoire, l'héritage multiple, ...)
- **Orienté Objet** : Tout est objet, Il faut concevoir ainsi. Contrairement au C++ qui autorise l'écriture du code des fonctions en dehors des classes, Java oblige le programmeur à définir les méthodes comme partie intrinsèque des classes.
- **Orienté réseau et distribué** : Développement d'applications réparties. Java permet d'accéder facilement à des données distantes sur le réseau, et de réaliser des applications client/serveur grâce aux classes paquetages de communication (java.net).
- **Multitâche (multi-thread)** : Java permet de concevoir des applications capables d'effectuer plusieurs actions (*thread*) en même temps. Un même programme peut par exemple à l'aide de trois *threads* faire un calcul, tout en chargeant une vidéo et en diffusant un son.

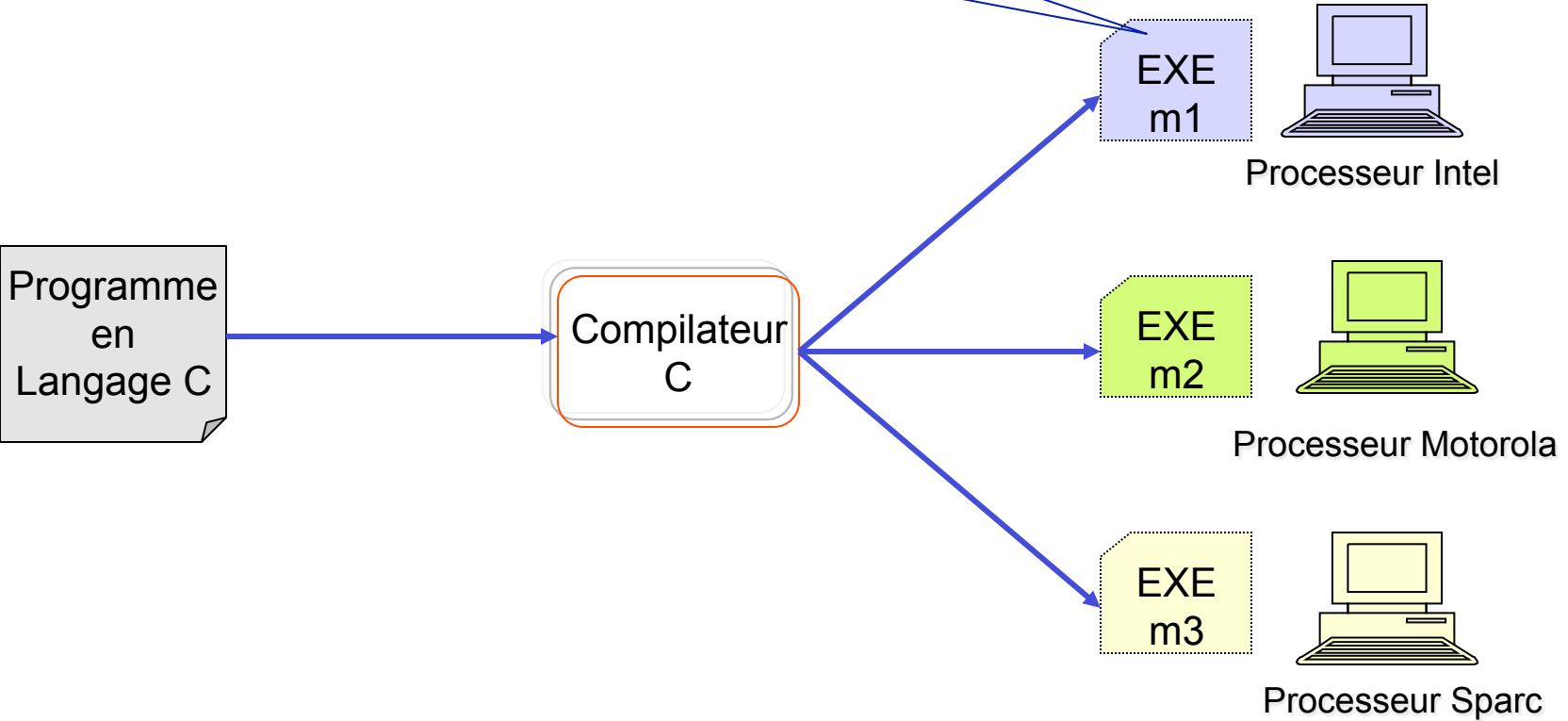
CARACTERISTIQUES DE JAVA

- **Dynamique** : Un programme Java charge les classes qu' il utilise (y compris les classes de base) en cours d' exécution. La modification d'une classe ne nécessite pas une recompilation de l'application pour prendre en compte les changements effectués, car les liens entre les objets sont résolus lors de l'exécution.
- **Robuste** : L'objectif de java est de permettre la réalisation de logiciels fiables. Les caractéristiques suivantes permettent d'atteindre ce objectif : **typage fort, pas d'accès aux pointeurs (réduisant les risques d'erreurs), gestionnaire de la mémoire, mécanisme de gestion d'exception.**
- **Sécurisé** : Eviter d' exécuter du code dommageable. De plus java possède des API destinés au cryptage, à la gestion de l'authentification, ...
- **Interpréte, Indépendant des architectures matérielles** : Le code produit par le compilateur n' est pas du code machine. C' est un code intermédiaire (*bytecode*) simple et rapide (plus rapide qu' un langage de script entièrement interprété) à traduire en langage machine par un interpréteur (*Java Virtual Machine - JVM*) gérant pour sa part les spécificités du système hôte. Il est cependant possible de recompiler le bytecode afin de produire du code natif et d' atteindre les performances identiques à celles du langage C.

CARACTERISTIQUES DE JAVA

(Compilation)

Le fichier généré dépend du processeur de l' ordinateur



CARACTERISTIQUES DE JAVA

(Compilation)

L'exécution d'un programme Java se fait instruction par instruction. Il s'agit d'une **interprétation**, où chaque instruction en bytecode est traduite en langage machine, puis exécutée. Il existe une technique de mémorisation des instructions déjà exécutées.

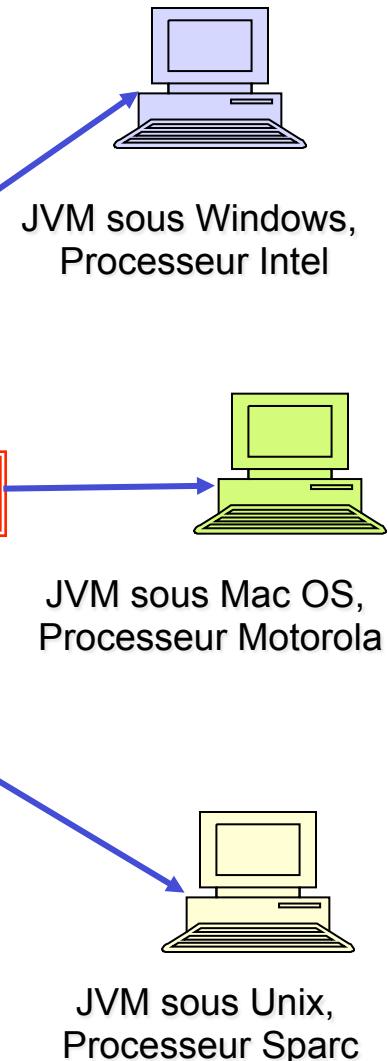
Le code source n'est pas **directement** traduit dans le langage de l'ordinateur



Le code source est d'abord traduit dans un langage **indépendant** de l'ordinateur



Machine Virtuelle Java (JVM) = Interprète



CARACTERISTIQUES DE JAVA (La JVM)

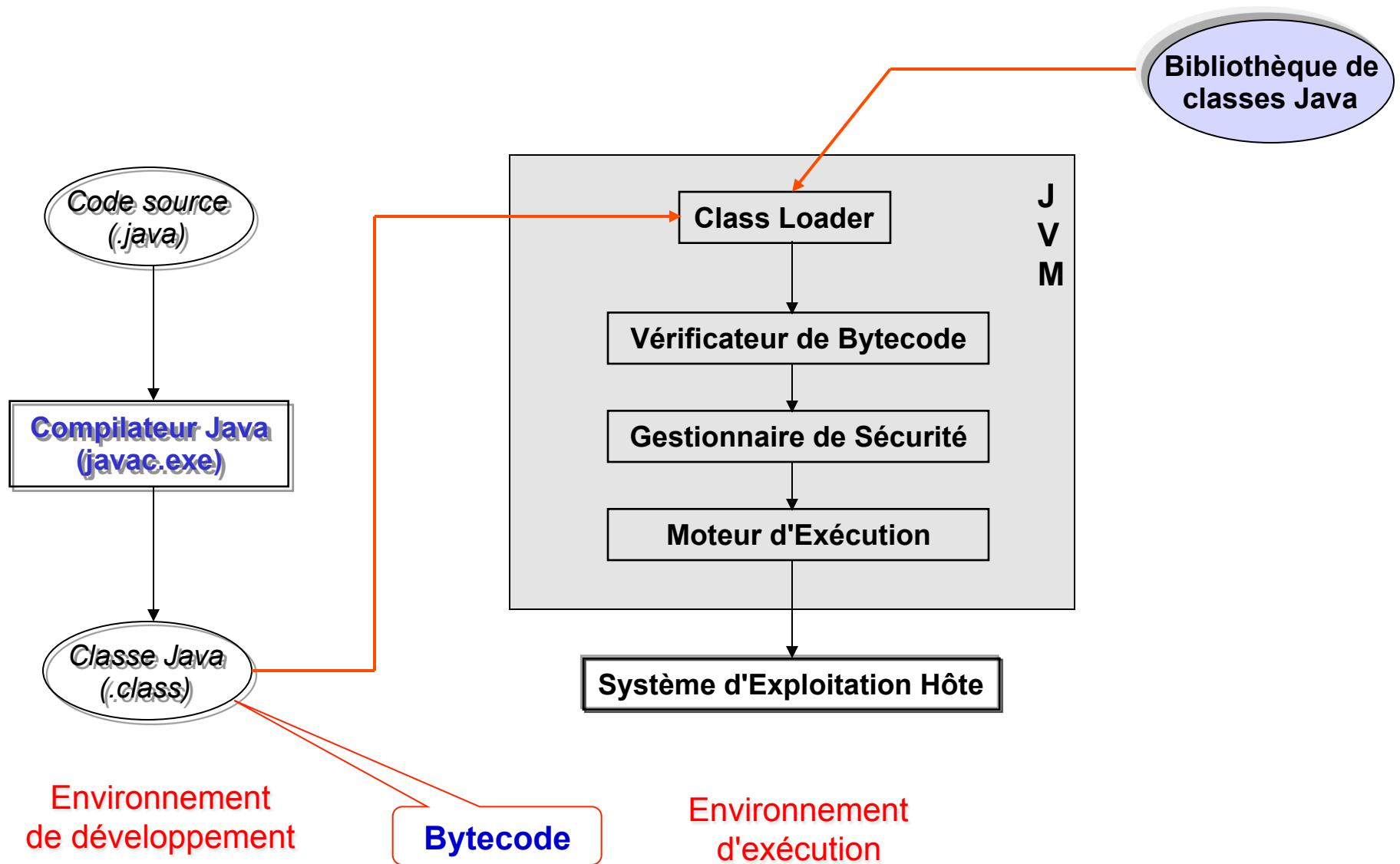
La JVM est un **programme** qui réalise principalement les trois tâches suivantes :

- lire les instructions en bytecode
- les traduire dans le langage natif du processeur de l' ordinateur
- lancer leurs exécution

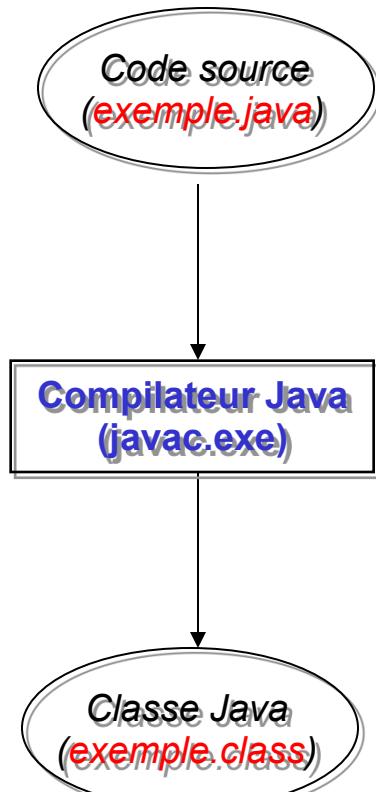
Ce programme **simule** (machine virtuelle) le comportement d'une vraie machine dont le langage natif est le bytecode.

Si un ordinateur possède une JVM alors il peut exécuter tout fichier compilé sur n'importe quel autre machine.

COMPILEATION ET EXECUTION DES PROGRAMMES



COMPILEATION ET EXECUTION DES PROGRAMMES

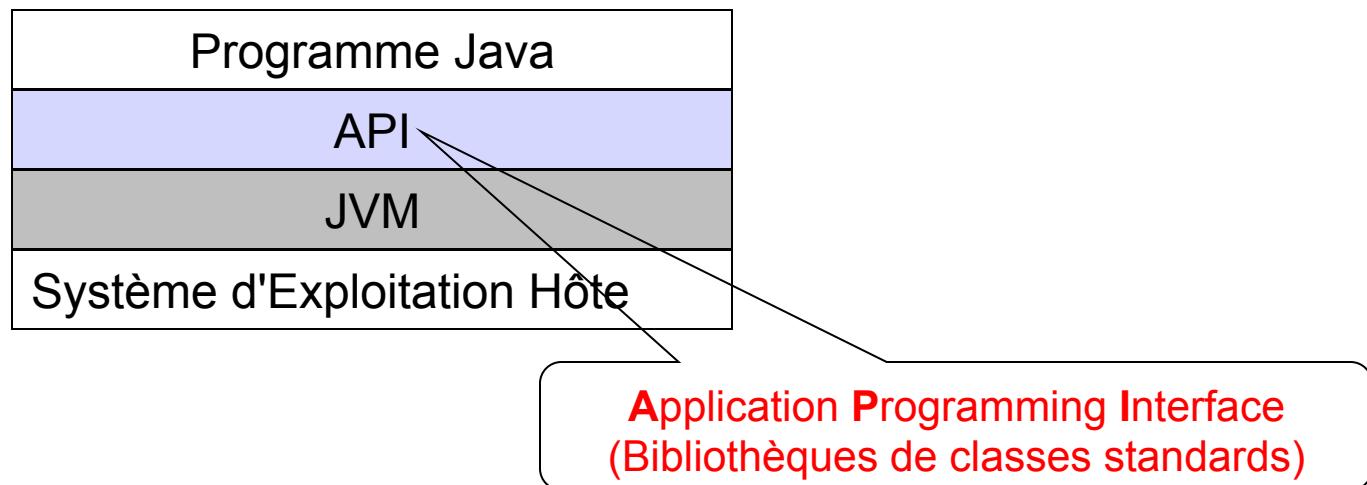


Le fichier **.class** doit être dans le répertoire courant ou dans un des emplacements indiqués par une option **-classpath** ou par une variable d'environnement **CLASSPATH**.

Environnement
de développement

ENVIRONNEMENT DE DEVELOPPEMENT

(La plate-forme JAVA)



ENVIRONNEMENT DE DEVELOPPEMENT

Le Java Development Kit (JDK) comprends :

- Toutes les classes du java (*J2SE*)
- Un compilateur (*javac.exe*)
- Une machine virtuelle (*java.exe*)
- Un debuggeur (*jdb.exe*)
- Un générateur de documentation technique (*javadoc.exe*)
- Un générateur de fichiers archives – JAR (*jar.exe*)
- Une visionneuse d'applets (*appletviewer.exe*)
- Divers outils pour le RMI
- ...

N.B. La variable d' environnement PATH doit inclure le répertoire qui contient les utilitaires Java (javac, java, ...)

ENVIRONNEMENT DE DEVELOPPEMENT

Un IDE (Integrated Development Environment) est un logiciel qui propose les fonctionnalités suivantes :

- Un éditeur à coloration syntaxique
- Des assistants de préfabrication de code
- Un explorateur de classes
- Une aide en ligne
- Un debugger graphique
- Un composeur d'interface graphique
- ...

Les IDE du marché :

Eclipse,

Netbeans,

Borland Jbuilder,

Oracle JDeveloper,

...

Chapitre 2 – Structure d'un programme Java

- Rappel
- Organisation des fichiers sources
- Éléments d'une classe Java
- Les variables
- Les méthodes
- Les constructeurs

STRUCTURE D'UN PROGRAMME JAVA (Rappel)

- En **programmation structurée**, un programme est constitué d'un ou de plusieurs **modules**. Chaque module est formé de la réunion de différentes procédures (fonctions) et de différentes structures de données **généralement indépendantes** de ces procédures.
- En **programmation orientée objet**, un programme met en œuvre différents **objets** (instances de classes). Chaque objet associe des données et des méthodes (fonctions) **agissant exclusivement sur les données**.

La notion de classe représente donc une extension de la notion de module. Les données et les fonctions traitant ces données, sont réunies ensemble dans une classe qui constitue un **nouveau type**.

STRUCTURE D'UN PROGRAMME JAVA (Rappel)

Une classe est définie par son nom, ses attributs (ses données) et ses méthodes (ses fonctions).

A chaque classe correspond des objets qui en sont les réalisations informatiques concrètes.

Les objets sont structurés par les attributs de la classe, et ils ont la capacité d'exécuter les fonctions définies au sein de la classe.

Un objet est donc créé (on dit **construit**) à partir d'une classe, et **résidé dans la mémoire centrale de l'ordinateur**.

STRUCTURE D'UN PROGRAMME JAVA

D'un point de vue syntaxique, un **programme Java** se présente sous la forme d'un ensemble de fichiers sources (extension `.java`).

Chaque fichier source peut contenir une ou plusieurs définitions de classes.

L'une de ces définitions de classes tient lieu de point d'entrée (programme principal), elle est lancée prioritairement au début du programme, pour l'exécution (le pilotage) du programme.

STRUCTURE D'UN PROGRAMME JAVA

(Organisation des fichiers sources)

Une seule classe dans un fichier source - Employe.java

```
public class Employe {  
    // Définition des variables d' instance  
    private String nom;  
    private int anneeNaissance;  
    // Définition du constructeur  
    public Employe (String n, int an){  
        nom = n;  
        anneeNaissance =an;  
    }  
    // Définition des méthodes  
    public void calculAge (){  
        int age = 2007 – anneeNaissance;  
        System.out.println ("Votre âge : " + age);  
    }  
    public void afficher (){  
        System.out.println( "Nom : " + nom + " Année de naissance : " + anneeNaissance );  
    }  
    public static void main (String [] args){ // Méthode principale  
        Employe e1 = new Employe ("Tobago", 1965);  
        Employe e2 = new Employe ("Bonanza", 1969);  
        e1.afficher();  
        e1.calculAge();  
        e2.afficher();  
        e2.calculAge();  
    }  
}
```

Le fichier doit porter le même nom que la classe

STRUCTURE D'UN PROGRAMME JAVA

(Organisation des fichiers sources)

Deux classes dans un seul fichier source - **Employe.java**

```
public class Employe {  
    private String nom;  
    private int anneeNaissance;  
    public Employe (String n, int an){  
        nom = n;  
        anneeNaissance =an;  
    }  
    public void calculAge (){  
        int age = 2007 – anneeNaissance;  
        System.out.println ("Votre âge : " + age);  
    }  
    public void afficher (){  
        System.out.println( "Nom : " + nom + " Année de naissance : " + anneeNaissance );  
    }  
}  
class TestEmploye {  
    public static void main (String [] args){ // Méthode principale  
        Employe e1 = new Employe ("Tobago", 1965);  
        Employe e2 = new Employe ("Bonanza", 1969);  
        e1.afficher();  
        e1.calculAge();  
        e2.afficher();  
        e2.calculAge();  
    }  
}
```

**Une seule classe publique par fichier source.
Le fichier doit porter le même nom que cette classe**

**La compilation du fichier source fournit 2 fichiers classes :
Employe.class et TestEmploye.class.
On lance l'exécution de la classe qui contient la méthode
main()**

STRUCTURE D'UN PROGRAMME JAVA

(Organisation des fichiers sources)

Deux classes dans un seul fichier source - **TestEmploye.java**

```
class Employe {  
    private String nom;  
    private int anneeNaissance;  
    public Employe (String n, int an){  
        nom = n;  
        anneeNaissance =an;  
    }  
    public void calculAge (){  
        int age = 2007 – anneeNaissance;  
        System.out.println ("Votre âge : " + age);  
    }  
    public void afficher (){  
        System.out.println( "Nom : " + nom + " Année de naissance : " + anneeNaissance );  
    }  
}  
  
public class TestEmploye {  
    public static void main (String [] args){ // Méthode principale  
        Employe e1 = new Employe ("Tobago", 1965);  
        Employe e2 = new Employe ("Bonanza", 1969);  
        e1.afficher();  
        e1.calculAge();  
        e2.afficher();  
        e2.calculAge();  
    }  
}
```

**Une seule classe publique par fichier source.
Le fichier doit porter le même nom que cette classe**

**La compilation du fichier source fournit 2 fichiers classes :
Employe.class et TestEmploye.class.
On lance l'exécution de la classe qui contient la méthode
main()**

STRUCTURE D'UN PROGRAMME JAVA

(Organisation des fichiers sources)

Deux classes dans deux fichiers sources

```
public class Employe {  
    private String nom;  
    private int anneeNaissance;  
    public Employe (String n, int an){  
        nom = n;  
        anneeNaissance =an;  
    }  
    public void calculAge (){  
        int age = 2007 – anneeNaissance;  
        System.out.println ("Votre âge : " + age);  
    }  
    public void afficher (){  
        System.out.println( "Nom : " + nom + " Année de naissance : " + anneeNaissance );  
    }  
}
```

Fichier Employe.java

```
public class TestEmploye {  
    public static void main (String [] args){ // Méthode principale  
        Employe e1 = new Employe ("Tobago", 1965);  
        Employe e2 = new Employe ("Bonanza", 1969);  
        e1.afficher();  
        e1.calculAge();  
        e2.afficher();  
        e2.calculAge();  
    }  
}
```

Fichier TestEmploye.java

STRUCTURE D'UN PROGRAMME JAVA

(Organisation des fichiers sources)

Synthèse

1. Un nom de classe commence par une majuscule.
2. Dans un fichier source contenant plusieurs classes, une seule classe doit définir une méthode *main()*.
3. Il y a au plus une définition de classe *public* par fichier source, **le nom du fichier source doit alors être égal au nom de cette classe.**
4. Plusieurs classes peuvent être dans un même fichier, mais il y aura autant de fichier.*.class* produit par le compilateur qu'il y a de classes dans le fichier *.java*.
5. Eviter les accents dans les noms des classes, et des variables

N.B. Si le fichier contient une seule classe, il n'est pas nécessaire de la déclarer publique.

STRUCTURE D'UN PROGRAMME

(Eléments d'une classe)

Classe = Brique de base d'un programme Java

Structure d'une classe

```
import ... ; // importer les classes prédéfinies à utiliser  
  
class NomDeClasse {  
    // Déclaration des Variables d' instance  
    // Définition des Constructeurs  
    // Définition des Méthodes  
}
```

STRUCTURE D'UN PROGRAMME

(Eléments d'une classe)

Fichier – *Client.java*

Class Client {

 String nom;
 String prenom;

Variables d'instance

 Client (String n, String p) {
 nom = n;
 prenom = p;
 }

Constructeurs

 Client () { //constructeur sans argument
 nom = " ";
 prenom = " ";
 }

 void afficher (){
 System.out.println("Nom : " + nom + "Prénom : " + prenom);
 }

Méthodes

}

STRUCTURE D'UN PROGRAMME

(Eléments d'une classe)

Constructeurs

Ils servent à créer des objets (instances) de la classe

Variables d'instance

Quand une instance est créée, son **état** (l'ensemble des valeurs de ses attributs) est conservé dans les variables d'instance.

Méthodes

Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message.

Les variables et les méthodes s'appellent les **membres** de la classe

STRUCTURE D'UN PROGRAMME

(Eléments d'une classe)



Une classe possède plusieurs facettes :

- un **type** qui décrit une structure de données (ses variables d'instance) et un comportement (ses méthodes).
- un **module** pour décomposer un programme en sous-programmes
- un **générateur d'objets** (grâce à ses constructeurs)

STRUCTURE D'UN PROGRAMME

(Eléments d'une classe)



Conventions pour les identificateurs

Le langage Java distingue les lettres majuscules des lettres minuscules.

- Les noms des classes commencent par une majuscule : Facture, Personne
- Les noms des variables et des méthodes commencent par une minuscule : afficher(), client
- Les mots contenus dans un identificateur commencent par une majuscule : FactureClient, FigureGeometrique, unClient, perimetreCercle
- Les noms des constantes sont en majuscule : TAUX, PI, ...
- Eviter d'utiliser des accents

STRUCTURE D'UN PROGRAMME

(Les variables)

Pour définir une classe, il existe trois sortes de variables :

- les variables d'instance,
- les variables de classe et
- les variables locales (à une méthode).

Les variables de classe seront traitées plus loin.

Les variables d'instance

- sont déclarées en dehors de toute méthode
- conservent l'état d'un objet
- sont accessibles et partagées par toutes les méthodes de la classe

Les variables locales

- sont déclarées à l'intérieur d'une méthode
- conservent une valeur utilisée pendant l'exécution de la méthode
- ne sont accessibles que dans le bloc dans lequel elles ont été déclarées

STRUCTURE D'UN PROGRAMME

(Les variables)

Déclaration d'une variable

La déclaration d'une variable indique au compilateur que le programme va utiliser une variable de ce nom et de ce type.

Toute variable doit être déclarée avant d'être utilisée. La déclaration se fait à l'aide de la syntaxe suivante :

<Type><nom> [=valeur d'initialisation]

Exemple :

```
int annee;  
double salaire;  
Client monClient;  
Figure f1;
```

STRUCTURE D'UN PROGRAMME

(Les variables)

Initialisation d'une variable

Une variable doit être initialisée avant d'être utilisée dans une expression.

Les **variables d'instance** (et les variables de classe étudiées plus loin) reçoivent automatiquement des valeurs par défaut, si elles ne sont initialisées par le programmeur.

L'utilisation d'une **variable locale** non initialisée par le programmeur provoque une erreur de compilation (**variable may not have been initialized**).

Il est possible d'initialiser une variable en la déclarant.

Exemple

```
double prime = 1500.0;  
double salaire = prime + 2500.0;
```

STRUCTURE D'UN PROGRAMME

(Les variables)

Déclaration / création d'un objet

La déclaration d'une variable destinée à référencer un objet ne suffit pas à créer cet objet. Pour créer un objet (instancier une classe) sous Java, il faut utiliser l'opérateur **new**.

L'opérateur **new** procède en trois étapes :

- (1) Alloue dynamiquement la mémoire de l'objet (en fonction des types des attributs). **Une initialisation par défaut des valeurs des attributs se produit.**
- (2) Appelle la méthode constructeur et exécute le constructeur
- (3) Retourne l'adresse de l'objet ainsi construit (**cette adresse est mise dans la variable**).

STRUCTURE D'UN PROGRAMME

(Les variables)

```
Employe e1 = new Employe("Durand" , "Rue de Paris " , 22 );
```

(1) Allocation dynamique de mémoire et initialisation

nom	Null
adresse	Null
age	0

(2) Appel du constructeur de la classe

nom
adresse
age

Durand
Rue de Paris
22

(3) Retour de l'adresse de l'objet

e1

@objet

La variable e1 contient l'adresse en mémoire de l'objet créé !

STRUCTURE D'UN PROGRAMME

(Les variables)

```
public static void main (String[] args){  
    Employe e1;  
    e1.calculSalaire();  
}
```

Déclare que l'on va utiliser une variable e1 qui référencera un objet de la classe Employe

Provoque une erreur de compilation : car la variable n'est pas initialisée.

STRUCTURE D'UN PROGRAMME

(Les variables)

```
public static void main (String[] args){  
    Employe e1 = null;  
    e1.calculSalaire();  
}
```

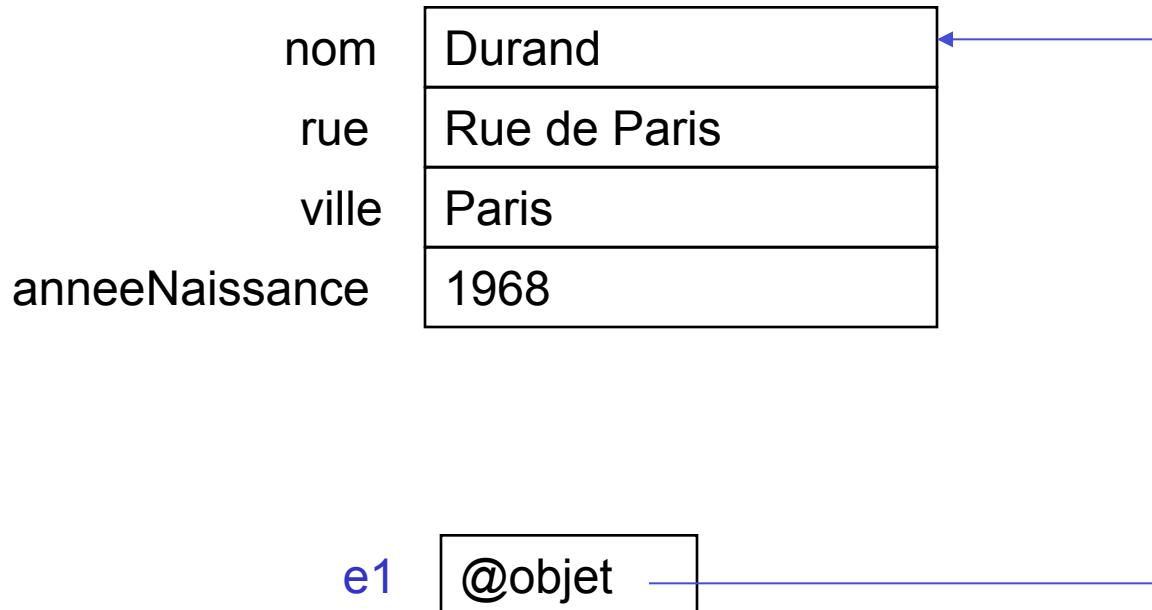
Déclare que l'on va utiliser une variable e1 qui référencera un objet de la classe Employe

Provoque une erreur d'exécution :
NullPointerException, car l'objet n'est pas encore créé.

STRUCTURE D'UN PROGRAMME

(Les variables)

```
Employe e1 = new Employe("Durand" , "Rue de Paris " , "Paris" ,1968);
```



Il est maintenant possible d'invoquer la méthode `calculSalaire()` sur l'objet référencé par `e1` : `e1.calculSalaire()`

STRUCTURE D'UN PROGRAMME

(Les méthodes)

Chaque objet (instance d' une classe) dispose des méthodes définies au niveau de la classe dont le code est conçu pour lui permettre de bien gérer les valeurs de ses attributs.

Un objet est donc **responsable de son propre état interne**. Il est souhaitable dans la mesure du possible que les méthodes agissent sur les valeurs de l' objet tout en retournant rien (void).

Une méthode s' exécute au sein d' un espace qui contient à la fois les variables de l' objet et les variables (arguments) de la fonction.

Dans l' exemple de la classe *Employe*, les objets (instance de la classe *Employe*) ont la capacité (à travers la méthode *modifierAdresse()*) de modifier chacun **leur propre rue et ville**.

STRUCTURE D'UN PROGRAMME

(Les méthodes)

Définition d'une méthode

Pour chaque méthode d'une classe, il faut indiquer :

- Le niveau d'accessibilité (par exemple *public* – voir le chapitre encapsulation)
- Le type de la valeur de retour
- Le nom de la méthode et ses arguments formels avec leurs types.
- Puis le code de définition de la méthode.

La spécification du nom, du type de retour, de la valeur d'accessibilité et des arguments d'une méthode s'appelle la *signature de la méthode*.

Le type de retour d'une méthode peut être soit un type primitif, soit une classe, soit le type **void** (qui signifie que la méthode ne renvoie aucune valeur).

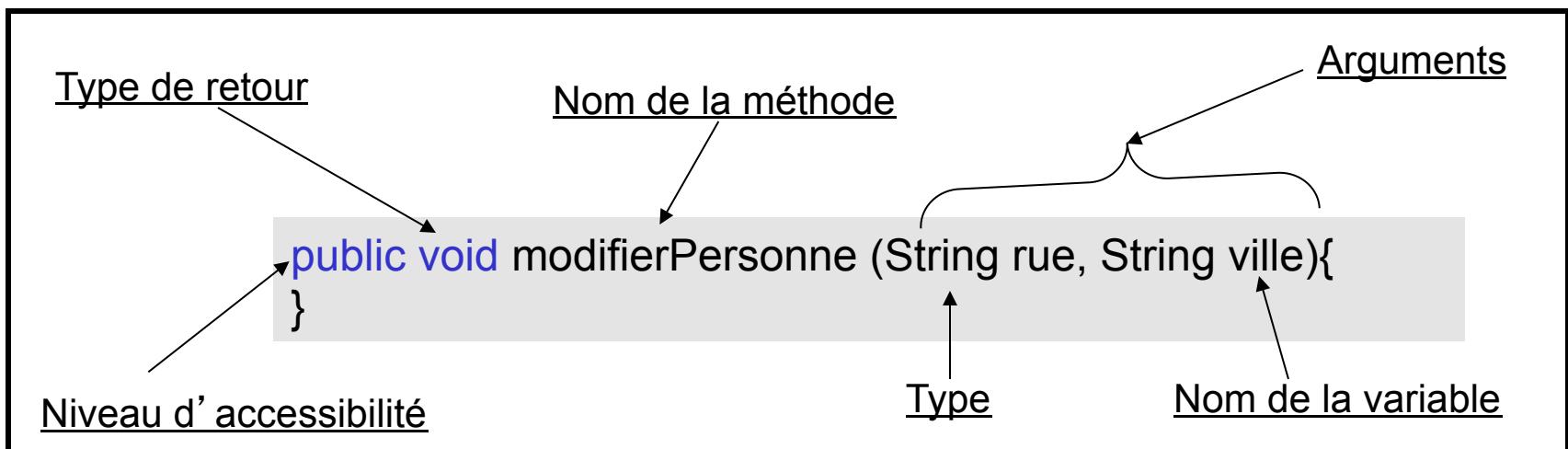
Le mot clé **return** est utilisé pour renvoyer une valeur.

STRUCTURE D'UN PROGRAMME

(Les méthodes)

Les arguments sont indiqués entre parenthèses après le nom de la méthode. Chaque argument est spécifié par son type et le nom de la variable.

Deux méthodes peuvent avoir le même nom, à condition d'avoir des arguments différents.



STRUCTURE D'UN PROGRAMME

(Les méthodes)

Exemple

```
public class Employe {  
    // Définition des variables d'instance  
    private String nom;  
    private int anneeNaissance;  
  
    ...  
  
    // Définition des méthodes  
    public void calculAge (){  
        int age = 2007 – anneeNaissance;  
        System.out.println ("Votre âge : " + age);  
    }  
    public void afficher (){  
        System.out.println( "Nom : " + nom + " Année de naissance : " +  
            anneeNaissance );  
    }  
}
```

STRUCTURE D'UN PROGRAMME

(Les méthodes)

Signature de la méthode main()

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

```
public class TestEmploye {  
    public static void main (String [] args){ // Méthode principale  
        Employe e1 = new Employe ("Tobago", 1965);  
        Employe e2 = new Employe ("Bonanza", 1969);  
        e1.afficher();  
        e1.calculAge();  
        e2.afficher();  
        e2.calculAge();  
    }  
}
```

STRUCTURE D'UN PROGRAMME

(Les méthodes)

Surcharge d'une méthode

Écrire plusieurs méthodes de **même nom** dans une classe. Les méthodes doivent différer par le nombre des arguments ou par les types des arguments.

Lors de la compilation, le compilateur vérifie qu'il y a compatibilité entre chaque appel des méthodes et la définition formelle des arguments des méthodes surchargées.

Puis, au moment de l'exécution, l'interpréteur choisit parmi les méthodes possibles celle qui convient d'exécuter.

Il est interdit de surcharger une méthode en changeant uniquement le type de retour.

```
void afficherPersonne (String nom, String rue, String ville){  
    System.out.println( "Nom : " + nom + "Rue : " + rue + "Ville :" ville );  
}  
  
void afficherPersonne (String nom){  
    System.out.println( "Nom : " + nom );  
}
```

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

Comme indiqué précédemment, un objet est une instance, un exemplaire construit dynamiquement sur le modèle que décrit la classe.

Pour cela, toute classe comporte au moins une **méthode particulière** appelée **constructeur**. Ses caractéristiques sont les suivantes :

- Il **porte le nom de la classe**, il n'est jamais appelé directement ; il est pris en compte lors de la demande de création de l'objet (avec l'opérateur **new**).
- Il ne retourne pas de valeur, **mais il ne faut pas indiquer void**.

Si le programmeur ne définit pas de constructeur, le compilateur en rajoute un (appelé **constructeur par défaut**) qui ne fait rien (**public NomClasse(){}**}).****

Dès qu'un constructeur est déclaré de manière explicite, le constructeur par défaut ne peut plus être invoqué.

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

Le constructeur permet essentiellement :

- de créer une instance
- d'initialiser les attributs de l' instance créée.

Il peut y avoir plusieurs constructeurs car plusieurs manières d' initialiser. Cependant, deux constructeurs d'une même classe ne peuvent avoir le même nombre et les mêmes types d'arguments (voir définition de la surcharge).

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

Fichier – *TestClient.java*

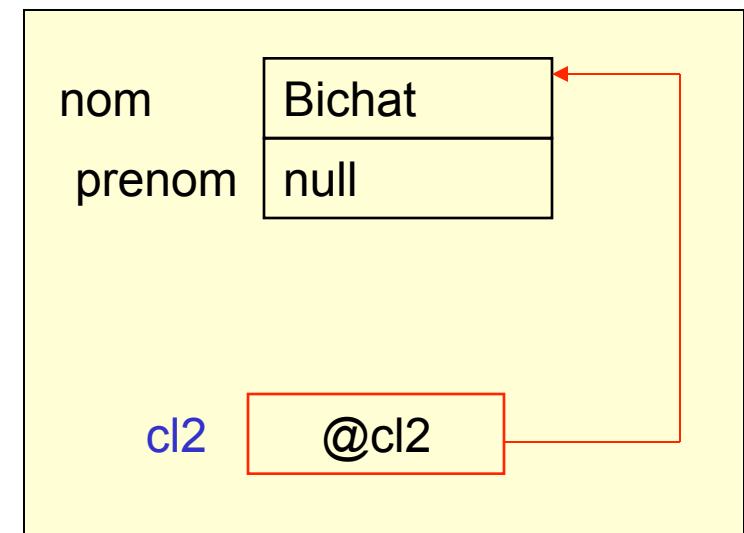
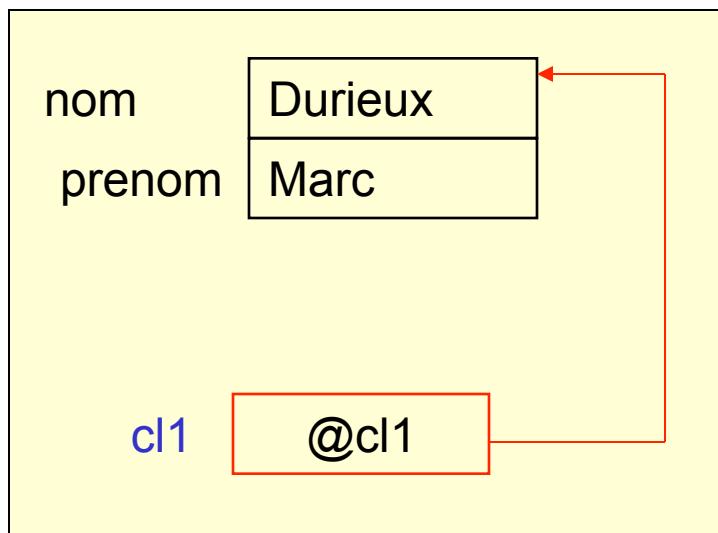
```
Class Client {  
    // Définition des attributs  
    String nom;  
    String prenom;  
  
    //Constructeurs  
    Client (String n, String p) {  
        nom = n;  
        prenom = p;  
    }  
    Client (String n) {  
        nom = n;  
        prenom = " ";  
    }  
}
```

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

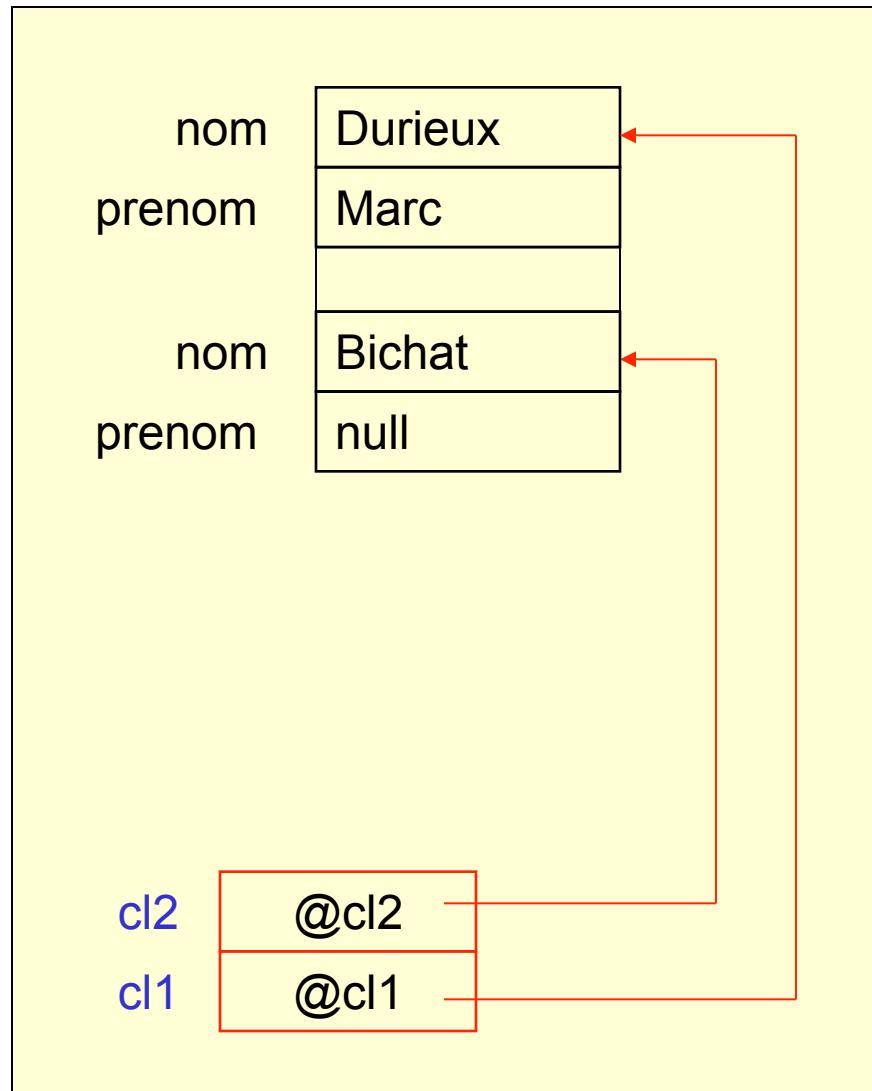
Fichier – *TestClient.java* (suite et fin)

```
public class TestClient {  
    // Définition de la méthode principale  
    public static void main(String[] args){  
        Client cl1, cl2;  
        cl1 = new Client("Durieux" , "Marc");  
        cl2 = new Client("Bichat");  
    }  
}
```



STRUCTURE D'UN PROGRAMME

(Les constructeurs)



STRUCTURE D'UN PROGRAMME

(Les constructeurs)

this

this est un mot clé qui est utilisé dans les deux cas suivants :

- pour référencer l'objet courant.
- pour représenter un des constructeurs de l'objet courant. Ce constructeur ne peut être appelé que par un autre constructeur de ce même objet.

Il est interdit de modifier *this*. Le code suivant est interdit : *this = valeur;*

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

this pour référencer l'objet courant

Le code d'une **méthode d'instance** (voir les méthodes de classe plus loin) désigne l'objet courant (l'objet qui a reçu le message – ou qui **exécute la méthode**) par le mot clé **this**.

Les membres de cet objet sont par conséquent préfixés par **this**.

Lorsqu'il n'y a pas d'ambiguïté, **this est optionnel** pour désigner un membre de l'instance courante

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

this pour référencer l'objet courant

```
class Client {  
    // Définition des attributs  
    String nom;  
    String prenom;  
    //Constructeurs  
    Client (String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    //méthode  
    public void setNom (String nom) {  
        this.nom = nom;  
    }  
}
```

Attribut de l'objet qui
exécute la méthode

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

this pour représenter un des constructeurs de l'objet courant

Le mot clé this, utilisé comme méthode, permet de représenter un des constructeurs de l'objet courant. Ce constructeur ne peut être appelé que par un autre constructeur de ce même objet.

L'appel à *this()* doit être **obligatoirement la première** instruction du constructeur.

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

```
class Client {  
    // Définition des attributs  
    String nom;  
    String prenom;  
    //Constructeurs  
    Client (String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    Client () { //constructeur sans argument  
        this(" ", " "); //appel du constructeur ci-dessus  
    }  
}
```

Notons que l'appel
Client (" ", " ") est incorrect

STRUCTURE D'UN PROGRAMME

(Les constructeurs)

```
class Segment {  
    int abscisse;  
    int ordonnee;  
    int longueur;  
    Segment (int abscisse, int ordonnee) {  
        this. abscisse = abscisse;  
        this. ordonnee = ordonnee;  
        longueur = ordonnee - abscisse;  
    }  
    Segment(Segment origine, int dep) {  
        //Appel du constructeur ci-dessus  
        this(origine. abscisse + dep, origine. ordonnee + dep);  
    }  
}
```

Chapitre 3 – Eléments du langage

- Types primitifs
- Types composites
- Opérateurs
- Transtypage
- Instructions conditionnelles et boucles
- Instruction d' interruption
- Tableaux

LES TYPES PRIMITIFS

On distingue deux grandes familles de types de variables : les types primitifs et les types « classe ». **On s' intéresse ici à la première famille.**

Les types primitifs de java permettent de manipuler des nombres entiers, des nombres réels, des caractères et des booléens. Ils sont semblables à ceux du C et C++, avec cependant les particularités suivantes :

- Tous les types entiers sont signés.
- Le type int est sur 32 bits, quelle que soit le processeur de la plate-forme d'accueil.
- Le type char est sur 16 bits (au lieu de 8).

LES TYPES PRIMITIFS

NOM	TAILLE (en bits)	
byte	8	Entiers signés
short	16	Entiers signés
int	32	Entiers signés
long	64	Entiers signés
float	32	Virgule flottante
double	64	Virgule flottante
boolean	1	true ou false
char	16	Non signé (Unicode)

LES TYPES COMPOSITES

Des classes d'emballage ont été prévues afin d'envelopper les types primitifs dans des objets. Elles fournissent des **méthodes permettant de manipuler facilement** les valeurs de ces types (par exemple des méthodes de conversion).

Types primitifs	Types composites
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

LES OPERATEURS

Arithmétiques :	+ , - , * , / , %
Affectations :	= , += , -= , *= ,
Unaires :	++ , --
Comparaisons logiques :	== , != , < , <= , > , >=
Logiques sur des booléens :	!, && ,
Ternaires :	?

```
int y = 2;  
y+= 10; //donne 12
```

```
int i =2;
```

```
int j = i++ // donne 2 pour j et 3 pour i car la valeur de l'opérande est retournée  
// avant qu'elle ne soit incrémentée.
```

```
Int i =2;
```

```
int j = ++i // donne 3 pour j et 3 pour i car la valeur de l'opérande est retournée  
// après qu'elle ne soit incrémentée.
```

```
int a = 2 ; int b = 8;  
int c = a>b ? a : b // c recevra comme valeur, la valeur max. entre a et b (soit 8).
```

TRANSTYPAGE

Le **transtypepage** est l'opération qui consiste à stocker dans une variable une valeur d'un autre type, qui sera "adaptée". (par exemple stocker dans une variable de type float, une valeur de type int).

Il est **implicite** lorsque l'on fait entrer le contenu d'une variable d'un type plus petit vers un type plus grand (par exemple un *int* dans un *long*).

La conversion implicite doit se faire suivant la hiérarchie suivante :

byte -> short -> int -> long -> float -> double
char -> int -> long -> double

Java refuse par exemple, de convertir implicitement une valeur de type *float* en *int*. Il faut donc dans ce cas demander un **transtypepage explicite**.

Un transtypepage **explicite** est déclaré à l'aide de l'opérateur dit de **cast**. Cet opérateur est le **nouveau type encadré par des parenthèses** et précédant la donnée concernée.

Exemple (un long dans un int) : int i = (*int*) variableLong;

TRANSTYPAGE

Exemples

```
int x = 14;  
long y= x; //transtypage implicite
```



```
int y;  
y= (int) 60.58; //transtypage explicite, la valeur 60 est affectée à y, perte d'information
```

```
int y ; float z;  
y = z+5; // sera rejetée par le compilateur
```



```
// Il faut donc faire un transtypage explicite  
y = (int) z+5;
```

INSTRUCTION CONDITIONNELLE

if ...else ...

```
if (expression booléenne) {  
    instructions  
}  
else {  
    instructions  
} // le bloc else est optionnel
```

Exemple

```
if (age < 18) {  
    System.out.println("Vous êtes mineur");  
}  
else {  
    System.out.println("Vous êtes majeur");  
}
```

INSTRUCTION CONDITIONNELLE

switch ...

```
switch (expression) { // entière ou caractère
    case valeur1:
        instructions;
        [break;]
    case valeur2 :
        instructions;
        [break;]
...
    default :
        instructions;
        [break;]
}
```

INSTRUCTION CONDITIONNELLE

switch ...

Exemple

```
switch (choix) { // variable caractère
    case 'o':
        System.out.println("Ouverture du fichier");
        break;
    case 'f':
        System.out.println("Fermeture du fichier");
        break;
    case 'x':
        System.out.println("Suppression du fichier");
        break;
    default :
        System.out.println("Merci de saisir une commande valide");
        break;
}
```

INSTRUCTION CONDITIONNELLE

switch ...

Exemple

```
switch (numMois) { // variable entière
    case 1: case 3: case 5: case 7: case 8:
    case 10: case 12:
        nombreDeJours = 31;
        break;
    case 4: case 6: case 9: case 11:
        nombreDeJours = 30;
        break;
    case 2 :
        nombreDeJours = 28;
        break;
    default :
        nombreDeJours = 0;
        System.out.println("Numéro de mois " + numMois + "incorrect" );
        break;
}
```

INSTRUCTION ITERATIVE

while et le do ... while

```
while(expression_booléenne) {  
    instructions;  
}  
  
do {  
    instructions;  
} while (expression_booléenne) ;
```

Exemple

```
int taille = 130;  
while(taille >230) {  
    System.out.println("la valeur saisie doit être inférieure à 230");  
}  
  
do {  
    System.out.println("la valeur saisie doit être inférieure à 230");  
} while (taille >230) ;
```

INSTRUCTION ITERATIVE

for ...

```
for( instruction_initiale ; condition_booléenne ; incrément) {  
    instructions;  
}
```

Exemple

```
Int j;  
for ( j=0 ; j < 10 ; j++) {  
    System.out.println (j)  
}
```

```
// Il est possible de déclarer la variable j dans la boucle for  
// la visibilité de j est alors limitée à la boucle  
for ( int j=0 ; j < 10 ; j++) {  
    System.out.println (j)  
}
```

INSTRUCTION D'INTERRUPTION

```
while (expression_booléenne) {  
    continue;  
    break;  
}  
  
do {  
    continue;  
    break;  
} while (expression_booléenne);
```

```
for( instruction_initiale ; condition_booléenne ; incrément) {  
    continue;  
    break;  
}
```

break permet de sortir d'une boucle

continue permet de sortir d'une boucle, mais en retournant à la condition booléenne

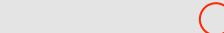
INSTRUCTION D'INTERRUPTION

Exemple

```
for ( int j=0 ; j < 20 ; j++) {  
    if( j == 4) continue ;  
    System.out.println ( "Valeur de j : " + j);  
    if( j == 6) break ;  
}
```



Résultat



Valeur de j : 0
Valeur de j : 1
Valeur de j : 2
Valeur de j : 3
Valeur de j : 5
Valeur de j : 6

Juste une illustration ...
Exemple à ne pas reproduire !!!

TABLEAUX (Définition)

Un tableau permet de regrouper plusieurs éléments de même type (primitif ou objet). On accède à chaque élément à l'aide d'un indice précisant le rang de l'élément dans le tableau.

En Java, la taille du tableau est définie au moment de sa création, et ne peut plus être changée par la suite.

Une fois créé, le tableau est vide, les cellules sont initialisées à 0 pour les tableaux de valeurs numériques, à *false* pour les booléens et à *null* pour les objets.

L'accès aux éléments du tableau se fait en spécifiant un nombre entier (de type byte, char, short ou int, **mais pas long**), indexé à partir de 0.

Un tableau est un objet, il possède un attribut *length* qui retourne la taille du tableau.

Une variable de type tableau contient donc la référence d'un objet un peu particulier. elle peut donc être passée en argument à une méthode (voir par exemple la méthode main).

TABLEAUX

(Déclaration)

Deux méthodes :

<type> [] nomTableau

<type> nomTableau []

Exemples

String [] tableauDeChaines;
int [] tableauDEntiers;

ou
ou

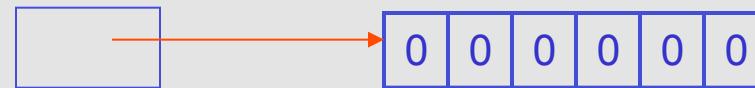
String tableauDeChaines [] ;
int tableauDEntiers [] ;

Remarque

L'instruction int[] tableauDEntiers déclare et définit la variable tableauDEntiers comme une référence sur un tableau d'entiers, mais ne réserve pas de place mémoire pour ce tableau. L'allocation doit se faire avec new en indiquant le nombre d'éléments.

tableauDEntiers = new int [6] ;

tableauDEntiers



TABLEAUX

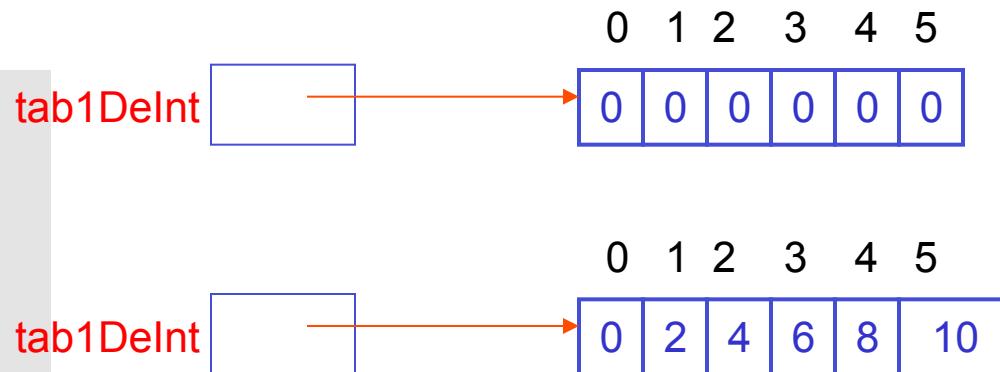
(Exemples)

Exemples

```
int [ ] tabDeInt = {10, 5, 6};  
for (int i=0; i< tabDeInt.length; i++) {  
    System.out.println( tabDeInt[i]+ " ");  
}
```

Une manière de déclarer et d'initialiser le tableau en une seule instruction

```
int [ ] tab1DeInt = new int[6];  
for (int i=0; i< tab1DeInt.length; i++) {  
    tab1DeInt[i] = 2 * i;  
}  
  
for (int i=0; i< tab1DeInt.length; i++) {  
    System.out.println( tab1DeInt[i]+ " ");  
}
```

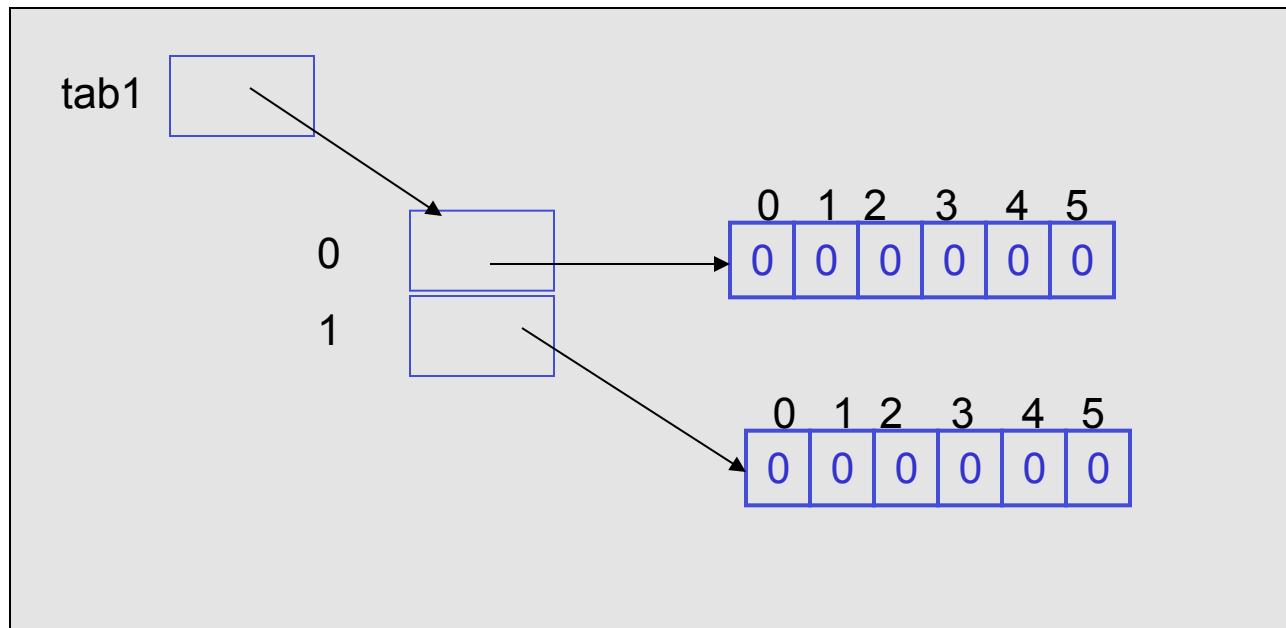


TABLEAUX MULTIDIMENSIONNELS

Les tableaux multidimensionnels sont des "tableaux de tableaux". Le nombre de dimensions est sans limite.

Exemple

```
int [ ] [ ] tab1 = new int [2] [6] ;
```



TABLEAUX MULTIDIMENSIONNELS

```
int [ ] [ ] tab1 = new int [2 ] [3 ] ;  
tab1[0][0] = 20 ;  tab1[0][1] = 30 ;  tab1[0][2] = 40 ;  
tab1[1][0] = 10 ;  tab1[1][1] = 20 ;  tab1[1][2] = 30 ;
```

```
for( int i=0; i< tab1.length ; i++) {  
    for (int j=0; j< tab1[i].length ; j++) {  
        System.out.println( tab1[i][j]) ;  
    }  
}
```

// Il est possible de créer un tableau multidimensionnel en plusieurs étapes

```
int [ ] [ ] tab ;  
tab = new int[2][ ];  
tab[0] = new int [10 ] ;  
tab[1] = new int [20 ] ;
```

Chapitre 4 – Membres de classe

- Variable de classe ou variable statique
- Méthode de classe ou méthode statique
- Bloc d'initialisation *static*
- Déclaration de constantes
- Exemples d'utilisation
 - Une bibliothèque personnelle de fonctions pour gérer la saisie clavier
 - La classe `java.lang.Math`

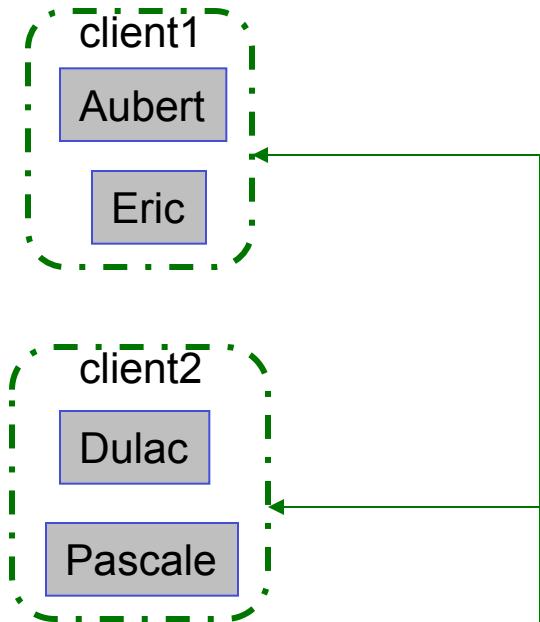
MEMBRES DE CLASSE

(Variable de classe ou variable statique)

Définition

```
Class Client {  
    String nom;  
    String prenom;  
    static int nbClients;  
}
```

1000



Variable de classe :

- est stockée dans la classe, elle existe dès que la classe est chargée
- est globale à **toutes les instances** de la classe
- on peut y accéder en dehors de toute instance
- est déclaré à l'aide du modificateur **static**

Variable d'instance :

- est stockée dans la mémoire de chaque objet
- définie dans la classe mais initialisée dans l'objet

MEMBRES DE CLASSE

(Variable de classe ou variable statique)

Définition

```
Client client1, client2;  
client1 = new Client();
```

```
client1.nbClients = 1000
```

client2.nbClients vaut aussi 1000

```
client2 = new Client();
```

client2.nbClients et client1.nbClients ont maintenant pour valeur : 2000

```
Client.nbClients = 2000
```

MEMBRES DE CLASSE

(Méthode de classe ou méthode statique)

Définition

Le modificateur **static** permet également de qualifier une méthode. La méthode (dite méthode de classe ou méthode statique) est alors **appelée directement par la classe**, sans nécessiter d'appel en provenance d'un objet.

Une méthode de classe exécute donc une action indépendante d'une instance particulière de la classe.

```
class Outils {  
    static double max(double x, double y){  
        if(x>y) return (x);  
        else return (y);  
    }  
}
```

Exemple d'appel

```
double c = Outils.max(5,11);
```

MEMBRES DE CLASSE

(Méthode de classe ou méthode statique)

Cas d'utilisation des méthodes de classe

On définit une méthode de classe lorsqu'elle doit intervenir avant la création des objets ou bien lorsqu'elle doit intervenir souvent, **sans avoir de rapport direct avec les objets.**

Il est par exemple plus correct de déclarer en *static* une méthode n'utilisant aucun attribut d'instance ou méthode d'instance de sa classe.

Une utilisation courante est la constitution de bibliothèques d'outils, comme par exemple la classe pré définie *Math* de Java.

Exemple de bibliothèque personnelle

```
class Outils {  
    static double max(double x, double y){  
        if(x>y) return (x);  
        else return (y);  
    }  
}
```

```
static double carre(double x){  
    return (x*x);  
}  
}
```

MEMBRES DE CLASSE

(Méthode de classe ou méthode statique)

Remarques importantes

Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante.

```
static double doublerPrime(){  
    return (this.prime * 2);  
}
```

Une méthode de classe **ne peut manipuler** (en lecture ou en écriture) des attributs d'instance de sa classe.

```
Class Client {  
    String nom;  
    String prenom;  
    static int nbClients;  
    static void afficherNbClients(){  
        System.out.println ("Nom : " +nom + "Prénom : " + prenom);  
        System.out.println ("Nombre de clients : " +nbClients);  
    }  
}
```

MEMBRES DE CLASSE (Méthode de classe ou méthode statique)

Remarques importantes – suite et fin

Cependant, une **méthode d'instance** peut manipuler des membres de classe, et ce, sans avoir besoin de créer une instance d'objet de cette classe.

```
Class Client {  
    String nom;  
    String prenom;  
    static int nbClients;  
  
    void afficher (){  
        System.out.println ("Nom : " +nom + "Prénom : " + prenom + "Nombre de  
                           clients : " + nbClients);  
    }  
}
```

Une *méthode de classe* ne peut être redéfinie dans les héritiers de la classe (voir chapitre sur l'Héritage).

MEMBRES DE CLASSE

(Bloc d'initialisation *static*)

Un bloc d'initialisation *static* permet d'initialiser les variables static complexes. Il est exécuté une seule fois, quand la classe est chargée en mémoire.

```
Class Tableau {  
    static int[] tab = new int[10];  
  
    static {  
        for(int i=0; i<tab.length; i++){  
            tab[i] = -1;  
        }  
    }  
}
```

MEMBRES DE CLASSE (Le modificateur final)

Un attribut déclaré avec le modificateur *static final* représente simplement une constante. Il sera donc nécessaire de toujours l'initialiser lors de sa déclaration. Sa valeur ne pourra pas être modifiée par la suite. **Les constantes sont nommées en majuscule**

Exemple : public **static final** double PI = 3.14;

Le modificateur *static final* permet également de qualifier une méthode ou une classe, il faut cependant respecter les contraintes suivantes :

- Une **méthode finale** ne peut pas être redéfinie dans une classe dérivée (voir chapitre sur l'héritage).
- Une **classe finale** ne peut pas être dérivée, elle ne pourra pas avoir d'héritier (voir chapitre sur l'héritage).

MEMBRES DE CLASSE

(Exemple d'utilisation – Une bibliothèque de fonctions pour gérer la saisie au clavier)

```
import java.io.*;
class Saisie {
    public static String lireChaine(String message){
        String ligne = null;
        try{
            //conversion d'un flux binaire en un flux de caractères (caractères Unicode)
            InputStreamReader isr = new InputStreamReader(System.in);
            //construction d'un tampon pour optimiser la lecture du flux de caractères
            BufferedReader br = new BufferedReader(isr);
            System.out.print(message);
            // lecture d'une ligne
            ligne = br.readLine();
        }
        catch (IOException e){
            System.err.println(e.getMessage());
        }
        return ligne;
   }// fin lireChaine
```

MEMBRES DE CLASSE

(Exemple d'utilisation – Une bibliothèque de fonctions pour gérer la saisie au clavier)

```
...
public static int lireEntier(String message){
    return Integer.parseInt(lireChaine(message));
}
public static double lireReel(String message){
    return Double.parseDouble(lireChaine(message));
}
}// fin classe Saisie
```

MEMBRES DE CLASSE

(Exemple d'utilisation – Une bibliothèque de fonctions pour gérer la saisie au clavier)

```
Class Client {  
    // Définition des attributs  
    private String nom;  
    private String prenom;  
    private int age;  
    private double solde;  
  
    //Constructeurs  
    public Client () {  
        nom = Saisie.lireChaine ("Votre nom ? ");  
        prenom = Saisie.lireChaine ("Votre prénom ? ");  
        age = Saisie.lireEntier ("Votre âge ? ");  
        solde = Saisie.lireReel("Votre solde ? ");  
    }  
}
```

Exemple d'utilisation de la classe Saisie

MEMBRES DE CLASSE

(Exemple d'utilisation – La classe java.lang.Math)

Cette classe proposée par Java, contient des **méthodes statiques** pour effectuer des opérations numériques telles que les fonctions trigonométriques, l'exponentielle, la racine carrée, ...

Quelques méthodes de cette classe

Valeur absolue :

type **abs** (type valeur) où type représente un type primitif numérique

Nombre aléatoire :

double **random()** : retourne un nombre aléatoire entre 0 et 1 ($0 \leq nb < 1$)

Puissances et racines :

double **pow(double a1, double a2)** : a1 à la puissance a2

double **sqrt (double a1)** : racine carrée de a1

Trigonométrie :

double **cos (double angle)** ; double **sin (double angle)** ;

double **tan (double angle)** ; ...

Chapitre 5 – Organisation des classes en Java

- Notion de paquetage (package)
- Les packages de Java

ORGANISATION DES CLASSES EN JAVA

(Notion de paquetage)

Définition

Il est possible de rassembler les classes selon des ensembles appelés **paquetages** (**package**). La notion de paquetage est proche du concept de **bibliothèque** que l'on rencontre dans d'autres langages.

Un paquetage est caractérisé par un nom qui est soit un identificateur, soit une suite d'identificateurs séparés par des points, comme par exemple :

chapitre3

chapitre3.utilitaires

java.awt.event

D'un point de vue technique, un package désigne un ensemble de classes compilées (fichiers .class), situées dans un même **répertoire**. Le répertoire où se situe le fichier compilé définit le nom du package, (en remplaçant les séparateurs de répertoires par des points).

Exemple :

Répertoire : java.awt.event Nom du package : java.awt.event

ORGANISATION DES CLASSES EN JAVA

(Notion de paquetage)

Attribution d'une classe à un paquetage

L'attribution d'un nom de paquetage se fait au niveau du fichier source, en plaçant en début du fichier, une instruction de la forme :

package nom_package; où *nom_package* représente le nom du package.

Toutes les classes d'un même fichier source appartiendront toujours à un même paquetage.

En l'absence de l'instruction *package* dans le fichier source, le compilateur considère que les classes appartiennent au paquetage par défaut (qui est unique pour une implémentation donnée).

ORGANISATION DES CLASSES EN JAVA

(Notion de paquetage)

Utilisation d'une classe d'un paquetage

Le compilateur recherche une classe dans le paquetage par défaut. Pour utiliser une classe appartenant à un autre paquetage, il est nécessaire de désigner la classe en utilisant une des deux syntaxes suivantes :

- (1) citer le nom du paquetage avec le nom de la classe.

Exemple :

```
bibliotheque.utilitaires.Date d = new bibliotheque.utilitaires.Date()  
d.getMonth();
```

- (2) Utiliser une instruction *import*.

Exemple :

```
import bibliotheque.utilitaires.Date  
Date d = new Date();  
d.getMonth();
```

N.B. On peut aussi importer toutes les classes d'un paquetage :

Exemple : *import bibliotheque.utilitaires.** ;

ORGANISATION DES CLASSES EN JAVA

(Les paquetages de Java)

La machine virtuelle de Java est livrée avec de nombreux paquetages dont voici les principaux :

java.lang regroupe les classes faisant partie intégrante du langage Java.

java.io permet de représenter les flux d'entrée et les flux de sortie.

java.net permet de développer des applications réseaux.

java.awt regroupe les classes permettant de réaliser des IHM.

javax.swing complète le package *java.awt*.

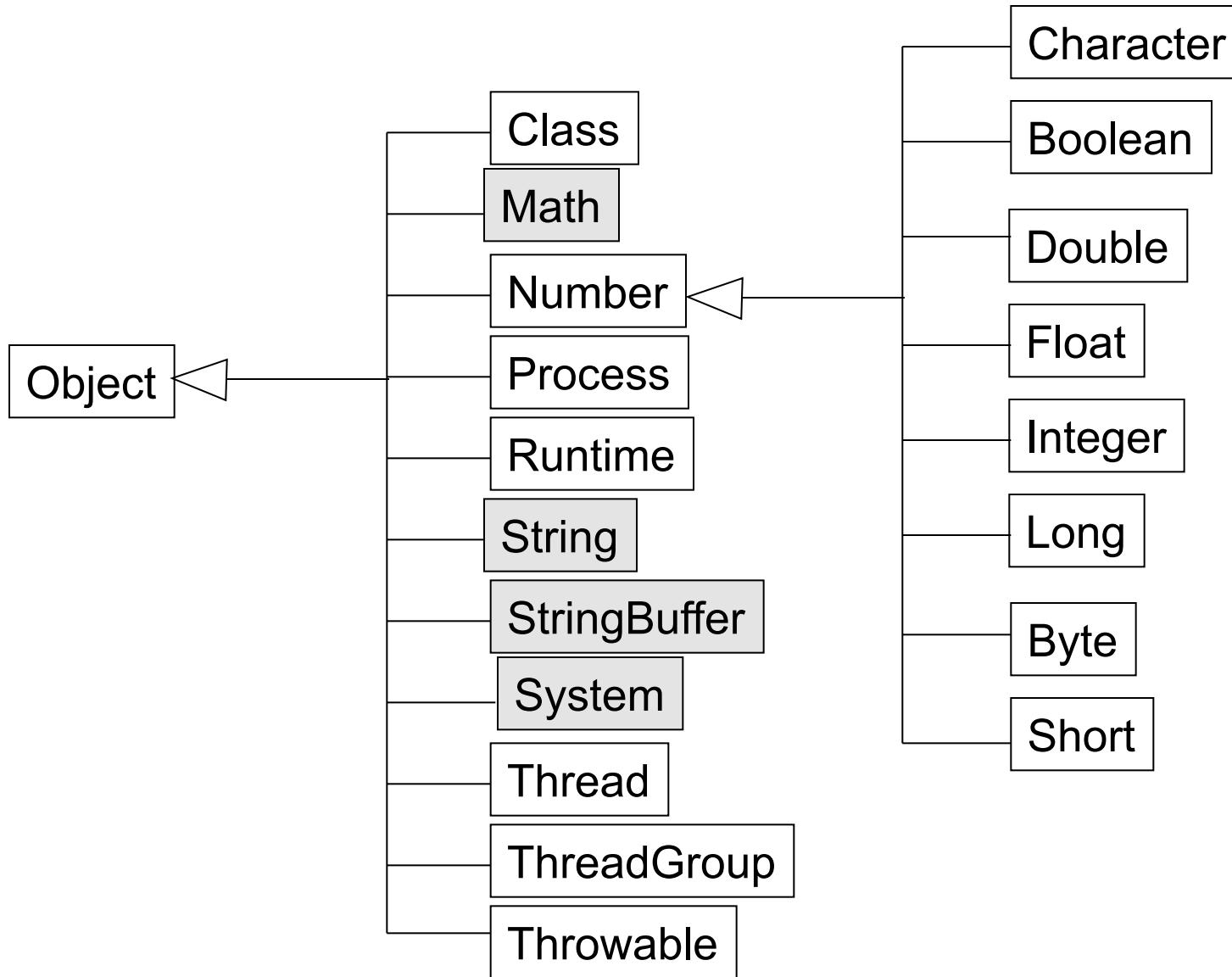
java.util propose des classes utilitaires.

java.sql permet de communiquer avec des bases de données.

N.B. Le package *java.lang* qui est **automatiquement** importé par le compilateur. Il contient les classes String, StringBuffer, Math, ...

ORGANISATION DES CLASSES EN JAVA

(Les paquetages java.lang)



Chapitre 6 – Les chaînes de caractères

- La classe `java.lang.String`
- La classe `java.lang.StringBuffer`
- La classe `java.util.StringTokenizer`

LES CHAINES DE CARACTERES

(java.lang.String)

Java propose une classe standard nommée `String`, pour manipuler les chaînes de caractères.

Les caractères des chaînes sont codés en UNICODE sur 16 bits.

la notation littérale `String chaine = "Bonjour"` est transformée par le compilateur en
`String chaine = new String("Bonjour");`

La classe `String` dispose de deux constructeurs, l'un sans argument créant une chaîne vide, l'autre avec un argument de type `String` qui en crée une copie.

Exemple :

```
String ch1 = new String() // ch1 contient la référence à une chaîne vide  
String ch2 = new String("Bonjour") // ch2 contient la référence à une chaîne  
contenant la suite "Bonjour"
```

chaine

@

Bonjour

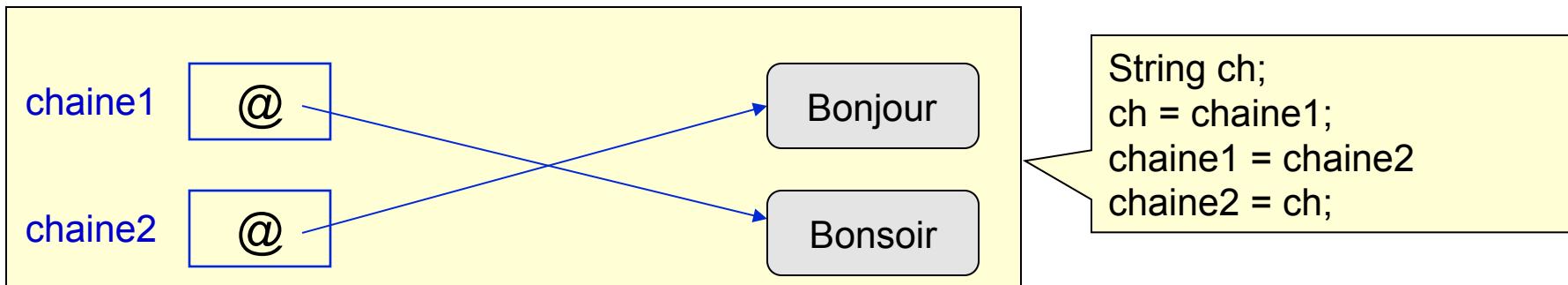
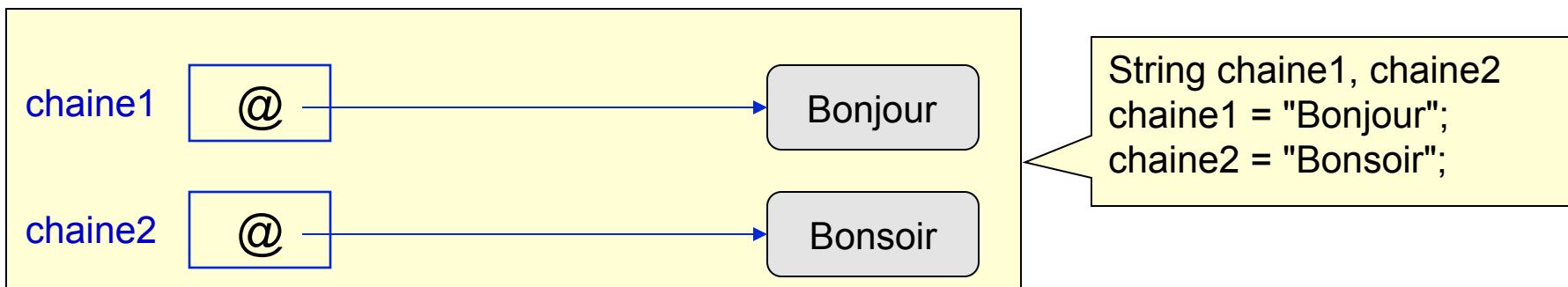
LES CHAINES DE CARACTERES

(java.lang.String)

Attention!!

Un objet de type String ne peut être modifiée. En cas de modification, les méthodes fournissent **un nouvel objet**.

Il ne faut cependant pas perdre de vue que les variables contiennent des **références** à des objets.



LES CHAINES DE CARACTERES

(java.lang.String)

Quelques méthodes de la classe String

int length () donne la taille de la chaîne.

char charAt(int position) donne un caractère à une position.

int indexOf(char car) retourne la position dans la chaîne du caractère passé en argument.

String substring (int posDebut) retourne une sous-chaîne

String substring (int posDebut, posFin) retourne une sous-chaîne

String replace (char oldchar, char newchar) remplace un caractère par un autre.

String concat(String chaine1) retourne un nouvel objet String, créé à partir de la concaténation de la chaîne dans laquelle est invoquée cette méthode et de la chaîne passée en argument.

LES CHAINES DE CARACTERES

(java.lang.String)

Quelques méthodes de la classe String

String trim() retourne une nouvelle chaîne de caractères sans les espaces en début et en fin de la chaîne.

String toUpperCase() transforme toutes les minuscules en majuscules.

String toLowerCase() transforme toutes les majuscules en minuscules.

boolean equals() teste l'égalité entre deux objets de type String

int compareTo(String autreChaine) compare avec une autre chaîne. Le résultat est :
0 si les deux chaînes sont identiques.

un nombre négatif si la chaîne en argument est plus grande

un nombre positif si la chaîne en argument est plus petite

N.B. Toutes ces méthodes fournissent un nouvel objet.

LES CHAINES DE CARACTERES

(java.lang.StringBuffer)

Java propose également la classe *StringBuffer* destinée elle aussi à la manipulation de chaînes, mais dans laquelle les objets sont modifiables.

La classe *StringBuffer* représente donc une chaîne de caractères modifiable. Elle est implantée de façon à **rendre performantes** les opérations de chaînes (concaténation, insertion, ...).

Les constructeurs

StringBuffer() : création d'un StringBuffer contenant 0 caractère avec une capacité par défaut de 16 caractères.

StringBuffer(int taille) : la taille est précisée en argument.

StringBuffer(String s) : création d'un StringBuffer à partir d'une chaîne.

Les méthodes

StringBuffer insert(int pos, Type argument) Type = char | String | Objet

StringBuffer append(String s) effectue une concaténation

StringBuffer delete(int debut, int fin) suppression

StringBuffer deleteCharAt(int pos)

StringBuffer replace(int debut, int fin, String chaine) remplacement

StringBuffer reverse() : inversion de la chaîne

LES CHAINES DE CARACTERES

(java.lang.StringBuffer)

Exemple

```
public void remplacer(StringBuffer buf, String oldChaine, String newChaine){  
    if(buf.indexOf (oldChaine)>0) //si oldChaine se trouve dans buf  
        buf.replace(buf.indexOf(oldChaine), oldChaine.length(), newChaine);  
}
```

LES CHAINES DE CARACTERES

(java.util.StringTokenizer)

Cette classe utilitaire permet de décomposer une chaîne en plusieurs chaînes à partir d'un délimiteur.

Constructeurs

StringTokenizer (String chaine) délimite la chaîne passée en argument avec le caractère espace (" ").

StringTokenizer (String chaine, String delimiteur) délimite la chaine en utilisant le délimiteur passé en argument.

Méthodes

int countTokens() retourne le nombre de sous-chaînes.

boolean hasMoreTokens() retourne true s'il existe encore un élément.

boolean hasMoreElements() retourne true s'il existe encore un élément.

String nextToken() retourne l'élément suivant

LES CHAINES DE CARACTERES

(java.util.StringTokenizer)

Exemple

```
import java.util.StringTokenizer

public class Tokenizer {
    public static void decoupeLigne(String ligne){
        StringTokenizer ligneToken =new StringTokenizer (ligne, ";");
        for(int i = 0 ; ligneToken.hasMoreTokens(); i++){
            String element = ligneToken.nextToken();
            System.out.println(element);
        }
    }//fin méthode

    public static void main(String[] args){
        String ligne = "Lundi;Mardi;Mercredi;Jeudi;Vendredi"
        Tokenizer tokenizer = new Tokenizer ();
        tokenizer. decoupeLigne(ligne);
    }//fin méthode main
}// fin classe
```

LES CHAINES DE CARACTERES

(java.util.StringTokenizer)

Exemple

```
import java.util.StringTokenizer

public static String[] getWords(String chaine, String delimiteur){
    StringTokenizer stoken = new StringTokenizer(chaine, delimiteur);
    int nbmots =stoken.countTokens();
    String[] stringres = new String[nbmots];
    for(int i=0;i<nbmots;i++){
        stringres[i]=stoken.nextToken();
    }
    return stringres;
}
```

Chapitre 7 – Les structures d’objets

- Les tableaux d’objets
- Les vecteurs d’objets

STRUCTURES D' OBJETS

(Les Tableaux d'objets)

Ce chapitre présente respectivement, les tableaux d' objets et des structures plus sophistiquées basées sur des classes prédéfinies du package *java.util*.

Les Tableaux d' objets

Un tableau d' objets doit répondre aux deux caractéristiques suivantes :

- Il a une **taille fixe**
- Il est défini par le type des objets : chaque case du tableau est implémentée selon le type choisi et pourra contenir **l' adresse** d' un objet de ce type (ou d' un type dérivé)

```
//Tableau d' employés  
//déclaration de la variable qui désigne le tableau  
Employe [ ] employes;  
  
//allocation en mémoire des cases nécessaires au tableau  
employes = new Employe [TAILLE];
```

Lorsque **new** crée un tableau d'objets, il n'y a que deux étapes (au lieu de trois) : (1) allocation mémoire (2) retour de l'adresse du tableau

STRUCTURES D' OBJETS

(Les Tableaux d'objets)

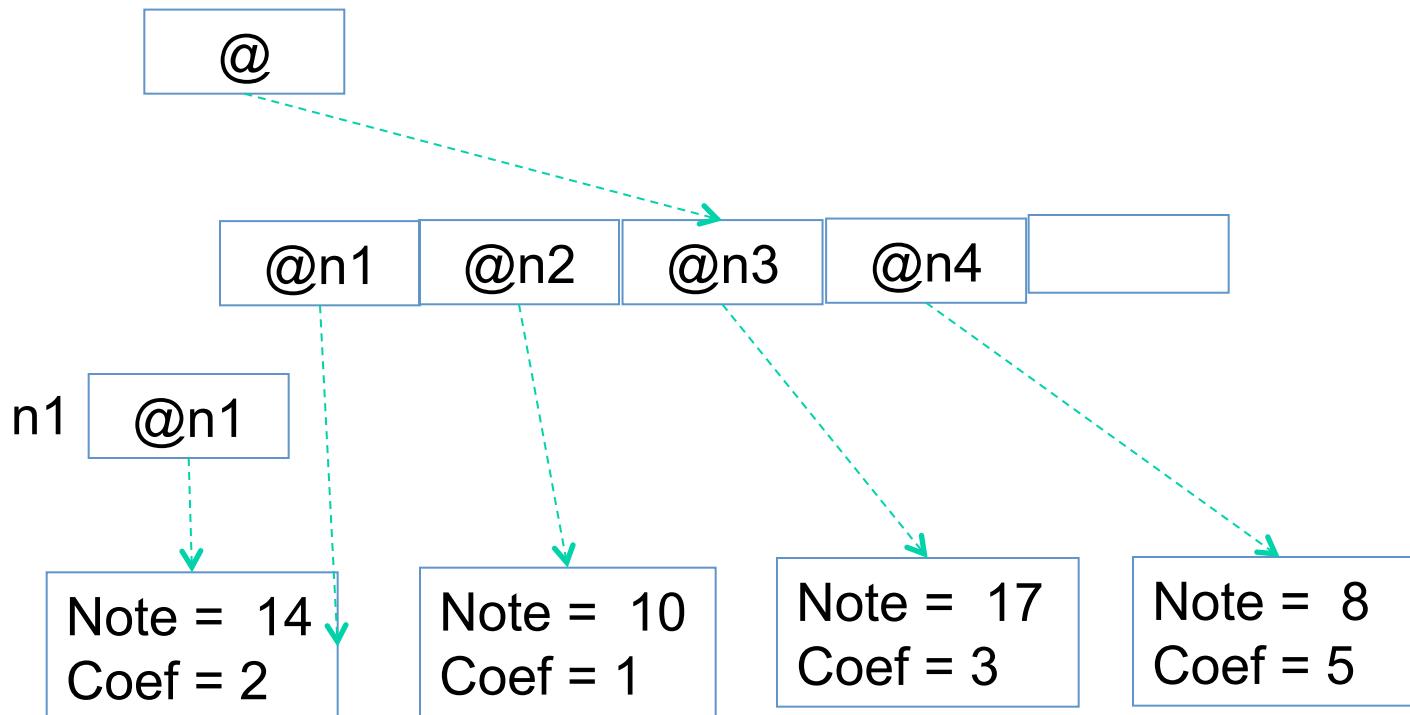
Notation [] notations = new Notation [5]

Notations [0] = n1

notations

@

Notation n1 = new Notation (14,2)

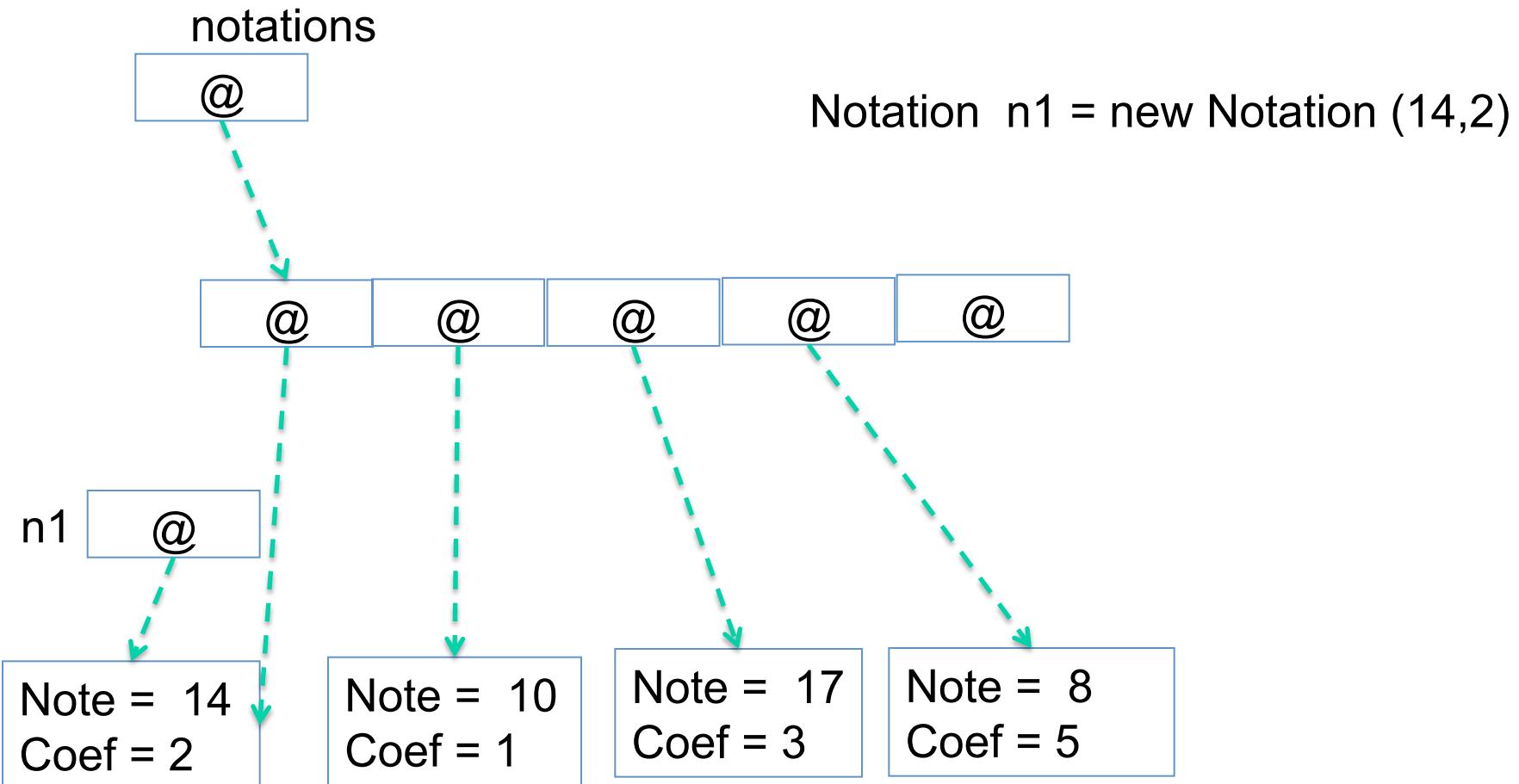


STRUCTURES D' OBJETS

(Les Tableaux d'objets)

Notation [] notations = new Notation [5]

Notations [0] = n1



STRUCTURES D' OBJETS

(Les vecteurs d'objets)

Les vecteurs d' objets sont construites à l'aide de la classe **Vector** du package **java.util**.

La classe **Vector** permet de créer des tableaux d'objets appelés **tableaux vectoriels**, dont **la taille est dynamique** . .

Remarque

Avant la version J2SE 5.0, la classe Vector permettait de créer des tableaux dont le contenu pouvait être hétérogène (objets de types Employe, Voiture, Client, ... dans le même vecteur) . La raison est que toutes les cases du tableau sont de type **Object** (type le plus général dans Java). **Il est cependant conseillé d'éviter cette pratique.**

A partir de la version J2SE 5.0, il est possible de créer des vecteurs d' objets **selon un type déterminé**.

STRUCTURES D' OBJETS

(Les vecteurs d'objets)

La classe Vector propose des **méthodes** (environ cinquante) qui permettent d'ajouter, de supprimer, de rechercher des éléments, ...

Constructeurs

Vector<TYPE>() construit un vecteur de taille initiale 10.

Vector<TYPE>(int taille) construit un vecteur en spécifiant une taille initiale.

Quelques Méthodes

addElement(Object element) ajoute l'élément à la fin.

insertElementAt(Object element, int indice) insère un élément à l'indice passé en argument.

STRUCTURES D' OBJETS

(Les vecteurs d'objets)

Quelques Méthodes (suite)

setElementAt(Object element, int indice) remplace l'élément positionné à l'indice passé par celui passé en argument.

removeElementAt(int indice) retire l'élément désigné par l'indice et décale les éléments qui le suivent à un indice inférieur.

elementAt(int indice) renvoie l'élément désigné par l'indice passé comme argument

isEmpty() renvoie true si le vecteur ne contient pas d'éléments

size() retourne le nombre d'éléments insérés dans le vecteur

STRUCTURES D' OBJETS

(Les vecteurs d'objets)

Nous pouvons par exemple déclarer un vecteur d' objets de type *Employe* en considérant la classe *Vector<Employe>*.

Exemple 1

```
import java.util.Vector

public class CollectionEmployes {
    private Vector<Employe> employes;
    private int nbEmp;

    public CollectionEmployes(){
        employes = new Vector<Employe>();
        nbEmp = 0;
    }//fin constructeur

    public void ajoutEmploye(Employe e){
        employes. add(e);
    }
    .....
}
```

STRUCTURES D' OBJETS

(Les vecteurs d'objets)

La version J2SE 5.0 propose également une nouvelle formulation de la **boucle for**. Cette formulation est illustrée par l' exemple ci-dessous.

```
import java.util.Vector

public class CollectionEmployes {
    private Vector<Employe> employes;
    private int nbEmp;

    public CollectionEmployes(){
        employes = new Vector<Employe>();
        nbEmp = 0;
    }//fin constructeur

    public void afficheEmployes(){
        // pour chaque variable emp de type Employe qui appartient à la collection d' objets
        // désignée par la variable employes.
        for(Employe emp : employes) {
            emp.affiche();
        }
    }
    .....
}
```

STRUCTURES D' OBJETS

(Les vecteurs d'objets avec J2SE 1.4.2)

Exemple 1 (avec J2SE 1.4.2)

```
import java.util.Vector

public class CollectionEmployes {
    private Vector employes;
    private int nbEmp;

    public CollectionEmployes(){
        employes = new Vector();
        nbEmp = 0;
    }//fin constructeur

    public void ajoutEmploye(Employe e){
        employes. addElement(e);
    }
.....
}
```

STRUCTURES D' OBJETS

(Les vecteurs d'objets avec J2SE 1.4.2)

Exemple 2 (avec J2SE 1.4.2)

```
import java.util.Vector

public class Collection {
    public static void main(String[] argv){
        //utilisation d'un vecteur
        Vector jours = new Vector();
        jours.addElement("Lundi");
        jours.addElement("Mardi");
        jours.addElement("Mercredi");
        for(int i=0;i< jours.size();i++){
            System.out.println(jours.elementAt(i));
        }
    }
}
```

...

STRUCTURES D' OBJETS

(Les vecteurs d'objets avec la classe **ArrayList**)

La classe **ArrayList** peut-être utilisée à place de la classe Vector

Constructeurs

ArrayList<TYPE>() la taille est initialisée à 10.

ArrayList<TYPE>(int taille) en spécifiant une taille initiale.

Quelques Méthodes

add(Object element) ajoute l'élément à la fin.

add(int indice, Object element) insère un élément à l'indice passé en argument.

STRUCTURES D' OBJETS

(Les vecteurs d'objets avec la classe **ArrayList**)

Quelques Méthodes (suite)

set(int indice, Object element) remplace l'élément positionné à l'index *indice*, par celui passé en argument.

remove(int indice) retire l'élément désigné par l'indice et décale les éléments qui le suivent à un indice inférieur.

get(int indice) renvoie l'élément désigné par l'indice passé comme argument

isEmpty() renvoie true si la structure ne contient pas d'éléments

size() retourne le nombre d'éléments insérés dans le vecteur

STRUCTURES D' OBJETS

(Les vecteurs d'objets avec la classe **ArrayList**)

Exemple

```
//utilisation d'un arrayList
ArrayList<String> jours = new ArrayList<String> ();
jours.add("Lundi");
jours.add("Mardi");
jours.add("Mercredi");
for(int i=0;i< jours.size();i++){
    System.out.println(jours.get(i));
}
//fin méthode
}// fin classe
```

Chapitre 8 – Héritage

- Définition et Intérêt
- Accès d'une classe dérivée aux membres d'une classe de base
- Exemple
- Le mot-clé *super*
- Redéfinition
- Classes abstraites
- Interfaces
- La super-classe Object
- Transtypage des objets

Chapitre 8 – Héritage, Classes abstraites et Interface

- Notion d' héritage
 - définition et intérêt
 - héritage en Java
 - exemple
- Héritage et redéfinition
- Utilisation du mot-clé *super*
- Notions de *classes abstraites* et *d'interfaces*

HERITAGE (Définition et intérêt)

L'héritage est une technique qui permet d'utiliser une (ou plusieurs) classe existante (classe de base) pour en créer une nouvelle, dite classe dérivée.

Exemple :

Créer la classe *Etudiant* à partir de la classe *Personne*

Créer la classe *Voiture* à partir de la classe *Vehicule*

La nouvelle classe **hérite** naturellement des membres (attributs et méthodes) de la classe de base *qu'elle pourra modifier ou compléter, sans toucher à la classe de base.*

Une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée.

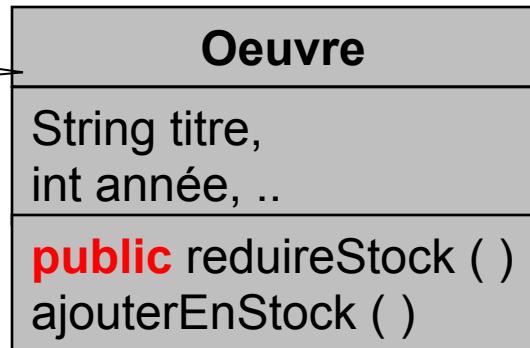
HERITAGE (Définition et intérêt)

La classe de base est encore appelée : *super-classe* ou *classe parente*.

La classe dérivée est appelée *sous-classe* ou *classe fille*.

L'objectif est de créer des classes **de plus en plus précises**, héritant des caractéristiques de classes plus générales.

Super-classe ou
Classe parente



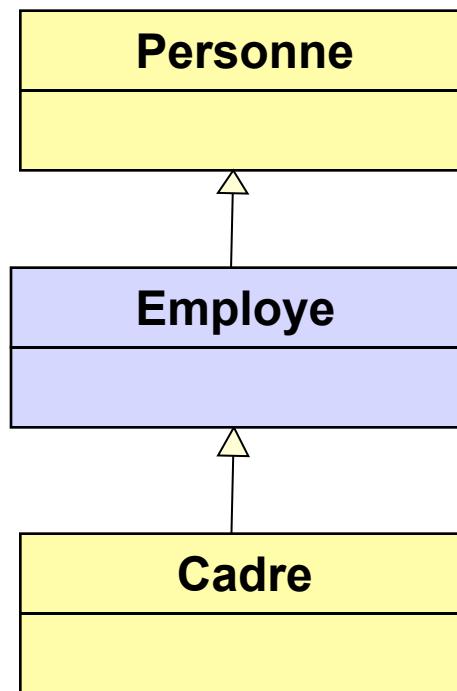
Sous-classe ou
Classe fille

HERITAGE

(Définition et intérêt)

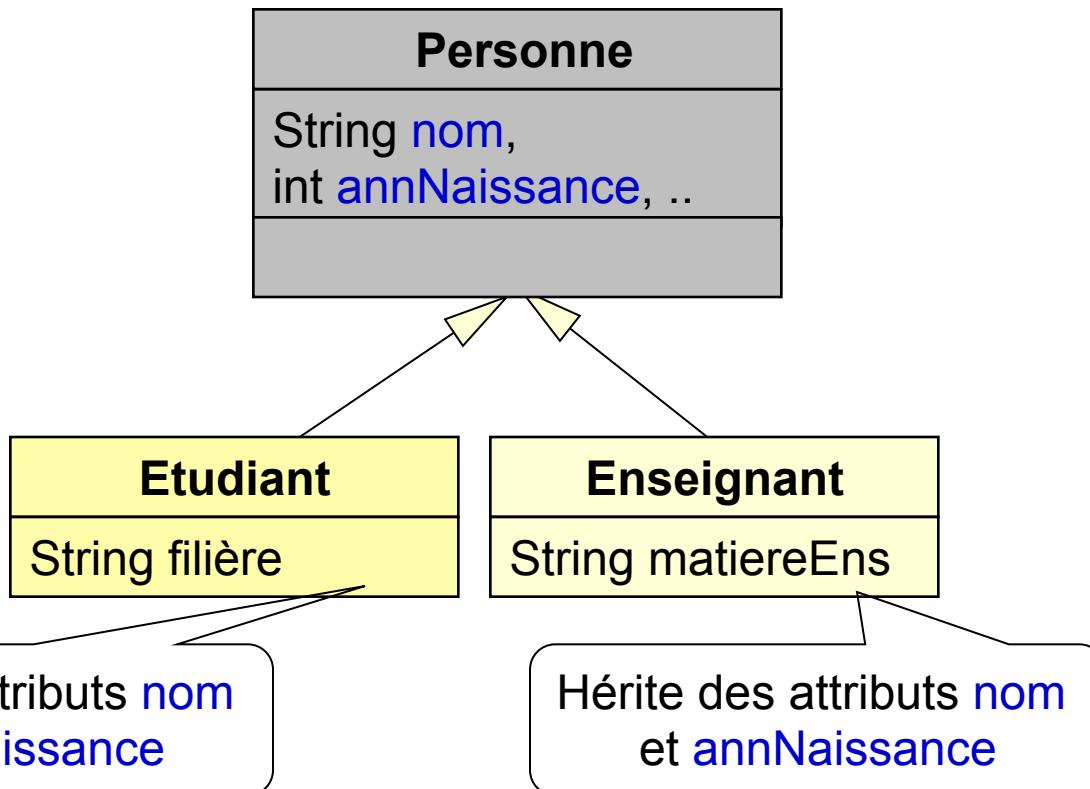
Héritage à plusieurs niveaux

Une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée.



HERITAGE (Définition et intérêt)

Un **objet** de la classe *Etudiant* ou *Enseignant* est **forcément** un objet de la classe *Personne*, il dispose donc de tous les membres de la classe *Personne*.

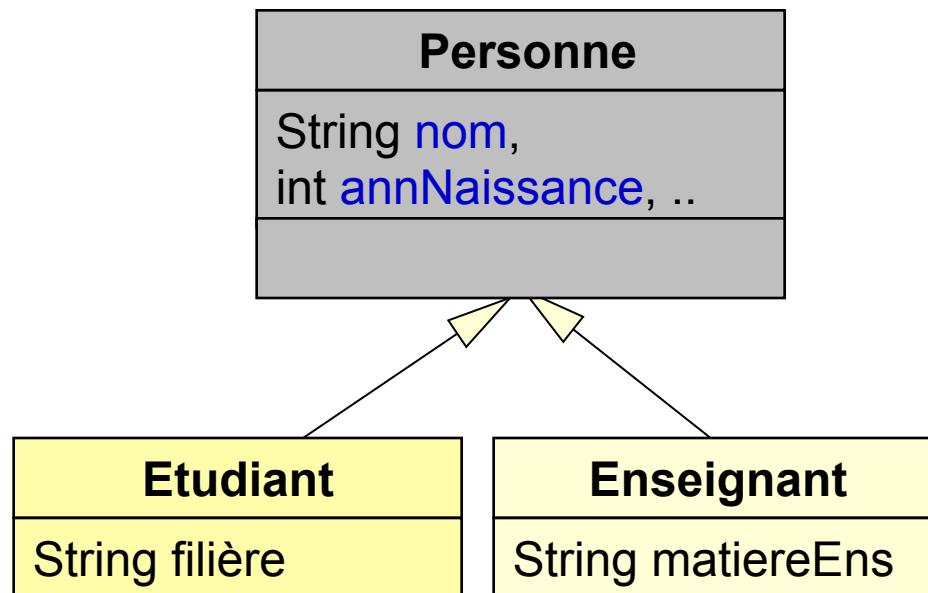


HERITAGE (Définition et intérêt)

Attention :

Un **objet** de la classe *Etudiant* ou *Enseignant* est **forcément** un objet de la classe *Personne*.

Par contre, un **objet** de la classe *Personne* **n'est pas forcément** un objet de la classe *Etudiant* ou *Enseignant* !!!!



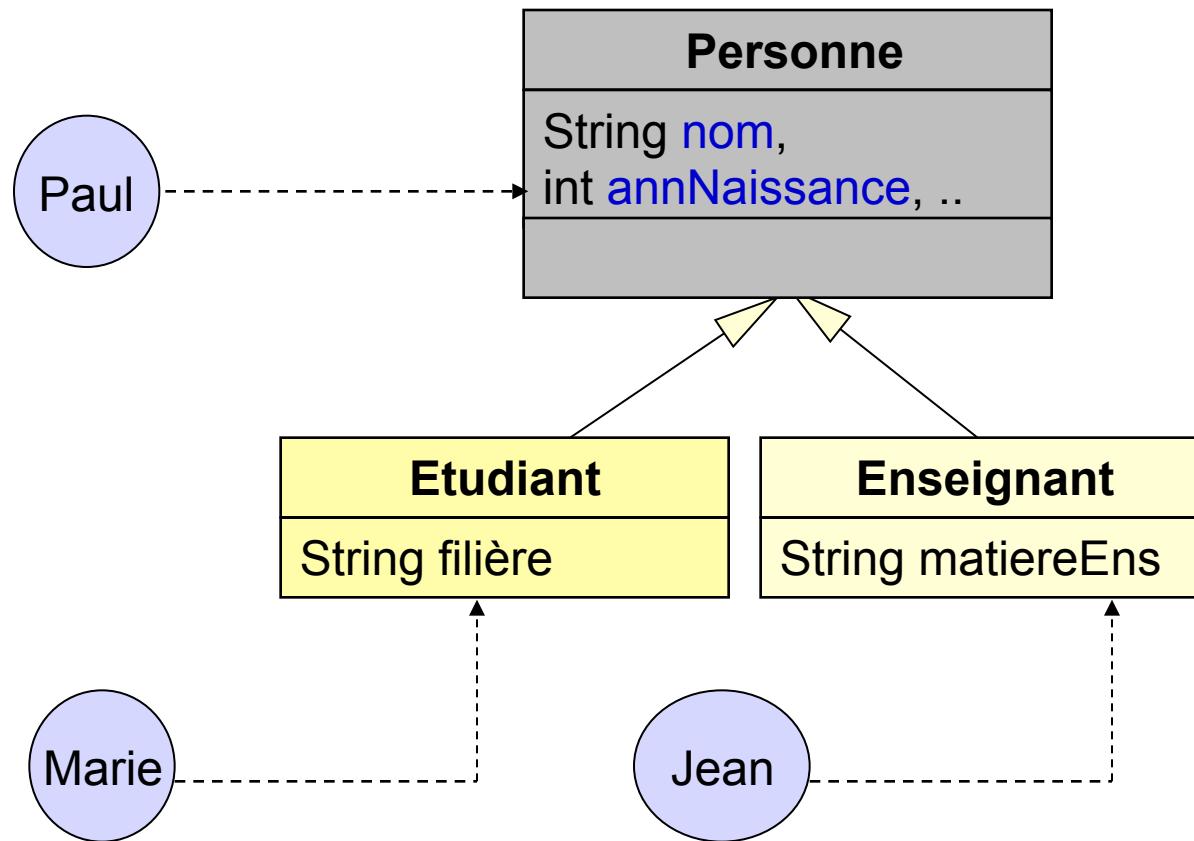
HERITAGE

(Définition et intérêt)

Attention :

Marie et Jean sont forcément des Personnes (Objets de la classe Personne).

On ne sait pas si Paul est Etudiant ou Enseignant ????



HERITAGE (Définition et intérêt)

Attention :

Un **objet** de la classe *Etudiant* ou *Enseignant* est **forcément** un objet de la classe *Personne*.

Un **objet** de la classe *Personne* **n'est pas forcément** un objet de la classe *Etudiant* ou *Enseignant* !!!!

Personne etudiant = new Personne();

Personne ens = new Personne()

Etudiant etudiant = new Etudiant()

HERITAGE

(Définition et intérêt)

Intérêts

(1) Spécialisation

La nouvelle classe **réutilise** une classe existante **en y ajoutant** des attributs et/ou des méthodes supplémentaires.

Par exemple on réutilise la classe *Employer* pour créer la classe *Commerciale* en y ajoutant une méthode *prime()*.

(2) Redéfinition

La nouvelle classe **redéfinit** les attributs et/ou les méthodes d'une classe existante de manière à **en changer** le sens ou le comportement.

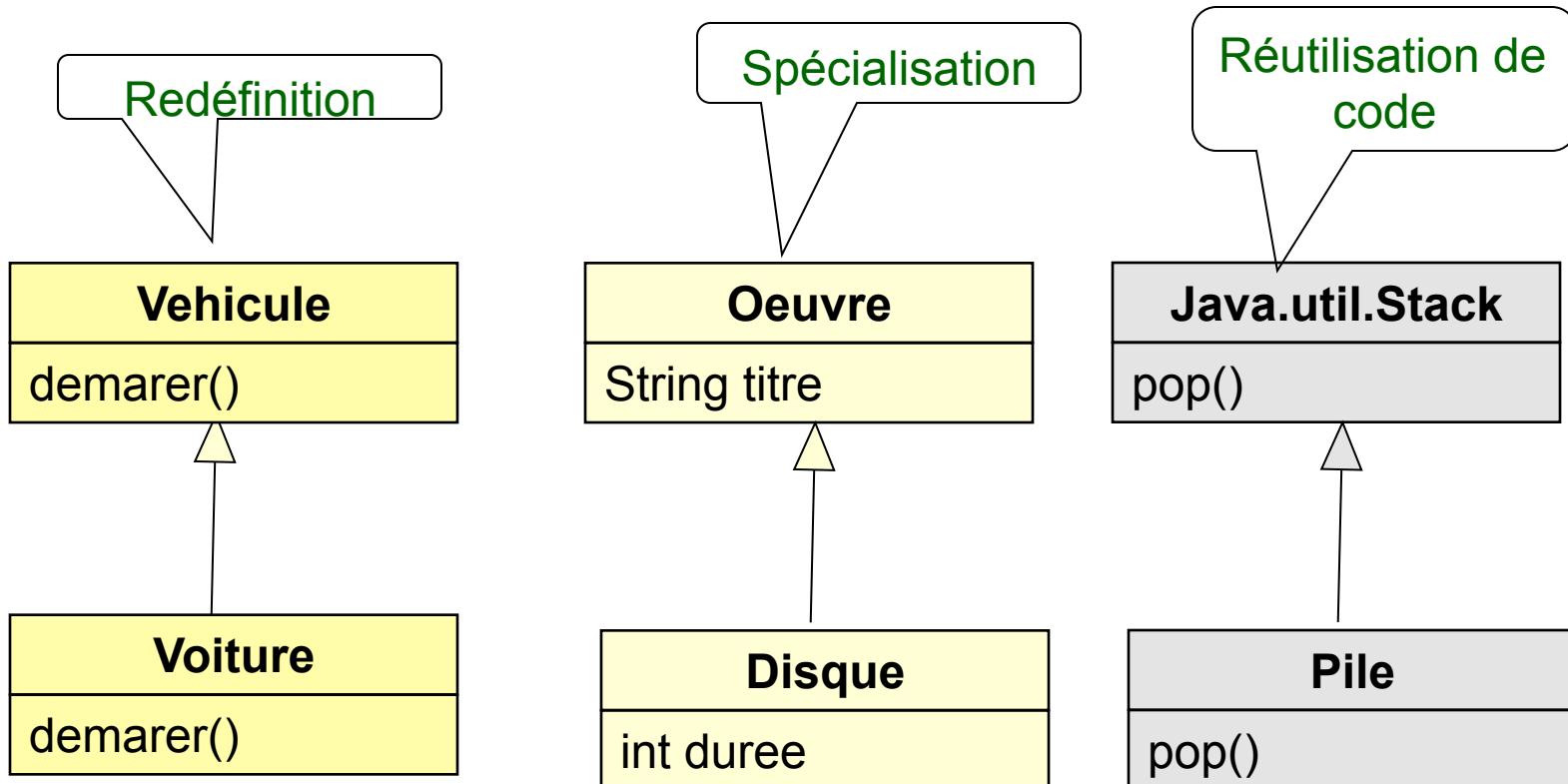
Par exemple on réutilise la classe *Vehicule* pour créer la classe *Voiture*, mais, en redéfinissant la méthode *avancer()*.

(3) Réutilisation de code

Réutiliser du code existant, **même si on ne possède pas les sources** de la classe de base.

HERITAGE

(Définition et intérêt)



HERITAGE (Héritage en Java)

En Java, une classe de base peut avoir plusieurs classes dérivées, mais une classe dérivée ne peut hériter que d'une seule classe de base : on parle d'*héritage simple*.

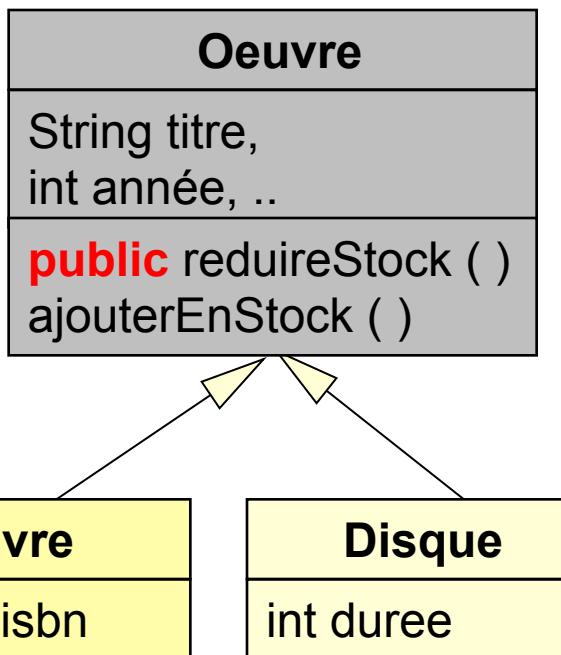
Java ne propose donc *pas l'héritage multiple* qui permet à une classe dérivée d'hériter de plusieurs classes quelconques.

On peut cependant partiellement contourner le problème grâce à la notion d'*Interface* : Java permet à une classe dérivée qui hérite déjà d'une classe de base *d'hériter* d'une ou plusieurs classes particulières appelées *Interfaces*.

HERITAGE

(Héritage en Java)

En java, on utilise le mot clé **extends** pour spécifier la classe dont on hérite.



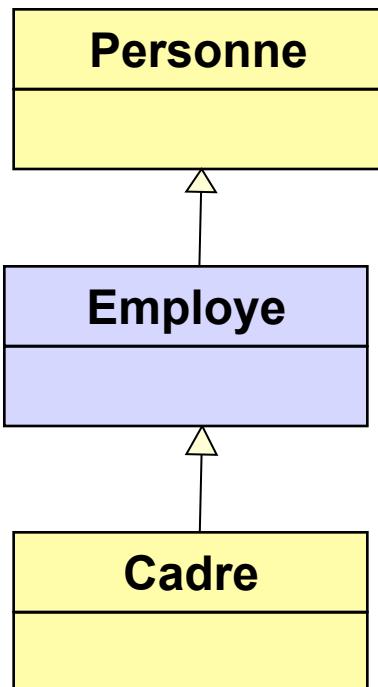
```
public class Livre extends Oeuvre {
    String isbn;
}

public class Disque extends Oeuvre {
    int duree;
}
```

HERITAGE

(Héritage en Java)

En java, on utilise le mot clé **extends** pour spécifier la classe dont on hérite.



```
public class Personne {  
    String nom, prenom;  
    byte age;}
```

```
public class Employe extends Personne {  
    String societe;  
    float salaire;  
    public double prime(){  
        return salaire * 0.1;}  
}
```

```
public class Cadre extends Employe {  
    public double prime(){  
        return salaire * 0.4;}  
}
```

HERITAGE

(Redéfinition et surcharge)

Rappel de la surcharge

Possibilité de définir des méthodes possédant le **même nom** mais dont **les arguments diffèrent**.

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

Redéfinition d' une méthode

On peut **redéfinir** une méthode définie dans une classe de base, à l'intérieur d'une classe dérivée. Il faut que la méthode redéfinie ait :

- même nom,
- mêmes types et nombre d'arguments,
- même type de retour, que la méthode d'origine

HERITAGE

(Redéfinition et surcharge)

```
public class Employe {  
    String societe;  
    float salaire;  
    public double prime(){  
        return salaire * 0.1;}  
}
```

```
public class Cadre extends Employe {  
    public double prime(){  
        return salaire * 0.4;}  
}
```

Redéfinition de la méthode

HERITAGE

(Redéfinition et surcharge)

```
public class Employe {  
    String societe;  
    float salaire;  
    public double prime(){  
        return salaire * 0.1;}  
}
```

Redéfinition
(Spécialisation de la méthode existante)

```
ublic class Cadre extends Employe {  
    public double prime(){  
        return salaire * 0.4;}  
  
    public double prime(float coef){  
        return salaire * coef;}  
}
```

Surcharge
(Ajout d'une méthode)

HERITAGE

(Redéfinition et surcharge)

Redéfinition d' une méthode

La redéfinition d'une méthode "**écrase**" le code de la méthode héritée.

Il est cependant possible de **réutiliser** le code de la méthode héritée grâce au mot clé **super**.

super permet de désigner explicitement l'instance d'une classe dont le type est celui de la classe mère.

Par exemple, dans la classe **Employe** (sous classe de Personne), **super** permet de désigner explicitement **une instance de la classe Personne**.

super permet donc l'accès aux attributs et méthodes redéfinies par la classe courante mais que l'on souhaite utiliser

HERITAGE

(Redéfinition et surcharge)

```
public class Employe {  
    private String societe;  
    private float salaire;  
    private float primeBase;  
    public void prime(){  
        ↑primeBase = salaire * 0.1;  
    }  
}
```

```
public class Cadre extends Employe {  
    private float primeSupp  
  
    public void prime(){  
        primeSupp = salaire*0.2;  
        super.prime();  
    }  
}
```

```
public class Principale {  
  
    public static void main(String[] args){  
        Cadre c = new Cadre();  
        c.prime();  
  
        ....  
    }  
}
```

HERITAGE

(Redéfinition et surcharge)

JVM et redéfinition d' une méthode

La JVM cherche la définition d' une méthode invoquée à partir de la classe de l' instance concernée. La première implantation rencontrée en remontant la hiérarchie de classes est celle choisie.

Ainsi, une hiérarchie de classes construite avec des liens d' héritage peut mettre en jeu, à plusieurs niveaux de l' arborescence, des méthodes ayant la même signature, mais dont le code est différent.

HERITAGE (Constructeurs)

Une classe n'hérite pas des constructeurs de la classe mère. Il faut toujours définir tous les constructeurs dont elle a besoin, même si ces derniers ne font que des appels aux constructeurs de la classe mère par *super(arguments)*.

N.B. L'appel au constructeur de la classe mère doit se faire absolument en première instruction.

Attention : quand il n'existe pas d'appel explicite d'un constructeur de la classe mère, Java insère implicitement l'appel **super()**.

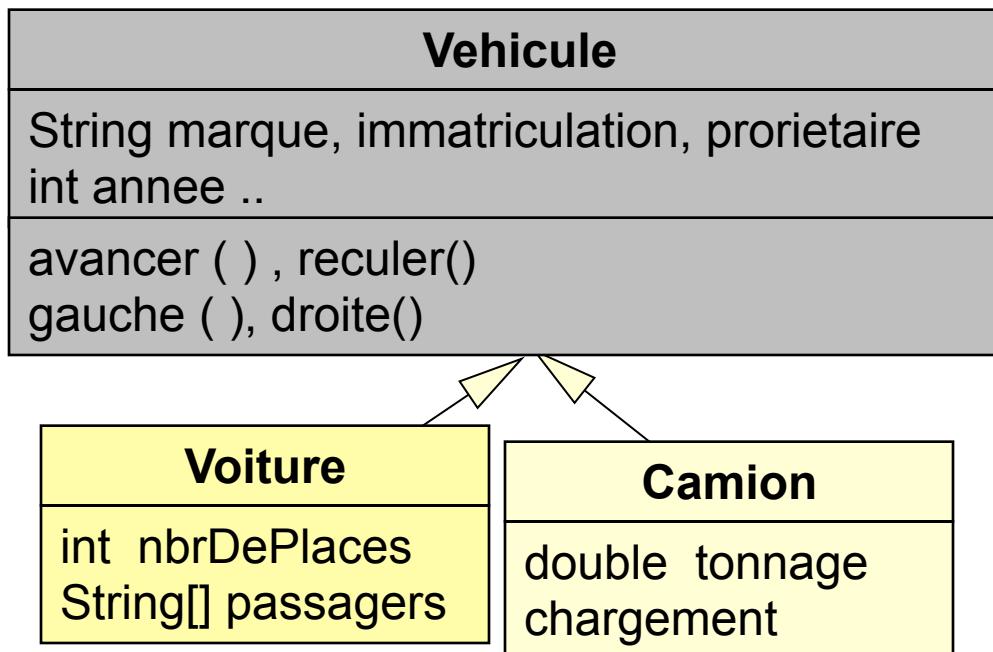
HERITAGE (Constructeurs)

```
public class Personne {  
  
    private String nom;  
    private short age;  
    public Personne(String nom, short age){  
        this.nom=nom; this.age = age;  
    }  
  
    public Personne (){  
        this (null, 0);  
    }  
}
```

```
public class Employe extends Personne {  
  
    String societe;  
    public Employe(String nom, short age, String societe){  
        super(nom,age);  
        this.societe=societe;  
    }  
}
```

HERITAGE

(Exemple)



HERITAGE (Exemple)

```
Class Vehicule {  
    private String marque;  
    private String immatriculation;  
    private String proprietaire;  
    private int annee;  
  
    Vehicule(String m, String p, String i, int a){  
        marque = m;  
        proprietaire = p;  
        immatriculation = i;  
        annee = a;  
    }  
    ...  
}
```

HERITAGE (Exemple)

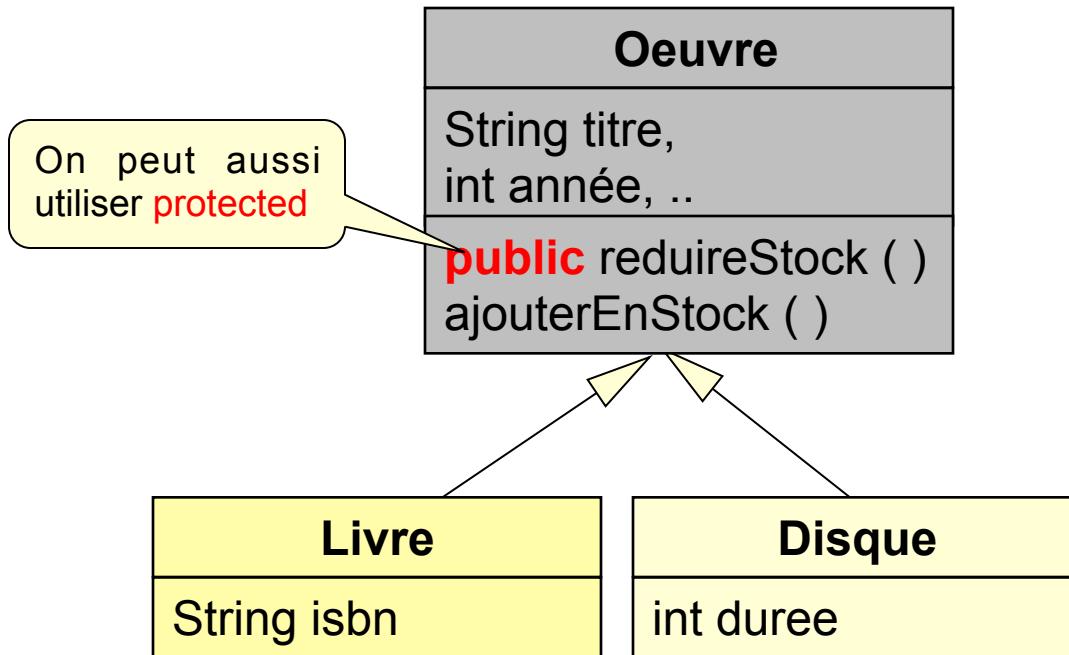
```
Class Voiture extends Vehicule {  
    private int nbrDePlace;  
    private String passagers [ ];  
  
    Voiture (int places, String m, String p, String i, int a){  
        super (m, p, i, a);  
        nbrDePlace = places;  
        passagers = new String [places];  
    }  
}
```

HERITAGE

(Accès d'une classe dérivée aux membres de sa classe de base)

Une **méthode** d'une classe dérivée n'accède pas aux membres (attributs et méthodes) privés de sa classe de base.

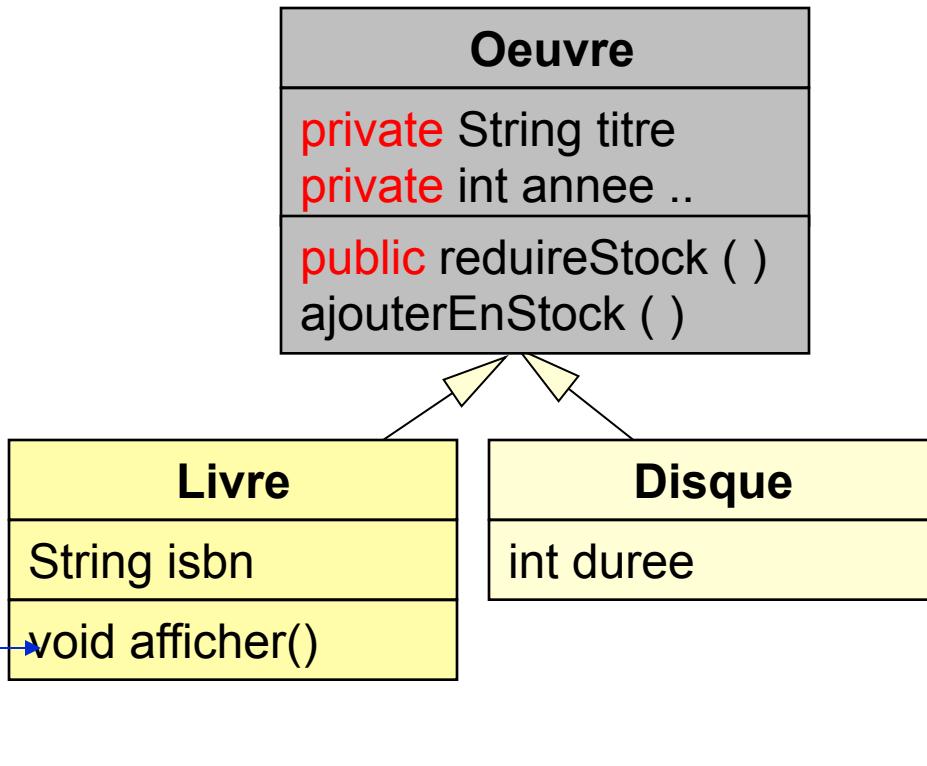
Une classe dérivée (un **objet** d'une classe dérivée) n'accède pas aux membres privés de sa classe de base.



Dans les classes **Livre** et **Disque**, le programmeur peut accéder aux membres **publiques** et **protégés** (voir chapitre Encapsulation) de la classe **Oeuvre**. Il peut par exemple invoquer la méthode `reduireStock()`.

HERITAGE

(Accès d'une classe dérivée aux membres de sa classe de base)

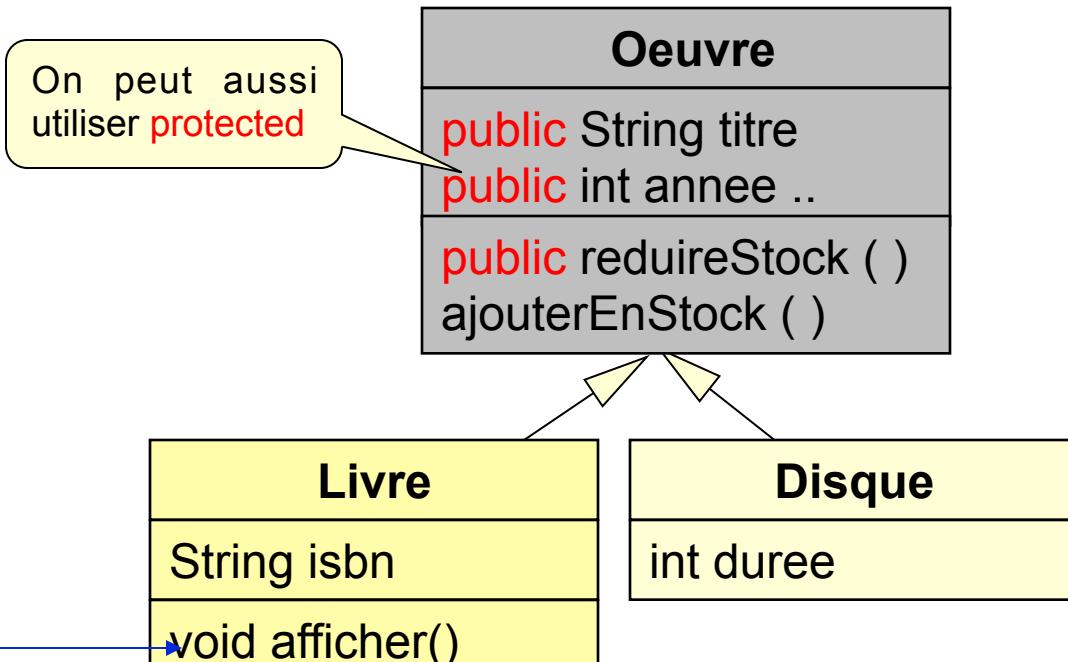


```
void afficher(){
    System.out.println("ISBN " + isbn);
    System.out.println("Titre " + titre);
    System.out.println("Année " + annee);
}
```

A blue line with an arrow points from the `afficher()` method in the **Livre** class to the corresponding code in the text box below. A red line with an arrow points from the `System.out.println("Titre " + titre);` line in the `afficher()` method to the `titre` attribute in the **Oeuvre** class, illustrating how the derived class can access members of its base class.

HERITAGE

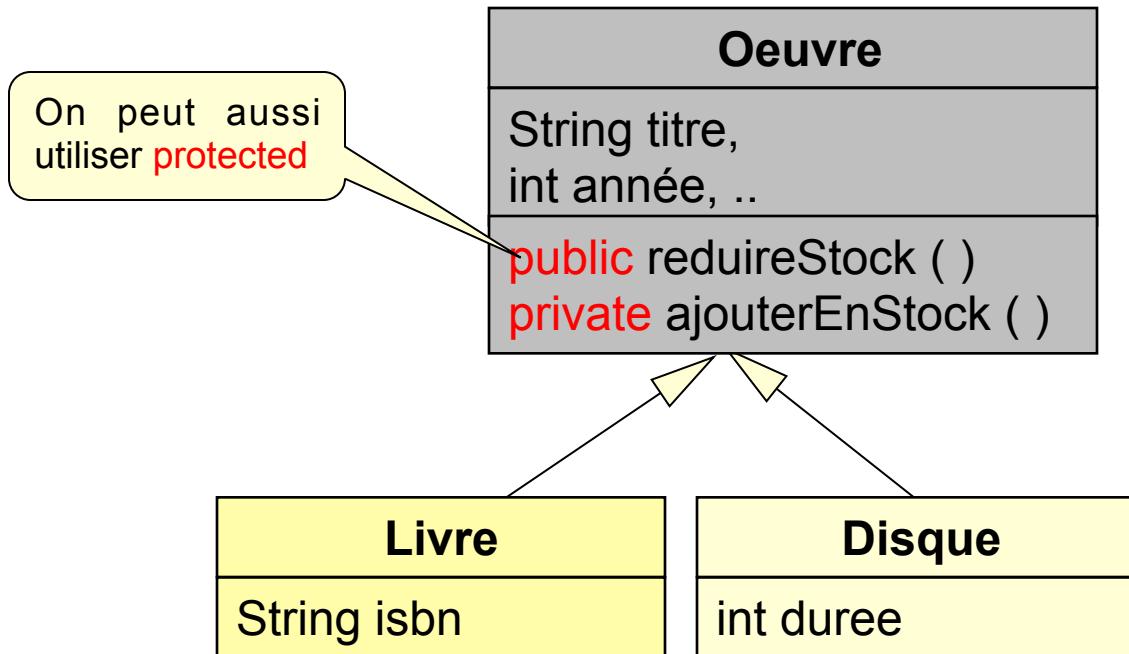
(Accès d'une classe dérivée aux membres de sa classe de base)



```
void afficher(){  
    System.out.println("ISBN " + isbn);  
    System.out.println("Titre " + titre);  
    System.out.println("Année " + annee);  
}
```

HERITAGE

(Accès d'une classe dérivée aux membres de sa classe de base)



```
Livre livre = new Livre();
livre.reduireStock()
livre.ajouterEnStock()
```

L'objet `livre` peut faire appel :
- aux méthodes de *Livre*
- mais aussi aux méthodes **publiques** et **protégées** de *Oeuvre*

HERITAGE

(Accès d'une classe dérivée aux membres de sa classe de base)

```
public class Employe {  
    private String nom;  
    private double salaire;  
    protected double prime( ){  
        return salaire*0.1;  
    }  
}
```

```
public class Cadre extends Employe {  
    private int codePrime;  
    public double prime( int code){  
        if(codePrime==code)  
            super.prime()  
        else  
            return getSalaire()*code*0.1;  
    }  
}
```

```
public class Principale {  
    public static void main(String[] args){  
        Cadre c = new Cadre(...);  
        c.prime(2);  
        c.prime();  
    }  
}
```

Appel de la méthode prime() de Cadre

Appel de la méthode prime() de Employe
Erreur: méthode uniquement accessible dans
Employe et dans ses sous-classes

HERITAGE

(Comment interdire l'héritage d'une classe ?)

Utilisation du mot-clé *final* devant une méthode

Le mot-clé *final* devant une méthode permet d'interdire une éventuelle redéfinition de la méthode.

Exemple : public *final* double prime(){...};

Utilisation du mot-clé *final* devant une classe

Le mot-clé *final* devant une classe permet d'interdire toute spécialisation ou héritage de la classe.

Exemple : public *final* class Employe{...}

Autre exemple : la classe **String** est finale

HERITAGE

(Comment interdire l'héritage d'une classe ?)

```
public class Employe {  
    private String nom;  
    private double salaire;  
    public final double prime( ){  
        return salaire*0.1;  
    }  
}
```

```
public class Cadre extends Employe {  
    private int codePrime;  
    public double prime( ){  
        if(codePrime >0)  
            super.prime()  
        else  
            return getSalaire()*code*0.1;  
    }  
}
```

Erreur : la méthode prime() est finale

HERITAGE

(Comment interdire l'héritage d'une classe ?)

```
public final class Employe {  
    private String nom;  
    private double salaire;  
    public double prime( ){  
        return salaire*0.1;  
    }  
}
```

```
public class Cadre extends Employe {  
    private int codePrime;  
    public double prime( int code){  
        if(codePrime==code)  
            super.prime()  
        else  
            return salaire*code*0.1;  
    }  
}
```

Erreur : la classe Employe est finale

HERITAGE

(Comment interdire l'héritage d'une classe ?)

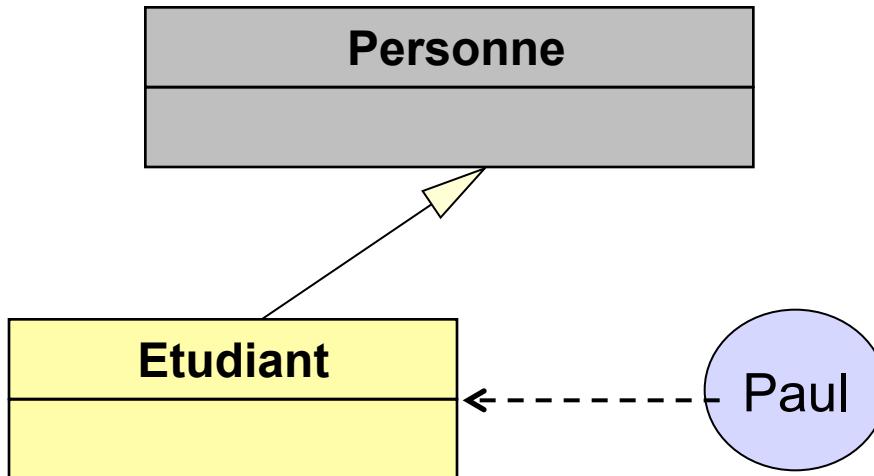
```
public class Collection extends Vector {  
}
```

OK : la classe Vector n'est pas finale

Erreur : la classe String est finale

```
public class Chaine extends String {  
}
```

HERITAGE (Polymorphisme)



Paul est une instance de la classe **Etudiant**, mais aussi une instance de la classe **Personne**.

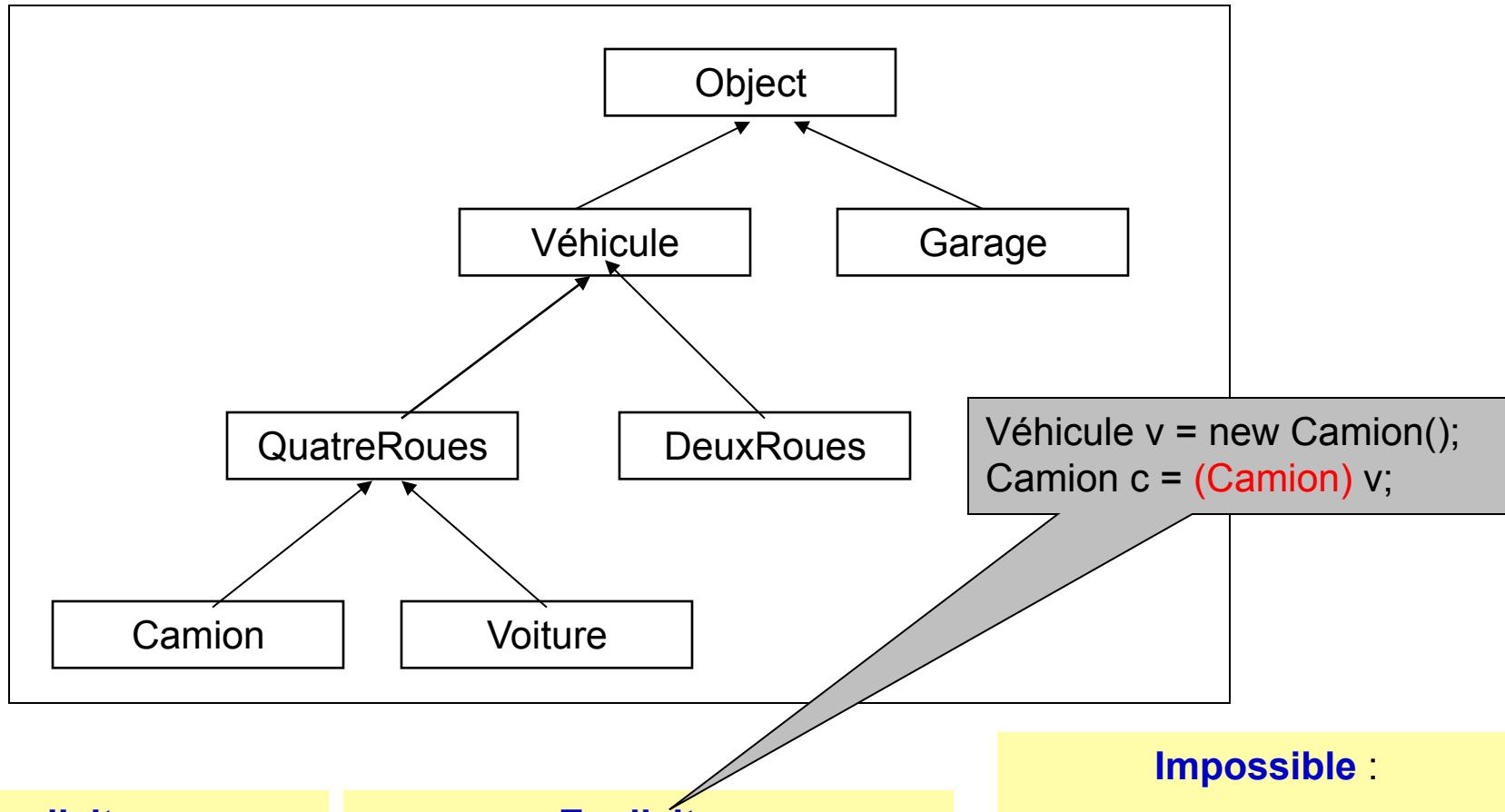
L'objet Paul peut donc être perçu, selon les besoins, comme instance de plusieurs classes : on parle alors de **Polymorphisme**.

Le polymorphisme, est également défini comme la capacité, pour une même méthode, de correspondre à plusieurs formes de traitement, selon l' objet qui contient cette méthode.

HERITAGE

(Polymorphisme et transtypage)

Le principe du transtypage des objets part du même principe que le transtypage des variables de types primitifs



Implicit :

Un Camion **est** un Véhicule

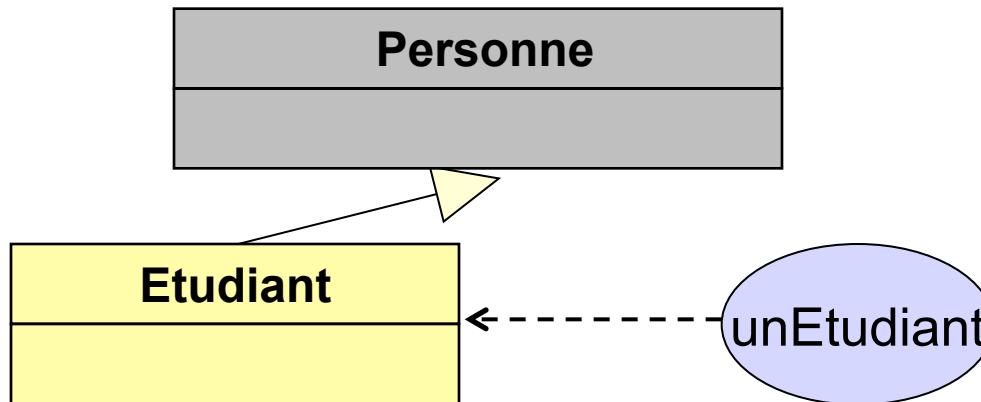
Explicit :

Un Véhicule **peut être** un Camion

Impossible :

Un Garage **n'est pas**
un Véhicule

HERITAGE (Polymorphisme – transtypage implicite)



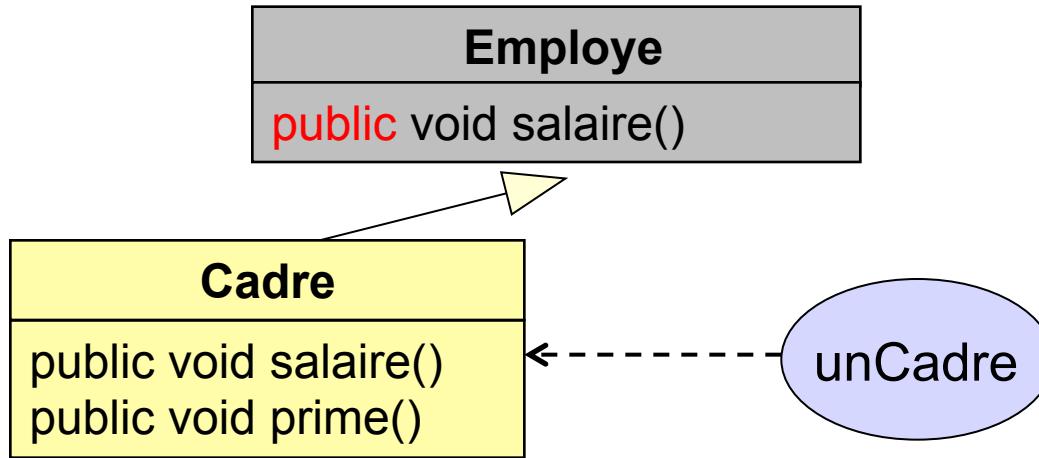
→ **Personne** unEtudiant = new **Etudiant**(...);

A une référence déclarée de la classe Personne, il est possible d'affecter une valeur qui est une référence vers un objet de la classe Etudiant.

Transtypage implicite ou surclassement ou encore upcasting

A une référence déclarée d'une classe donnée, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle de ses sous-classe directe ou indirecte.

HERITAGE (Polymorphisme – transtypage implicite)



A la compilation, l'objet (surclassé) `unCadre` est vu comme un objet de type *Employe* (type de la référence utilisée pour le désigner) .

```
Employe unCadre = new Cadre(...);
```

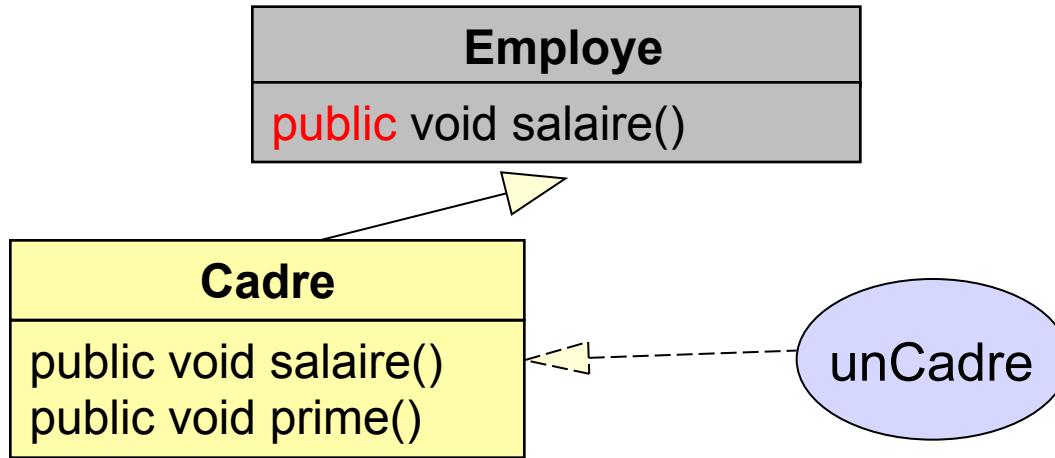
```
unCadre.salaire();  
unCadre.prime();
```

utilisation de la méthode `salaire()` de la classe *Employe*

Erreur : cette méthode n'est pas disponible dans *Employe*

HERITAGE

(Polymorphisme – transtypage implicite)



A l'exécution, l'objet (surclassé) *unCadre* est vu comme un objet de type *Cadre* (type effectif de l'objet).

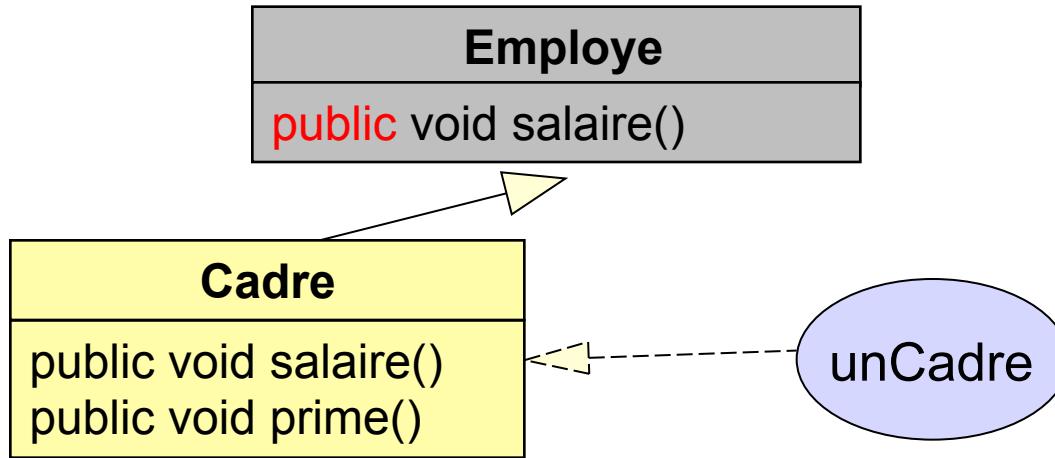
`Employe unCadre = new Cadre(...);`

`unCadre.salaire();`

Les attributs de la classe **Cadre** sont initialisés!!

C'est la méthode `salaire()` de la classe **Cadre** qui est appelée !!

HERITAGE (Polymorphisme – transtypage implicite)

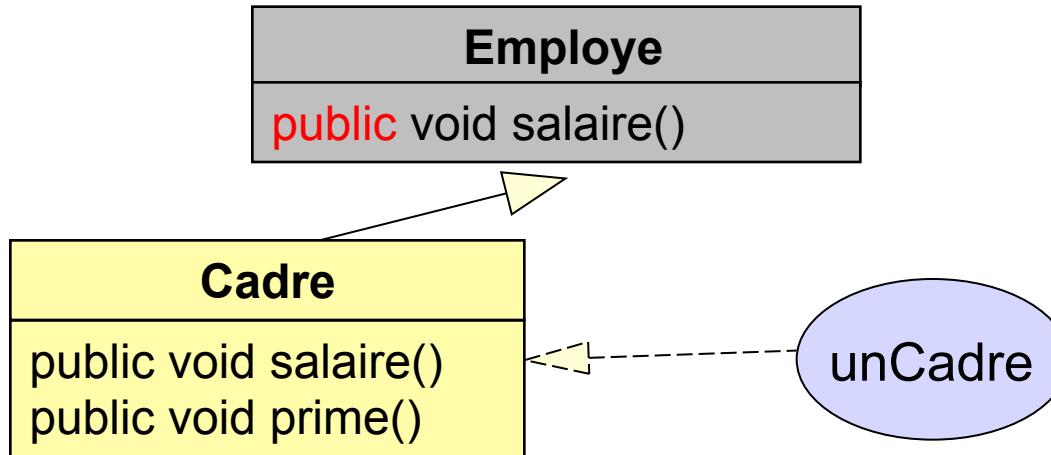


La méthode à exécuter est déterminée à l'exécution et non à la compilation : *lien dynamique (ou dynamic binding)*.

Conséquence:

Lorsqu'une méthode d'un objet est accédée au travers d'une référence surclassée, c'est la méthode telle qu'elle est définie au niveau de la **classe effective** de l'objet qui est exécutée.

HERITAGE (Polymorphisme – transtypage implicite)

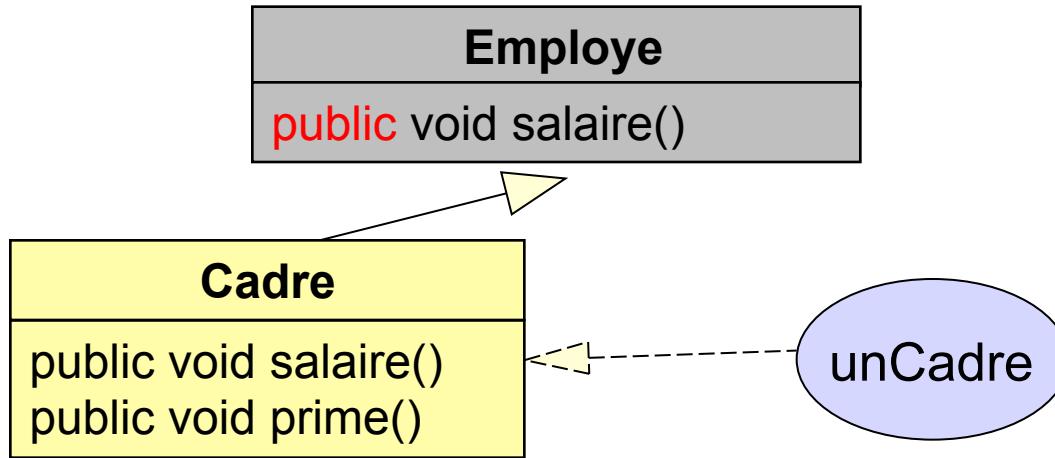


```
Employe unCadre = new Cadre(...);  
unCadre.salaire();
```

A la compilation

- (1) un objet de la classe **Cadre** est considéré
- (2) cet objet (**unCadre**) est vu comme une référence à un objet de la classe **Employe**
- (3) le compilateur empêche donc d'accéder aux méthodes spécifiques de la classe **Cadre**.

HERITAGE (Polymorphisme – transtypage implicite)

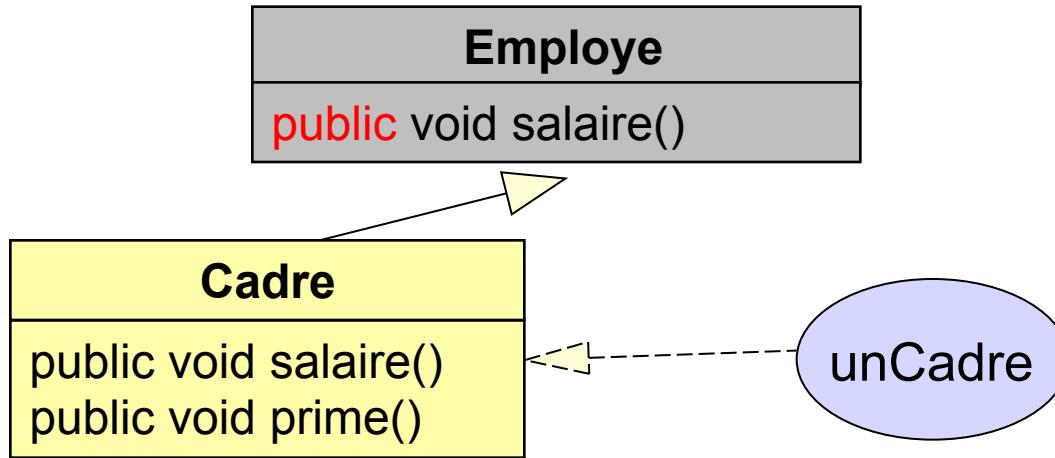


```
Employe unCadre = new Cadre(...);  
unCadre.salaire();
```

A l'exécution

- (1) cet objet (`unCadre`) est vu comme une référence à un objet de la classe **Cadre**
- (2) la méthode `salaire()` de **Cadre** est exécutée.

HERITAGE (Polymorphisme – transtypage explicite)



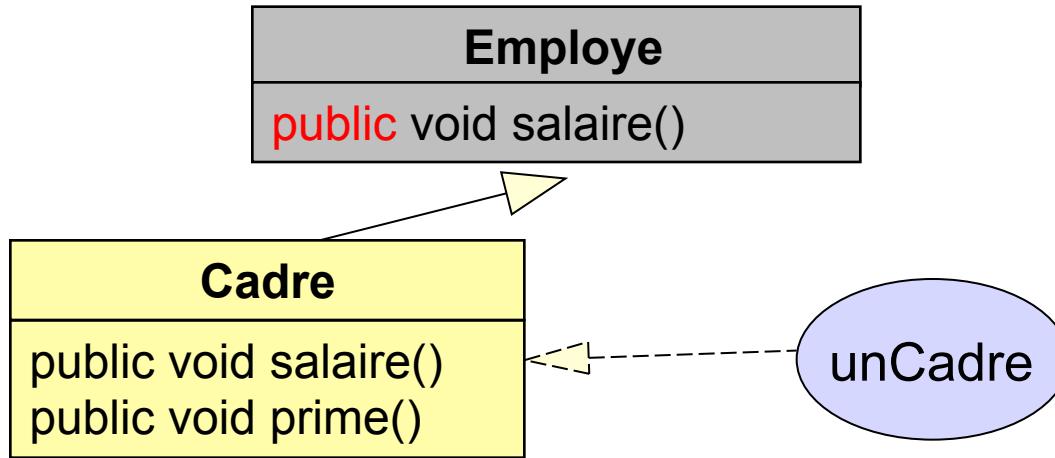
```
Employe unEmploye = new Cadre(...);  
unEmploye.prime(); // Erreur
```

```
Cadre unCadre;  
unCadre = (Cadre) unEmploye;
```

// on peut maintenant utiliser les méthodes spécifiques de Cadre
`unCadre.prime(); //Pas d'erreur`

//On peut aussi directement écrire :
`(Cadre) unEmploye.prime();`

HERITAGE (Polymorphisme – transtypage explicite)



```
Employe unEmploye = new Cadre(...); //Upcasting  
unEmploye.prime(); // Erreur
```

```
(Cadre) unEmploye.prime(); //Downcasting
```

Le *downcasting* (transtypage explicite) permet essentiellement d'obtenir l'accès aux méthodes cachées par le *upcasting* (transtypage implicite).

HERITAGE

(Classes abstraites)

Notions de Méthode abstraite et de classe abstraite

Une *méthode abstraite* est une méthode définie uniquement par sa signature, sans la définition de son code. Pour déclarer une méthode abstraite, on utilise le mot-clé *abstract*.

Une classe abstraite est une classe dont au moins une méthode est abstraite. Une classe abstraite doit aussi être déclarée en utilisant le mot-clé *abstract*.

On ne peut pas instancier une classe abstraite. Il faut créer une classe héritière et implanter les méthodes abstraites.

HERITAGE

(Classes abstraites)

```
public abstract class FormeGeometrique{  
    protected Color couleur;  
    abstract double getSurface();  
    abstract double getPerimetre();  
  
    public void setCouleur(Color couleur){  
        this.couleur = couleur;  
    }  
}
```

```
public class Rectangle extends FormeGeometrique{  
    double longueur, largeur;  
    public double getSurface(){  
        return longueur * largeur;  
    }  
  
    public double getPerimetre(){  
        return 2* (longueur + largeur);  
    }  
}
```

HERITAGE (Exemple Interface)

```
Public interface FormeGeometrique{  
    abstract double getSurface();  
    abstract double getPerimetre();  
}
```

```
public class Rectangle implements FormeGeometrique{  
    double longueur, largeur;  
    public double getSurface(){  
        return longueur * largeur;  
    }  
    public double getSurface(){ }  
  
    public double getPerimetre(){  
        return 2* (longueur + largeur);  
    }  
}
```

HERITAGE (Interfaces)

Java permet à une classe dérivée qui hérite déjà d'une classe, d'hériter d'une ou plusieurs classes abstraites particulières appelées *interfaces*. Cette extension permet de construire des programmes dans un cadre proche de celui de l'héritage multiple.

Définition

Une **interface** est une classe abstraite ayant les caractéristiques suivantes :

- Toutes les méthodes sont abstraites et public.
- Contient uniquement des variables statiques et constantes.
- Toute classe peut hériter (implémenter) d'une ou plusieurs interfaces. Une interface peut hériter d'une ou plusieurs interfaces, **mais elle ne peut pas hériter d'une classe**.

HERITAGE

(Interfaces)

Syntaxe

Une interface est définie par le mot clé *interface*.

Il n' est pas nécessaire de mettre *abstract* au début de chaque méthode.

Les variables sont nécessairement *static*, *final* ; il n' est pas nécessaire de faire figurer ces termes.

Si une classe hérite d' une interface, il faut le mentionner avec le mot clé *implements*; Si une interface hérite d' une autre interface, il faut utiliser le mot clé *extends*.

HERITAGE (La classe Object)

Lorsqu'une classe n'hérite de rien, elle hérite par défaut de la classe **Object**. Cette classe définie dans Java est la racine de Java. Toutes les classes héritent directement ou indirectement de la classe **Object**.

La classe **Object** définit par exemple quelques méthodes intéressantes :

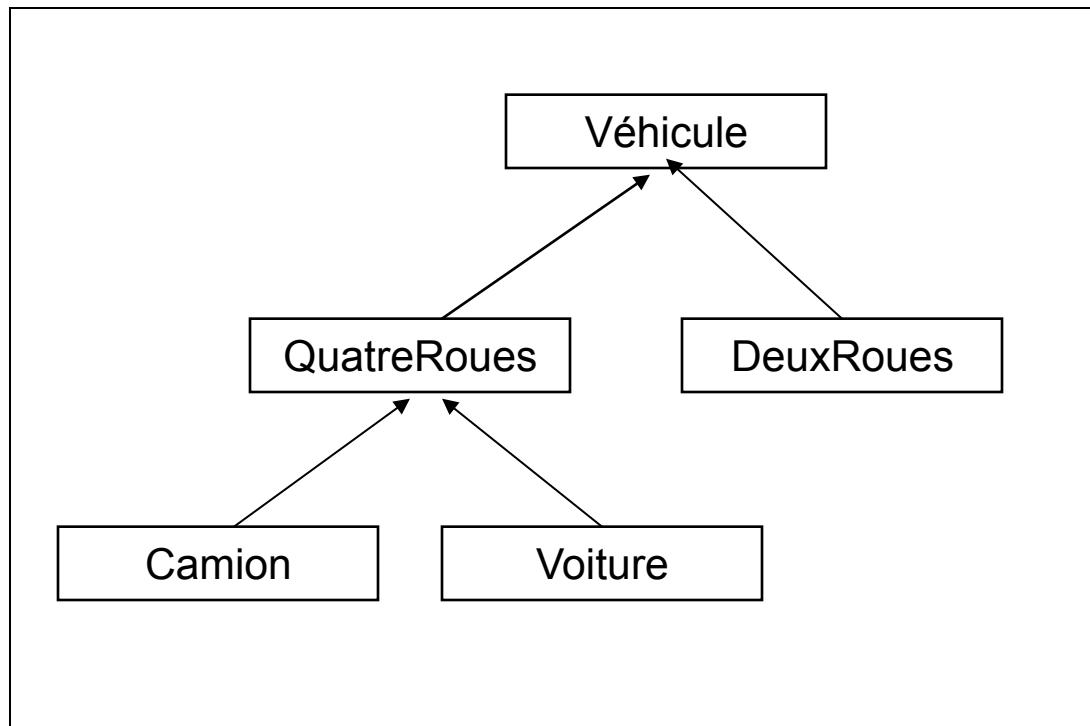
public String **toString()** : retourne une représentation textuelle de l'objet

public boolean **equals(Object)** : permet de comparer l'objet à un autre objet

Toutes ces méthodes peuvent évidemment être redéfinies dans les classes que nous définissons.

HERITAGE (Hiérarchie de classes)

Il arrive souvent d'avoir à utiliser ce mécanisme pour construire les classes d'une même famille en les structurant de manière arborescente : les classes situées au bas de l'arborescence sont plus spécifiques et plus riches en information que les classes du sommet.



Chapitre 9 – Encapsulation

- Objectif
- Définition
- Démarche
- Modificateurs de visibilité

ENCAPSULATION (Objectif)

Un programme Java est composé de plusieurs classes. Lors de l'exécution du programme, plusieurs objets instances de classes différentes, sont en interaction en mémoire. Dans ce contexte, chaque objet doit être responsable de son état interne. Pour renforcer cette responsabilité, il faut interdire à un objet d'une classe de modifier directement les valeurs d'un autre objet (d'une autre classe).

Il s'agit donc, de limiter l'accès aux classes ou aux membres des classes, dans le but de protéger le contenu d'un objet. L'idée est de contrôler comment votre classe sera utilisée par les autres classes. Certains membres d'une classe ne seront utilisés qu'à l'intérieur de la classe elle-même, et doivent être masqués des autres classes qui pourraient interagir avec cette classe.

Ce principe qui consiste à protéger les objets est désigné par le terme *encapsulation*.

ENCAPSULATION

(Définition démarche)

Définition

L'encapsulation est le processus consistant à empêcher les variables d'une classe d'être lues ou modifiées par d'autres classes.

Démarche

La démarche consiste à utiliser des *modificateurs de visibilité*, pour définir des *règles de visibilité* entre les classes et entre les attributs et méthodes qu'elles contiennent.

Ces règles dépendent de l' appartenance ou non à un même paquetage.

ENCAPSULATION

(Modificateurs de visibilité)

Visibilité des classes

public : sera visible partout. Il faut que la source de la classe soit dans un fichier ayant le même nom qu'elle.

friendly : ne sera visible qu'à l'intérieur du même package.

private : une classe *private* doit être interne à une autre.

Visibilité des membres

private : ne sera visible qu'à l'intérieur d'une classe.

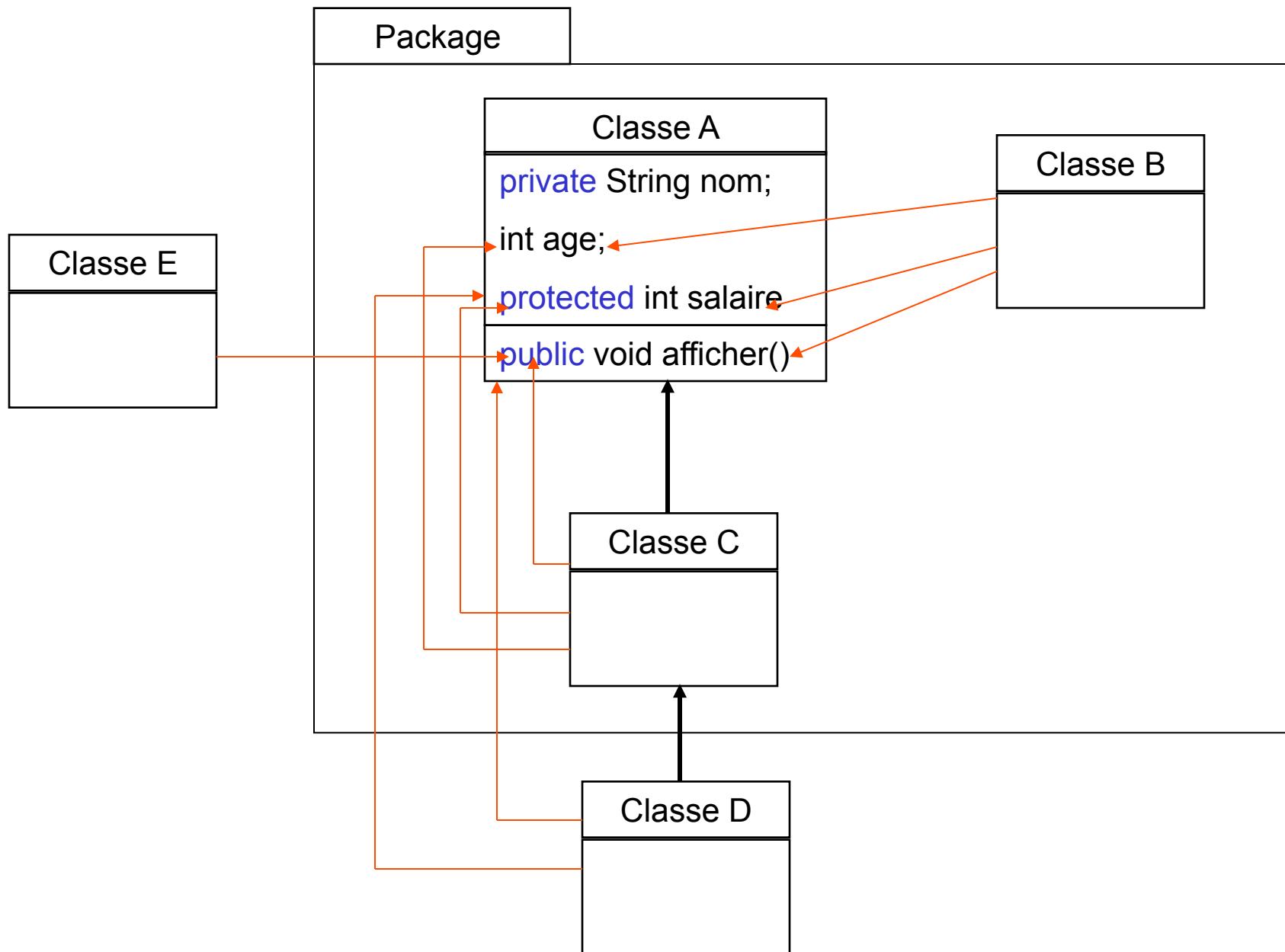
public : sera visible partout (membres dédiés au monde extérieur).

friendly : ne sera visible qu'aux classes du même package.

protected : ne sera visible qu'à l'intérieur d'une classe, mais peut être manipulé par les méthodes des classes héritières (indépendamment du package).

ENCAPSULATION

(Modificateurs de visibilité)



Chapitre 10 – Gestion des erreurs – Les exceptions

- Définition
- Comment capturer une exception
- Propagation d'une exception
- Exceptions utilisateur
- Synthèse

LES EXCEPTIONS (Définition)

La gestion des erreurs dans java est basée sur la notion *d'Exception*.

Une exception correspond à un événement (ex. division par zéro, ...) anormal ou inattendu qui se manifeste au cours de l'exécution d'un programme. Cet événement se manifeste sous la forme d'un signal véhiculé par un objet (de type *Throwable*) qui est lancé et se propage en remontant au travers de l'imbrication des blocs de code et les appels emboîtés de fonctions depuis la source jusqu'à la méthode «main()».

Les exceptions permettent de traiter les erreurs prévisibles (ex. fichier inexistant, mauvaise saisie, ...).

Le principe général consiste à isoler les instructions susceptibles de provoquer des exceptions (des erreurs), puis de prévoir un traitement d'erreur sur ces instructions.

LES EXCEPTIONS (Définition)

Si une exception n'est pas capturée par le bloc de code qui l'a générée, l'exception se propage vers le bloc directement entourant. Si aucun bloc de méthode ne capture l'exception, elle se propage vers la méthode appelante et ainsi de suite.

Si l'exception n'est jamais capturée, elle finira sa «remontée» à la méthode «main()» et conduira l'interpréteur Java à s'arrêter en affichant un message d'erreur et une trace de la pile

LES EXCEPTIONS

(Comment capturer une exception)

```
→try {  
    k = divise(y,x);  
}  
→catch (ArithmeticException e) {  
    //traitement de l' exception  
    System.out.println("Exception arithmétique");  
    System.out.println("Message : " + e.getMessage());  
    return;  
}  
→}  
finally {  
    System.out.println(" opération terminée !!");  
}
```

Les instructions de ce bloc seront exécutées qu'il y ait ou non exception

Contient les différents traitements pour chaque type d'erreur

Contient les instructions susceptibles de provoquer une erreur

Le bloc finally est optionnel

LES EXCEPTIONS

(Comment capturer une exception)

Le bloc *finally*

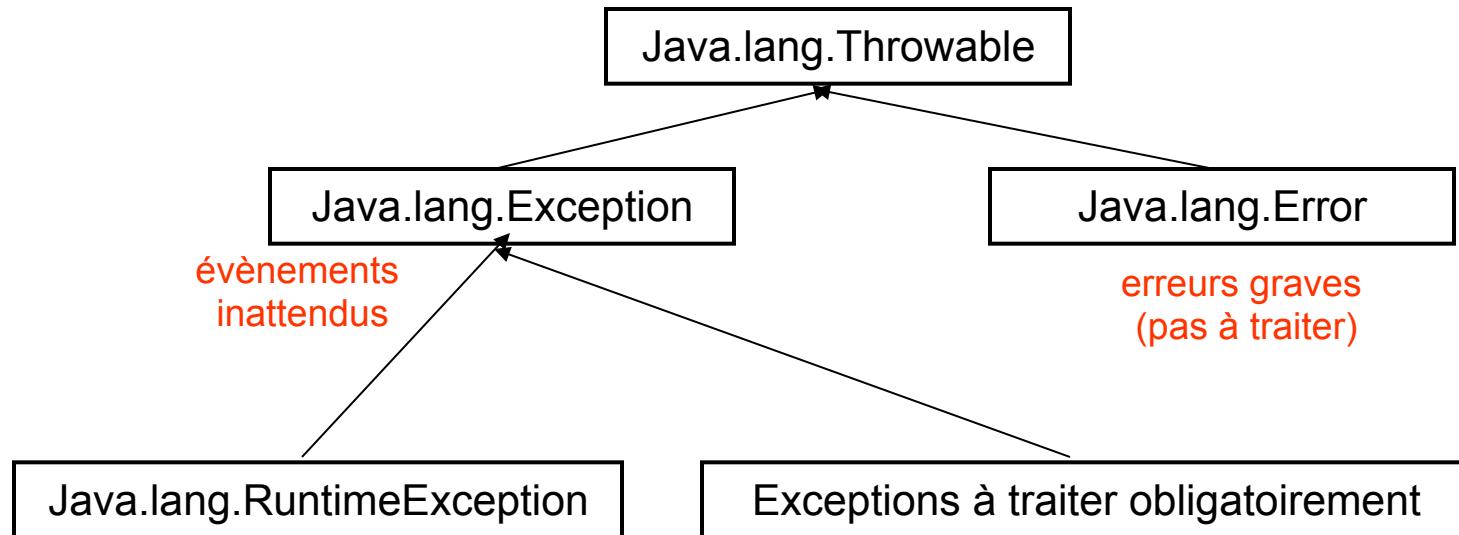
Un bloc *finally* permet de laisser le programme dans un état cohérent (qu'une exception ait été déclenchée ou non).

A la différence des instructions placées à la suite du *catch*, si une instruction *return* est exécutée dans un bloc de *catch*, le code contenu dans le bloc *finally* sera quand même exécuté avant le retour vers l'appelant.

LES EXCEPTIONS

(Comment capturer une exception)

Chaque exception est un objet (représente le problème rencontré), transmis par l'instruction qui la génère. On distingue les différents types des exceptions en fonction des classes auxquelles elles appartiennent.



Pas d'obligation de traiter ces erreurs.

Exemples : `ArithmaticException`, `NullPointerException`, ...

Obligation de traiter ces erreurs.

Exemples : `IOException`, `AWTException`, ...

Si une méthode est susceptible de générer une exception (à traiter) elle doit obligatoirement, soit capturer et traiter cette exception, soit prévenir qu'elle ne la traite pas (en utilisant la clause `throws`)

LES EXCEPTIONS

(Comment capturer une exception)

Toutes les classes d'exception héritent de la classe **Throwable**. Elle propose des méthodes facilitant la gestion des erreurs.

String **toString()** : retourne une chaîne contenant le type de l'exception suivi d'un message.

String **getMessage()** : retourne le message de l'exception.

void **printStackTrace()** : affiche la trace de la pile

LES EXCEPTIONS

(Comment capturer une exception)

```
Class FonctionMath {  
    static int moyenne(String [ ] notes) {  
        int somme = 0, note, nbNotes = 0;  
        for (int i = 0; i<notes.length;i++){  
            try {  
                note = Integer.parseInt(notes[i]);  
                somme += note;  
                nbNotes++;  
            }  
            catch (NumberFormatException e) {  
                System.out.println(" La "+(i+1)+"eme note n'est pas entière ");  
            }  
        }//fin for  
        return somme/nbNotes;  
    }  
    public static void main (String [ ] argv){  
        System.out.println(" La moyenne est " + moyenne(argv));  
    }  
}
```

LES EXCEPTIONS

(Propagation d'une exception)

```
public String getLigne() {  
    String ligne;  
    FileInputStream f = new FileInputStream (fichier)  
}
```

Cette instruction est susceptible de provoquer une exception de type ***IOException***

LES EXCEPTIONS

(Propagation d'une exception)

Première possibilité :

Traitement de l'exception

```
public String getLigne() {  
    String ligne;  
    FileInputStream f = new FileInputStream (fichier)  
}
```

Placer l'instruction dans un bloc *try* et attraper (*catch*) une exception de type *IOException*

LES EXCEPTIONS

(Propagation d'une exception)

Première possibilité

```
public String getLigne() {  
    String ligne;  
    try {  
        FileInputStream f = new FileInputStream (fichier)  
    }  
    catch (IOException e){  
        System.out.println("Pb ouverture de fichier");  
        System.out.println(e.getMessage());  
    }  
}
```

Placer l'instruction dans un bloc *try* et attraper (*catch*) une exception de type *IOException*

LES EXCEPTIONS

(Propagation d'une exception)

Deuxième possibilité :

Propagation de l'exception

```
public String getLigne() {  
    String ligne;  
    FileInputStream f = new FileInputStream (fichier)  
}
```

Ne pas traiter l'exception et confier ce traitement à la méthode appelante

LES EXCEPTIONS

(Propagation d'une exception)

La propagation d'une exception consiste à ne pas traiter celle-ci, mais à la faire remonter à l'appelant. Pour ignorer une exception obligatoire, il faut le stipuler explicitement dans la clause «**throws**»

Appelant

```
Public void ligneFichier() {  
    String ligne;  
    try {  
        ligne = fichier.getLigne ();  
    }catch(IOException e){  
        //Traitement de l'exception  
        // IOException  
    }  
}
```

Appelé

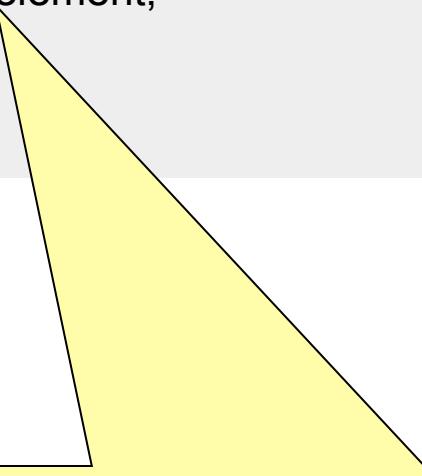
```
Public String getLigne() throws IOException {  
    String ligne;  
    FileInputStream f = new  
        FileInputStream (fichier)  
}
```

L'ouverture du fichier n'est pas placée dans un bloc try. Donc si une IOException se produit, elle sera automatiquement remontée à l'appelant

LES EXCEPTIONS

(Propagation d'une exception)

```
public void add(Object element) {  
    tab[compteur] = element;  
    compteur++;  
}//fin add
```



Cette instruction est susceptible de lancer une exception de type **IndexOutOfBoundsException**

LES EXCEPTIONS

(Propagation d'une exception)

Remarque : c'est Java qui lance l'exception au moment où l'erreur se produit !!!

```
public void add(Object element)
try {
    tab[compteur] = element;
    compteur++;
}
catch (IndexOutOfBoundsException e) {
    System.out.println("Pb dépassement taille tableau");
    System.out.println(e.getMessage());
}
}//fin add
```

LES EXCEPTIONS

(Propagation d'une exception)

Propagation d'une Exception

Le programmeur a également la possibilité de **lancer explicitement une exception**. Pour ce faire, il utilise le mot clé ***throw*** qui prend en argument un objet de type Exception. La méthode qui lance une exception doit la déclarer dans sa clause « ***throws*** ».

```
public void add(Object element) throws IndexOutOfBoundsException{
```

```
    if (tab.length <= compteur)
```

On lance explicitement
l'exception

```
        throw new IndexOutOfBoundsException("limite atteinte");
```

```
    else {
```

```
        tab[compteur] = element;
```

```
        compteur ++;
```

```
}
```

LES EXCEPTIONS (Exception utilisateur)

Pour définir des exceptions personnalisées, il suffit de définir des classes dérivées de la classe *Throwable* ou de l'une de ses sous-classes. Par convention, les nouveaux types d'exception définis par le programmeur dérivent de la classe *Exception*. La classe *Exception* contient un attribut de type *String* qui sert à décrire le message associé à l'erreur.

```
class PileException extends Exception {  
    private int codeErreur;  
    public PileException(String message, int code){  
        super ("Erreur Pile : " + message);  
        codeErreur = code;  
    }  
    public int getCodeErreur(){  
        return codeErreur;  
    }  
}
```

LES EXCEPTIONS

(Exception utilisateur)

```
public void empile(int element) throws PileException{  
    if (pile.length <= sommet)  
        throw new PileException( "pile pleine " , -1 ) ;  
    else {  
        pile[sommet] = element;  
        sommet ++;  
    }  
}//fin empile
```

LES EXCEPTIONS

(synthèse)

Exception obligatoire

Pour gérer une *exception obligatoire*, deux alternatives s'offrent au développeur :

- (1) Traiter cette exception à l'aide des blocs *try catch*.
- (2) Propager (à l'aide de l'instruction *throws*) cette exception à la méthode appelante (qui hérite à son tour de la responsabilité de traiter ou non l'exception).

N.B. Lorsqu'une méthode déclenche une exception, la machine virtuelle va remonter la suite des invocations de méthodes jusqu'à atteindre une méthode qui accepte de traiter l'exception. Si aucune méthode n'est trouvée alors la machine virtuelle affiche l'intitulé de l'exception levée (`e.getMessage()`) et s'arrête.

LES EXCEPTIONS (Synthèse)

Exception non-obligatoire

Dans le cas d'une exception *non-obligatoire*, deux alternatives s'offrent également au développeur :

- (1) Se ramener au cas d'une exception obligatoire

- (2) Déléguer le traitement à la machine virtuelle. Dans ce cas, la machine virtuelle affiche un message et arrête le programme.

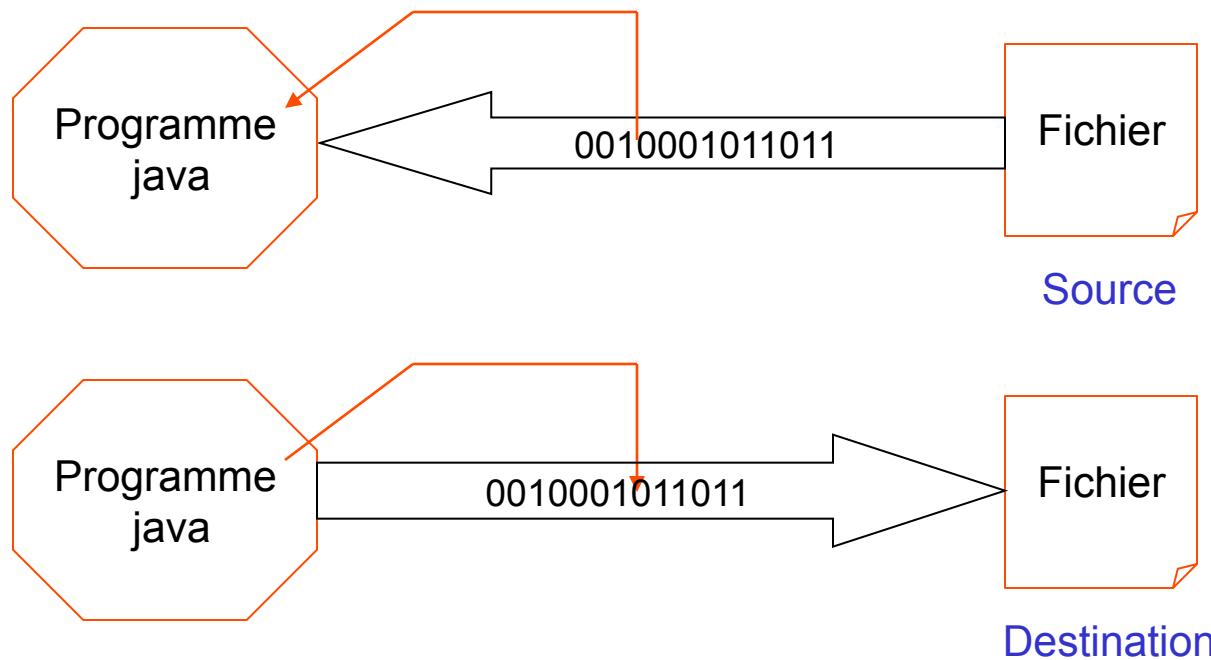
Chapitre 11 – Les entrées-sorties

- Principe
- Lecture / Écriture de données binaires
- Lecture / Écriture de suite de caractères
- Convertir un flux binaire en un flux de caractères
- La classe File
- Les Fichiers
- La sérialisation des objets

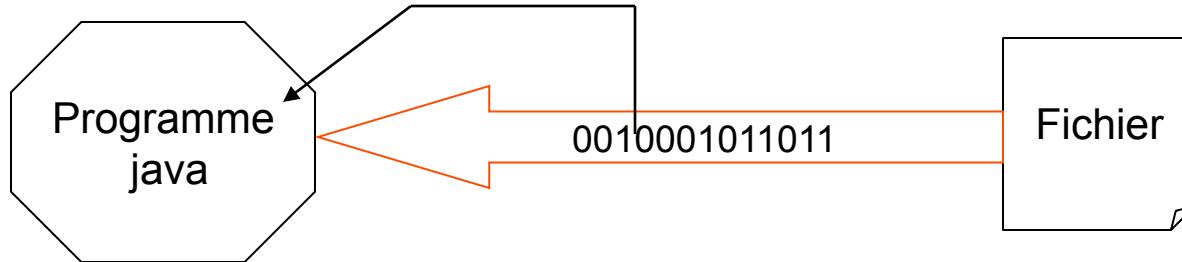
LES ENTREES-SORTIES

En Java, la manipulation des entrées-sorties est basée sur le concept de **flux** (**stream** en anglais).

Un flux est un ensemble de données (de type octets, caractères, objet, ...) émis par une **source** (lecture) ou envoyé vers une **destination** (écriture). La source et la destination peuvent être très variées : un fichier, une zone de mémoire, un périphérique, ...



LES ENTREES-SORTIES

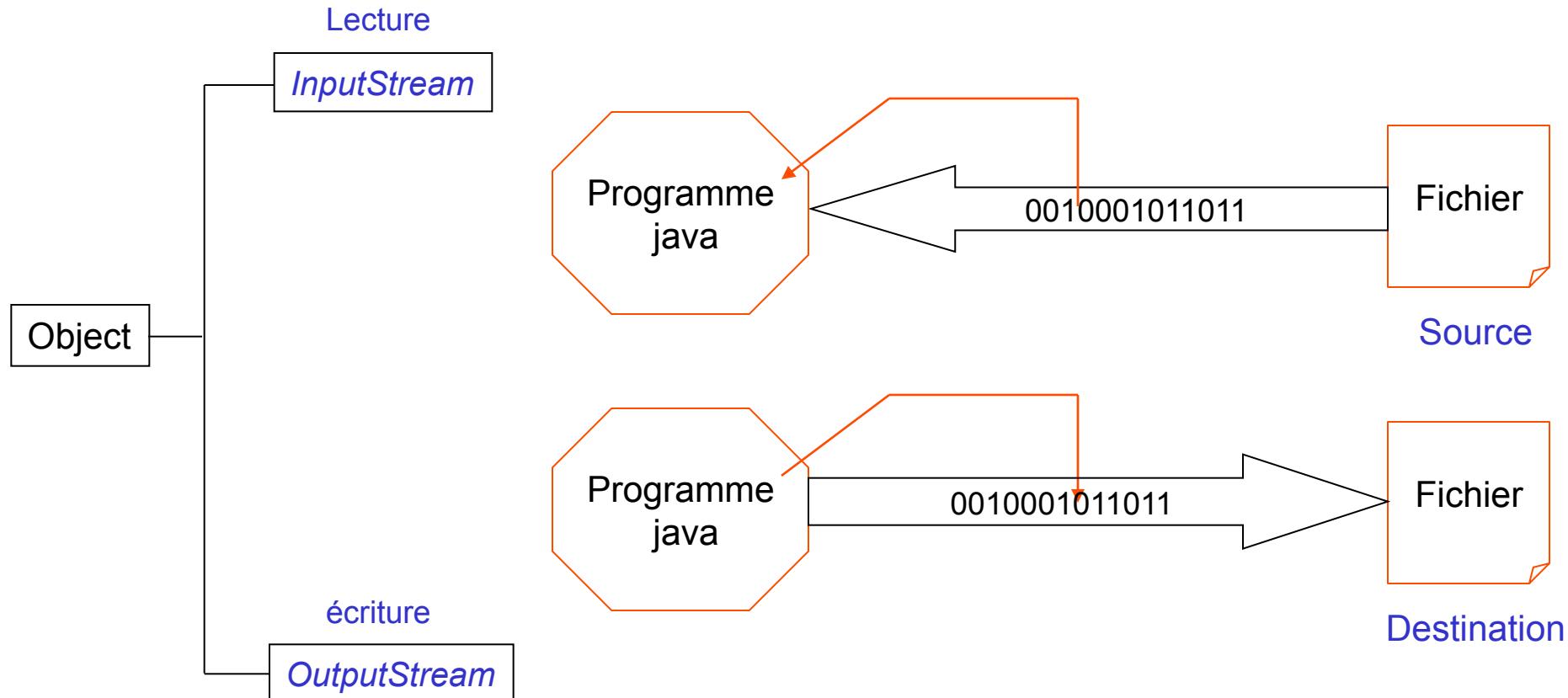


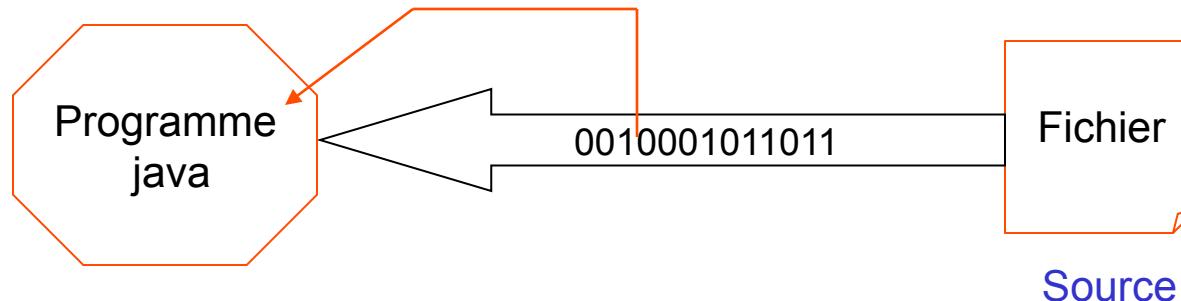
Un flux est représenté en java comme un objet. Le package **java.io** propose différentes interfaces et classes permettant de manipuler ces flux **de manière similaire, quelque soit la plate-forme hôte**.

Ce package contient une cinquantaine de classes que l'on peut répartir en trois grandes catégories :

- classes de gestion des flux d'octets
- classes de gestion des flux de caractères
- classes spécifiques à la gestion des fichiers

Lecture / écriture de données binaires (octets)





Les principales méthodes de la classe InputStream

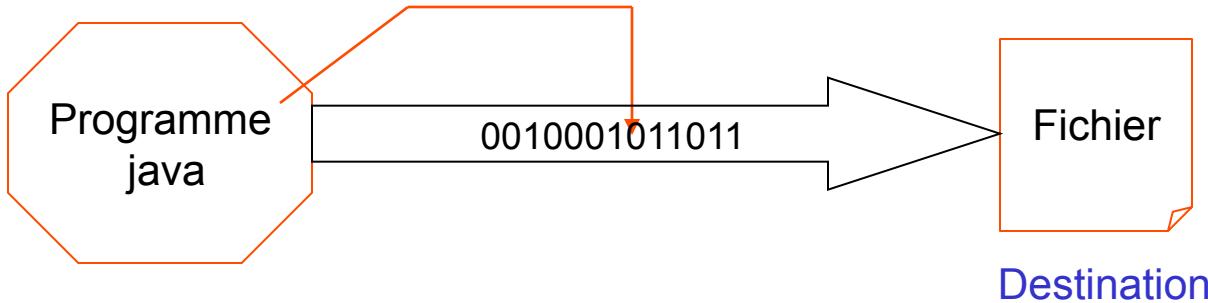
la classe **InputStream** a pour but de définir le concept de lecture de flux, elle propose trois méthodes :

int read() : lecture d'un octet dans le flux (cet octet est stocké dans un int).

int read(byte[] tableau) : lecture des octets disponibles dans le flux vers le tableau de byte passé en argument. Elle retourne le nombre d'octets lus (qui sera inférieur ou égal à la taille du tableau) ou –1 s'il n'y a pas d'octets disponibles.

int read(byte[] tableau, int offset, int longueur) : identique à la précédente mais insertion des bytes dans l'espace défini par les arguments offset et longueur.

void close() : permet de refermer le flux.



Les principales méthodes de la classe OutputStream

la classe `OutputStream` a pour but de définir le concept d'écriture vers les flux, elle propose trois méthodes :

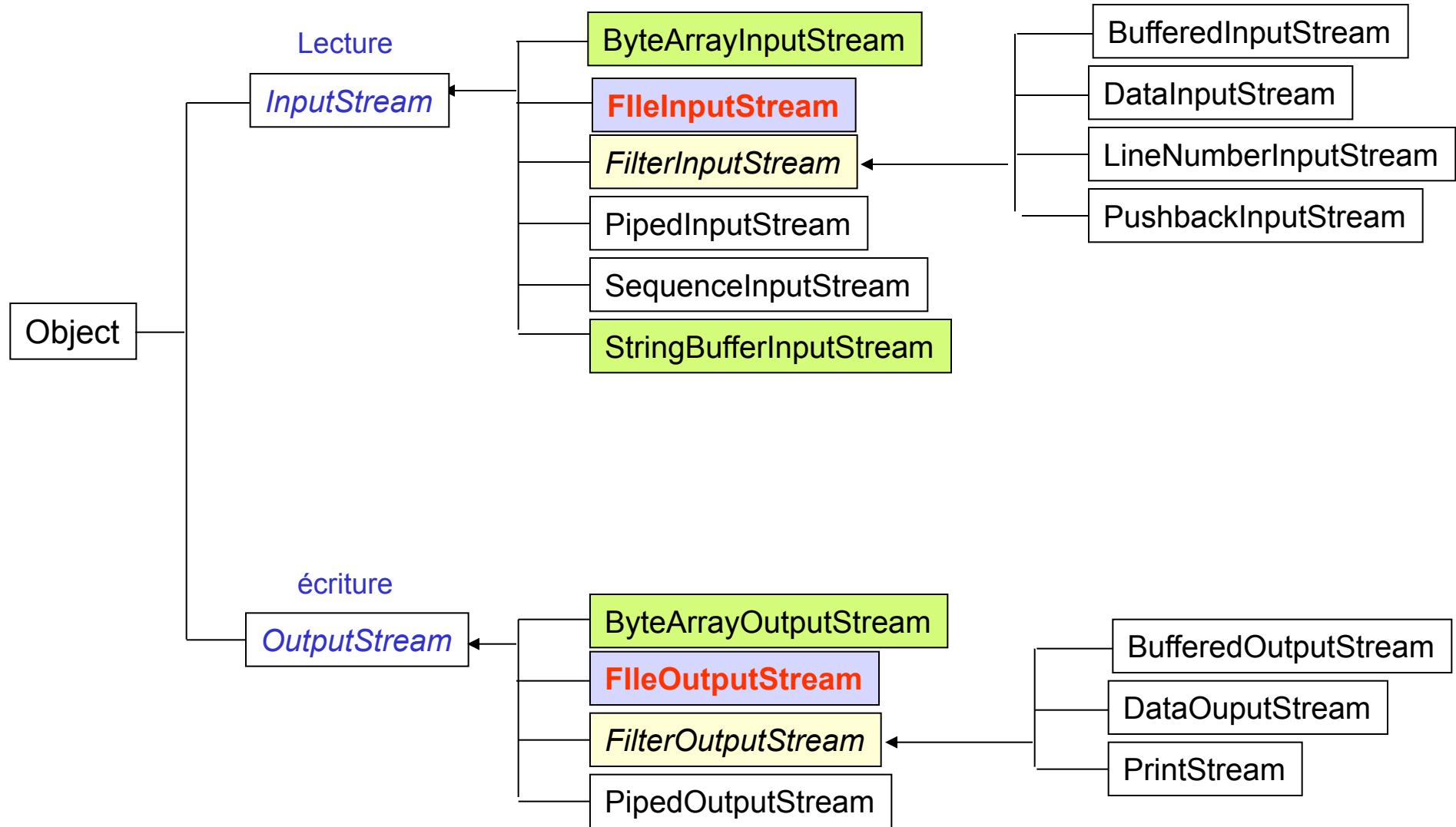
`void write(int octet)`

`void write (byte[] tableau)`

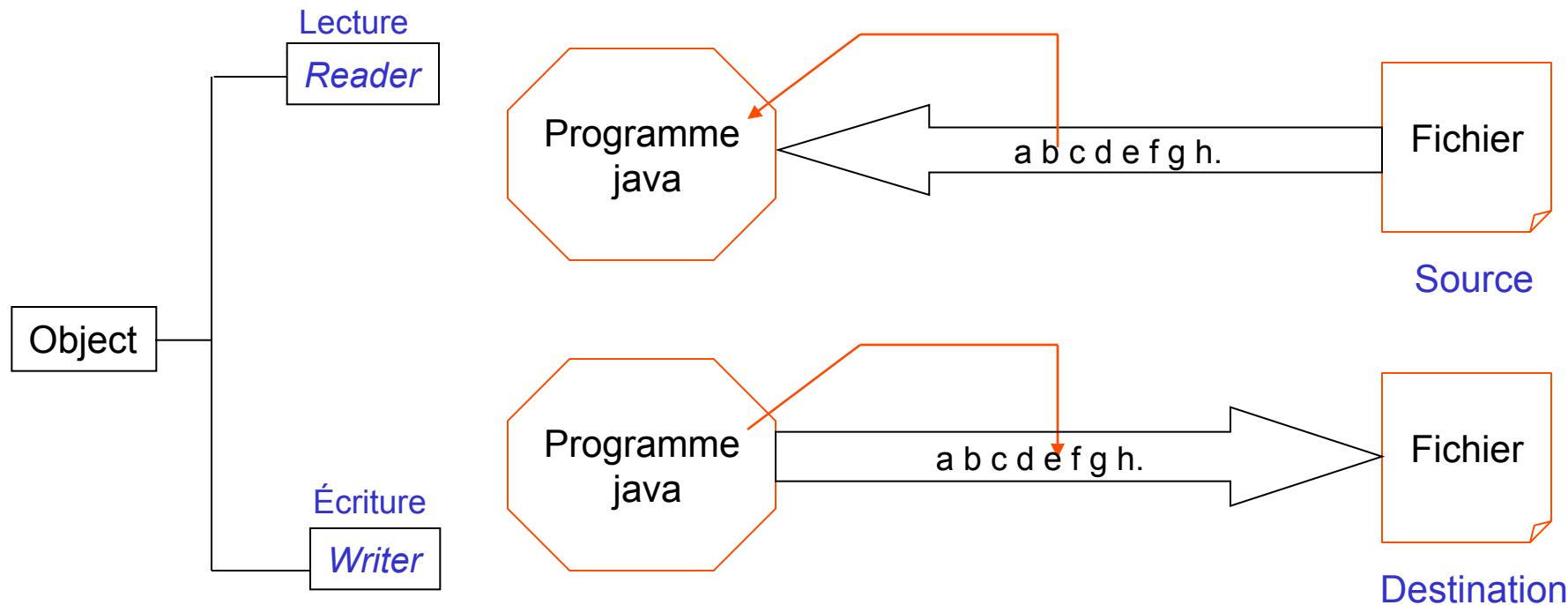
`void write (byte[] tableau, int offset, int longueur)`

`void close()`

Lecture / écriture de données binaires (octets)



Lecture / écriture de suite de caractères Unicode



Les principales méthodes de la classe Reader

la classe **Reader** propose aussi trois méthodes de lecture, mais qui liront des caractères codés sur 16 bits :

int read() : lecture d'un caractère disponible dans le flot

int read(char[] tableau) : lecture des caractères disponibles dans le flux vers le tableau de char passé en argument. Elle retourne le nombre de caractères lus (qui sera inférieur ou égal à la taille du tableau) ou –1 s'il n'y a pas de caractères disponibles.

int read(char[] tableau, int offset, int longueur) : identique à la précédente mais insertion des caractères dans l'espace défini par les arguments offset et longueur.

void close() : permet de refermer le flux.

Les principales méthodes de la classe Writer

la classe **Writer** a pour but de définir le concept d'écriture vers les flux de caractères, elle propose cinq méthodes :

void write(int caractere)

void write (byte[] tableau)

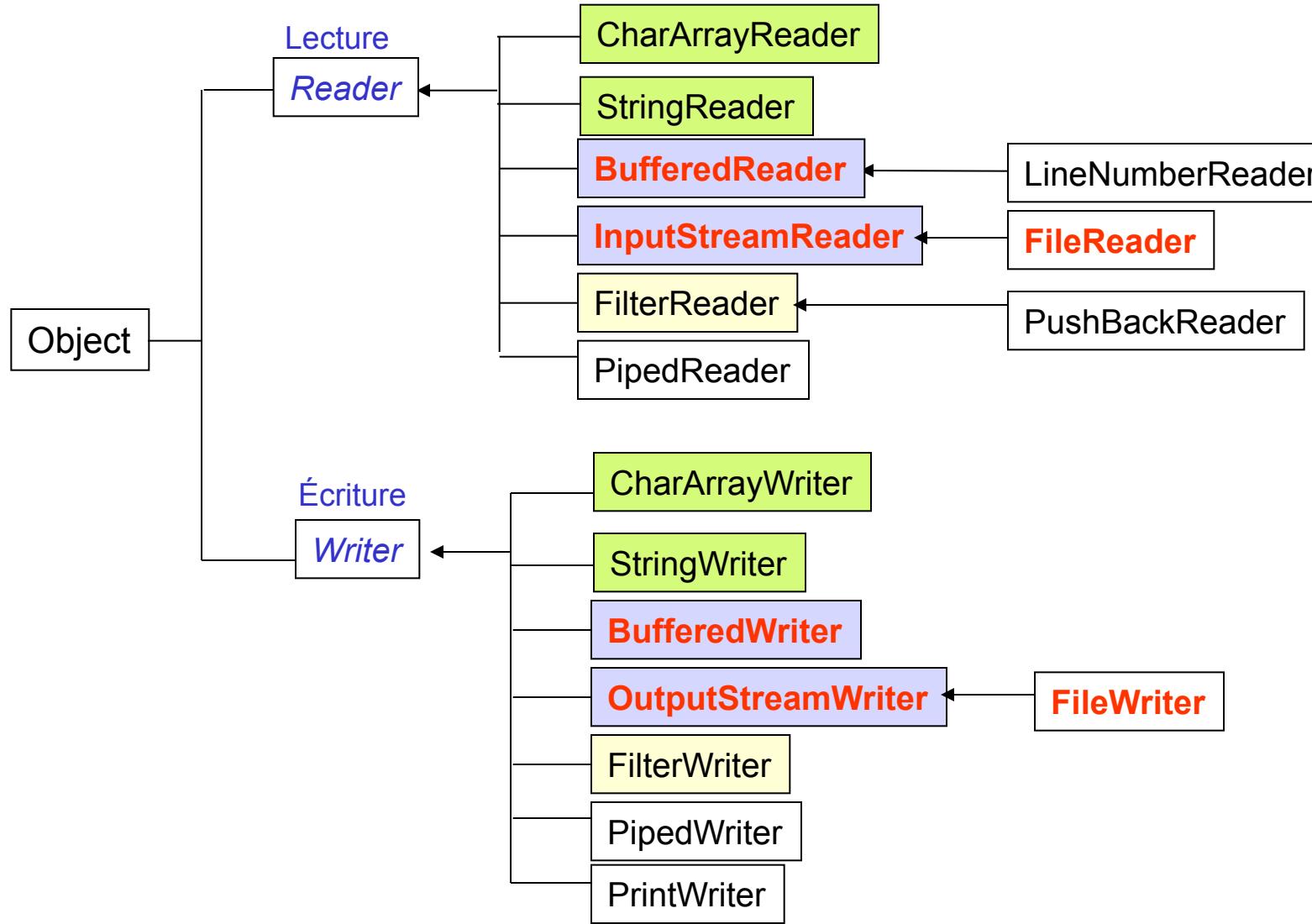
void write (byte[] tableau, int offset, int longueur)

void write (String chaine, int offset, int longueur)

void write (String chaine)

void close()

Lecture / écriture de suite de caractères Unicode



Convertir un flux binaire en un flux de caractères avec les classes :
InputStreamReader et *OutputStreamWriter*

Ces deux classes représentent une sorte de pont entre le monde binaire et le monde caractère.

InputStreamReader : permet la lecture en mode caractère d'un flux binaire.

OutputStreamWriter : permet l'écriture de caractères vers un flux binaire.

Exemple d'utilisation : sur tous les Système d'Exploitation, les fichiers sont composés d'octets. Il est donc nécessaire de faire une conversion avant affichage à l'écran.

BufferedReader et BufferedWriter

Le principal intérêt de la classe *BufferedReader* est la méthode *readLine()* qui va lire une ligne de texte.

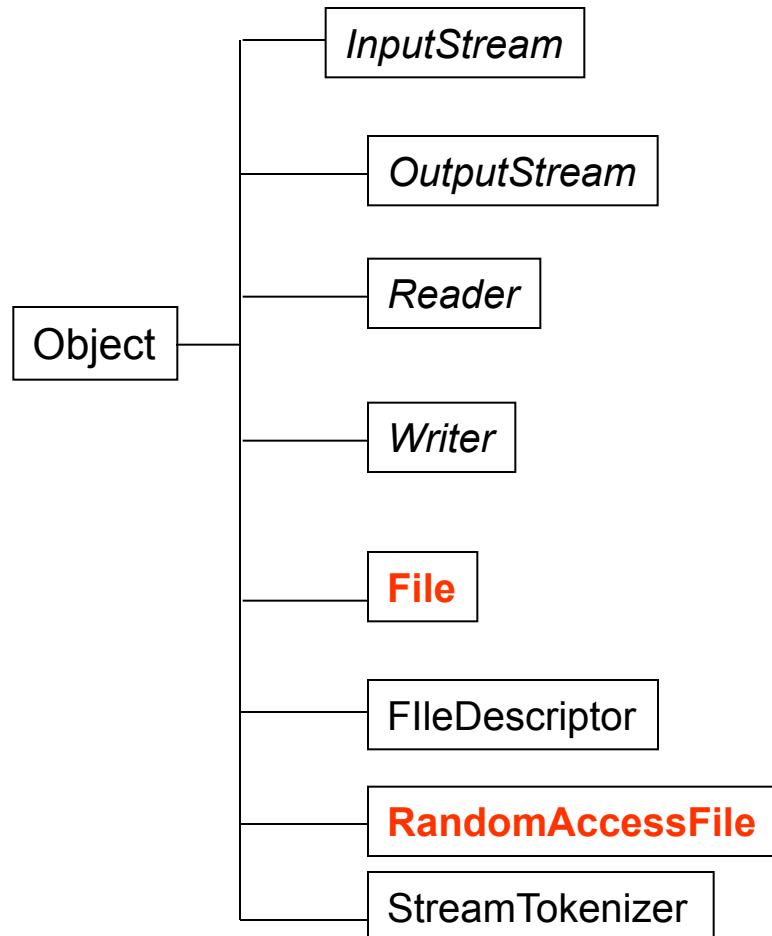
La classe *BufferedWriter* possède la méthode *newLine()* qui écrit dans le flux un caractère de fin de ligne

MEMBRES DE CLASSE

Exemple d' utilisation

```
import java.io.*;
class Saisie {
    public static String lireChaine(String message){
        String ligne = null;
        try{
            //conversion d'un flux binaire en un flux de caractères (caractères Unicode)
            InputStreamReader isr = new InputStreamReader(System.in);
            //construction d'un tampon pour optimiser la lecture du flux de caractères
            BufferedReader br = new BufferedReader(isr);
                System.out.print(message);
                // lecture d'une ligne
                ligne = br.readLine();
            }
            catch (IOException e){
                System.err.println(e.getMessage());
            }
            return ligne;
        } // fin lireChaine
```

java.io



LA CLASSE – java.lang.System

La classe `System` permet de lire ou d'écrire sur la console Java. Elle propose pour cela trois propriétés statiques représentant trois fichiers standards :

`System.in` est de type `java.io.BufferedInputStream`

`public int read ()` : attend une entrée et retourne son code
`public long skip(long nbOctets)` : se déplace d ' un nombre d ' octets dans le flot.

`System.out` et `system.err` sont de type `java.io.PrintStream`

`public void print(tout type java)`
`public void println(tout type java)`

LA CLASSE – java.lang.System

Exemple

```
import java.io.*;  
  
class EntreeSorties {  
    public static int saisie () throws IOException{  
        String chaine;  
        char car;  
        while((car = (char) System.in.read ( )) != '\n') {  
            chaine = chaine + car;  
        }  
        int valeur = Integer.parseInt(chaine);  
        return valeur;  
    }  
}
```

LA CLASSE – java.lang.System

Exemple

```
import java.io.*;  
  
class EntreeSorties {  
    public static int saisie () throws IOException{  
        StringBuffer sb = new StringBuffer();  
        String chaine  
        char car;  
        while((car = (char) System.in.read ( )) != '\n') {  
            sb.append(car);  
        }  
        chaine = sb.toString();  
        int valeur = Integer.parseInt(chaine);  
        return valeur;  
    }  
}
```

LA CLASSE – java.lang.System

Exemple

```
import java.io.*;  
  
Public class EntreeSorties1 {  
    public static void main (String [] args) {  
        BufferedReader br;  
        // System.in étant de type flux binaire, l'utilisation de InputStreamReader permet la  
        // transformation en caractères  
        br = new BufferedReader (new InputStreamReader (System.in));  
        System.out.println("Entrez des phrases suivies de Entrée puis une chaîne  
                        vide pour finir");  
        try{  
            while(true){  
                String chaine = br.readLine();  
                if(chaine.length() == 0) {  
                    System.out.println("Vous avez entré : " + chaine);  
                    exit(0);  
                }  
            }  
        }catch (IOException e){}  
    }  
}
```

LA CLASSE - java.io.File

Cette classe donne une représentation objet d'un fichier ou d'un répertoire. Ses méthodes permettent par exemple : nommage, droits d'accès, informations diverses ...

Exemple :

```
File f = new File(" texte.txt");
System.out.println("texte.txt" + f.getPath()); //chemin des répertoires
if(f.exists( )) {
    // savoir si on a le droit de lire ou d'écrire
    System.out.println("droits" + f.canRead() + f.canWrite( )); //
}
```

LES FICHIERS

FileInputStream et FileOutputStream

FileInputStream et *FileOutputStream* sont des implémentations de *InputStream* et *OutputStream* pour la gestion de fichiers binaires.

Les différents constructeurs prennent toujours : soit [un objet de type File](#), soit [une chaîne de caractères](#) représentant le chemin du fichier.

Les méthodes *close()* et *flush()* doivent être invoquées à la fin de toute écriture, sous peine de voir les données écrites perdues.

LES FICHIERS

FileInputStream et FileOutputStream

Exemple de lecture d'un fichier binaire :

```
FileInputStream fis;
byte[] b = new byte[1024];
try {
    fis = new FileInputStream("client.dat");

    while (fis.read(b) != - 1){
        String s = new String (b); // création d'une chaîne à partir de b
        System.out.println (s);
    }
} catch (FileNotFoundException e) {... }
catch (IOException e) { ... }
```

LES FICHIERS

FileInputStream et FileOutputStream

Exemple d'écriture dans un fichier binaire:

```
FileOutputStream fos;
String s = new String ("Bonjour !!!");
int longueur = s.length ();
byte [ ] buffer = new byte[longueur];
s.getBytes(0, longueur-1, buffer, 0); // maintenant remplacé par s.getBytes()
try {
    fos = new FileOutputStream("client.dat");
    for(int i = 0; i < longueur; i++)
        fos.write(buffer[i]); // écriture octet par octet
}
catch (FileNotFoundException e) {... }
catch (IOException e) { ... }
```

LES FICHIERS FileReader et FileWriter

FileReader *FileWriter* sont des implémentations de *Reader* et *Writer* pour la gestion de fichiers de textes.

Les différents constructeurs prennent toujours : soit [un objet de type File](#), soit une [chaîne de caractères](#) représentant le chemin du fichier.

Les méthodes *close()* et *flush()* doivent être invoquées à la fin de toute écriture, sous peine de voir les données écrites perdues.

LES FICHIERS

FileReader et FileWriter

Exemple

```
import java.io.*;
public class Copy {
    public static void main(String[] args) {
        try{
            File inputF = new File("../fichier1.txt");
            File outputF = new File("../fichier2.txt");
            FileReader in = new FileReader(inputF);
            FileWriter out = new FileWriter(outputF);
            int c; //The character read, as an integer in the range 0 to 65535
                   //or -1 if the end of the stream has been reached
            while((c=in.read())!=-1){
                out.write(c);
            }
            in.close();
            out.close();
        }catch (IOException e){
            System.out.println("Pb entrée sortie :" + e.getMessage());
        }
    }//fin main
}
```

Les Fichiers à Accès Aléatoire

La classe ***RandomAccessFile*** permet de lire ou d'écrire dans un fichier sur des positions aléatoires.

Les constructeurs sont :

RandomAccessFile(File fichier, String mode)

RandomAccessFile(String chemin, String mode)

L'argument mode permet de choisir le type d'accès :

r en lecture seule

rw en lecture écriture

Ces constructeurs renvoient l'exception *FileNotFoundException*

Les Fichiers à Accès Aléatoire

Quelques méthodes

long getFilePointer() : donne la position du pointeur

long length() : donne la taille du fichier en octets

void seek(long pos) : modifie la position du pointeur

//Lecture

byte readByte() : lecture d'un octet

char readChar() : lecture d'un caractère (2 octets)

double readDouble() et float readFloat() : lecture d'un nombre réel

int readInt() et long readLong() : lecture d'un nombre entier

...

Les Fichiers à Accès Aléatoire

Quelques méthodes

//Écriture

void write(byte[] b) : écriture d'octets

void writeChar(int v) : écriture d'une valeur de type caractère

void writeChars(String s) : écriture d'une chaîne de caractères

void writeDouble(double d) : écriture d'un nombre réel (8 octets)

void writeFloat(float f) : écriture d'un nombre réel (sur 4 octets)

void writeInt(int i) : écriture d'un nombre entier (4 octets)

void writeLong(long l) : écriture d'un nombre entier

Les Fichiers à Accès Aléatoire

Exemple - fichier structuré

(Création d'un fichier à accès aléatoire structuré)

Structure et taille d'un enregistrement

int codeClient	→	4 octets
String nomClient (sur 20 caractères maxi)	→	40 octets
String adresseClient (sur 256 caractères maxi)	→	512 octets
double solde	→	8 octets
	→	564 octets

Les Fichiers à Accès Aléatoire

```
import java.io.*;  
  
public class Lecture {  
  
    public static void main(String[] args) {  
        try {  
            RandomAccessFile rf = new RandomAccessFile("../test.bin", "r");  
  
            int pos = Integer.parseInt(args[0]);  
  
            System.out.println("Lecture de l'élément en position :" + pos);  
  
            rf.seek(pos*564);  
  
            //Lecture de l'enregistrement  
            int codeclient = rf.readInt();  
            System.out.println("Code client :" + codeclient);  
  
            ...  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Les Fichiers à Accès Aléatoire

```
byte[] b = new byte[40];
rf.readFully(b);
System.out.println("Nom client :" + new String(b, "UTF-16BE") );

b = new byte[512];
rf.readFully(b);
System.out.println("Email client :" + new String(b, "UTF-16BE") );

double solde = rf.readDouble();
System.out.println("Montant client :" + solde);

}catch (IOException e){
    System.out.println("Erreur IO : "+e.getMessage());
}
}

}
```

La sérialisation des objets

La plate forme java permet de sauvegarder les objets en tant que tels au sein de fichiers. Ce traitement est appelé sérialisation. Elle permet aussi de restaurer (en tant objet) les objets sauvegardés.

Pour ce faire, il faut déclarer *Serializable* les classes concernées et mettre en place des flux de sortie pour réaliser la sauvegarde, ainsi que des flux d'entrée pour effectuer la restauration des objets.

Pour illustrer ce point, considérons un programme qui gère une collection d'objets (*CollectionEmploye*) de type *Employe*.

Il faut tout d'abord déclarer sérialisables les classes *Employe* et *CollectionEmploye* :

```
class Employe implements Serializable { ...}
```

```
class CollectionEmploye implements Serializable { ...}
```

La sérialisation des objets

```
public class SerialCollectionEmploye {  
    public static void main (String args[]){  
        CollectionEmploye employes = new CollectionEmploye();  
        employes.remplir();  
        employes.afficherEmployes();  
  
        try {  
            FileOutputStream f = new FileOutputStream ("emps.obj");  
            ObjectOutputStream s = new ObjectOutputStream(f);  
            s.writeObject(employes);  
            s.flush();  
        }  
        catch (IOException e){System.out.println(" Erreur E/S ");}  
    }// fin main  
}
```

La sérialisation des objets

```
public class RestaureCollectionEmploye {  
    public static void main (String args[]){  
        /* On introduit ici un constructeur sans argument qui ne fait rien car le compilateur  
        impose de créer tout d'abord un objet de type CollectionEmploye pour initialiser le  
        traitement*/  
  
        CollectionEmploye employes = new CollectionEmploye();  
  
        try {  
            FileInputStream f = new FileInputStream ("emps.obj");  
            ObjectInputStream s =  new ObjectInputStream(f);  
            employes = (CollectionEmploye ) s.readObject();  
        }  
        catch (IOException e){System.out.println(" Erreur E/S ");}  
        catch (ClassNotFoundException e){System.out.println(" Pb classe ");}  
  
        employes.afficherEmployes();  
  
    }// fin main  
  
}
```

