Programmation Orientée Objet en JAVA

Dr Abdoulaye GUISSE

Département GIT
Ecole Polytechnique de Thiès › EPT

Plan général du cours

- **Bases du langage Java** : Présentation ; Bases syntaxiques ; Notions techniques (ClassPath, Jar) ; Outils de développement
- **Fondamentaux P.O.O** : Classes, héritages, Polymorphisme, Interfaces, Paquetages de base
- **API Avancées** : Classes de test, String, Vector, ArrayList, Arrays, Enumération, HashTable, Gestion des fichiers
- **Bonnes Pratiques** : Normes, Techniques spécifiques, Gestion de la mémoire (garbage Collector), Javadoc, Ant, Maven, Validation des données, Tests unitaires (JUNIT), Internationalisation
- **Interfaces graphiques** : Bases, Construction avec JBuilder, API AWT, SWING, SWT, et JFace
- **Gestion des base de données** : JDBC, Hibernate
- **Compléments pour Développeur avancé** : Applets, Threads d'exécution, Programmation TCP-IP, Java RMI, Applications Distribuées, Développement Mobile en JAva

Bibliographie

- Développons en JAVA - J. M. Doudoux - version 2014
- Les cahiers du Programmeur Java - Emmanuel Puybaret - 2006
- Apprentissage du Langage Java - Serge Tahé - 2002

Chapitre 2 : Fondamentaux P.O.O

Concept de classe

Un ensemble d'objets ayant les mêmes caractéristiques.

- **Une classe est une description abstraite d'un objet**.
- Les caractéristiques regroupent les **attributs** (ou propriétés) et des **méthodes** (ou fonctions), les manipulant.
- **Instancier une classe consiste à créer un objet sur son modèle**.
- **Entre classe et objet** il y a, en quelque sorte, le **même rapport qu'entre type et variable**.

```
public class C1{
   type1 p1;          // propriété p1
   type2 p2;          // propriété p2
   …
   type3 m3(…){       // méthode m3
      …
   }
   type4 m4(…){       // méthode m4
      …
   }
   …
}
```

Concept de classe

Une classe se compose de deux parties : un en-tête et un corps

- Le corps divisé en 2 sections : **la déclaration attributs, et la définition des méthodes**.
- Les attributs et les méthodes sont pourvues d'états de **visibilité qui gèrent leur accessibilité par les composants hors de la classe**.

```
import java.io.*;

public class Personne{
  // attributs
  private String prenom;
  private String nom;
  private int age;

  // méthode
  public void initialise(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
  public void identifie(){
    System.out.println(prenom+","+nom+","+age);
  }
}
```

Concept de classe

Les états de visibilité sont appelés modificateurs de classe

- ***abstract*** : la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
- ***final*** : la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
- ***private*** : la classe n'est accessible qu'à partir du fichier où elle est définie
- ***public*** : la classe est accessible partout

Convention de nommage en Java

Les noms des classes

- 1ère lettre en majuscule
- Donner des noms simples et descriptifs
- Mélange de minuscule, majuscule avec la première lettre de chaque mot en majuscule
- N'utiliser que les lettres [a-z] et [A-Z] et [0-9] : Ne pas utiliser de tiret '-', d'underscore '_', ou d'autres caractères ($, *, accents, ...).

Convention de nommage en Java

Les noms des caractéristiques : attributs, méthodes, arguments de méthodes, variables locales, constante

- 1ère lettre en minuscule
- Mélange de minuscule, majuscule avec la première lettre de chaque mot en majuscule
- Variable d'une seule lettre pour un usage local : **int i, j, k, m ;**
- Ne pas commencer les noms avec '$' ou '_' bien que ce soit possible.
- N'utiliser que les lettres [a-z] et [A-Z] et [0-9] : Ne pas utiliser de tiret '-', d'underscore '_', ou d'autres caractères ($, *, accents, ...).
- **Exception pour les constantes** : Tout est en majuscule et Séparer les mots par underscore '_'

Attributs d'une classe

- Les données d'une classe sont contenues dans des **variables nommées attributs**.
- Ce sont des variables qui peuvent être des **variables d'instances**, des **variables de classes** ou des **constantes**.

Variables **d'instance**

- Une variable **d'instance** nécessite simplement une déclaration de la variable dans le corps de la classe.
- **Chaque instance de la classe a accès à sa propre occurrence de la variable**.

```
public class MaClasse {
   public int valeur1 ;
   int valeur2 ;
   protected int valeur3 ;
   private int valeur4 ;
}
```

Attributs d'une classe

Variables **de classe**

- Les variables de classes sont définies avec le mot clé **static**.
- **Chaque instance de la classe partage la même variable**.

```
public class MaClasse {
    static int compteur ;
}
```

Constantes

- Les constantes sont définies avec le mot clé **final** : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées.

```
public class MaClasse {
    final double pi=3.14 ;
}
```

Méthodes de classes

Les méthodes sont des fonctions

- Elles implémentent les traitements de la classe.
- La syntaxe de la déclaration d'une méthode est :
- **modificateurs type_retourné nom_méthode (arg1, ...) ...**
  // définition des variables locales et du bloc d'instructions.
- Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise **void**.

Les arguments de types primitifs passés par valeur, les objets par référence

- Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis.
- **Un objet est transmis comme argument par sa référence qui désigne son emplacement mémoire.**
- Il est possible de modifier l'objet grâce à ses méthodes mais il n'est pas possible de remplacer la référence.

Méthodes de classes

"**return**" d'une méthode

- La valeur de retour de la méthode doit être transmise par l'instruction **return**.
- Elle indique la valeur que prend la méthode et termine celle ci : toutes les instructions qui suivent **return** sont donc ignorées.
- Il est possible d'inclure une instruction **return** dans une méthode de type **void** : cela permet de quitter la méthode.

```
int add(int a, int b) {
   return a + b; }
```

Spécificité du **main**

- *public static void main (String args[]) ...* : **Sans return**
- Si la déclaration de la méthode main() diffère, une exception **java.lang.NoSuchMethodError** sera levée par la JVM

Méthodes de classes

Surcharge de méthode

- La surcharge d'une méthode permet de **définir plusieurs fois une même méthode avec des arguments différents**.
- Le compilateur choisi la méthode qui doit être **appelée en fonction du nombre et du type des arguments**.
- Il est donc possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes.

```
class affiche{
    public void afficheValeur(int i) {
        System.out.println(" nombre entier = " + i);
    }

    public void afficheValeur(float f) {
        System.out.println(" nombre flottant = " + f);
    }
}
```

Méthodes de classes

Constructeurs

- La déclaration d'un objet est suivie d'une sorte **d'initialisation** par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une **valeur de départ**.
- Elle n'est systématiquement invoquée que lors de la création d'un objet.
- **Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe**
- **Et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur.**
- **On peut surcharger un constructeur.**

Méthodes de classes

Constructeur par défaut

- **La définition d'un constructeur est facultative.**
- **Si aucun constructeur n'est explicitement défini dans la classe, le compilateur va créer un constructeur par défaut sans argument.**
- Dès qu'un constructeur est explicitement défini, le compilateur considère que le programmeur prend en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, n'est pas mis en oeuvre.
- Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres en plus des autres constructeurs.

Méthodes de classes

Plusieurs manières de définir un constructeur :

1. le **constructeur simple** : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.
   public MaClasse() { }
2. le **constructeur avec initialisation fixe** : il permet de créer un constructeur par défaut.
   public MaClasse() {nombre = 5 ; }
3. le **constructeur avec initialisation des variables** : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur.
   public MaClasse(int valeur) { nombre = valeur ; }

Méthodes de classes

Rôle des accesseurs

- L'encapsulation permet de sécuriser l'accès aux données d'une classe.
- Ainsi, les données déclarées **private** à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe.
- Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet.

Méthodes de classes

Accesseurs

- Un **accesseur** est une méthode publique qui donne l'accès à une variable d'instance privée.
- Par convention, les accesseurs en **lecture** commencent par **get** et les accesseurs en **écriture** commencent par **set**.
- Pour un attribut de type booléen, il est possible de faire commencer l'accesseur en lecture par **is** au lieu de **get**.

```
private int valeur = 13;

public int getValeur(){
    return(valeur);
}

public void setValeur(int val) {
    valeur = val;
}
```

Modificateurs d'accès

Modificateurs d'attribut

- **public** : la variable est accessible aux méthodes des autres classes
- **private** : l'usage de la variable est réservé aux autres méthodes de la même classe
- **protected** : la variable ne peut être invoquée que par des méthodes de la classe ou de ses sous-classes
- **final** : la variable est une constante et ne peut être modifiée (redéfinition lors de l'héritage interdite)
- **static** : la variable appartient simultanément à tous les objets de la classe. Il est inutile d'instancier la classe pour appeler la variable.
- **volatile** : la variable peut être changée par un périphérique ou de manière asynchrone.

Modificateurs d'accès

Modificateurs de méthode

- **public** : la méthode est accessible aux méthodes des autres classes
- **private** : l'usage de la méthode est réservé aux autres méthodes de la même classe
- **protected** : la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous-classes
- **final** : la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
- **static** : la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
- **synchronized** : la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
- **native** : le code source de la méthode est écrit dans un autre langage.

- **Sans modificateur**, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.

La création d'un objet : instancier une classe

Déclaration

- Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré.
- La déclaration est de la forme : **nom_de_classe *nom_de_variable***
- Exemple : **MaClasse *m*** ; **String *chaine*** ;

Mot clé **new**

- L'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable
- Exemples : ?
  m = new MaClasse() ;
  MaClasse m = new MaClasse() ;
- Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

La durée de vie d'un objet

- **Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.**

Durée en 3 étapes :

1. La déclaration de l'objet et l'instanciation grâce à l'opérateur new.
2. L'utilisation de l'objet en appelant ses méthodes
3. La suppression de l'objet : elle est automatique en Java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction delete comme en C++.

La variable this

Accès à l'objet courant :

- L'instruction ***this.prenom=P*** signifie que l'attribut ***prenom*** de l'objet courant (***this***) reçoit la valeur P.
- Le mot clé ***this*** désigne l'objet courant : celui dans lequel se trouve la méthode exécutée.
- ***this*** est aussi utilisé quand l'objet doit appeler une méthode en se passant lui-même en paramètre de l'appel.

L'opérateur instanceof

- L'opérateur instanceof permet de déterminer la classe de l'objet qui lui est passé en paramètre.
- La syntaxe est : ***objet instanceof classe***
- Nécessité d'effectuer un casting (conversion) pour accéder aux données et méthodes de la classe.

```
void afficheChaine(Object o) {
   if (o instanceof MaClasse)
   {
      MaClasse m = (MaClasse) o;
      System.out.println(m.getChaine());
      // OU System.out.println( ((MaClasse) o).getChaine() );
   }
}
```

Chapitre 2 : Fondamentaux P.O.O

Principe d'héritage

- L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution.
- Elle définit une relation entre deux classes :
  - une classe mère ou super classe
  - une classe fille ou sous-classe qui hérite de sa classe mère

Grâce à l'héritage

- **Les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre.**
- **Les sous-classes peuvent redéfinir les variables et les méthodes héritées.**
  - **Les variables son redéclarées sous le même nom avec un type différent.**
  - **Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.**

Principe d'héritage

Hiérarchie de classes

- L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super classes et de sous-classes.
- Une classe qui hérite d'une autre est une sous-classe et celle dont elle hérite est une super classe.
- Une classe peut avoir plusieurs sous-classes.
- Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.

- Exemple : Object est la classe parente de toutes les classes en Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

Mise en oeuvre de l'héritage

Mot réservé **extends**

- **class Fille extends Mere { ... }**
- On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.

Comment hériter ?

Variables

- Les variables définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.
- Pour appeler le constructeur de la classe mère, il suffit d'écrire **super(paramètres) ;** avec les paramètres adéquats.
- Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement mais via les méthodes héritées.
- Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur protected.

Méthodes

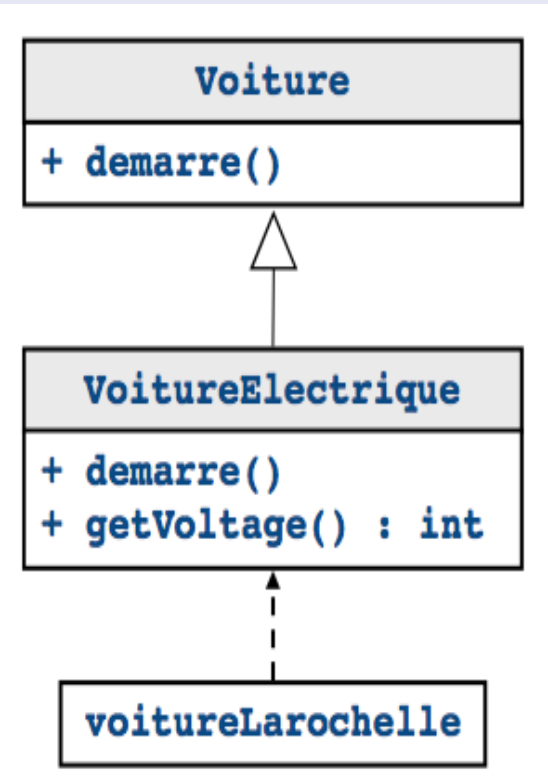
- Les méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.
- Pour invoquer une méthode d'une classe mère, il suffit de la préfixée par **super**. Exemple : **super.identite() ;**

Principe du polymorphisme

Variables

- Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé.
- L'héritage définit un **cast** implicite de la classe fille vers la classe mère : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous-classes.
- Il est possible de surcharger une méthode héritée : la forme de la méthode à exécuter est choisie en fonction des paramètres associés à l'appel.

Principe du polymorphisme : Exemple 1

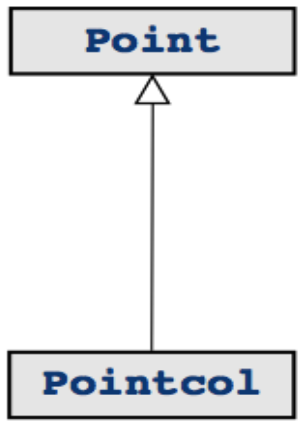



```
public class Test {
    public static void main (String[] argv) {
        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

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

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

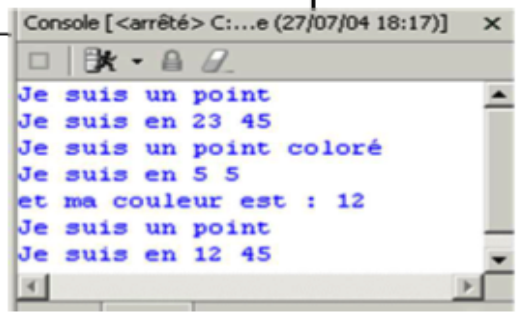
Principe du polymorphisme : Exemple 2

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

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




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



Chapitre 2 : Fondamentaux P.O.O

Classes abstraites

Présentation

- Syntaxe :
  abstract class A {
  *****
  }
- Une classe abstraite est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation.
- Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée.
- **Mais on peut aussi trouver des méthodes dites abstraites, c'est- à-dire dont on ne fournit que la signature et le type de la valeur de retour.**

```
abstract class A
{  public void f() {  .....  }          // f est définie dans A
   public abstract void g(int n) ;      // g n'est pas définie dans A ; on n'en
                                        // a fourni que l'en-tête
}
```

Classes abstraites

Règles

- **Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot-clé *abstract* devant sa déclaration.**
- Une méthode abstraite doit obligatoirement être déclarée public, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.
- Dans l'en-tête dune méthode déclarée abstraite, les noms d'arguments muets doivent figurer :
  public abstract void g(int) ; // erreur : nom d'argument (fictif) obligatoire
- Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base. Dans ce cas, elle reste simplement abstraite.

Classes abstraites

```
abstract class A
{  public abstract void f1() ;
   public abstract void f2 (char c) ;
      .....
}
abstract class B extends A        // abstract obligatoire ici
{  public void f1 () {  ..... }   // définition de f1
      .....                       // pas de définition de f2
}
```

Interfaces

Rôle

- Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super classes. **Ce mécanisme n'existe pas en Java.**
- **Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement.**

Principe

- **Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite.**
- **Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle-ci.**
- **Plusieurs interfaces peuvent être implémentées dans une même classe.**

Interfaces

Mise en œuvre

- La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot-clé **interface** à la place de **class** :

```
public interface I
{  void f(int n) ;    // en-tête d'une méthode f (public abstract facultatifs)
   void g() ;         // en-tête d'une méthode g (public abstract facultatifs)
}
```

- Une interface peut être dotée des mêmes droits d'accès qu'une classe.
- Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes ou des constantes.
- Il n'est pas nécessaire de mentionner les méthodes les mots-clés **public** et **abstract**.

Interfaces

Implémentation

- Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot-clé **implements**
- Une même classe peut implémenter plusieurs interfaces.

```
public interface I1
{  void f() ;
}
public interface I2
{  int h() ;
}
class A implements I1, I2
{  // A doit obligatoirement définir les méthodes f et h prévues dans I1 et I2
}
```

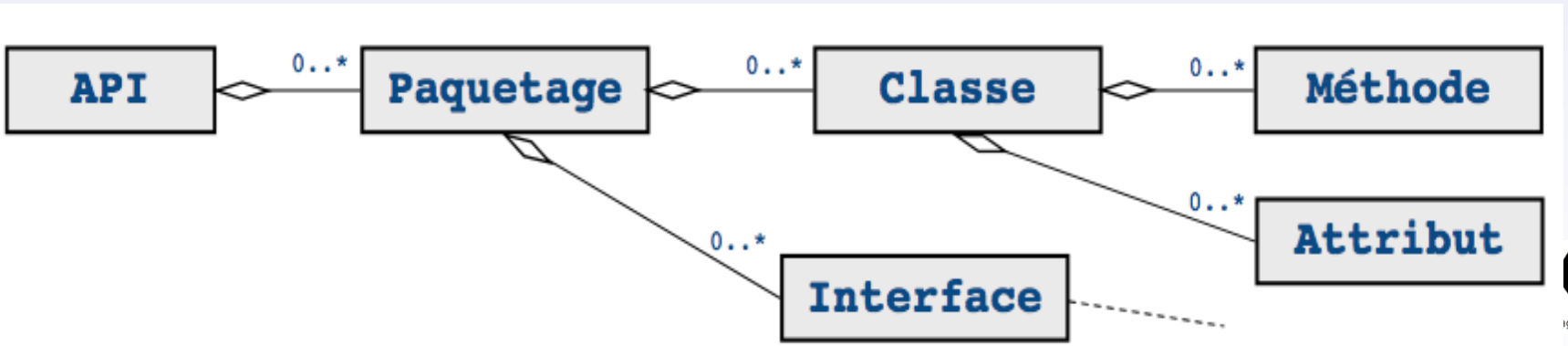
- Pour instance une interface : **public interface I { .....}**
  I i ; // i est une référence à un objet d'une classe implémentant l'interface I
  I i = new I(...) ; //Erreur car impossible
  class A implements I { ..... }
  I i= newA(...) ; //OK

Chapitre 2 : Fondamentaux P.O.O

Paquetages

Rôle

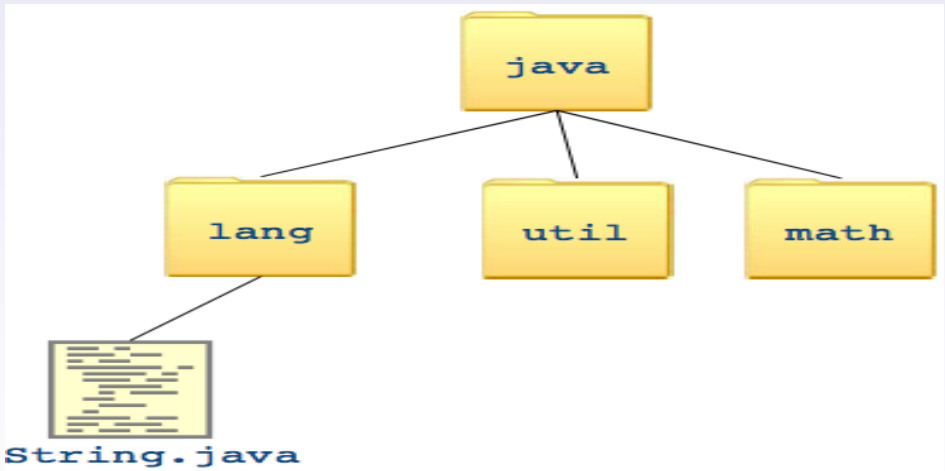
- Le langage Java propose une définition très claire du mécanisme d'empaquetage qui permet de classer et de gérer les API externes.
- Un package est donc un groupe de classes associées à une fonctionnalité.
- Exemple : **java.lang** : rassemble les classes de base Java (Object, String, ...).
  java.util : rassemble les classes utilitaires (Collections, Date, ...).
  java.io : lecture et écriture.



Paquetages

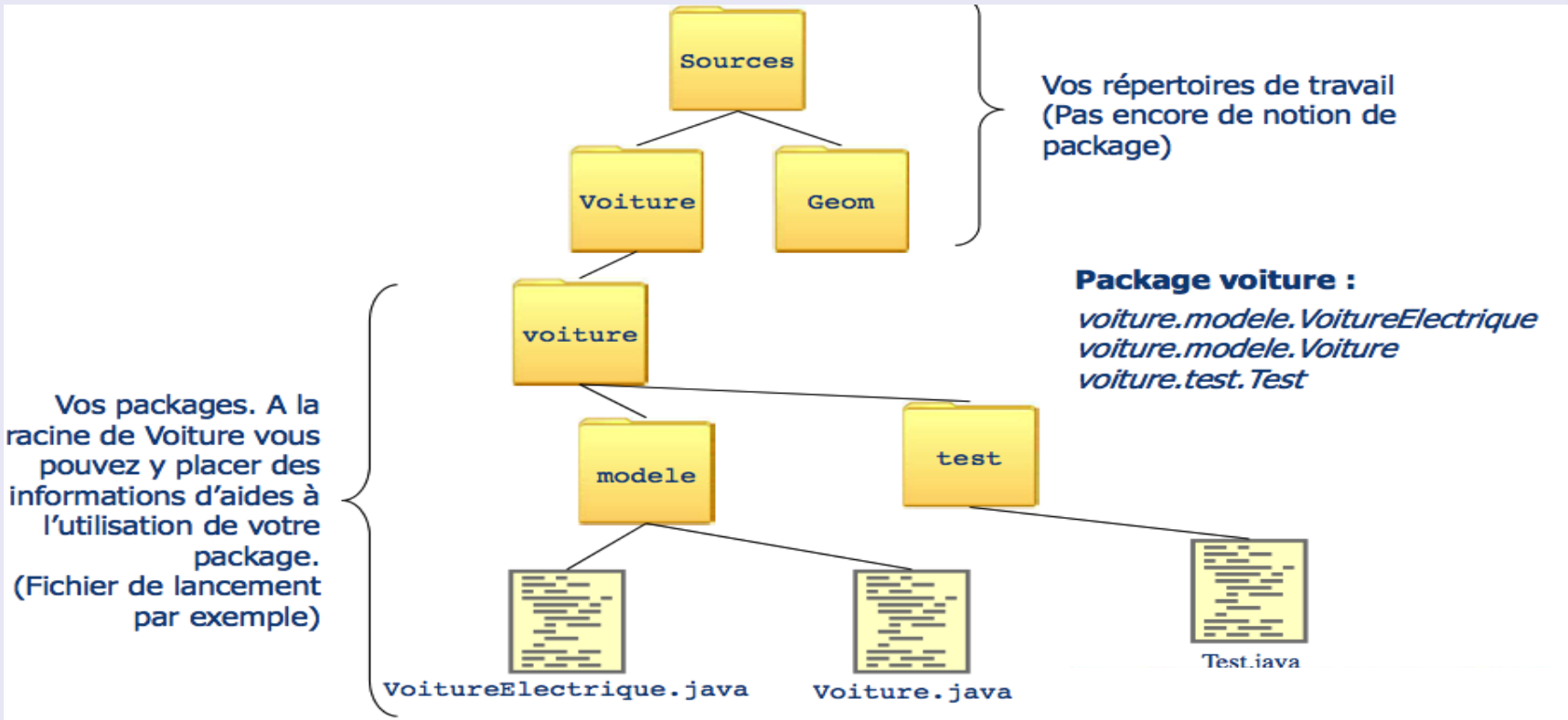
Existence physique

- A chaque classe Java correspond un fichier.
- A chaque package (sous-package) correspond un répertoire
- A une hiérarchie de packages correspond une hiérarchie de répertoires dont les noms coïncident avec les composants des noms de package.
- Exemple de la classe String :



Paquetages

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



Merci de votre attention,
et
Rendez-vous au prochain cours,
pour le reste du cours.