

# Java CM2

Olivier Marchetti

Laboratoire d'informatique de Paris 6 – Pôle SoC – Sorbonne Université

7 octobre 2022



# Plan

- 1 Surcharge – Ellipse
  - Surcharge de méthodes
  - Ellipse
- 2 Héritage
  - Définition
  - Héritage et construction d'objet
  - Héritage et type
- 3 Redéfinition
  - Redéfinition de méthode d'instance
  - Polymorphisme
  - La méthode `toString()`
  - Redéfinition de méthode de classe
- 4 Notion de paquetage
  - Définition
  - L'instruction `import`
  - Compilation



- 1 Surcharge – Ellipse
  - Surcharge de méthodes
  - Ellipse
- 2 Héritage
  - Définition
  - Héritage et construction d'objet
  - Héritage et type
- 3 Redéfinition
  - Redéfinition de méthode d'instance
  - Polymorphisme
  - La méthode toString()
  - Redéfinition de méthode de classe
- 4 Notion de paquetage
  - Définition
  - L'instruction import
  - Compilation

# La surcharge de méthodes

La surcharge<sup>1</sup> consiste à définir dans une même classe différentes méthodes ayant le même nom mais des arguments différents.

```
1 class MethodeSurcharge {
2     static float methodeInverse(int n) {
3         return 1 / ((float) n);
4     }
5
6     static float methodeInverse(int n, boolean verif) {
7         if (verif && (n == 0)) {
8             System.out.println("Division par zéro !");
9             return 0.0f;
10        }
11        else {
12            return methodeInverse(n);
13        }
14    }
15
16    public static void main(String args[]) {
17        int nb1 = 4;
18        int nb2 = 0;
19        System.out.println("Inverse(" + nb1 + ") = "
20            + methodeInverse(nb1));
21        System.out.println("Inverse(" + nb2 + ") = "
22            + methodeInverse(nb2, true));
23    }
24 }
```

```
[12:54][Prog pc666 :]$ java MethodeSurcharge
Inverse(4) = 0.25
Division par zéro !
Inverse(0) = 0.0
[12:54][Prog pc666 :]$
```

- ▶ La méthode invoquée est celle présentant une correspondance valable entre les arguments :
  - effectifs,
  - formels.
- ▶ Cette méthode sera dite « surchargée ».

<sup>1</sup>En anglais : *overloading method*.

# La surcharge de méthodes – intérêt

- ▶ Un bon langage de programmation se doit d'offrir :

- 1 des facilités d'écriture (constructions élégantes),
- 2 une lisibilité/intelligibilité du code.

La surcharge permet cela en évitant d'alourdir le code source avec davantage de noms méthodes offrant des services proches.

- ▶ Exemple classique : la surcharge du constructeur.

```

1 class Etudiant {
2     private String nom;
3     private String prenom;
4     private int numEtudiant;
5
6     Etudiant(String n, String p, int numEtu) {
7         nom = n;
8         prenom = p;
9         numEtudiant = numEtu;
10    }
11
12    Etudiant(String n, String p) {
13        nom = n;
14        prenom = p;
15        numEtudiant = -1; // Inscription non faite.
16    }

```

```

17     void afficherEtudiant() {
18         String finMessage = "";
19         if (numEtudiant >= 0) {
20             finMessage = ", num : " + numEtudiant;
21         }
22         System.out.println("nom : " + nom +
23                             ", prenom : " + prenom +
24                             finMessage);
25     }
26
27     public static void main(String args[]) {
28         Etudiant e1 = new Etudiant("Inzescaille",
29                                     "Lucy",
30                                     7);
31         Etudiant e2 = new Etudiant("Dutchad",
32                                     "Toumai");
33         e1.afficherEtudiant();
34         e2.afficherEtudiant();
35     }
36 }

```



# La surcharge de méthodes – écueil

La surcharge **ne pourrait fonctionner uniquement en reposant sur des types de retour différents !**

```
class ZutAlors {  
    ...  
    void methodeMauvaiseIdee() {  
        ...  
    }  
  
    int methodeMauvaiseIdee() {  
        ...  
        return resultat;  
    }  
    ...  
}
```

Pourquoi ?

# La surcharge de méthodes – limites

Un programmeur écrit :

```
1 class Fraction {  
2     static String afficherFraction(short a, int b) {  
3         return a + "/" + b;  
4     }  
5  
6     static String afficherFraction(int a, long b) {  
7         return a + "/" + b;  
8     }  
9  
10    static String afficherFraction(int a, short b) {  
11        return a + "/" + b;  
12    }  
13  
14    public static void main(String args[]) {  
15        short a = 13;  
16        int b = 11;  
17        long l = 42;  
18        System.out.println(afficherFraction(a, b));  
19        System.out.println(afficherFraction(b, a));  
20        System.out.println(afficherFraction(l, l));  
21        System.out.println(afficherFraction(a, a));  
22    }  
23 }
```

À la compilation,

javac a deux possibilités. Ce sera la troisième méthode qui sera appelée.

et deux erreurs sont détectées :

javac ne trouve aucune méthode acceptant deux long ;

javac détecte que l'appel est ambigu (deux appels étant possibles).

## Appel de méthode et hiérarchie des types

javac s'appuie sur la hiérarchie des types pour déterminer, en fonction des types des arguments, la meilleure méthode à appeler (cf. ligne 19).

# Méthode avec un nombre d'arguments variable : l'ellipse

Alternative : méthode avec un nombre d'arguments variable **de même type**.

## ► Syntaxe :

```
typeDeRetour nomMethode([listeArgType], type ... arg) {  
    ...  
    [return [resultat];]  
}
```

- Toujours placée en fin de liste.
- arg est presque comme un tableau :
  - syntaxe avec crochet,
  - syntaxe *for each*,
  - champ `length`.
- Limitations :
  - une seule ellipse possible par méthode.

## ► Exemple :

```
class ExempleEllipse {  
    static void moyenneUE(String matiere, int ... notes) {  
        float moyenne = 0.0f;  
        int somme = 0;  
        for(int note : notes) {  
            somme += note;  
        }  
        System.out.println("Moyenne en " + matiere + " : " +  
            ((float) somme) / notes.length );  
    }  
  
    public static void main(String args[]) {  
        moyenneUE("Java", 10, 12, 15, 18, 5);  
        moyenneUE("C", 15, 15, 13, 16, 15, 14);  
    }  
}
```

```
[20:17] [Prog pc666 :]$ java ExempleEllipse  
Moyenne en Java : 12.0  
Moyenne en C : 14.666667
```



## 1 Surcharge – Ellipse

- Surcharge de méthodes
- Ellipse

## 2 Héritage

- Définition
- Héritage et construction d'objet
- Héritage et type

## 3 Redéfinition

- Redéfinition de méthode d'instance
- Polymorphisme
- La méthode toString()
- Redéfinition de méthode de classe

## 4 Notion de paquetage

- Définition
- L'instruction import
- Compilation

# Héritage – motivations

Un informaticien souhaite refaire un système de gestion des étudiants. Il identifie trois types d'étudiants :

❶ Étudiant :

- Nom
- Prénom
- Âge
- Filière

❷ Étudiant sportif de haut-niveau :

- Nom
- Prénom
- Âge
- Filière
- Sport

❸ Étudiant en alternance :

- Nom
- Prénom
- Âge
- Filière
- Entreprise

Doit-il alors dupliquer les champs, et écrire des constructeurs et des méthodes quasiment identiques ?

⇒ JAVA permet de structurer ces classes intelligemment !

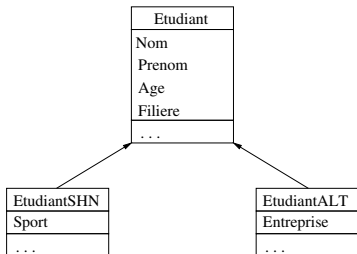
# Héritage – généralités & diagramme UML

L'héritage permet de définir des classes en s'appuyant sur des classes existantes :

- ▶ allègement du code à écrire (pas de redondance) ;
- ▶ lisibilité accrue ;
- ▶ structuration hiérarchique du projet.

La modélisation UML prévoit ce cas de figure.

Exemple :



# Héritage – syntaxe en JAVA

Pour définir une classe B héritant d'une classe A, on utilisera le mot clé extends :

```
class A {  
    ...  
}  
  
class B extends A {  
    ...  
}
```

Nous dirons que la classe B

- ▶ « hérite de » ou « étend » ou « dérive de »
- ▶ a pour super-classe

la classe A.

De plus, un objet défini sur le modèle de classe B dispose en plus des champs et méthodes propres à sa classe :

- ▶ des champs de la classe A,
- ▶ des méthodes de la classe A.

# Héritage – le constructeur : la méthode `super()`

Après avoir écrit les champs de la classe `Etudiant`, on peut définir le constructeur ainsi :

```
Etudiant(String n, String p, int a, String f) {  
    nom = n;  
    prenom = p;  
    age = a;  
    filiere = f;  
}
```

Faut-il reprendre ces instructions pour le constructeur de `EtudiantSHN` ?

Inutile et risqué !

La méthode `super()` (avec ou sans argument), placée au début, permet d'invoquer le constructeur de la super-classe. Ainsi :

```
EtudiantSHN(String n, String p, int a, String f, String s) {  
    super(n, p, a, f);  
    sport = s;  
}
```

On parle alors de « chaînage des constructeurs ».

# Héritage – retour sur l'exemple

Implémentation simple et minimaliste de nos classes :

```

1  class Etudiant {
2      String nom;
3      String prenom;
4      int age;
5      String filiere;
6
7      Etudiant(String n, String p, int a,
8              String f) {
9          nom = n;
10         prenom = p;
11         age = a;
12         filiere = f;
13     }
14 }
15
16 class EtudiantSHN extends Etudiant {
17     String sport;
18
19     EtudiantSHN(String n, String p, int a,
20               String f, String s) {
21         super(n, p, a, f);
22         sport = s;
23     }
24 }

```

```

25 class EtudiantALT extends Etudiant {
26     String entreprise;
27
28     EtudiantALT(String n, String p, int a,
29               String f, String e) {
30         super(n, p, a, f);
31         entreprise = e;
32     }
33 }
34
35 class ExempleHéritageEtudiant {
36     public static void main(String args[]) {
37         Etudiant e1 = new Etudiant("Baez", "Joan", 22,
38                                   "Musique");
39         EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
40                                         "Science", "Course");
41         EtudiantALT e3 = new EtudiantALT("French", "Melinda", 20,
42                                         "Gestion", "Microsoft");
43         System.out.println(e1 + " " + e2 + " " + e3);
44     }
45 }

```

```
[15:58][Prog pc666 :]$ java ExempleHéritageEtudiant
Etudiant@2a139a55 EtudiantSHN@15db9742 EtudiantALT@6d06d69c
```

Qu'est-ce que cet étrange affichage ?

# Héritage – chaînage de constructeur 1/2

Lors de la création d'objet appartenant à une classe héritante, il y a construction de cet objet par récursion sur l'arbre d'héritage :

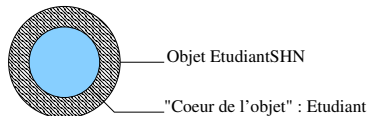
```
class Etudiant {
    ...
    Etudiant(String n, String p, int a,
              String f) {
        nom = n;
        prenom = p;
        age = a;
        filiere = f;
    }
}

class EtudiantSHN extends Etudiant {
    ...
    EtudiantSHN(String n, String p, int a,
                 String f, String s) {
        super(n, p, a, f);
        sport = s;
    }
}
...
class ExempleHeritageEtudiant {
    public static void main(String args[]) {
        EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
                                           "Science", "Course");
        ...
    }
}
```

❶ **new EtudiantSHN** : appel au constructeur de la classe EtudiantSHN.

❷ **super(...)** : appel au constructeur de la super-classe.

❸ Une fois la construction « interne » de l'objet terminée, reprise de la construction au niveau du constructeur EtudiantSHN.



# Héritage – chaînage de constructeur 2/2

La première ligne du constructeur doit appeler celui de la super-classe.

Sinon, appel implicite à un constructeur par défaut sans argument !

```
1 class Etudiant {
2     Etudiant() {
3         System.out.println("Etudiant");
4     }
5 }
6
7 class EtudiantALT extends Etudiant {
8     EtudiantALT() {
9         System.out.println("EtudiantALT");
10    }
11
12    EtudiantALT(int alt) {
13        this();
14        System.out.println("EtudiantALT avec argument");
15    }
16 }
17
18 class EtudiantFC extends EtudiantALT {
19     EtudiantFC() {
20         super(20);
21         System.out.println("EtudiantFC");
22     }
23
24     public static void main(String args[]) {
25         new EtudiantFC();
26     }
27 }
```

```
[10:01][Prog pc666 :]$ java EtudiantFC
Etudiant
EtudiantALT
EtudiantALT avec argument
EtudiantFC
```

- 1 Ligne 25 : création d'un objet EtudiantFC.
- 2 Ligne 20 : appel au constructeur avec argument de la super-classe.
- 3 Ligne 13 : appel au constructeur sans argument de la classe.  

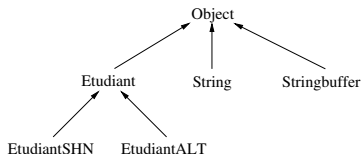
this()      (au début)
- 4 Ligne 9 : appel implicite au constructeur de la super-classe.



# Héritage – terminologie & propriétés

- ▶ Toute classe hérite implicitement de la classe `Object`.
- ▶ Une classe ne peut hériter que d'au plus une seule classe.

⇒ l'ensemble des classes  
forme une arborescence !



- ▶ Le relation d'héritage est une relation transitive, c'est-à-dire :
  - soit une classe B qui hérite d'une classe A ;
  - soit une classe C qui hérite de la classe B ;⇒ la classe C hérite de la classe A.
- ▶ Si la définition d'une classe est précédée du mot clé `final` alors cette classe ne pourra être étendue.

# Héritage – référencement des objets et affectation

Une variable d'un type A peut

- ❶ référencer tout objet,
- ❷ être affectée par une variable,

d'un type dérivé de A.

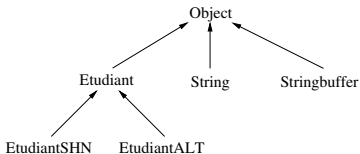
Exemple :

```
Object obj = new StringBuffer("Hello World !");
String message = "Hello World !"
obj = message;
```

## Règle d'or

**« Qui décrit le moins peut référencer le plus. »**

Exemples :



Une variable de type :

- ▶ Object peut référencer un objet de n'importe quel type ;
- ▶ Etudiant peut aussi référencer un objet de type EtudiantSHN ou EtudiantALT.

# Héritage – conversion de types références

javac vérifie le typage des variables et des opérations d'affectation.

```
class A {}

class B extends A {
    public static void main(String args[]) {
        A a = new A();    // ok
        a = new B();      // ok
        B b = new B();    // ok
        b = a;
    }
}
```

```
[10:42][Prog pc666 :]$ javac HeritageTypeConversion.java
HeritageTypeConversion.java:8: error: incompatible types:
                        A cannot be converted to B
b = a;
~
1 error
```

vérification « statique » de javac

À l'exécution, les affectations auraient été pourtant cohérentes en type.

On peut forcer javac à accepter ce code avec un cast().

```
class A {}

class B extends A {
    public static void main(String args[]) {
        A a = new A();
        a = new B();
        B b = new B();
        b = (B) a; // ok, mais risqué !
    }
}
```

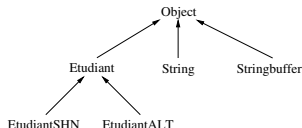
```
[10:42][Prog pc666 :]$ javac HeritageTypeConversion.java
[10:42][Prog pc666 :]$
```

transtypage explicite

Attention : il faut être sûr de son coup !

# Héritage – gestion des types par javac

Examinons de nouveau l'arborescence d'héritage :



```

1 class HeritageGestionDesTypes {
2     public static void main(String args[]) {
3         Etudiant e1 = (Math.random() > 0.5) ? new EtudiantSHN( ... ) : new EtudiantALT( ... );
4     }
5 }

```

- 1 Un étudiant sportif de haut-niveau/en alternance est bien un étudiant.

```

1 class HeritageGestionDesTypes {
2     public static void main(String args[]) {
3         Etudiant e1 = (Math.random() > 0.5) ? new EtudiantSHN( ... ) : new EtudiantALT( ... );
4         EtudiantSHN e2 = e1;
5     }
6 }

```

- 2 Un étudiant sportif de haut niveau est-il n'importe quel étudiant ?

⇒ Pas forcément et javac veille à ce que cela reste cohérent !

# Héritage & type – l'opérateur instanceof

On appelle classe propre d'un objet le type de cet objet en mémoire.

► Exemple :

```
Object obj = new StringBuffer("Hello World !");
String message = "Hello World !"
obj = message;
```

- obj est de type Object pour le compilateur,
- obj désigne ici cependant à l'exécution un objet de type StringBuffer.

Une instance est aussi une instance de toutes les classes dont elle hérite.

```
1 class ExempleHéritageEtType {
2     public static void main(String args[]) {
3         Object obj = new EtudiantALT("Melinda", "French", 20,
4             "Gestion", "Microsoft");
5         System.out.print("obj est-il une instance de Object ?");
6         System.out.println(obj instanceof Object);
7         System.out.print("obj est-il une instance de Etudiant ?");
8         System.out.println(obj instanceof Etudiant);
9         System.out.print("obj est-il une instance de EtudiantSHN ?");
10        System.out.println(obj instanceof EtudiantSHN);
11    }
12 }
```

```
[17:02][Prog pc666 :]$ java ExempleHéritageEtType
obj est-il une instance de Object ? true
obj est-il une instance de Etudiant ? true
obj est-il une instance de EtudiantSHN ? false
```

L'opérateur instanceof examine l'objet en mémoire retourne true si et seulement s'il est une instance du type spécifié, false sinon.

# Héritage & type – transtypage explicite & exécution

À l'exécution, la JVM s'assure de la cohérence des affectations avec la classe propre des objets.

```
class A {}  
  
class B extends A {  
    public static void main(String args[]) {  
        A a = new A();  
        B b = (B) a;  
    }  
}
```

```
[20:50][Prog pc666 :]$ javac HeritageMauvaiseConversion.java  
[20:50][Prog pc666 :]$ java B  
Exception in thread "main" java.lang.ClassCastException:  
    A cannot be cast to B  
    at B.main(HeritageMauvaiseConversion.java:6)
```

vérification « dynamique » par la JVM

## Héritage & conversion de types

L'arborescence d'héritage définit une hiérarchie de types et :

- ▶ javac rejette à la compilation,
- ▶ la JVM, en s'appuyant sur la classe propre des objets, rejette à l'exécution,

toutes les affectations allant à l'encontre de cette hiérarchie.

- 1 Surcharge – Ellipse
  - Surcharge de méthodes
  - Ellipse
- 2 Héritage
  - Définition
  - Héritage et construction d'objet
  - Héritage et type
- 3 Redéfinition
  - Redéfinition de méthode d'instance
  - Polymorphisme
  - La méthode `toString()`
  - Redéfinition de méthode de classe
- 4 Notion de paquetage
  - Définition
  - L'instruction `import`
  - Compilation

# Redéfinition de méthodes d'instance : motivation

Rappel : le langage JAVA permet de structurer intelligemment le code en classes. Les classes dérivées partagent :

- ▶ les champs,
- ▶ les méthodes.

La redéfinition<sup>1</sup> est une réécriture d'une définition d'une méthode héritée.

```

1 class Bonjour {
2     void saluer(String s) {
3         System.out.println("Bonjour " + s);
4     }
5 }
6
7 class Hello extends Bonjour {
8     void saluer(String s) {
9         System.out.println("Hello " + s);
10    }
11 }
12
13 class RedefinitionInstance {
14     public static void main(String args[]) {
15         Bonjour b = new Bonjour();
16         b.saluer("Annick");
17         Hello h = new Hello();
18         h.saluer("Alexandre");
19     }
20 }
```

```

[15:01][Prog pc666 :]$ java RedefinitionInstance
Bonjour Annick
Hello Alexandre
```

## Contraintes

- ① La signature devra être identique.
- ② Pas de diminution du modificateur d'accès.

## Intérêts

- ① Compléter/ajuster facilement une méthode aux besoins propres de l'objet.
- ② Conserver un seul et même nom de méthode.

<sup>1</sup>En anglais : *method overriding*.



# Redéfinition de méthodes vs. surcharge

Il ne faut pas confondre redéfinition et surcharge : une méthode de la super-classe peut parfaitement être surchargée.

```

1  class Bonjour {
2      void saluer(String s) {
3          System.out.println("Bonjour " + s);
4      }
5  }
6
7  class Hello extends Bonjour {
8      void saluer(String prenom, String nom) {
9          System.out.println("Hello " + prenom +
10             " " + nom);
11      }
12  }
13
14  class RedefinitionEtSurcharge {
15      public static void main(String args[]) {
16          Bonjour b = new Bonjour();
17          b.saluer("Annick");
18          Hello h = new Hello();
19          h.saluer("Alexandre");
20          h.saluer("Ada", "Lovelace");
21      }
22  }
    
```

```

[15:14][Prog pc666 :]$ java RedefinitionEtSurcharge
Bonjour Annick
Bonjour Alexandre
Hello Ada Lovelace
    
```

À l'exécution :

- appel de la méthode saluer() de la classe Bonjour.
- appel de la méthode saluer() héritée.
- appel de la méthode saluer() surchargée.

# Redéfinition de méthodes et types des objets

La rigueur sur les types donne une contrepartie très intéressante à l'exécution :

Un objet appellera la méthode la plus adaptée à sa classe propre !

```

1  class Bonjour {
2      void saluer() {
3          System.out.println("Bonjour");
4      }
5  }
6
7  class Hello extends Bonjour {
8      void saluer() {
9          System.out.println("Hello");
10     }
11 }
12
13 class ExemplePolymorphisme {
14     public static void main(String args[]) {
15         Bonjour obj = null;
16         for (int i = 0; i < 10; i++) {
17             if (Math.random() < 0.5) {
18                 obj = new Bonjour();
19             }
20             else {
21                 obj = new Hello();
22             }
23             obj.saluer();
24         }
25     }
26 }

```

```

[15:51][Prog pc666 :]$ java ExemplePolymorphisme
Bonjour
Hello
Hello
Bonjour
Bonjour
...

```

Cette propriété est le polymorphisme.

Remarques :

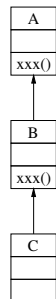
● null permet d'initialiser une variable qui ne pointe nulle-part.

Le type de la variable obj permet de référencer des objets de type Bonjour et Hello.

● l'objet appelle la méthode qui lui correspond le mieux.

# Redéfinition de méthodes – terminologie & propriétés

- ▶ Soient A, B et C trois classes telles que :
  - A contient une méthode `xxx()` ;
  - B hérite de A et contient une redéfinition de la méthode `xxx()`
  - C hérite de B.
- ▶ Une instance de la classe C appelant la méthode `xxx()` appellera la méthode redéfinie dans la classe B.
- ▶ À l'appel d'une méthode `xxx()`, un objet sélectionnera lors de l'exécution la première méthode `xxx()` trouvée en remontant l'arborescence depuis sa classe propre.
- ▶ Une méthode précédée du mot clé `final` ne pourra plus être redéfinie.



# Redéfinition de méthodes – le retour du mot clé super

Imaginons que nous décidions de reprendre l'affichage de nos objets Etudiant :

```
void afficher() {  
    System.out.print(nom + " " +  
                    prenom + " " +  
                    age + " " +  
                    filiere);  
}
```

Faut-il reprendre ces instructions pour la redéfinir dans la classe EtudiantSHN ?

Inutile et risqué !

Une méthode préfixée par super appellera sa dernière redéfinition/définition en remontant l'arborescence depuis la super-classe de la classe courante.

```
void afficher() {  
    super.afficher();  
    System.out.print(" " + sport);  
}
```

Complétion par « chaînage » de la méthode afficher() dans la classe EtudiantSHN.

super.super.nomMethode() est **interdit** en JAVA.

# Redéfinition de méthodes – la méthode toString() 1/2

Retour sur l'exemple :

```
class Etudiant {
    ...
}

class EtudiantSHN extends Etudiant {
    ...
}

class EtudiantALT extends Etudiant {
    ...
}

class ExempleHeritageEtudiant {
    public static void main(String args[]) {
        Etudiant e1 = new Etudiant("Baez", "Joan", 22,
                                   "Musique");
        EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
                                           "Science", "Course");
        EtudiantALT e3 = new EtudiantALT("French", "Melinda", 20,
                                           "Gestion", "Microsoft");
        System.out.println(e1 + " " + e2 + " " + e3);
    }
}
```

```
[15:58][Prog pc666 :]$ java ExempleHeritageEtudiant
Etudiant@2a139a55 EtudiantSHN@15db9742 EtudiantALT@6d06d69c
```

- ▶ La classe `Object` dispose d'une méthode retournant une représentation textuelle de tout objet :

```
public String toString()
```

- ▶ Cette méthode retourne alors une chaîne de caractères avec :
  - le nom de la classe,
  - un identifiant numérique<sup>1</sup>.
- ▶ `println()` appelle implicitement `toString()` lorsqu'elle reçoit une variable de type référence.

<sup>1</sup><https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString->

# Redéfinition de méthodes – la méthode toString() 2/2

Reprise de l'exemple avec redéfinition de la méthode toString() :

```
class Etudiant {
    ...
    public String toString() {
        return nom + " " + prenom + " " +
            age + " " + filiere;
    }
}

class EtudiantSHN extends Etudiant {
    ...
    public String toString() {
        return super.toString() + " " + sport;
    }
}

class EtudiantALT extends Etudiant {
    ...
    public String toString() {
        return super.toString() + " " + entreprise;
    }
}
...
```

```
class ExempleAvecToString {
    public static void main(String args[]) {
        Etudiant e1 = new Etudiant("Baez", "Joan", 22,
            "Musique");
        EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
            "Science", "Course");
        EtudiantALT e3 = new EtudiantALT("French", "Melinda", 20,
            "Gestion", "Microsoft");
        System.out.println(e1 + "\n" + e2 + "\n" + e3);
    }
}
```

À l'exécution :

```
[18:06][Prog pc666 :]$ java ExempleAvecToString
Baez Joan 22 Musique
Turing Alan 22 Science Course
French Melinda 20 Gestion Microsoft
```

Le programmeur redéfinira **systématiquement** la méthode toString().

# Redéfinition de méthodes – facilité d'écriture

Le polymorphisme permet d'écrire très simplement des codes élégants.

Exemple :

- ▶ soit `tabEtu[]` un tableau d'objets `Etudiant`,
- ▶ puisqu'un objet `EtudiantSHN` ou `EtudiantALT` est aussi un objet `Etudiant`, ce tableau peut référencer aussi ces objets.

Pour obtenir un *listing* de toutes les informations, l'on écrira :

```
Etudiant tabEtu[] = {...};  
...  
for (Etudiant etu : tabEtu) {  
    System.out.println(etu);  
}
```

⇒ chaque objet appellera sa propre méthode `toString()`.

# Redéfinition de méthodes : les méthodes de classe

L'appel effectué dépendra :

- ▶ du nom de la classe préfixant la méthode :

```
1 class Bonjour {  
2     static void saluer() {  
3         System.out.println("Bonjour");  
4     }  
5 }  
6  
7 class Hello extends Bonjour {  
8     static void saluer() {  
9         System.out.println("Hello");  
10    }  
11 }  
12  
13 class ExempleRedefMethStat {  
14     public static void main(String args[]) {  
15         Bonjour.saluer();  
16         Hello.saluer();  
17     }  
18 }
```

```
[14:19][Prog pc666 :]$ java ExempleRedefMethStat  
Bonjour  
Hello
```

- ▶ du **type** de la variable préfixant la méthode :

```
19 class ExempleRedefMethStatVar {  
20     public static void main(String args[]) {  
21         Bonjour b = null;  
22         b.saluer();  
23         Hello h = null;  
24         h.saluer();  
25         b = new Hello();  
26         b.saluer();  
27         (new Hello()).saluer();  
28         ((Bonjour) h).saluer();  
29     }  
30 }
```

```
[14:27][Prog pc666 :]$ java ExempleRedefMethStatVar  
Bonjour  
Hello  
Bonjour  
Hello  
Bonjour
```

Plus de polymorphisme !



- 1 Surcharge – Ellipse
  - Surcharge de méthodes
  - Ellipse
- 2 Héritage
  - Définition
  - Héritage et construction d'objet
  - Héritage et type
- 3 Redéfinition
  - Redéfinition de méthode d'instance
  - Polymorphisme
  - La méthode `toString()`
  - Redéfinition de méthode de classe
- 4 Notion de paquetage
  - Définition
  - L'instruction `import`
  - Compilation

# Paquetages JAVA – définition

- ▶ Un paquetage est un ensemble de classes ou d'interfaces<sup>1</sup>.
- ▶ En JAVA, un paquetage joue le rôle de librairie.
- ▶ Pour mettre les classes d'un fichier dans un paquetage, on place sur la première ligne l'instruction :

```
package nompaquetage;
```

Un nom de paquetage :

- s'écrit par convention en minuscule;
- est une suite d'identificateurs séparés par des points;

```
package universite.modeleetudiant;
```

- renseigne sur la localisation de fichiers sources/*bytecodes*.

⇒ sources/*bytecodes* devront se trouver dans un répertoire :  
universite/modeleetudiant/

- ▶ Une classe n'appartient qu'à un seul paquetage.

<sup>1</sup>cf. prochain support.

# Paquetages JAVA – retour sur les modificateurs de visibilité

## Rappel sur les modificateurs de visibilité :

<code>public</code>	accessible depuis toute classe.
sans modificateur (visibilité paquetage)	accessible depuis toute classe du paquetage.
<code>protected</code>	idem paquetage & accessible depuis les classes « héritantes ».
<code>private</code>	accessible seulement depuis la classe hôte.

Le paquetage définit un niveau de visibilité « englobant » les précédents.

Une classe a alors deux niveaux de visibilité possibles :

```
package universite.modeleetudiant;

class Etudiant {
    ...
}
```

- ▶ visibilité paquetage (aucun modificateur).

```
package universite.modeleetudiant;

public class Etudiant {
    ...
} // Le fichier devra s'appeler Etudiant.java.
```

- ▶ visibilité totale (modificateur `public`).

# Paquetages JAVA – retour sur les modificateurs de visibilité

## Rappel sur les modificateurs de visibilité :

public	accessible depuis toute classe.
sans modificateur (visibilité paquetage)	accessible depuis toute classe du paquetage.
protected	idem paquetage & accessible depuis les classes « héritantes ».
private	accessible seulement depuis la classe hôte.

Le paquetage définit un niveau de visibilité « englobant » les précédents.

Une classe a alors deux niveaux de visibilité possibles :

```
package universite.modeleetudiant;

class Etudiant {
    ...
}
```

- ▶ visibilité paquetage (aucun modificateur).

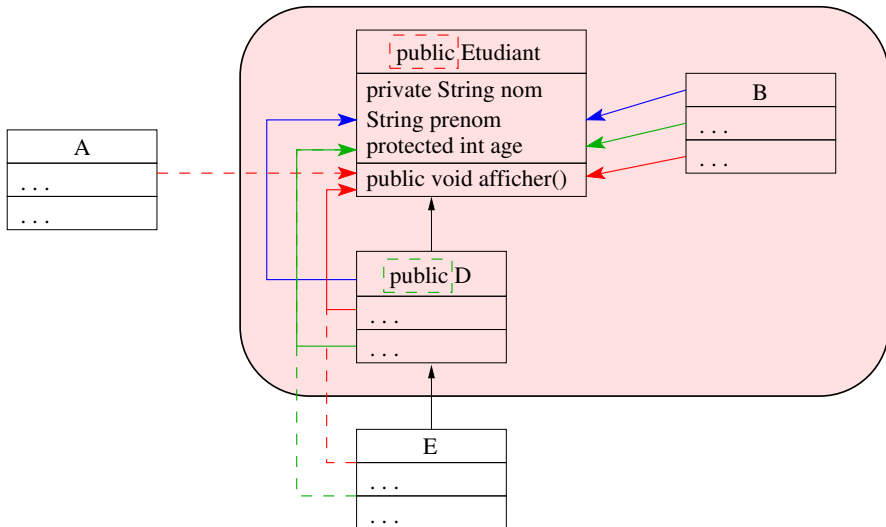
```
package universite.modeleetudiant;

public class Etudiant {
    ...
} // Le fichier devra s'appeler Etudiant.java.
```

- ▶ visibilité totale (modificateur public).

# Paquetage JAVA – synthèse des modificateurs de visibilité

paquetage



# Paquetage – interfacier avec l'instruction import 1/2

Il est nécessaire pour la JVM de bien identifier et localiser les *bytecodes*.  
En JAVA, on distingue :

▶ les nom courts de classes

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

▶ les noms complets de classes

```
java.util.Scanner clavier = new Scanner(System.in);
```

Utile pour distinguer deux classes de même nom issues de paquetages différents.

L'instruction `import` permet d'utiliser un nom court d'une classe.

```
import nompaquetage.NomClasse; // En début de fichier.
```

Pour importer toutes les classes d'un paquetage, utiliser l'opérateur `*` :

```
import nompaquetage.*; // OK.
```

```
import nompaquetage.*.*; // INTERDIT !
```

## Le paquetage `java.lang`

Il contient des classes basiques telles que `String` ou `System`.

⇒ JAVA dispense de mettre une instruction `import` !

# Paquetage – l'instruction `import static` 2/2

L'instruction `import` peut être utilisée avec le mot clé `static` :

► Syntaxe avec sélection :

```
import static paquetage.Classe.champOuMethode;
```

► Syntaxe avec métacaractère :

```
import static paquetage.Classe.*;
```

⇒ dispense de préfixer les champs ou méthodes statiques invoqués.

Exemple : reprise du code source CM1-transparent 65

```
class ExempleMathematique {  
    public static void main(String args[]) {  
        double valeurs[] = {Math.PI / 6,  
                             Math.PI / 4,  
                             Math.PI / 3};  
        for (double val : valeurs) {  
            System.out.printf("%.3f\n",  
                               Math.cos(val));  
        }  
    }  
}
```



# Paquetage – l'instruction `import static` 2/2

L'instruction `import` peut être utilisée avec le mot clé `static` :

► Syntaxe avec sélection :

```
import static paquetage.Classe.champOuMethode;
```

► Syntaxe avec métacaractère :

```
import static paquetage.Classe.*;
```

⇒ dispense de préfixer les champs ou méthodes statiques invoqués.

Exemple : reprise du code source CM1-transparent 65

► **Ligne 1** : permet d'utiliser directement le champ `out`

```
import static java.lang.System.out;  
import static java.lang.Math.*;  
  
class ExempleMathematique {  
    public static void main(String args[]) {  
        double valeurs[] = {PI / 6,  
                             PI / 4,  
                             PI / 3};  
        for (double val : valeurs) {  
            out.printf("%.3f\n",  
                        cos(val));  
        }  
    }  
}
```

► **Ligne 2** : permet d'utiliser directement

PI,  
cos().



# Paquetages – compiler les fichiers/exécuter un programme

Avec les paquetages, la compilation des fichiers devient plus délicate.

- ▶ Pour la compilation, il faut indiquer à javac où :
  - écrire les fichiers *bytecodes* (i.e. *.class*) avec l'option `-d` :

```
[20:30] [Prog pc666 :]$ javac -d chemin fichiersJava
```

**Attention** : l'option `-d` ne crée pas ce sous-répertoire `chemin`.

- chercher les fichiers *bytecodes* avec l'option `-cp` ou `-classpath` :

```
[20:30] [Prog pc666 :]$ javac -cp chemin1:chemin2:...:cheminN fichiersJava
```

Dans un système comme GNU-LINUX, il existe une variable d'environnement `CLASSPATH`. Avec l'interpréteur `bash`, on écrira :

```
[20:30] [Prog pc666 :]$ CLASSPATH=chemin1:chemin2:...:cheminN  
[20:30] [Prog pc666 :]$ javac fichiersJava
```


On pourra aussi utiliser la commande `export` au sein du fichier `.bashrc`.

- ▶ `javac` compilera toutes les dépendances nécessaires.
- ▶ Généralement, les fichiers *bytecodes* sont dans un répertoire et les fichiers sources dans un autre (généralement `src`).

# Paquetage – un exemple complet 1/2

Supposons que nous ayons un fichier `ExemplePackage.java` tel que :


```
class ExemplePackage {
    public static void main(String args[]) {
        Etudiant e1 = new Etudiant("Baez", "Joan", 22,
                                   "Musique");
        EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
                                         "Science", "Course");
        EtudiantALT e3 = new EtudiantALT("French", "Melinda", 20,
                                         "Gestion", "Microsoft");
        System.out.println(e1 + "\n" + e2 + "\n" + e3);
    }
}
```



- ▶ localisé dans un répertoire paquetage avec :
  - un sous-répertoire `cls` vide,
  - un sous-répertoire `src`.
- ▶ `cls` et `src` ont des sous-répertoires `universite/modeleetudiant`.
- ▶ des classes et des constructeurs `Etudiant`, `EtudiantsSHN`, et `EtudiantALT` **déclarés en public**.


```
public class Etudiant {
    String nom;
    String prenom;
    int age;
    String filiere;

    public Etudiant(String n, String p,
                    int a, String f) {
        nom = n;
        prenom = p;
        age = a;
        filiere = f;
    }
}
```




```
public class EtudiantSHN extends Etudiant {
    String sport;

    public EtudiantSHN(String n, String p,
                       int a, String f,
                       String s) {
        super(n, p, a, f);
        sport = s;
    }
}
```



```
public class EtudiantALT extends Etudiant {
    String entreprise;


    public EtudiantALT(String n, String p,
                       int a, String f,
                       String e) {
        super(n, p, a, f);
        entreprise = e;
    }
}
```



# Paquetage – un exemple complet 1/2

Supposons que nous ayons un fichier `ExemplePackage.java` tel que :

```
class ExemplePackage {
    public static void main(String args[]) {
        Etudiant e1 = new Etudiant("Baez", "Joan", 22,
                                   "Musique");
        EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
                                         "Science", "Course");
        EtudiantALT e3 = new EtudiantALT("French", "Melinda", 20,
                                         "Gestion", "Microsoft");
        System.out.println(e1 + "\n" + e2 + "\n" + e3);
    }
}
```




- ▶ localisé dans un répertoire paquetage avec :
  - un sous-répertoire `cls` vide,
  - un sous-répertoire `src`.
- ▶ `cls` et `src` ont des sous-répertoires `universite/modeleetudiant`.
- ▶ des classes et des constructeurs `Etudiant`, `EtudiantsSHN`, et `EtudiantALT` **déclarés en public**.

```
package universite.modeleetudiant;

public class Etudiant {
    String nom;
    String prenom;
    int age;
    String filiere;


    public Etudiant(String n, String p,
                    int a, String f) {
        nom = n;
        prenom = p;
        age = a;
        filiere = f;
    }
}
```



```
package universite.modeleetudiant;

public class EtudiantSHN extends Etudiant {
    String sport;


    public EtudiantSHN(String n, String p,
                       int a, String f,
                       String s) {
        super(n, p, a, f);
        sport = s;
    }
}
```



```
package universite.modeleetudiant;

public class EtudiantALT extends Etudiant {
    String entreprise;

    public EtudiantALT(String n, String p,
                       int a, String f,
                       String e) {
        super(n, p, a, f);
        entreprise = e;
    }
}
```



# Paquetage – un exemple complet 1/2

Supposons que nous ayons un fichier `ExemplePackage.java` tel que :

```
import universite.modeleetudiant.*;

class ExemplePackage {
    public static void main(String args[]) {
        Etudiant e1 = new Etudiant("Baez", "Joan", 22,
                                   "Musique");
        EtudiantSHN e2 = new EtudiantSHN("Turing", "Alan", 22,
                                         "Science", "Course");
        EtudiantALT e3 = new EtudiantALT("French", "Melinda", 20,
                                         "Gestion", "Microsoft");
        System.out.println(e1 + "\n" + e2 + "\n" + e3);
    }
}
```

- ▶ localisé dans un répertoire paquetage avec :
  - un sous-répertoire `cls` vide,
  - un sous-répertoire `src`.
- ▶ `cls` et `src` ont des sous-répertoires `universite/modeleetudiant`.
- ▶ des classes et des constructeurs `Etudiant`, `EtudiantsSHN`, et `EtudiantALT` **déclarés en public**.

```
package universite.modeleetudiant;

public class Etudiant {
    String nom;
    String prenom;
    int age;
    String filiere;

    public Etudiant(String n, String p,
                    int a, String f) {
        nom = n;
        prenom = p;
        age = a;
        filiere = f;
    }
}
```

```
package universite.modeleetudiant;

public class EtudiantSHN extends Etudiant {
    String sport;

    public EtudiantSHN(String n, String p,
                       int a, String f,
                       String s) {
        super(n, p, a, f);
        sport = s;
    }
}
```

```
package universite.modeleetudiant;

public class EtudiantALT extends Etudiant {
    String entreprise;

    public EtudiantALT(String n, String p,
                       int a, String f,
                       String e) {
        super(n, p, a, f);
        entreprise = e;
    }
}
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires `cls` et `src` :

```
[18:28][paquetage pc666 :]$ ls  
cls  ExemplePackage.java  src  
[18:28][paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires `cls` et `src` :

```
[18:28][paquetage pc666 :]$ ls  
cls  ExemplePackage.java  src  
[18:28][paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
```



# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28][paquetage pc666 :]$ ls  
cls  ExemplePackage.java  src  
[18:28][paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/  
total 0  
[18:28][paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires `cls` et `src` :

```
[18:28][paquetage pc666 :]$ ls  
cls  ExemplePackage.java  src  
[18:28][paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/  
total 0  
[18:28][paquetage pc666 :]$ ls src/universite/modeleetudiant/
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28][paquetage pc666 :]$ ls  
cls  ExemplePackage.java  src  
[18:28][paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/  
total 0  
[18:28][paquetage pc666 :]$ ls src/universite/modeleetudiant/  
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls  
cls  ExemplePackage.java  src  
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/  
total 0  
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/  
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls  
cls ExemplePackage.java src  
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/  
total 0  
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/  
EtudiantALT.java Etudiant.java EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java  
[18:28] [paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls  
cls ExemplePackage.java src  
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/  
total 0  
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/  
EtudiantALT.java Etudiant.java EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java  
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires `cls` et `src` :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```



# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
[18:29] [paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
[18:29] [paquetage pc666 :]$ ls
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
[18:29] [paquetage pc666 :]$ ls
cls  ExemplePackage.class  ExemplePackage.java  src
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
[18:29] [paquetage pc666 :]$ ls
cls  ExemplePackage.class  ExemplePackage.java  src
```

Exécution du programme :

```
[18:29] [paquetage pc666 :]$
```

# Paquetage – un exemple complet 2/2

État du répertoire courant paquetage et des sous-répertoires cls et src :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier ExemplePackage.java :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
[18:29] [paquetage pc666 :]$ ls
cls  ExemplePackage.class  ExemplePackage.java  src
```

Exécution du programme :

```
[18:29] [paquetage pc666 :]$ java -cp cls:. ExemplePackage
```

# Paquetage – un exemple complet 2/2

État du répertoire courant `paquetage` et des sous-répertoires `cls` et `src` :

```
[18:28] [paquetage pc666 :]$ ls
cls  ExemplePackage.java  src
[18:28] [paquetage pc666 :]$ ls -l cls/universite/modeleetudiant/
total 0
[18:28] [paquetage pc666 :]$ ls src/universite/modeleetudiant/
EtudiantALT.java  Etudiant.java  EtudiantSHN.java
```

Compilation du paquetage :

```
[18:28] [paquetage pc666 :]$ javac -d cls/ src/universite/modeleetudiant/*.java
[18:28] [paquetage pc666 :]$ ls cls/universite/modeleetudiant/
EtudiantALT.class  Etudiant.class  EtudiantSHN.class
```

Compilation du fichier `ExemplePackage.java` :

```
[18:28] [paquetage pc666 :]$ javac -cp cls ExemplePackage.java
[18:29] [paquetage pc666 :]$ ls
cls  ExemplePackage.class  ExemplePackage.java  src
```

Exécution du programme :

```
[18:29] [paquetage pc666 :]$ java -cp cls:. ExemplePackage
universite.modeleetudiant.Etudiant@15db9742
universite.modeleetudiant.EtudiantSHN@6d06d69c
universite.modeleetudiant.EtudiantALT@7852e922
```