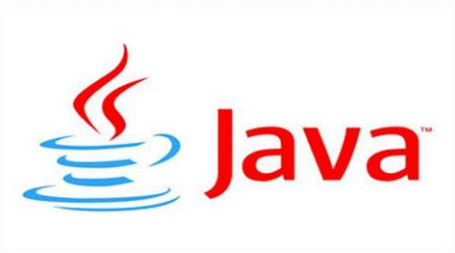
Faculté des Sciences
كلية العلوم
جامعة ابن طفيل Université Ibn Tofaïl

Département d'Informatique
Filière SMI - Semestre 6
2020-2021

Programmation Orientée Objet

Application avec Java



Pr. Khadija Louzaoui

Chapitre 2

Classes et Objets

chapitre 2

- Définition d'une classe
- Création d'instances
- Notion de référence
- Méthode : toString()
- Getters et Setters
- Constructeur
- Méthodes : equals() et hashCode()
- Membres de la classe
- Membres statiques de la classe
- Associations entre classes
- Surcharge d'une méthode
- Encapsulation

Rappel

Qu'est ce qu'une classe en POO?

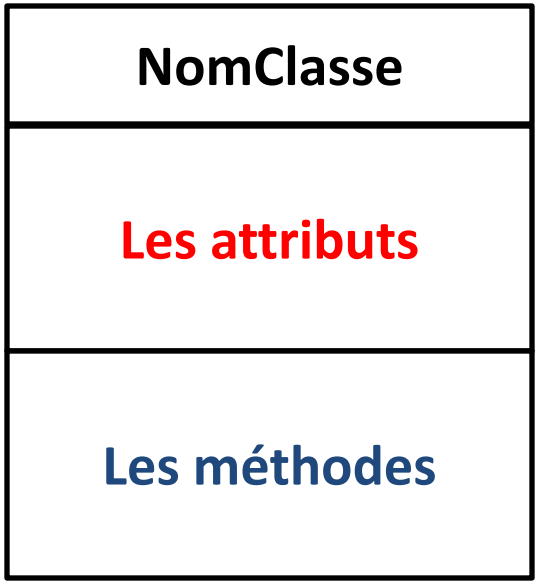
- Ça correspond à un plan, un moule, un modèle...
- C'est une description abstraite d'un type d'objets.
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (**attributs**) et dynamiques (**méthodes**).

Qu'est ce que c'est la notion d'instance?

- Une instance correspond à un objet créé à partir d'une classe (via le **constructeur**).
- L'instanciation : création d'un objet d'une classe
- **instance ≡ objet**

Définition d'une classe

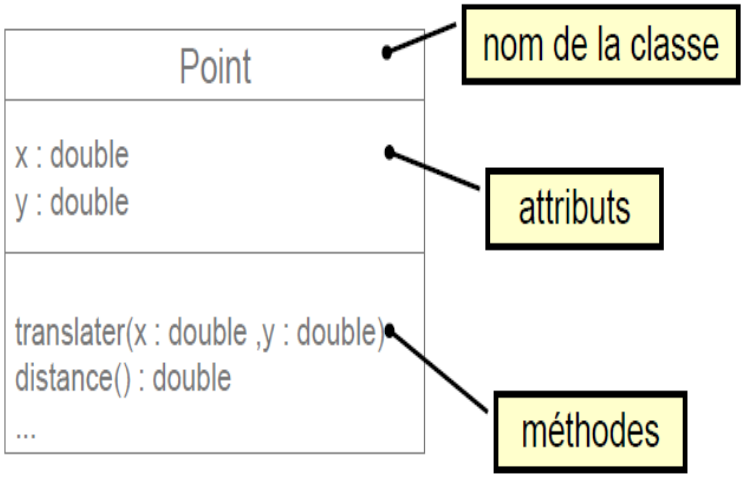
De quoi est composé une classe?



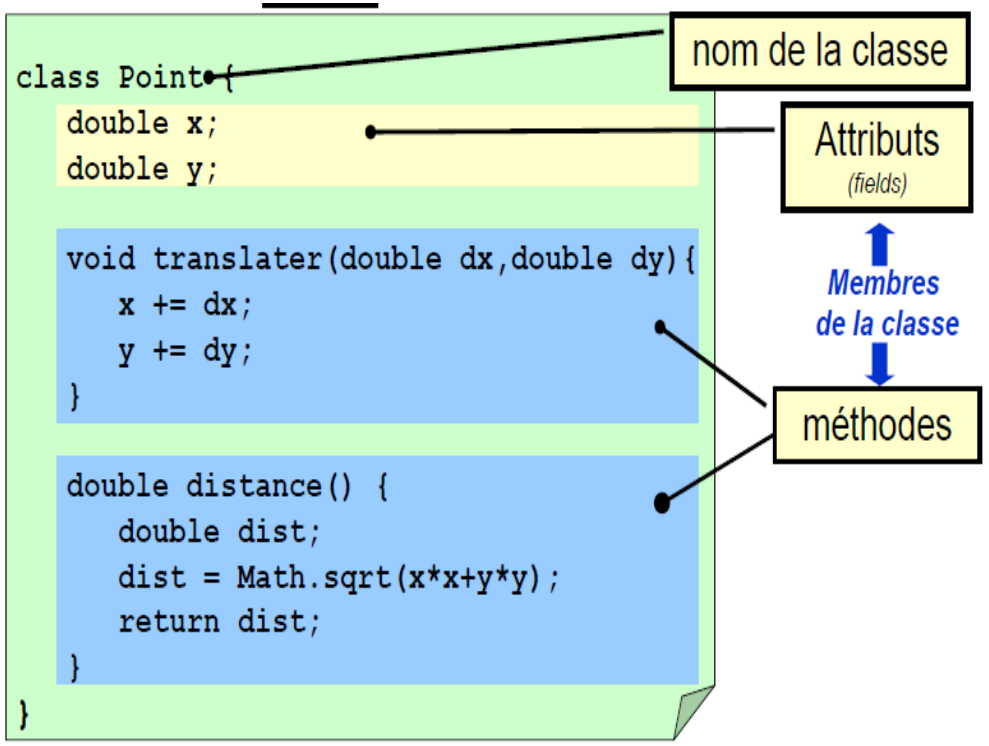
Attribut : [visibilité] + type + nom
Méthode : [visibilité] + valeur de retour + nom + arguments ≡ **signature**
(exactement comme les fonctions en procédurale)

Définition d'une classe

Notation UML



Syntaxe Java



Définition d'une classe

Eléments d'une classe

- Les constructeurs (il peut y en avoir plusieurs) servent à créer les instances (les objets) de la classe.
- Quand une instance est créée, son état est conservé dans les variables d'instance (attributs).
- 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

Définition d'une classe

Exemple : classe Livre

```
public class Livre {
  private String titre, auteur;
  private int nbPages;
  // Constructeur
  public Livre(String unTitre, String unAuteur) {
    titre = unTitre;
    auteur = unAuteur;
  }
  public String getAuteur() {       // accesseur
    return auteur;
  }
  public void setNbPages(int nb) { // modificateur
    nbPages = nb;
  }
}
```

Variables d'instance

Constructeurs

Méthodes

Définition d'une classe

Rôles d'une classe

- Une classe est
  - un type qui décrit une structure (variables d'instances) et un comportement (méthodes)
  - un module pour décomposer une application en entités plus petites
  - un générateur d'objets (par ses constructeurs)
- Une classe permet d'encapsuler les objets : les membres **public** sont vus de l'extérieur mais les membres **private** sont cachés

Définition d'une classe

Particularité du Java

- Toutes les classes héritent implicitement (pas besoin d'ajouter extends) d'une classe mère **Object**.

- La classe Object contient plusieurs méthodes telles que **toString()** (pour transformer un objet en chaîne de caractère),**clone()** (pour cloner)...

- Le mot-clé **this** permet de désigner l'objet courant.

- Contrairement à certains LOO, le **this** n'est pas obligatoire si aucune ambiguité ne se présente.

Définition d'une classe

Exemple d'application

- Commençons par créer un nouveau projet Java
- Créons deux packages

  org.exemple.test et org. exemple.model
- Créons deux classes
  - Une classe Personne dans org.exemple.model contenant trois attributs : num, nom et prénom.
  - Une classe Main dans org.exemple.test contenant le public static void main dans lequel on instanciera la classe Personne.

Contenu
de la classe Personne

```
package org.exemple.model;

public class Personne {
    int num;
    String prenom;
    String nom;
}
```

Définition d'une classe

Exemple d'application

Remarques

- En Java, toute classe a un **constructeur par défaut** sans paramètres.
- Par défaut, la visibilité des attributs, en Java, est **package**.
- Donc, les attributs ne seront pas accessibles depuis la classe Main qui se situe dans un package différent (org.exemple.test)
- Donc, changeons la visibilité des trois attributs de la classe Personne

Définition d'une classe

Exemple d'application

Nouveau contenu de la classe Personne

```
package org.exemple.model;

public class Personne{
    public int num;
    public String prenom;
    public String nom;
}
```

Contenu de la classe Main

```
package org.exemple.test;

public class Main {

    public static void main(String[] args) {
    // TODO Auto-generated method stub

    }

}
```

Création d'instances

- Une **instance**, ou un **objet**, d'une classe est créée en 3 temps

```
Pixel p1 = new Pixel(1,3);
```

  - **new** est l'opérateur d'instanciation: comme il est suivi du nom de la classe, il sait quelle classe doit être instanciée
    - Il initialise chaque champ à sa valeur par défaut
    - Il peut exécuter un bloc d'initialisation éventuel
    - Il retournera la référence de l'objet ainsi créé
  - **Pixel** est le nom de la classe
  - **(1,3)** permet de trouver une fonction d'initialisation particulière dans la classe, qu'on appelle un **constructeur**

Création d'instances

- Une instance d'une classe est créée par un des **constructeurs** de la classe.
- Une fois qu'elle est créée, l'instance
  - a son propre état interne (les valeurs des variables).
  - partage le code qui détermine son comportement (les méthodes) avec les autres instances de la classe.

Création d'instances

Exemple d'application

Si on voulait créer un objet de la classe Personne avec les valeurs 1, Mohammed et Amarni

Etape 1 : déclaration d'un objet (**objet non créé**)

```
Personne personne;
```

Etape 2 : instanciation de la classe Personne (**objet créé**)

```
personne = new Personne();
```

On peut faire déclaration + instanciation

```
Personne personne = new Personne();
```

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "Amrani";
personne.prenom = "Mohammed" ;
```

Création d'instances

Exemple d'application

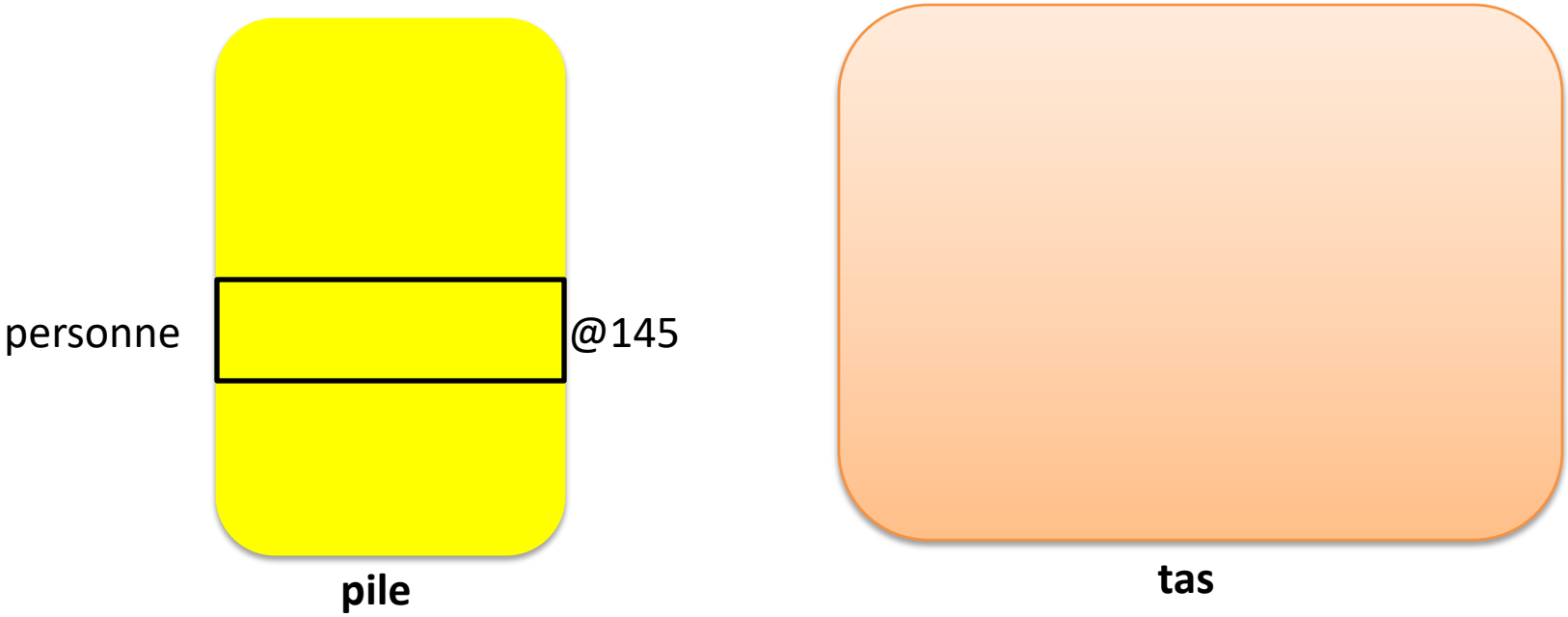
Contenu de la classe Main

```
package org.exemple.test;
import org.exemple.model.*;

public class Main {

    public static void main(String[] args) {
        Personne personne=new Personne();
        personne.num=1;
        personne.nom="Amrani";
        personne.prenom="Mohammed";
    }
}
```

Notion de référence

Création d'une variable personne de type Personne

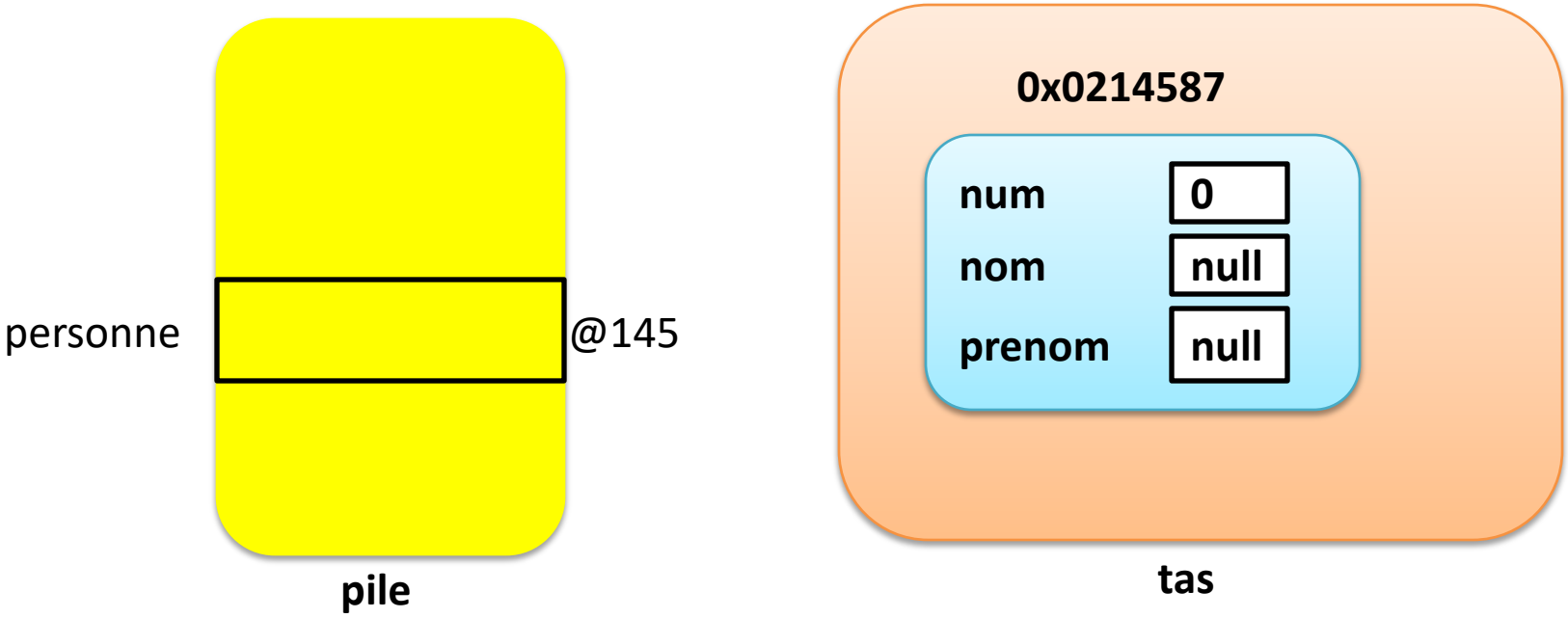
```
public class Main {
    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.num=1;
        personne.nom="Amrani";
        personne.prenom="Mohammed";
    }
}
```



Notion de référence

Instanciation de la classe Personne → création d'objet

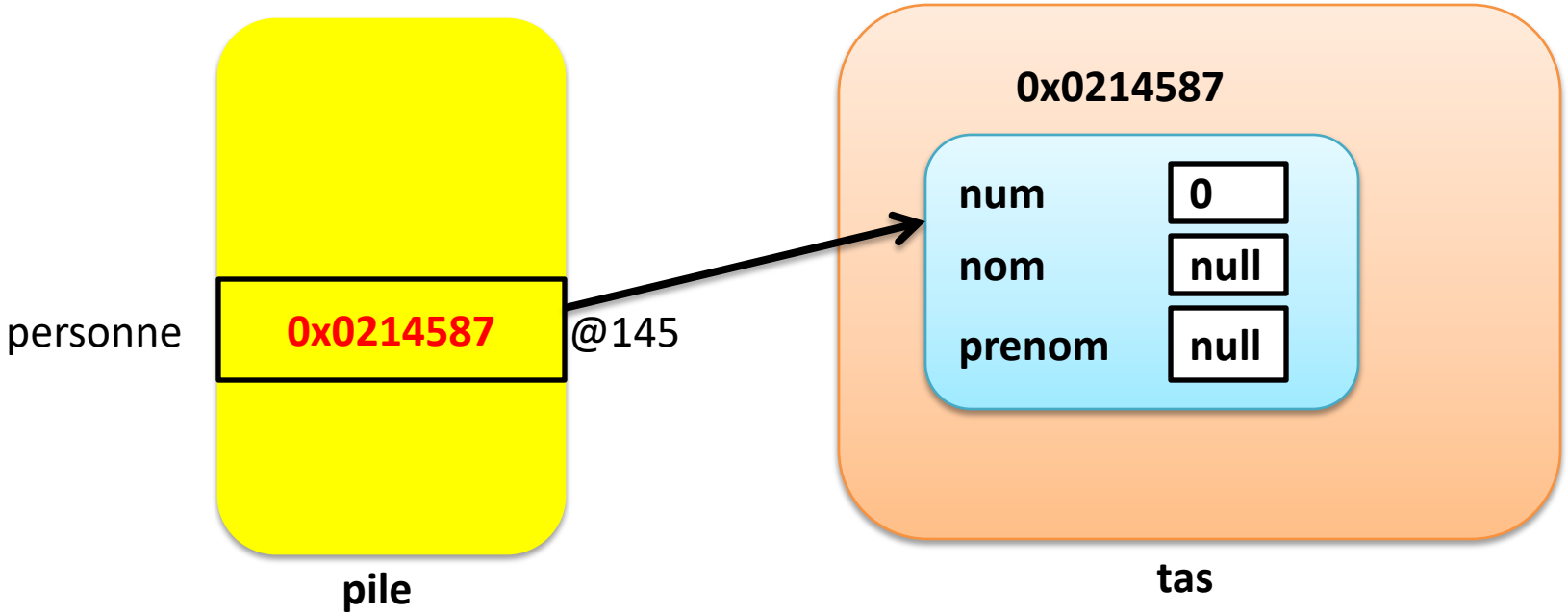
```
public class Main {
    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.num=1;
        personne.nom="Amrani";
        personne.prenom="Mohammed";
    }
}
```



Notion de référence

Lier l'objet créé à la variable *personne* → *personne* est la **référence** de l'objet crée

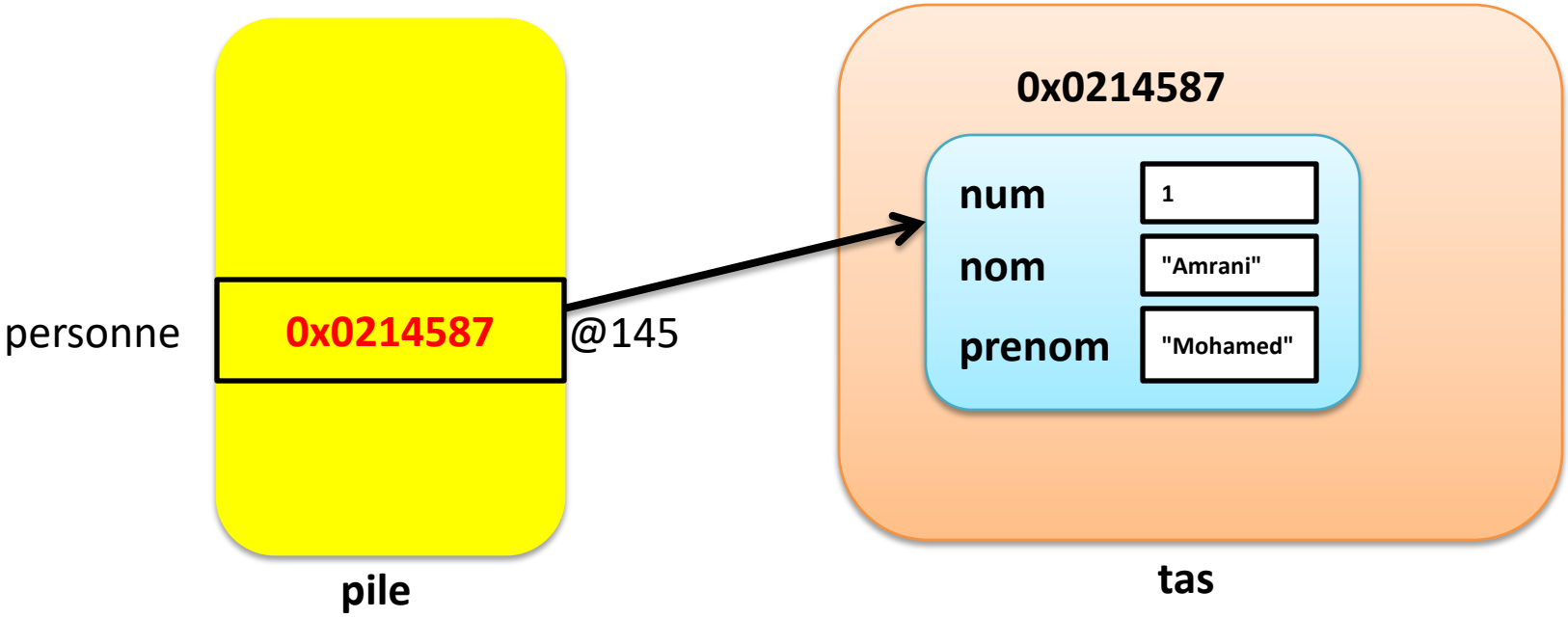
```
public class Main {
    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.num=1;
        personne.nom="Amrani";
        personne.prenom="Mohammed";
    }
}
```



Notion de référence

En utilisant la référence *personne* on peut accéder aux attributs de l'objet

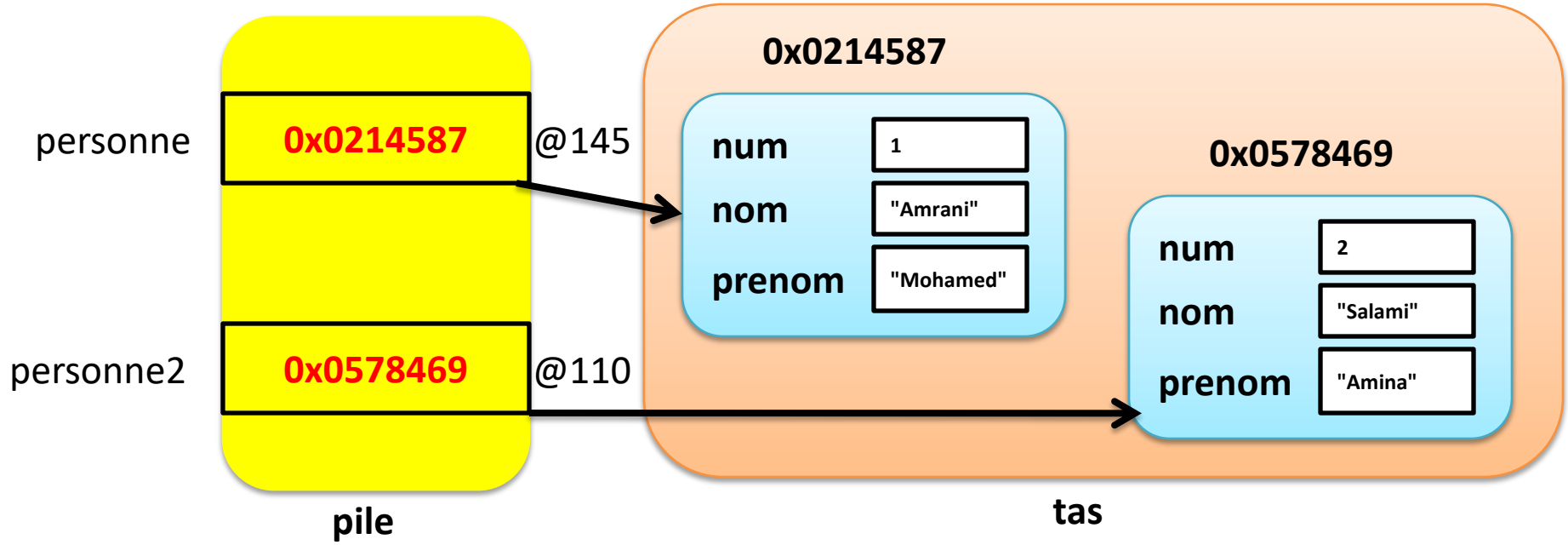
```
public class Main {
    public static void main(String[] args) {
        Personne personne = new Personne();
        personne.num=1;
        personne.nom="Amrani";
        personne.prenom="Mohammed";
    }
}
```



Notion de référence

Lier l'objet créé à la variable *personne2* → *personne2* est la **référence** de l'objet crée

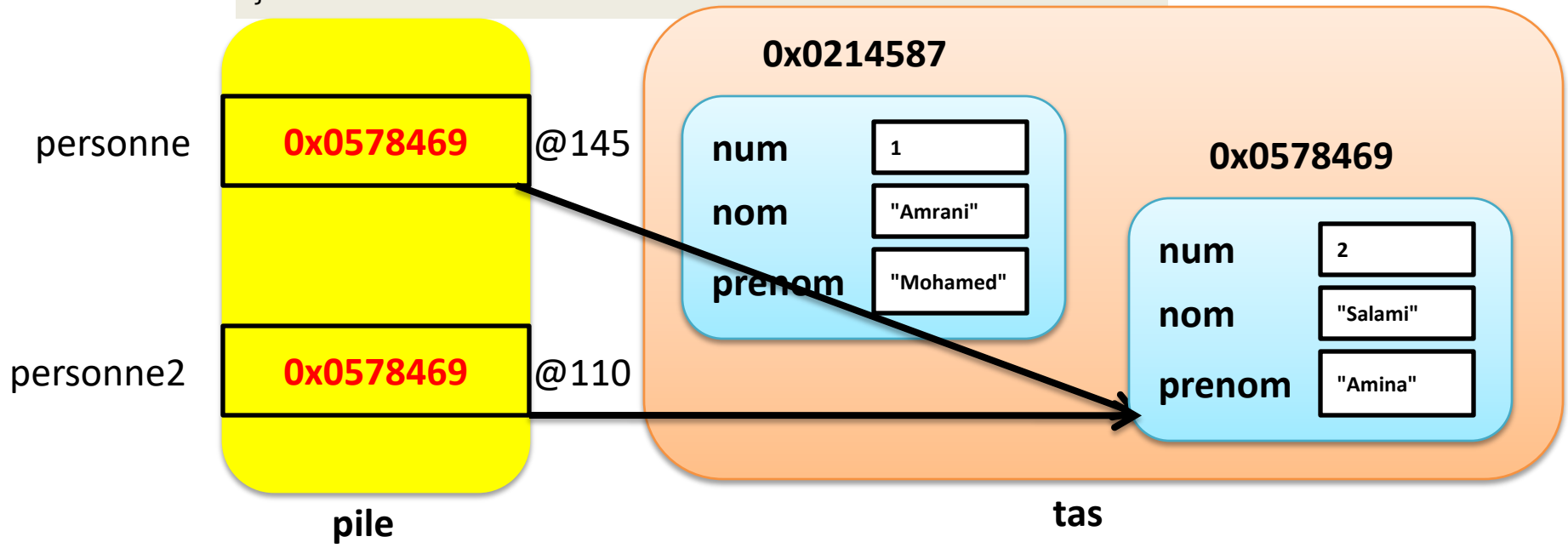
```
public class Main {
      public static void main(String[] args) {
            Personne personne = new Personne();
            personne.num=1;
            personne.nom="Amrani";
            personne.prenom="Mohammed";
            Personne personne2 = new Personne();
            personne.num=2;
            personne.nom="Salami";
            personne.prenom="Amina";
      }
}
```



Notion de référence

personne et *personne2* contiennent la même valeur d'adresse de l'objet

```
public class Main {
      public static void main(String[] args) {
            Personne personne = new Personne();
            personne.num=1;
            personne.nom="Amrani";
            personne.prenom="Mohammed";
            Personne personne2 = new Personne();
            personne.num=2;
            personne.nom="Salami";
            personne.prenom="Amina";
            Personne = personne2
      }
}
```



toString()

Exemple d'application

Pour être sûr que les valeurs ont bien été affectées aux attributs, on affiche

```
System.out.println(personne);
```

Contenu de la classe Main

```
package org.exemple.test;
import org.exemple.model.*;

public class Main {

    public static void main(String[] args) {
        Personne personne=new Personne();
        personne.num=1;
        personne.nom="Mohammed";
        personne.prenom="Amrani";
        System.out.println(personne);
    }
}
```

En exécutant, le résultat est :

org.exemple.model.Personne@7a81197d

toString()

Pour afficher les détails d'un objet, il faut que la méthode toString() soit implémentée

- Il est conseillé d'inclure une méthode **toString** dans toutes les classes que l'on écrit.
- Cette méthode renvoie une chaîne de caractères qui décrit l'instance.
- Une description compacte et précise peut être très utile lors de la mise au point des programmes.
- **System.out.println(*objet*)** affiche la valeur retournée par ***objet*.toString()**.

toString()

Exemple d'application

Le code de la méthode toString()

```
@Override
public String toString() {
        return "Peronne [num="+num+",nom="+nom+",prenom="+prenom+"]";
}
```

@Override

- Une annotation (appelé aussi décorateur par MicroSoft)
- Pour nous rappeler qu'on redéfinit une méthode qui appartient à la classe mère (ici Object)

En exécutant, le résultat est :

```
Peronne [num=1,nom=Mohammed,prenom=Amrani]
```

Setter et Getter

- Deux types de méthodes servent à donner accès aux variables depuis l'extérieur de la classe :
  - les accesseurs en lecture pour **lire** les valeurs des variables ; « accesseur en lecture » est souvent abrégé en « accesseur » ; ***getter*** en anglais
  - les accesseurs en écriture, ou modificateurs, ou mutateurs, pour modifier leur valeur ; ***setter*** en anglais

Setter

Exemple d'application

Supposant que l'on n'accepte pas de valeur négative pour l'attribut num de la classe Personne

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à **private**)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (**les setter**)

Convention

- Mettre la visibilité **private** ou **protected** pour tous les attributs
- Mettre la visibilité **public** pour toutes les méthodes

Setter

Exemple d'application

Mettons la visibilité **private** pour tous les attributs de la classe Personne

```
package org.exemple.model;

public class Personne {
    private int num;
    private String nom;
    private String prenom;

@Override
public String toString() {
    return "Peronne [num="+num+", nom="+nom+", prenom="+prenom+"]";
        }
}
```

Dans la classe Main, les trois lignes suivantes sont soulignées en rouge

```
5      public static void main(String[] args) {
6          Personne personne=new Personne();
7          personne.num=1;
8          personne.nom="Mohammed";
9          personne.prenom="Amrani";
10         System.out.println(personne);
11
12     }
```

Setter

Les attributs sont privés, donc aucun accès direct ne sera autorisé

- Solution : Générer les setters
  - Faire clic droit sur la classe Personne
  - Aller dans Source > Generate Getters and Setters...
  - Cliquer sur chaque attribut et cocher la case setNomAttribut(...)
  - Valider

Setter

Exemple d'application

Les trois méthodes générées

```
public void setNum(int num) {
        this.num = num;
}
public void setNom(String nom) {
        this.nom = nom;
}
public void setPrenom(String prenom) {
        this.prenom = prenom;
}
```

Modifions **setNum()** pour ne plus accepter de valeurs inférieures à 1

```
public void setNum(int num) {
    if (num>=1) {
        this.num = num;
    }
}
```

Setter

Exemple d'application

Mettons à jour la classe Main

```
package org.exemple.test;
import org.exemple.model.*;

public class Main {

public static void main(String[] args) {
    Personne personne=new Personne();
    personne.setNum(1);
    personne.setNom("Mohammed");
    personne.setPrenom("Amrani");
    System.out.println(personne);

   }
}
```

En exécutant, le résultat est :

```
Peronne [num=1,nom=Mohammed,prenom=Amrani]
```

Setter

Exemple d'application

Mettons à jour la classe Main

```
package org.exemple.test;
import org.exemple.model.*;

public class Main {

public static void main(String[] args) {
    Personne personne=new Personne();
    personne.setNum(-1);
    personne.setNom("Mohammed");
    personne.setPrenom("Amrani");
    System.out.println(personne);

   }
}
```

En exécutant, le résultat est :

```
Peronne [num=0,nom=Mohammed,prenom=Amrani]
```

Getter

Exemple d'application

Si on voulait afficher les attributs (privés) de la classe **Personne**, un par un, dans la classe **Main** sans passer par le **toString()**

Démarche

1. Définir des méthodes qui retournent les valeurs des attributs (**les getter**) .

- Solution : Générer les getters

  - Faire clic droit sur la classe Personne
  - Aller dans Source > Generate Getters and Setters...
  - Cliquer sur chaque attribut et cocher la case getNomAttribut()
  - Valider

Getter

Exemple d'application

Les trois méthodes générées

```
public int getNum() {
          return num;
}
public String getNom() {
          return nom;
}
public String getPrenom() {
          return prenom;
}
```

Mettons à jour la classe Main

```
package org.exemple.test;
import org.exemple.model.*;
public class Main {
public static void main(String[] args) {
     Personne personne=new Personne();
     personne.setNum(-1);
     personne.setNom("Amrani");
     personne.setPrenom("Mohammed");
     System.out.println(personne.getNum()+" "+ personne.getNom()+" "+
     personne.getPrenom());
   }
}
```

En exécutant, le résultat est :

```
1 Amrani Mohammed
```

Constructeur

- Constructeurs d'une classe :
  - méthodes particulières pour la création d'objets de cette classe
  - méthodes dont le nom est identique au nom de la classe

- rôle d'un constructeur
  - effectuer certaines initialisations nécessaires pour le nouvel objet créé une fois que la mémoire est réservée par **new**
  - n'a pas de type de retour (même void)

- toute classe JAVA possède au moins un constructeur
  - si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoqué

Constructeur

- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe

- Pour générer un constructeur avec paramètre
  1. Faire clic droit sur la classe Personne
  2. Aller dans Source > Generate Constructor using Fields...
  3. Vérifier que toutes les cases sont cochées
  4. valider

Constructeur

Exemple d'application

Le constructeur généré

```
public Personne(int num, String nom, String prenom) {
this.num = num;
this.nom = nom;
this.prenom = prenom;
}
```

Pour préserver la cohérence, il faut que le constructeur contrôle la valeur de l'attribut num

```
public Personne(int num, String nom, String prenom) {
if (num>=1) {
this.num = num;
}
this.nom = nom;
this.prenom = prenom;
}
```

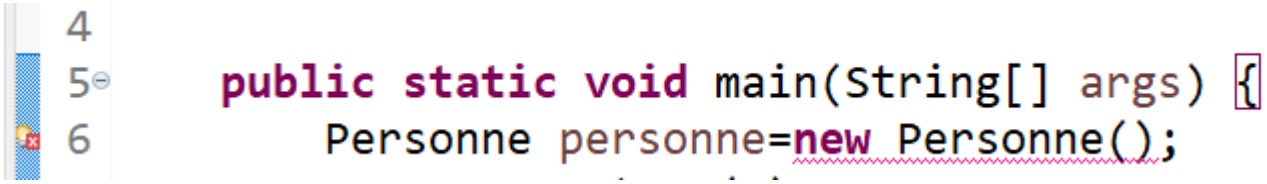
On peut aussi appelé le setter dans le constructeur

```
public Personne(int num, String nom, String prenom) {
this.setNum(num);
this.nom = nom;
this.prenom = prenom;
}
```

Constructeur

Exemple d'application

Dans la classe Main, la ligne suivante est soulignée en rouge



```
4
5⊖    public static void main(String[] args) {
6         Personne personne=new Personne();
```

Explication

- Le constructeur par défaut **a été écrasé** (il n'existe plus)

Solution

- Générer un constructeur sans paramètre

Constructeur

Plusieurs constructeurs (surcharge)

- Plusieurs constructeurs peuvent cohabiter dans la même classe.
- Ils ont typiquement des rôles différents et offrent des « services» complémentaires à l'utilisateur.
- La plateforme Java différencie entre les différents constructeurs déclarés au sein d'une même classe en se basant sur le nombre des paramètres et leurs types.

On ne peut pas créer deux constructeurs ayant le même nombre et types de paramètres

```
Personne (int num){
   this.num=num;
}
Personne (int num){
   this.num=num;
}
```

Erreur de compilation

Constructeur

Exemple d'application

Quel constructeur va choisir le compilateur lors de la création de l'objet?

```
package org.exemple.test;
import org.exemple.model.*;
public class Main {

public static void main(String[] args) {
    Personne personne=new Personne();
    personne.setNum(1);
    personne.setNom("Amrani");
    personne.setPrenom("Mohammed");
    System.out.println(personne.getNum()+" "+ personne.getNom()+" " +
    personne.getPrenom());
    Personne personne2=new Personne(2, "Salami", "Amina");
    System.out.println(personne2);
}

}
```

En exécutant, le résultat est :

```
1 Amrani Mohammed
Peronne [num=2, nom=Salami, prenom=Amina]
```

Constructeur

L'objet « courant » : this

- Chaque objet a accès à une référence à lui même
  - **la référence this**
- Dans le corps d'une méthode, le mot clé **this** désigne l'instance sur laquelle est invoquée la méthode. Ce mot clé est utilisé dans 3 circonstances :

1. pour accéder aux attributs de l'objets
2. pour comparer la référence de l'objet invoquant la méthode à une autre référence
3. pour passer la réference de l'objets invoquant la méthode en paramètre d'une autre méthode

Constructeur

L'objet « courant » : this

Implicitement quand dans le corps d'une méthode un attribut est utilisé, c'est un attribut de l'objet courant

this essentiellement utilisé pour lever les ambiguités

```
class Point {
    double x;
    double y;
    void translater(int dx, int dy) {
        x += dx; y += dy;
    }
        <==>  this.x += dx; this.y += dy;

    double distance() {
        return Math.sqrt(x*x+y*y);
    }
    void placerAuPoint(double x ,double y1){
        this.x = x;
        y = y1;
    }
}
```

Constructeur

L'objet « courant » : this

Un constructeur peut appeler d'autres constructeurs

Quand c'est possible, il est préférable qu'il y en ait un « le plus général » et que les autres y fassent appel

- –Plutôt que de dupliquer le code dans plusieurs constrcuteurs
- –L'appel à un autre constructeur de la même classe se fait par this(...)
- – Position de this(…) : première instruction

```
public class Point {
   private double x;
   private double y;

   // constructeurs
   private Point(double x, double y) {

      this.x = x;
      this.y = y;
   }

   public Point(Point p) {

      this(p.x,p.y);
   }

   public Point() {

      this(0.0,0.0);
   }

   ...
}
```

Intérêt :

- Factorisation du code.
- Un constructeur général invoqué par des constructeurs particuliers.

Possibilité de définir des constructeurs privés

Constructeur

Exemple d'application

Appel à un constructeur de la classe Personne par this(…)

```
public Personne(int num, String nom, String prenom) {
        this.setNum(num);
        this.nom = nom;
        this.prenom = prenom;
}
public Personne() {
        this(0,null,null);
}
public Personne (Personne p) {
        this(p.num,p.nom,p.prenom);
}
```

Mettons à jour la classe Main

```
public class Main {

public static void main(String[] args) {
     Personne personne1=new Personne(1,"Amrani","Mohammed");
     Personne personne2=new Personne();
     Personne personne3=new Personne(personne1);
     System.out.println(personne1);
     System.out.println(personne2);
     System.out.println(personne3);
    }
}
```

Constructeur

Exemple d'application

Appel à un constructeur de la classe Personne par this(…)

```
public Personne(int num, String nom, String prenom) {
          this.setNum(num);
          this.nom = nom;
          this.prenom = prenom;
}
public Personne() {
          this(0,null,null);
}
public Personne (Personne p) {
          this(p.num,p.nom,p.prenom);
}
```

Mettons à jour la classe Main

```
public class Main {

public static void main(String[] args) {
     Personne personne1=new Personne(1,"Amrani","Mohammed");
     Personne personne2=new Personne();
     Personne personne3=new Personne(personne1);
     System.out.println(personne1);
     System.out.println(personne2);
     System.out.println(personne3);
    }
}
```

equals() et hashCode()

Exemple d'application

En testant l'égalité entre les deux objets *personne1* et *personne3*

```
public class Main {

public static void main(String[] args) {
    Personne personne1=new
    Personne(1,"Amrani","Mohammed");
    Personne personne2=new Personne();
    Personne personne3=new Personne(personne1);
    System.out.println(personne1);
    System.out.println(personne2);
    System.out.println(personne3);
    System.out.println(personne3.equals(personne1));
   }
}
```

En exécutant, le résultat est :

```
Peronne [num=1, nom=Amrani, prenom=Mohammed]
Peronne [num=0, nom=null, prenom=null]
Peronne [num=1, nom=Amrani, prenom=Mohammed]
false
```

equals() et hashCode()

- Par défaut, deux objets d'une même classe sont **égaux** s'ils pointent sur le même espace mémoire (identiques).
- Pour **comparer** les valeurs, il faut redéfinir les méthodes equals() et hashCode().
- equals() permet de définir comment tester sémantiquement l'égalité de deux objets.
- hashCode() permet de retourner la valeur de hachage d'un objet (elle ne retourne pas un identifiant unique).
- Si une classe redéfinit equals(), alors elle doit aussi redéfinir hashCode().
- equals() et hashCode() doivent utiliser les mêmes attributs.

equals() et hashCode()

equals() et hashCode() doivent respecter les contraintes suivantes

- **Symétrie** : si a.equals(b) retourne true alors b.equals(a) aussi.
- **Réflexivité** : a.equals(a) retourne true si a est un objet non null.
- **Transitivité** : si a.equals(b) et b.equals(c) alors a.equals(c).
- **Consistance** : si a.equals(b) alors a.hashCode() == b.hashCode().
- a.equals(null) retourne false.

Des objets égaux au sens de **equals** doivent avoir le même **hashcode**

Redéfinition de equals() implique en général redéfinition de hashCode()

Code par défaut de equals() dans Object

```
public boolean equals(Object obj) {
  if (this == obj)
return true;
}
```

equals() et hashCode()

Le Code généré

```
@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        Personne other = (Personne) obj;
        if (nom == null) {
            if (other.nom != null)
                return false;
        }
        else if (!nom.equals(other.nom))
                return false;
        if (num != other.num)
                return false;
        if (prenom == null) {
             if (other.prenom != null)
                return false;
        }else if (!prenom.equals(other.prenom))
                return false;
        return true;
}
```

Exemple d'application

```
@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
result = prime * result +
((nom == null) ? 0 :
nom.hashCode());
result = prime * result + num;
result = prime * result +
((prenom == null) ? 0 :
prenom.hashCode());
return result;
}
```

equals() et hashCode()

Exemple d'application

En retestant le code précédent

```
public class Main {

public static void main(String[] args) {
    Personne personne1=new
    Personne(1,"Amrani","Mohammed");
    Personne personne2=new Personne();
    Personne personne3=new Personne(personne1);
    System.out.println(personne1);
    System.out.println(personne2);
    System.out.println(personne3);
    System.out.println(personne3.equals(personne1));
   }
}
```

En exécutant, le résultat est :

```
Peronne [num=1, nom=Amrani, prenom=Mohammed]
Peronne [num=0, nom=null, prenom=null]
Peronne [num=1, nom=Amrani, prenom=Mohammed]
true
```

Membres de la classe

Attributs:

- Variables qui sont utilisées pour caractériser l'instance d'un objet créée par un constructeur
- Déclarées dans une classe, mais en dehors des méthodes
- Appelées aussi membres ou variables de champ
- Créées quand un objet est créé et détruites quand l'objet est détruit

Méthodes:

- Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message
- Déclarées dans une classe
- Sont définies par
  - le nom
  - les paramètres avec leur types (optionnel)
  - le type de retour
  - un qualificateur (public, protected, private)
  - et le corps de la méthode (instructions)
- Utilisation

  référence.méthode(arguments)

```
personne.setNum(2);
```

Membres statiques de la classe

Déclarés avec le mot-clé **static**, les membres statiques sont liés à la classe et non à une instance particulière (un objet)

- **Attributs:** sa valeur n'est pas propre à un objet mais à la classe (le champ lui-même, et bien sûr sa valeur, est la même pour tous es objets de la classe)
  - On y accède à partir du nom de la classe ou de la référence à n'importe quel objet de la classe
- **Méthodes:** son code ne dépend pas d'un objet de la classe et peut être exécuté même si aucun objet existe (e.g. main)
- **Bloc de code:** indique que celui-ci ne sera exécuté qu'une fois
- **Classes internes:** des classes standard déclarées à l'intérieur d'une autre classe

Tout code utilisé dans les membres statiques ne peut pas faire référence à l'instance courante (**this**)

Membres statiques de la classe

Variable de classe

- Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur **static**)
- Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

```
public class Employe {
private String nom, prenom;
private double salaire;
private static int nbEmployes = 0;
// Constructeur
public Employe(String n, String p) {
nom = n;
prenom = p;
nbEmployes++;
}
. . .
}
```

Membres statiques de la classe

Méthode de classe

- Une méthode de classe (modificateur **static** en Java) exécute une action indépendante d'une instance particulière de la classe

- Une méthode de classe peut être considérée comme un message envoyé à une classe

- Exemple :

```
public static int getNbEmployes() {
        return nbEmployes;
}
```

Membres statiques de la classe

Exemple d'application

- Et si nous voudrions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe Personne).

- Pour créer un attribut contenant le nombre des objets créés à partir de la classe **Personne**.
- Cet attribut doit être static, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut.

Ajoutons un attribut statique nbrPersonnes à la liste d'attributs de la classe Personne

```
private static int nbrPersonnes;
```

Membres statiques de la classe

Exemple d'application

Incrémentons notre compteur de personnes dans les constructeurs

```
public Personne(int num, String nom, String prenom) {
    this.setNum(num);
    this.nom = nom;
    this.prenom = prenom;
    nbrPersonnes++;
}
public Personne() {
    this(0,null,null);
}
public Personne (Personne p) {
    this(p.num,p.nom,p.prenom);
}
```

Générons un getter pour l'attribut static nbrPersonnes

```
public static int getNbrPersonnes() {
        return nbrPersonnes;
}
```

Membres statiques de la classe

Exemple d'application

Testons cela dans la classe Main

```
public class Main {
    public static void main(String[] args) {
    Personne personne1=new Personne(1,"Amrani","Mohammed");
    System.out.println("Objet crée n°:"+ personne1.getNbrPersonnes());
    System.out.println(personne1);
    Personne personne2=new Personne();
    System.out.println("Objet crée n°:"+ personne2.getNbrPersonnes());
    System.out.println(personne2);
    Personne personne3=new Personne(personne1);
    System.out.println("Objet crée n°:"+ personne3.getNbrPersonnes());
    System.out.println(personne3);
   }
}
```

En exécutant, le résultat est :

```
Objet crée n°:1
Peronne [num=1, nom=Amrani, prenom=Mohammed]
Objet crée n°:2
Peronne [num=0, nom=null, prenom=null]
Objet crée n°:3
Peronne [num=1, nom=Amrani, prenom=Mohammed]
```

Associations entre classes

Relation d'association :

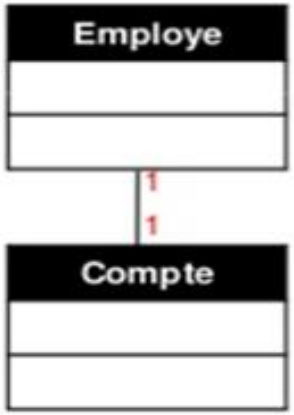
- Deux classes sont en association lorsque certains de leurs objets ont besoin s'envoyer des messages pour réaliser un traitement.
- Une association possède les cardinalités (des valeurs de multiplicité) qui représentent le nombre d'instances impliquées.

Traduction en java :

- Les associations entre classes sont tout simplement représentées par des références. Les classes associées possèdent en attribut une ou plusieurs références vers l'autre classe.
- Le nombre de référence dépend de la cardinalité. Lorsque la cardinalité est 1 ou 0..1, il n'y a qu'une seul référence de l'autre classe

Associations entre classes

Associations un à un ([1-1]) (bidirectionnelle)

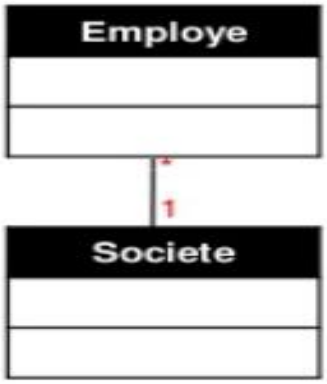



```
class Employe{
        public Compte compte;
        public void setCompte(Compte compte) {
                this.compte=compte;
        }
        public Compte getCompte() {
                return compte;
        }

}
```

```
class Compte{
        public Employe employe;
        public void setEmploye(Employe employe) {
                this.employe=employe;
        }
        public Employe getEmploye() {
                return empl;oye
        }

}
```

Associations entre classes

Associations un à plusieurs ([1-n])

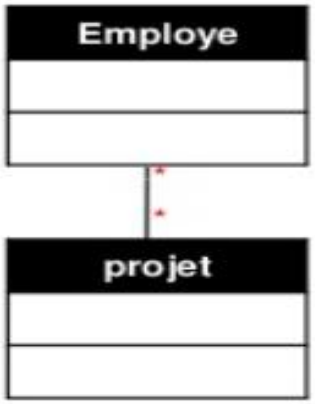



```
class Employe{
        public Societe societe
        public void setSociete(Societe
societe) {
                this.societe=societe;
        }
        public Compte getSociete() {
                return societe;
        }

}
```

```
class Societe{
        public Employe[] employes;
        public void setEmployes(Employe[]
employes) {
                this.employe=employe;
        }
        public Employe[] getEmployes() {
                return empl;oyes
        }
}
```

Associations entre classes

Associations plusieurs à plusieurs ([n-n])




```
class Employe{
        public Projet[] projets;
        public void setProjets(Projet[]
projets) {
                this.projets=projets;
        }
        public Projet[] getProjets() {
                return projets;
        }

}
```

```
class Projets{
        public Employe[] employes;
        public void setEmployes(Employe[]
employes) {
                this.employe=employe;
        }
        public Employe[] getEmployes() {
                return empl;oyes
        }
}
```

Associations entre classes

Exemple d'application

Exercice

1. Définir une classe **Adresse** avec trois attributs privés (rue, codePostal et ville de type chaîne de caractère)
2. Définir un **constructeur** avec trois paramètres, les **getters**, les **setters** et la méthode **toString()**
3. Dans la classe **Personne**, ajouter un attribut **adresse de type Adresse** et définir un nouveau constructeur à quatre paramètres et le getter et le setter de ce nouvel attribut (association [1-1] unidirectionnelle)
4. Dans la classe Main, créer deux objets, un objet de type Adresse et un de type Personne et lui attribuer l'adresse créée précédemment
5. Afficher tous les attributs de cet objet de la classe Personne

Surcharge d'une méthode

Les méthodes comme les constructeurs peuvent être surchargées: (overloaded)

- Leur nom est le même
- Le nombre ou le type de leurs paramètres varie

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents.

Objectif: fournir plusieurs définitions pour la même méthode

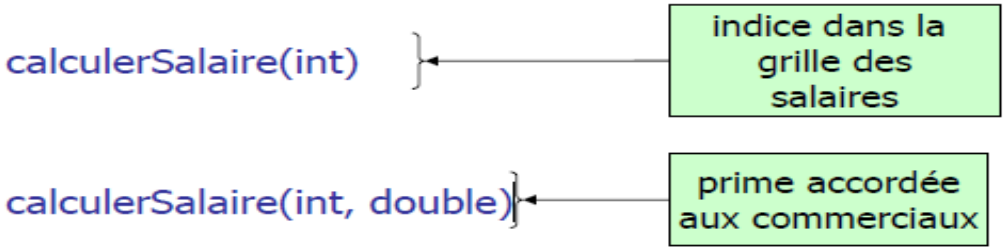
Le compilateur doit pouvoir trouver celui qui convient le mieux au « site d'appel », c'est à dire au nombre et au type des arguments

Si aucune méthode ne correspond exactement, le compilateur peut prendre une méthode approchée en fonction du typage

Surcharge d'une méthode

En Java :

- On peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :



- Il est interdit de surcharger une méthode en changeant le type de retour
  - Autrement dit, on ne peut différencier deux méthodes par leur type de retour

- Par exemple: il est interdit d'avoir ces deux méthodes dans une classe

```
int calculerSalaire(int)
double calculerSalaire(int)
```

- On peut pas surcharger les opérateurs définis par l'utilisateur.

Encapsulation

- L'**encapsulation** est une manière de définir une classe de telle sorte que ses attributs ne puisse pas être directement manipulés de l'extérieur de la classe, mais seulement indirectement par l'intermédiaire des méthodes.

- Un des avantages de cette approche est la possibilité de redéfinir la représentation interne des attributs, sans que cela affecte la manipulation externe d'un objet de cette classe.

- L'encapsulation facilite donc la mise à jour des applications.

Encapsulation

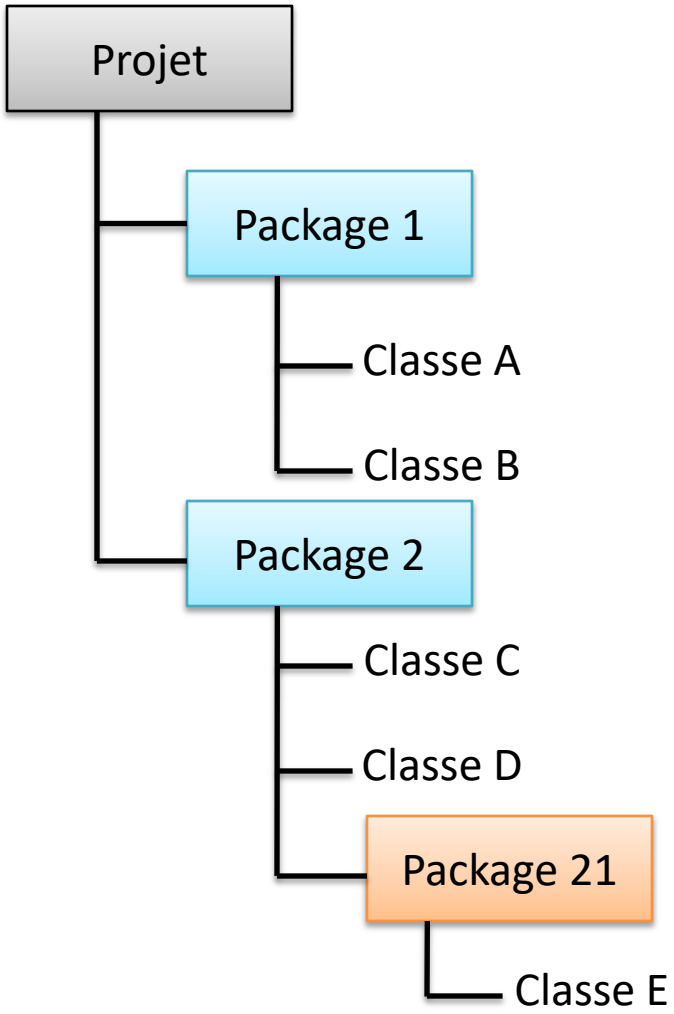
- On peut en voir l'intérêt dans un projet de développement réalisé par plusieurs développeurs (ce qui est généralement le cas en entreprise).

- Chaque développeur est responsable d'une partie du projet.
  - Si les classes dont il est le responsable sont proprement encapsulées,
  - il pourra modifier par la suite leurs représentations internes, sans que cela perturbe le travail des autres développeurs du projet susceptibles d'utiliser ces classes.

Encapsulation

- Le principe d'encapsulation dit qu'un objet ne doit pas exposer sa représentation interne au monde extérieur.
  - Représentation interne d'un objet
    - Les attributs
    - Les méthodes
  - Monde extérieur
    - Les autres classes dans la même package
    - Les classes des autres packages
- Les types d'autorisation d'accès
  - public
  - Aucun , Par défaut
  - private
  - protected

Encapsulation



Les packages

- Package= Répertoire
- Les classes Java peuvent être regroupé dans des packages
- Déclaration d'un package :
  - package nompackage
- Utilisation d'un package
  - 1. nom du package suivi du nom de la classe

```
java.util.Date now = new java.util.Date() ;
System.out.println(now) ;
```

  - 2. import de la classe du package

```
import java.util.Date ;
 …
Date now = new Date() ;
System.out.println(now) ;
```

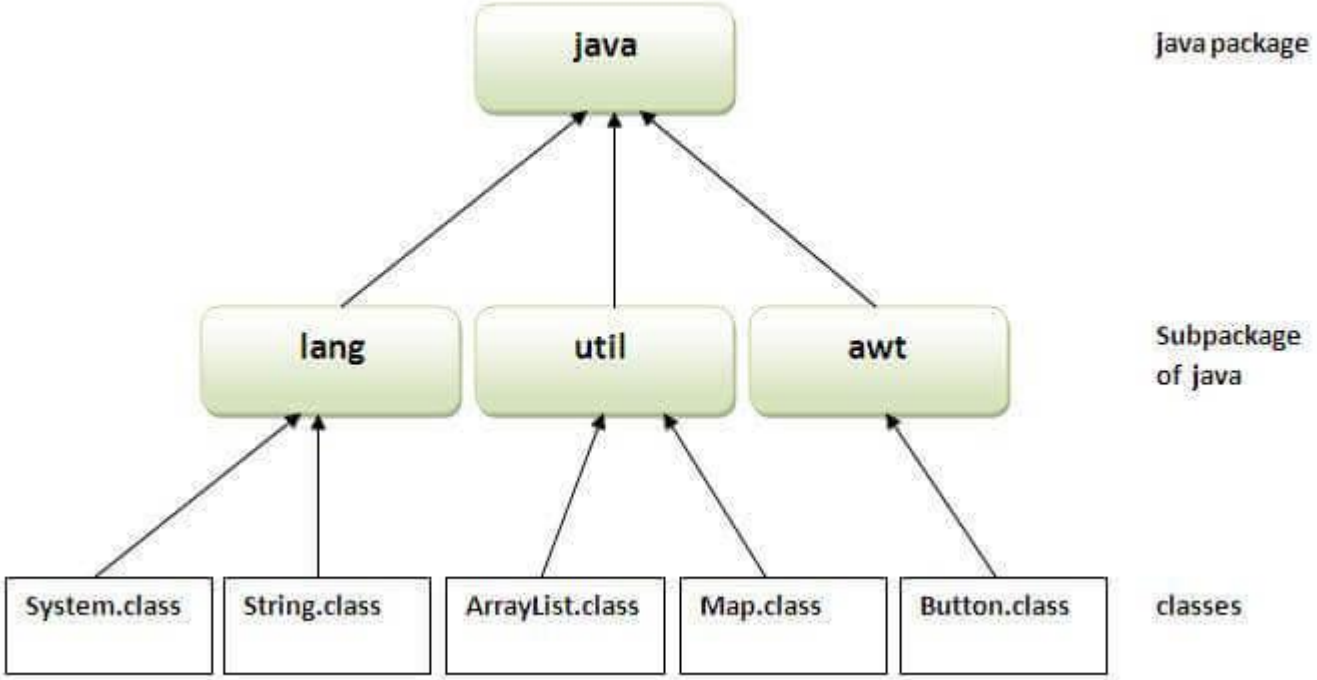
  - 3.import de tout le package

```
import java.util.*;
…
Date now = new Date() ;
System.out.println(now) ;
```

Encapsulation

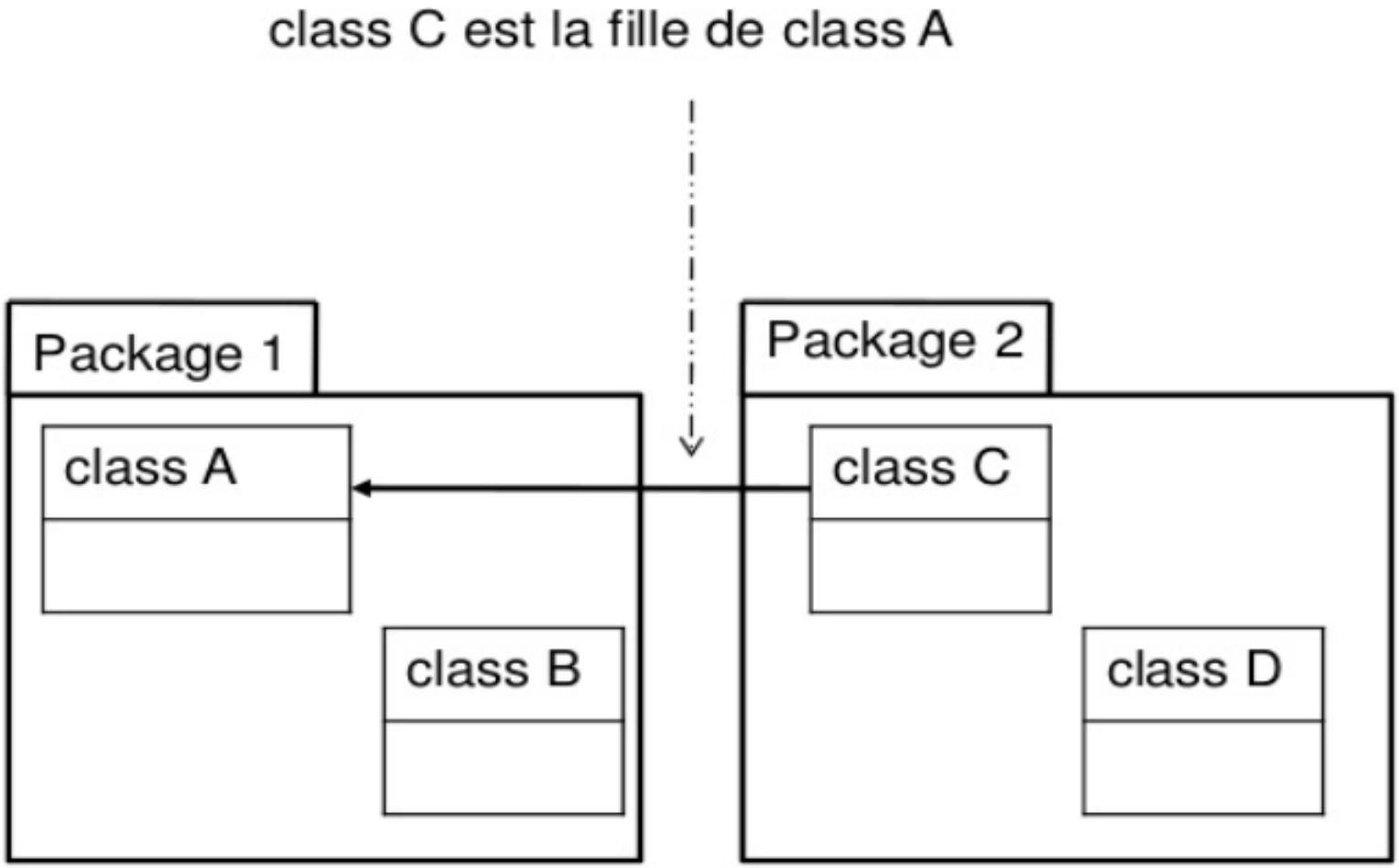
Les packages

- Les packages Les packages sont eux-mêmes organisés hiérarchiquement. : on peut définir des sous-packages, des sous-sous-packages,



Encapsulation

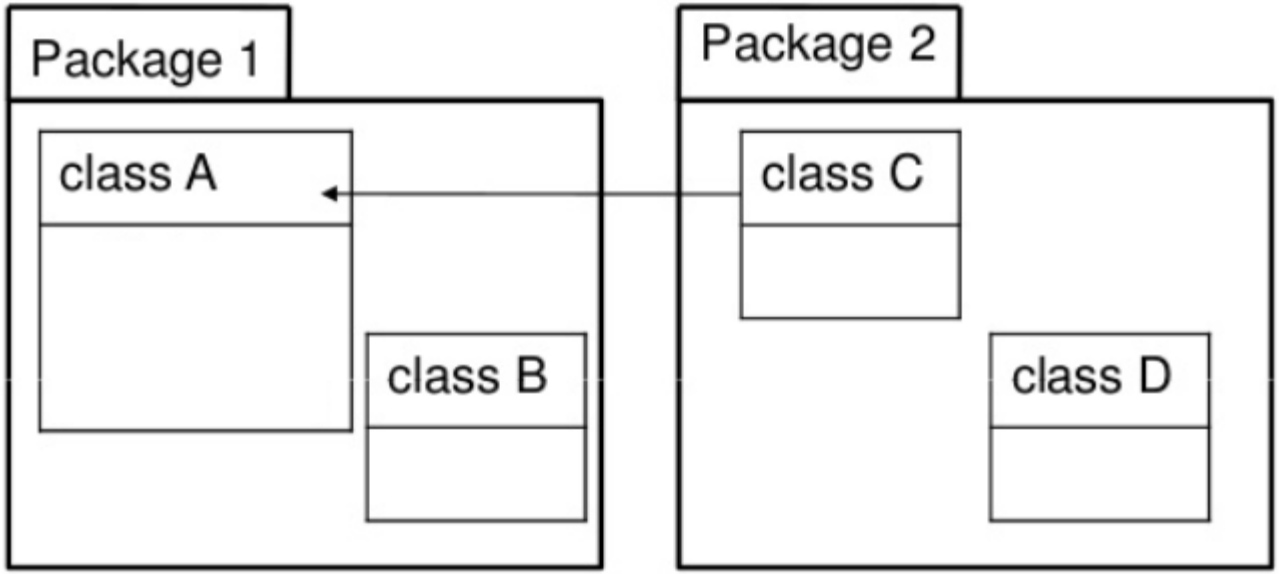
Encapsulation des classes



Encapsulation

Encapsulation des classes

La classe ***public***



`public classe` A --> La classe ***public*** est visible depuis n'importe quelle classe du projet.

```
public class A {
...

}
```

```
class B {
     A a;
...
}
```

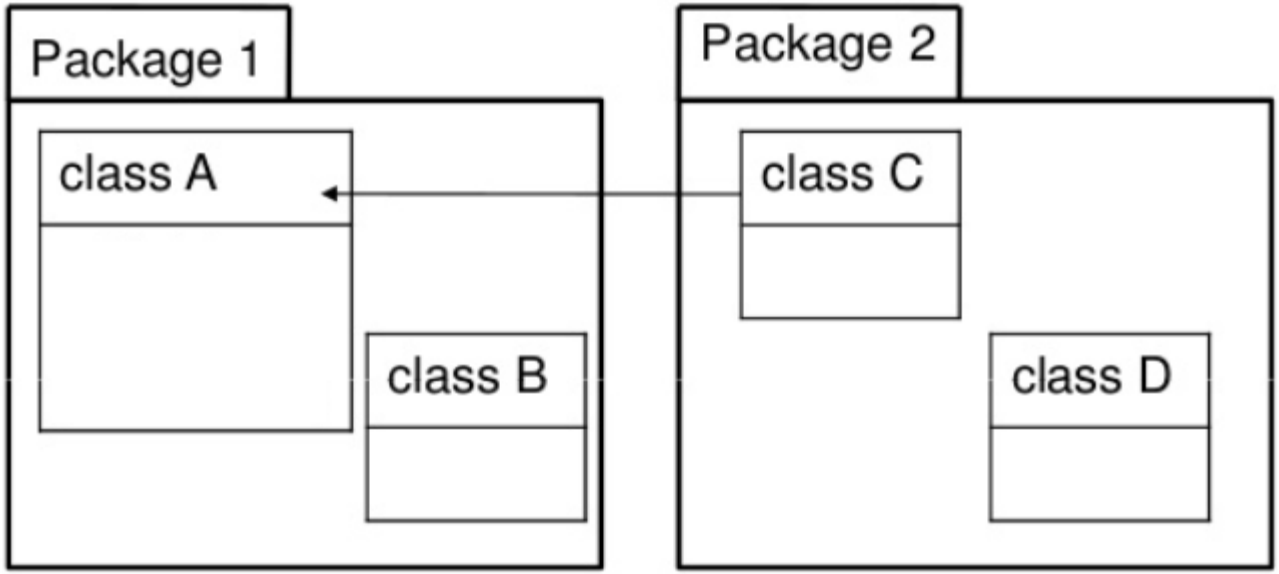
```
class c extends A {
          A a;
...
}
```

```
class D {
          A a;
...
}
```

Encapsulation

Encapsulation des classes

La classe ***default***



`classe` A --> La classe ***default*** est visible seulement par les classes de son package.

```
class A {
...

}
```

```
class B {
     A a;
...
}
```

✓

```
class c extends A {
          A a;
...
}
```

✗

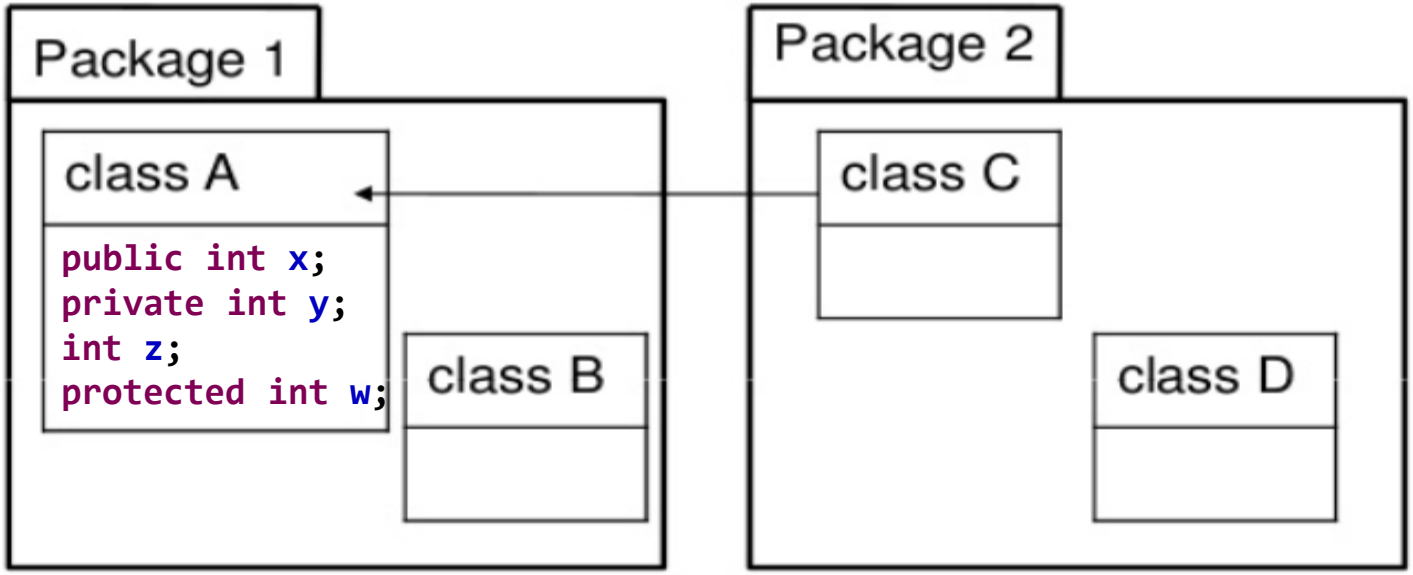
```
class D {
          A a;
...
}
```

✗

Encapsulation

Encapsulation des attributs

L'attribut ***public***



`public int x;` --> L'attribut ***public*** est visible par toutes les classes.

```
Public class A {
public int x;

...
}
```

```
class B {
     A a=new A();
     int t=a.x;
...
}
```

✓

```
class c extends A {
          A a;
          int t=a.x;
...
}
```

✓

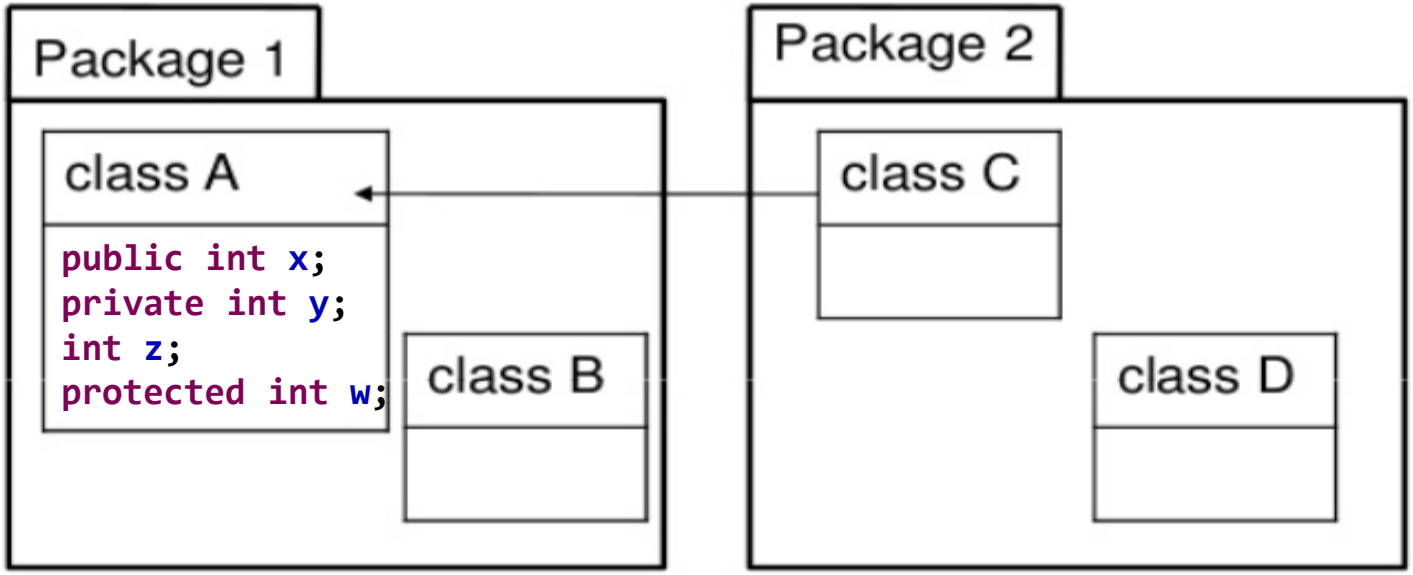
```
class D {
          A a;
          int t=a.x;
...
}
```

✓

Encapsulation

Encapsulation des attributs

L'attribut ***private***



`private int x;` --> L'attribut ***private*** n'est accessible que depuis l'intérieur même de la classe .

```
Public class A {
private int y;

...
}
```

```
class B {
     A a=new A();
     int t=a.y;
...
}
```

✗

```
class c extends A {
          A a;
          int t=a.y;
...
}
```

✗

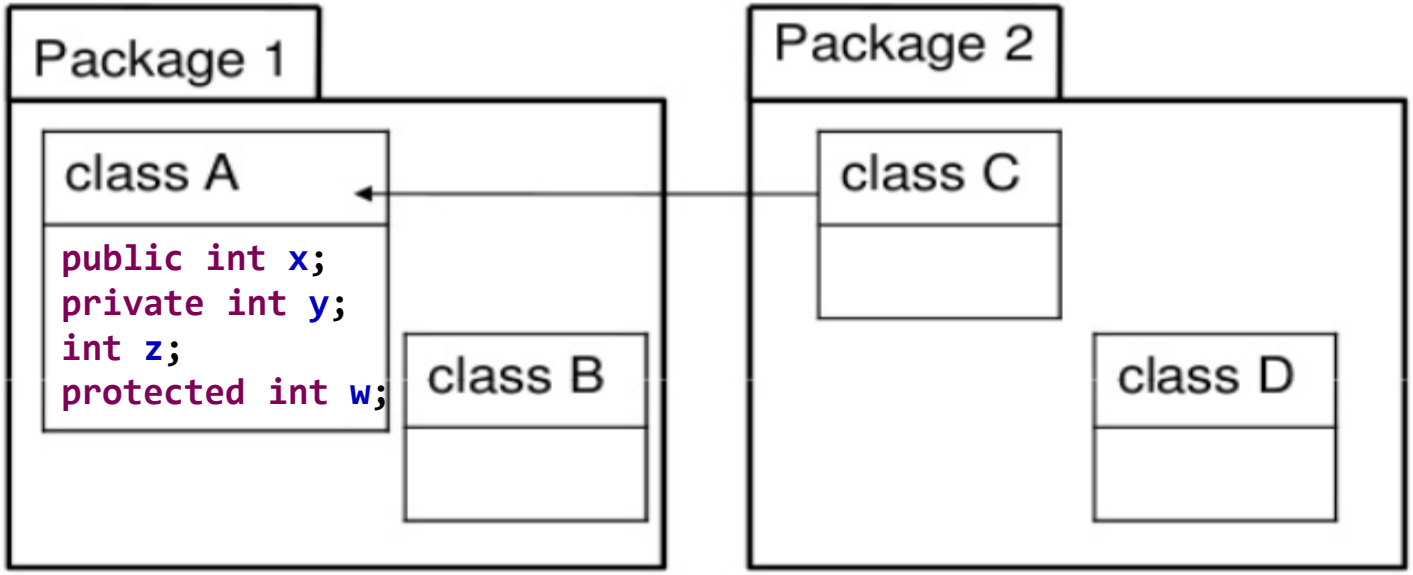
```
class D {
          A a;
          int t=a.y;
...
}
```

✗

Encapsulation

Encapsulation des attributs

L'attribut ***default***



`int z;` --> L'attribut ***default*** n'est accessible que depuis les classes faisant partie du même package .

```
Public class A {
int z;

...
}
```

```
class B {
     A a=new A();
     int t=a.z;
...
}
```

✓

```
class c extends A {
          A a;
          int t=a.z;
...
}
```

✗

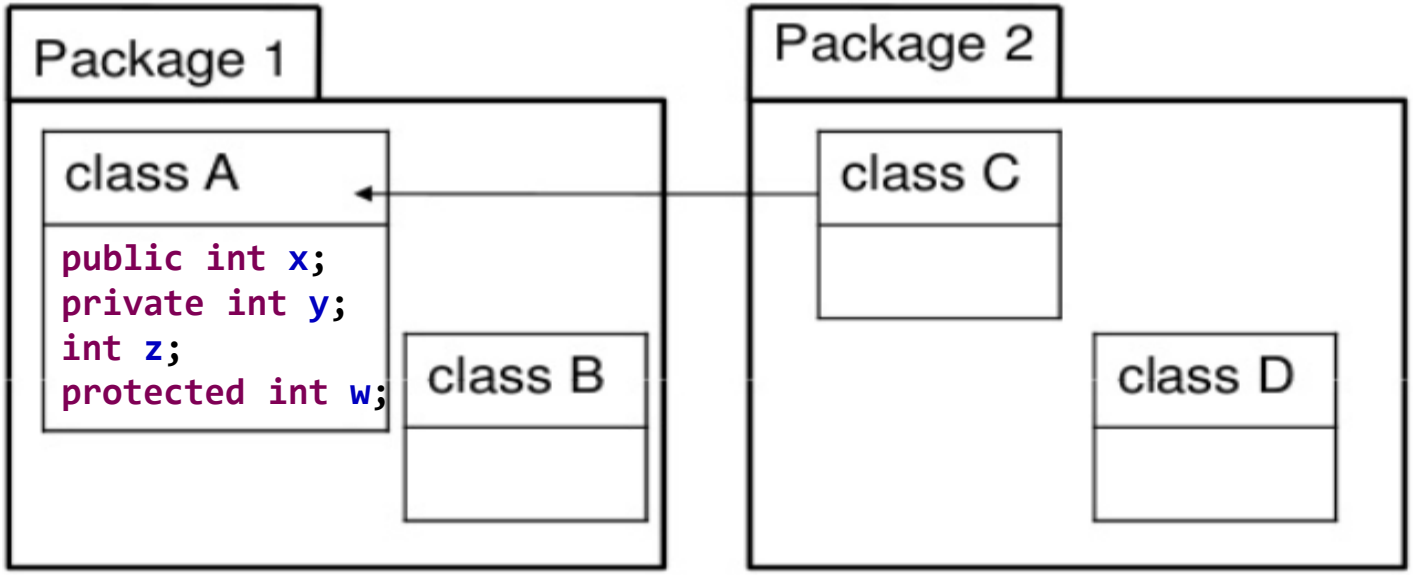
```
class D {
          A a;
          int t=a.z;
...
}
```

✗

Encapsulation

Encapsulation des attributs

L'attribut ***protected***



`protected int w;` --> L'attribut ***protected*** est accessible uniquement aux classes d'un même package et à ses classe filles (même si elles sont définies dans un package différents).

```
Public class A {
protected int w;

...
}
```

```
class B {
     A a=new A();
     int t=a.w;
...
}
```

✓

```
class c extends A {
          A a;
          int t=a.w;
...
}
```

✓

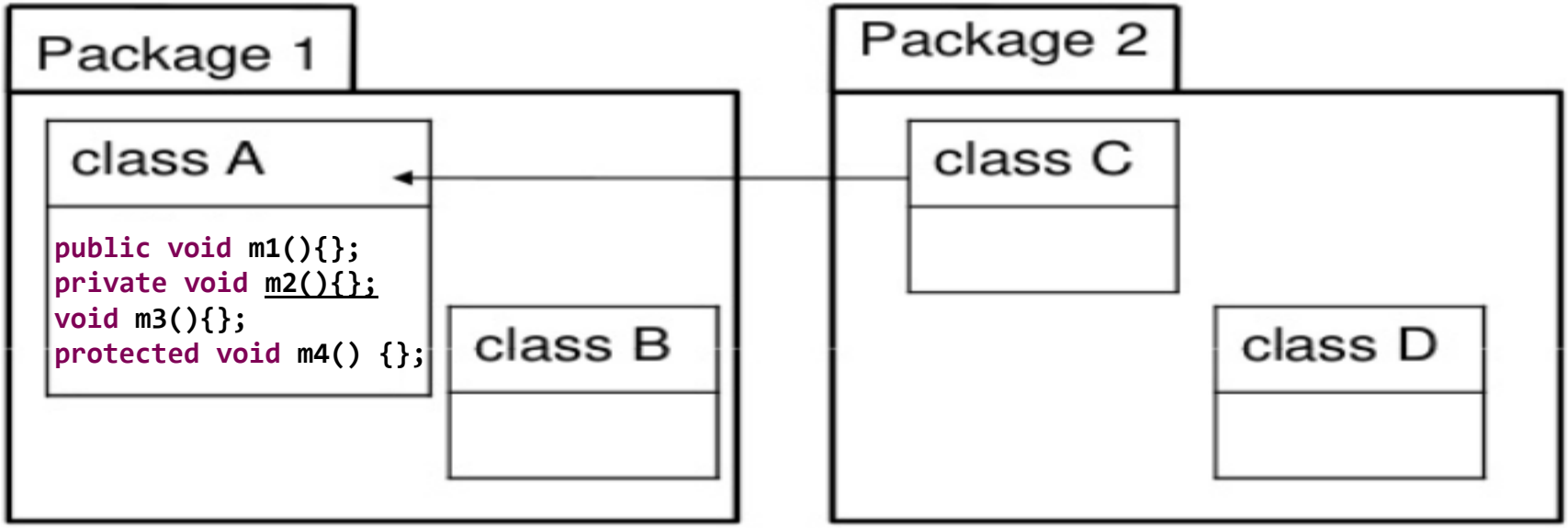
```
class D {
          A a;
          int t=a.w;
...
}
```

✗

Encapsulation

Encapsulation des méthodes

méthode ***public***



`public void m1()` --> la méthode ***public*** est accessible par toutes les classes du projet.

```
Public class A {
public void m1(){}
...

}
```

```
class B {
     A a=new A();
     a.m1();
...
}
```

✓

```
class c extends A {
          A a;
          a.m1();
...
}
```

✓

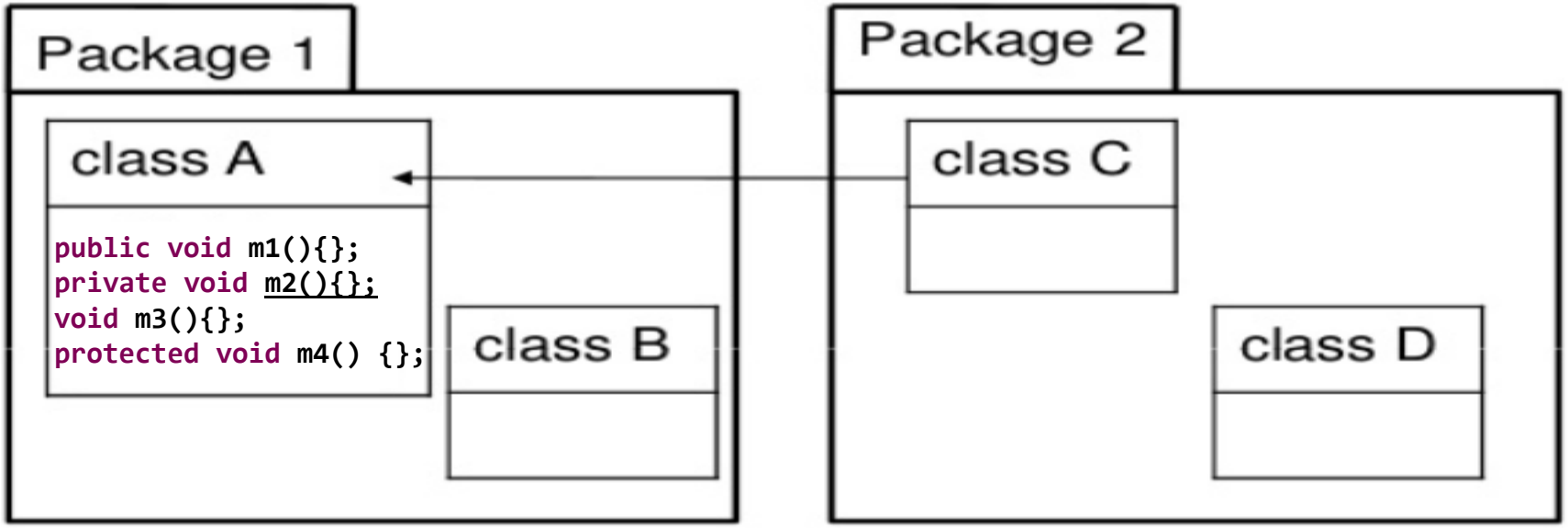
```
class D {
          A a;
          a.m1();
...
}
```

✓

Encapsulation

Encapsulation des méthodes

méthode ***private***



`private void m2()` --> la méthode ***private*** n'est accessible que depuis l'intérieur même de la classe .

```
Public class A {
void m2(){}
...

}
```

```
class B {
     A a=new A();
     a.m2();
...
}
```

✗

```
class c extends A {
          A a;
          a.m2();
...
}
```

✗

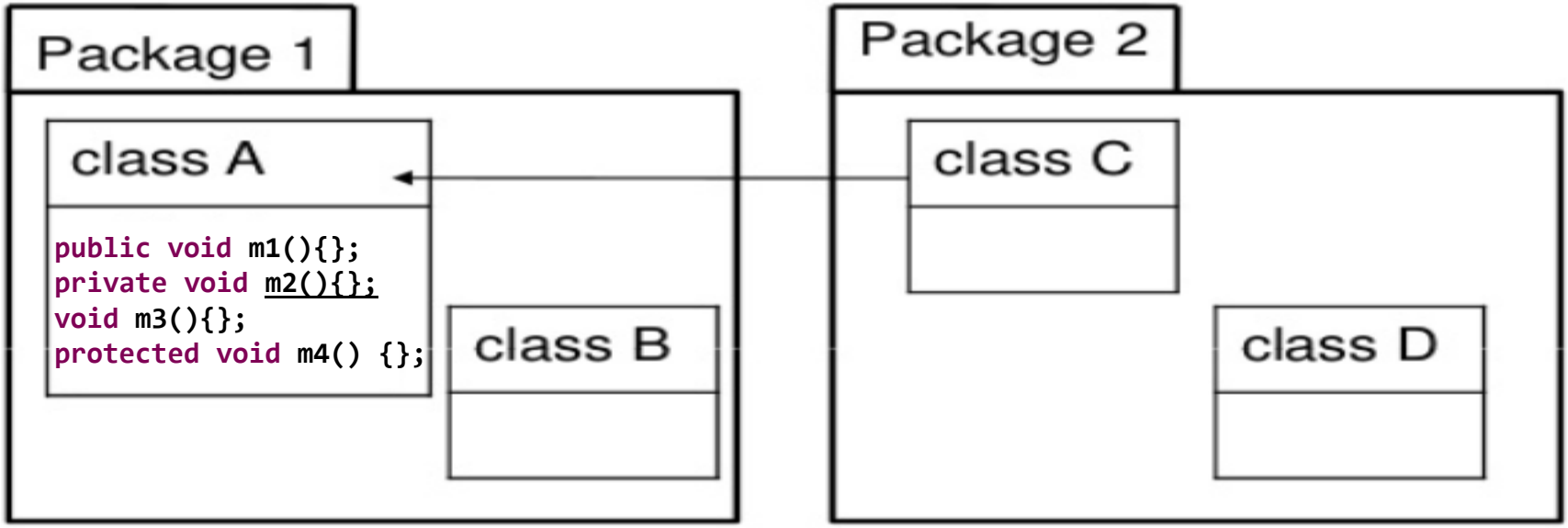
```
class D {
          A a;
          a.m2();
...
}
```

✗

Encapsulation

Encapsulation des méthodes

méthode ***default***



`void m3()` --> la méthode ***default*** n'est accessible que depuis les classes faisant partie du même package .

```
Public class A {
void m3(){}
...

}
```

```
class B {
     A a=new A();
     a.m3();
...
}
```

✓

```
class c extends A {
          A a;
          a.m3();
...
}
```

✗

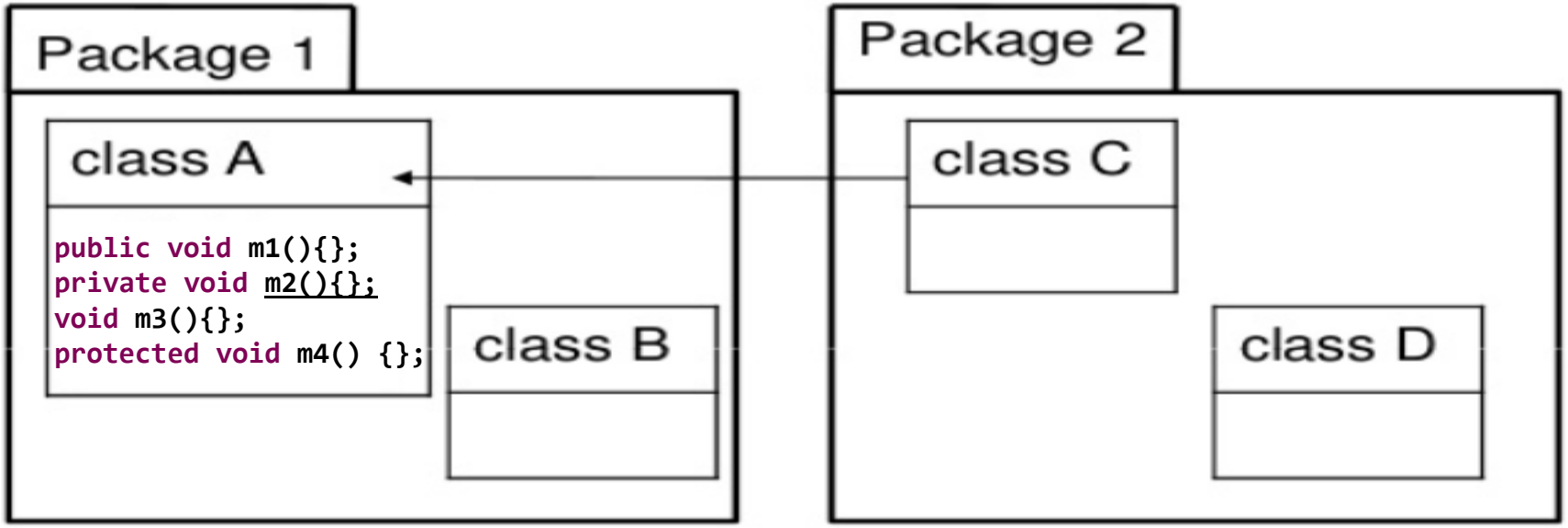
```
class D {
          A a;
          a.m3();
...
}
```

✗

Encapsulation

Encapsulation des méthodes

méthode ***protected***



`protected void m4()` --> la méthode ***protected*** est accessible uniquement aux classes d'un même package et à ses classe filles (même si elles sont définies dans un package différents).

```
Public class A {
void m4(){}
...

}
```

```
class B {
     A a=new A();
     a.m4();
...
}
```

✓

```
class c extends A {
          A a;
          a.m4();
...
}
```

✓

```
class D {
          A a;
          a.m4();
...
}
```

✗

Encapsulation

- Dans le but de renforcer le contrôle de l'accès **aux attributs** d'une classe, il est recommandé de les déclarer private.

- Pour la manipulation des attributs private, on utilise:
  - Un mutateur (setter): une méthode qui permet de définir la valeur d'un attribut particulier.
  - Un accesseur (getter): une méthode qui permet d'obtenir la valeur d'un attribut particulier.

- Le setter et le getter doivent être déclarés **public**