

Java

Les classes

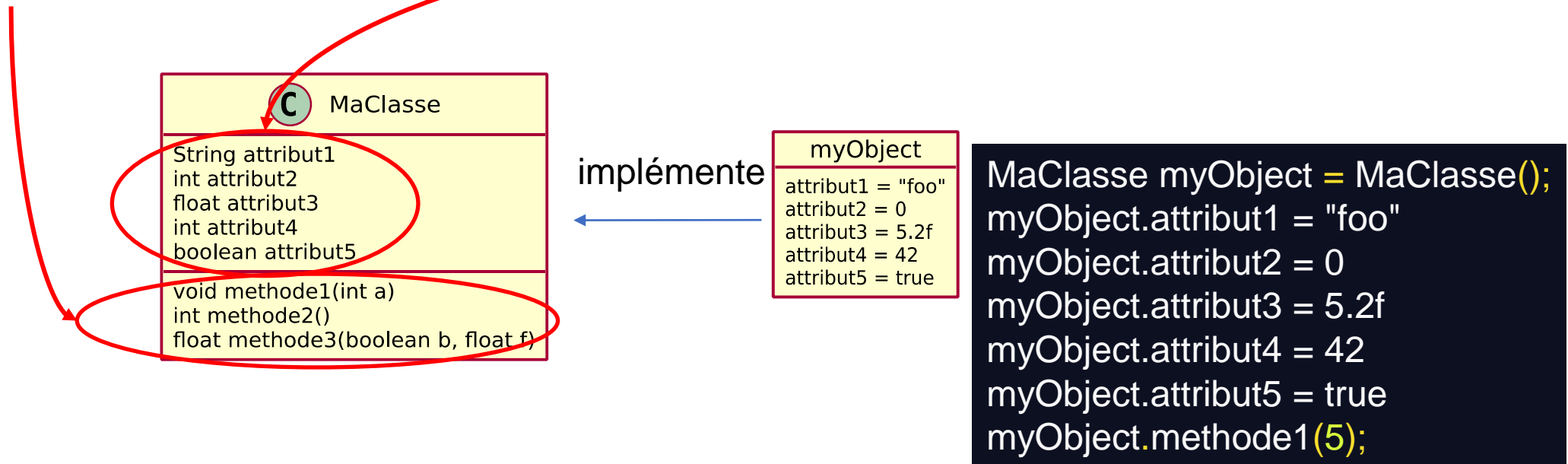
Rappels sur la Programmation Orientée Objet (POO)

Qu'est-ce qu'un objet ?

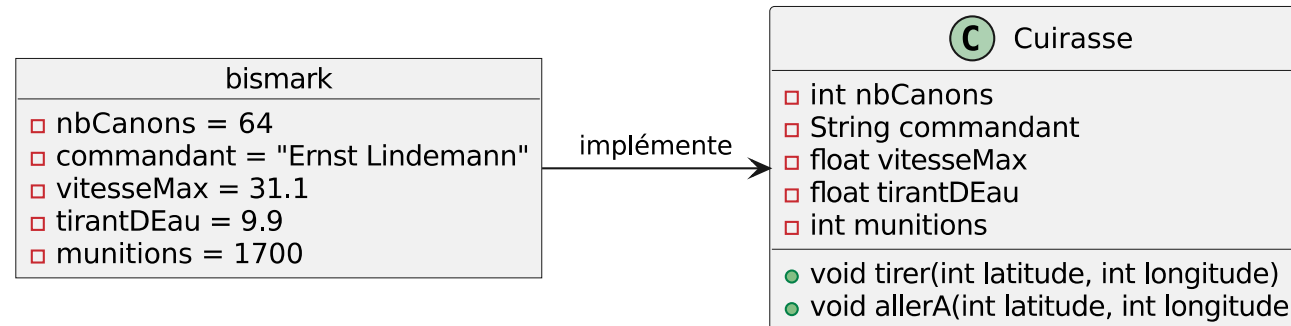
- Un objet est un conteneur encapsulant à la fois des informations et des mécanismes concernant un sujet.
- Les objets sont régulièrement utilisés comme représentation simplifiée (une abstraction) d'une entité du monde réel
- Cette abstraction ne conserve que les caractéristiques jugées pertinentes
- Cette démarche est forcément subjective, guidée par le besoin
- Par exemple, dans certaines applications, un pseudo et un âge peuvent suffire à modéliser une personne, pour d'autres on devra ajouter le nom complet, l'adresse, le numéro de sécurité sociale...

Les classes

- Une classe :
 - Est une méta-classe
 - Définit les caractéristiques des objets (**attributs**), ainsi que ses mécanismes (**méthodes**)

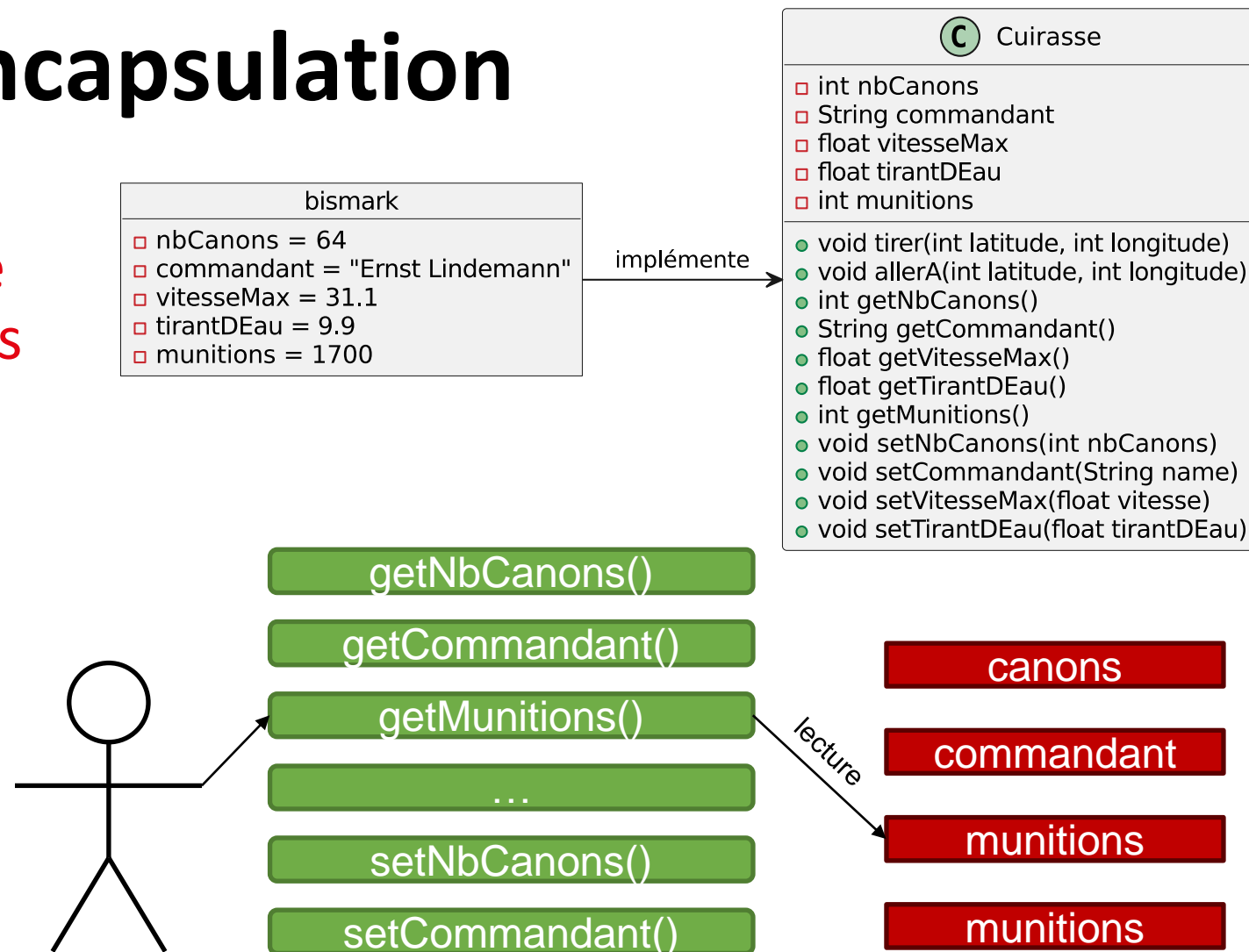


Un exemple



Encapsulation

- On va cacher l'état interne de l'objet, et le manipuler *via* des méthodes
- Cela permet de contrôler l'accès aux données
- Par exemple, pour vérifier les munitions du cuirassé, on utilise un **accesseur** (en anglais, *getter*)



Un accès aux données maîtrisé

- L'encapsulation permet de contrôler :
 - Quelles informations sur l'état de l'objet sont consultables et sous quelle forme
 - La manière dont l'utilisateur peut modifier l'état de l'objet.
- Dans notre exemple :
 - Pas de modification directe du nombre de la vitesse par le getter
 - Modification indirecte de la vitesse *via* `tirer()`
- Modifier un attribut:
 - On utilise une autre méthode, `setAttribute()`

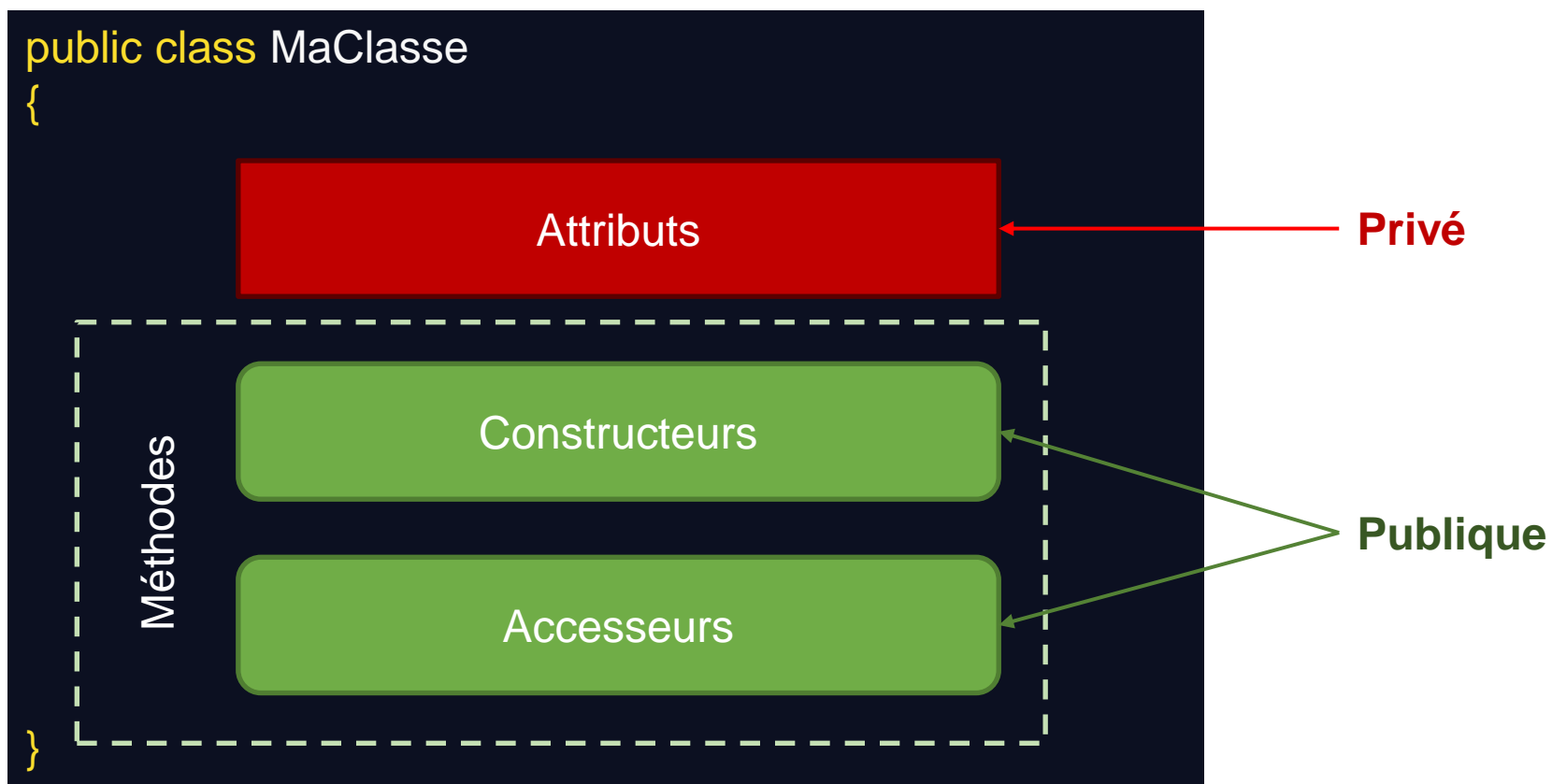
Définition d'une classe

Définition

- Une classe MaClasse est définie dans le fichier MaClasse.java
- Une classe possède des membres : des attributs et des méthodes
- Chaque membre possède une visibilité :
 - Privée : Uniquement accessible depuis le code des méthodes de la classe
 - Publique : Accessible depuis n'importe où
 - Protected: visible des classes enfants de la classe
 - Package private (défaut) : visible des classes du package
- Pour le moment, partez du principe que :
 - Les attributs sont privés
 - Les méthodes sont publiques

Définition

- Nous vous conseillons la structure type suivante :



Classe Fraction : attributs

```
public class Fraction
{
    private int numerator
    private int denominator

    ...
}
```

- La visibilité privée garantit techniquement l'encapsulation.
- Les règles d'initialisation par défaut s'appliquent.

Les accessoires

Classe Fraction : accesseurs

Visibilité (publique généralement)

Déclaration de méthode classique

```
public class Fraction
{
    ...
    public int getNumerator()
    {
        return this.numerator;
    }
    public void setNumerator(int numerator)
    {
        this.numerator = numerator;
    }
}
```

Accès en lecture : **getter**

Accès en écriture : **setter**

Référence à l'instance courante

L'auto-référence **this**

- C'est un argument implicite qui est automatiquement passé lors d'appels de méthodes.
- C'est une référence vers l'instance qui a appelé la méthode, que l'on peut interpréter comme « référence à moi même » (d'ailleurs appelée "self" dans certains langages).
- On parle d'auto référence.

L'auto-référence **this**

Appel à la méthode

`maFraction.setNumerator(42)`

maFraction

this

numerator 42
denominator 3

42

numerator

```
public class Fraction
{
    ...
    public void setNumerator(int numerator)
    {
        this.numerator = numerator;
    }
}
```

L'auto-référence **this**

- **this** est particulièrement utile pour éviter des conflits de nom avec
- des variables locales.
- L'oubli de **this** est une erreur fréquente dans les setters.

```
public void setNumerator int numerator {
    numerator = numerator
}
```

Que fait ce code ?



Recommandation : préfixer les noms des membres par **this** dans le code des méthodes

Des attributs privés assurent des données valides

- La visibilité privée des attributs rend la modification directe impossible et force à passer par les *setters*

```
maFraction.denominator = 0;
```

Erreur de compilation

```
maFraction.setDenominator(0);
```

OK

- On peut faire en sorte que la méthode `setDenominator` refuse de modifier le dénominateur (ou mieux, lève une exception) si la nouvelle valeur est 0.

Les constructeurs

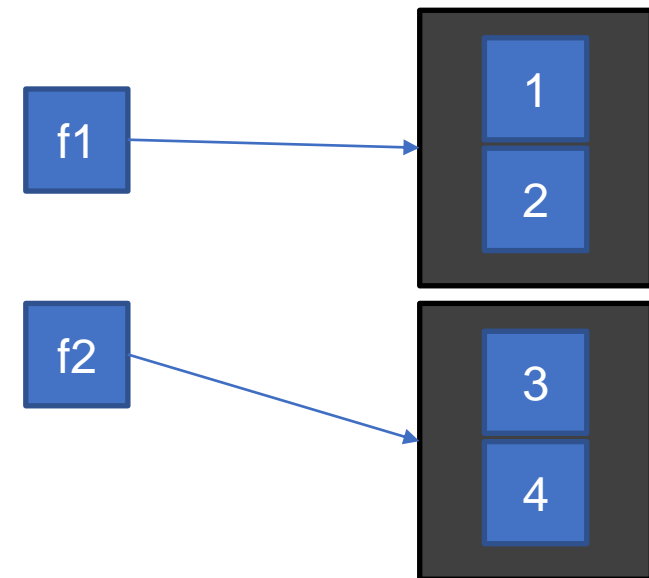
Instanciation d'une classe

- L'instanciation consiste à créer un objet en mémoire à partir d'un modèle qu'est la classe.
- Chaque instance a son **identité propre** (deux instances peuvent avoir des valeurs d'attributs identiques, elles sont tout de même discernables).
- Une vision concrète de cette identité peut être l'adresse en mémoire.

Constructeur

- On instancie une classe en appelant une méthode spéciale appelée constructeur initialise les attributs avec les valeurs données en argument.
- Le constructeur a le même nom que la classe, et est appelé en utilisant l'opérateur **new**

```
Fraction f1 = new Fraction(1,2);
Fraction f2 = new Fraction(3,4);
```

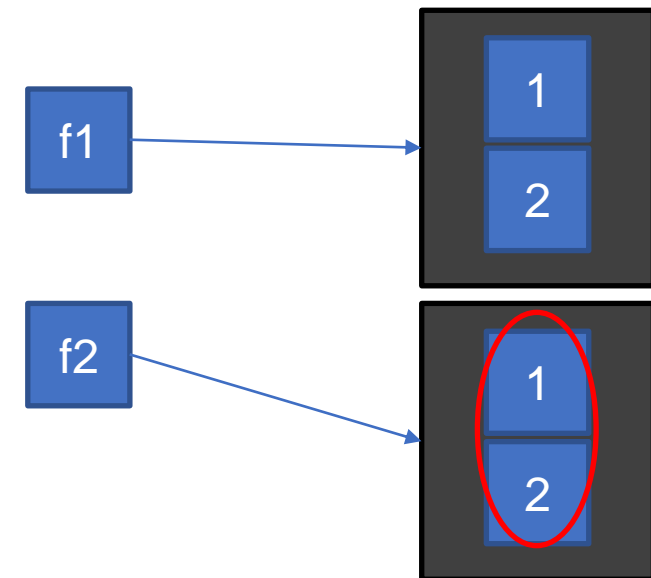


Constructeur

- On instancie une classe en appelant une méthode spéciale appelée constructeur initialise les attributs avec les valeurs données en argument.
- Le constructeur a le même nom que la classe, et est appelé en utilisant l'opérateur **new**

```
Fraction f1 = new Fraction(1,2);
Fraction f2 = new Fraction(1,2);
```

Les fractions f1 et f2 ont les mêmes valeurs d'attributs mais sont bien deux entités distinctes



Classe Fraction : constructeurs

```
public class Fraction
{
    public Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
    ...
}
```

Constructeur et accesseurs



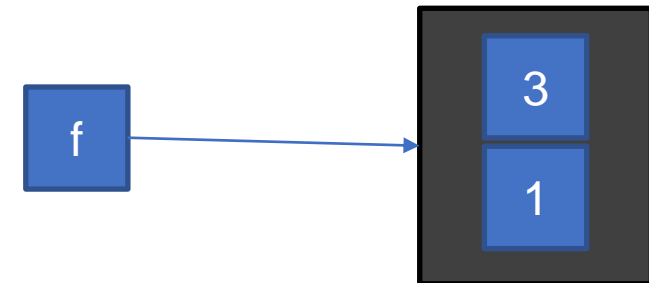
S'appuyer sur les accesseurs dans le constructeur permet d'appliquer les mêmes règles de vérification à la construction et lors de la vie de l'objet.

```
public class Fraction
{
    public Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
    ...
}
```

Plusieurs constructeurs

- Une classe peut avoir plusieurs constructeurs, ce qui permet de l'instancier de plusieurs manières différentes.
- Par exemple, on peut permettre d'instancier une fraction juste avec un numérateur, dans ce cas le dénominateur sera affecté à 1 par défaut :

```
Fraction f = new Fraction(3);
```



- Pour éviter la redondance de code, les constructeurs peuvent s'appeler les uns les autres.

Classe Fraction avec 2 constructeurs

```
public class Fraction
{
    // numérateur + dénominateur
    public Fraction(int numerator, int denominator) {
        this.setNumerator(numerator);
        this.setDenominator(denominator);
    }
    // numérateur uniquement
    public Fraction(int numerator) {
        this.setNumerator(numerator);
        this.setDenominator(1);
    }
    ...
};
```

Redondant

Classe Fraction avec 2 constructeurs

```
public class Fraction
{
    // numérateur + dénominateur
    public Fraction(int numerator, int denominator) {
        this.setNumerator(numerator);
        this.setDenominator(denominator);
    }
    // numérateur uniquement
    public Fraction(int numerator) {
        this(numerator, 1);
    }
    ...
};
```

Les packages

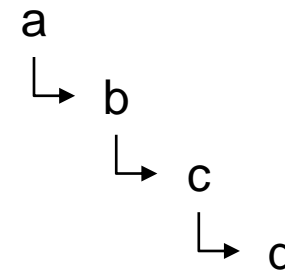
Classe Fraction avec 2 constructeurs

- Toute classe Java appartient à un package
- Les packages une structure arborescente. A chaque package est associé un répertoire :

Nom du package :

a.b.c.d

Répertoires sur le disque :



Importer des classes de packages existants

- Pour importer une classe :


```
import <Nom du package>.<Nom de la classe>
```

- Exemple : `import java.util.Scanner`

- Pour importer toutes les classes d'un package (*wildcard import*) :

```
import <Nom du package>
```

- Exemple : `import java.swing.*`

-  À utiliser avec modération ! (quand vous utilisez beaucoup de classes d'un même package), car possède les inconvénients suivants :

- Très peu d'information sur la provenance des classes.
- Forte augmentation de risque de conflits de noms.

Créer ses propres packages

Si on veut placer une classe C dans son propre package :

- D'ajouter la ligne `package <nom du package>` au début du fichier C.java
- Placer le fichier C.java dans une arborescence correspondant au nom du package
- Exemple : Pour ajouter la classe C au package **foo.bar** :

Nom du package :

```
package foo.bar
```

```
...
```

Arborescence sur le disque :

```
foo
└─ bar
    └─ C.java
```

Les membres statiques

Membre statique

- Un membre statique est associé à une classe (et non à une instance)
- On déclare un membre statique en ajoutant le mot clé **static**
- On accède à un membre statique `m` d'une classe `C` en écrivant : `C.m`
 - Un attribut statique existe même si la classe n'a jamais été instanciée
 - Dans le code d'une méthode statique, **this** n'a aucun sens.
- **Utilisations :**
 - Partager une information entre toutes les instances d'une classe (sorte de variable globale dans la portée de la classe)
 - Modéliser une information propre à la classe (ex : compteur d'instances)
 - Définir des fonctions utilitaires
 - Définir des constantes

Exemple : fonctions mathématiques

Définition :

```
public class Math{  
    public static float square(float x){  
        return x*x;  
    }  
    ...  
}
```

Utilisation :

```
float x = 2;  
float x2 = Math.square(x);
```

Exemple : constantes mathématiques

Définition :

Utilisation :

```
public class Math{
    public static final float PI = 3.14f;
    ...
}
```

```
float x = Math.PI / 2;
```

Le membre est public mais le mot clé final permet d'en figer la valeur (non modifiable une fois initialisée)

Les énumérations *

Les énumérations

- Une énumération permet de modéliser un type qui possède un nombre fini de valeurs
- Exemple :

```
public enum Color{  
    RED,  
    BLUE,  
    GREEN  
}
```