Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Introduction à la Programmation Objet : Programmation orientée-objet

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Objectifs du cours de la semaine

- Introduire les notions d'**encapsulation** et d'**abstraction**
- **Objets**, instances et **classes**
- Classes en Java
- Variables d'instance
- **Méthodes** d'instance
- Encapsulation et Interface : `public` et `private`
- L'objet `this` et le **masquage**
- Constructeurs

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Support MOOC

Vidéos et Quiz :

`https://www.coursera.org/learn/programmation-orientee-objet-java/home/week/1`

`https://www.coursera.org/learn/programmation-orientee-objet-java/home/week/2`

- ☞ Semaines 1 et 2
- ☞ Notes de cours et BOOC, semaine 1
- ☞ Notes de cours et BOOC, semaine 2

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Objets : 1ère introduction

Vous avez appris à écrire des programmes de plus en plus complexes.

Il faut donc maintenant des outils pour **organiser ces programmes** de façon plus efficace.

C'est l'un des objectifs principaux de la notion d'**objet**.

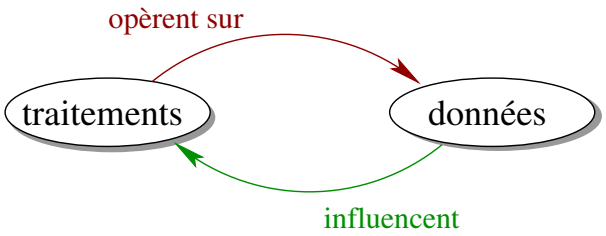
Les objets permettent de mettre en œuvre dans les programmes les notions :

- d'encapsulation
- d'abstraction
- d'héritage
- et de polymorphisme

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Programmation impérative/procédurale [rappel]

Dans les programmes que vous avez écrits jusqu'à maintenant, les notions de variables/types de **données** et de **traitement** de ces données étaient séparées :



Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Programmation procédurale : exemple

```
class Geometrie {

  public static void main(String[] args) {
    double largeur = 3.0;
    double hauteur = 4.0;

    System.out.println("Surface du rectangle : "
                       + surface(largeur, hauteur));
  }

  static double surface(double largeur,
                        double hauteur) {
    return (largeur * hauteur);
  }
```


Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Notions d'encapsulation

Le **principe d'encapsulation** consiste à *regrouper* dans le même objet informatique («concept»), données et traitements qui lui sont spécifiques :

- Les données inclues dans un objet seront appelées les **attributs** de cet objet ;
- Les traitements/fonctions défini(e)s dans un objet seront appelées les **méthodes** de cet objet.

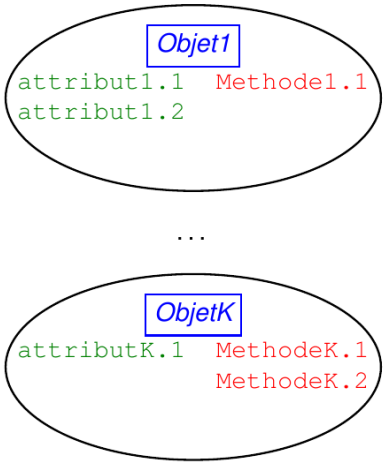
Résumé du premier principe de l'encapsulation :

OBJET
=
attributs
+
méthodes

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Notion d'encapsulation (2)

Les objets sont définis par leurs attributs et leurs méthodes :



Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Notion d'abstraction (1)

Pour être véritablement intéressant, un objet doit permettre un certain degré d'**abstraction**.

Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :

- des caractéristiques communes à tous les éléments
- des mécanismes communs à tous les éléments

☞ description **générique** de l'ensemble considéré : se focaliser sur l'essentiel, cacher les détails.

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

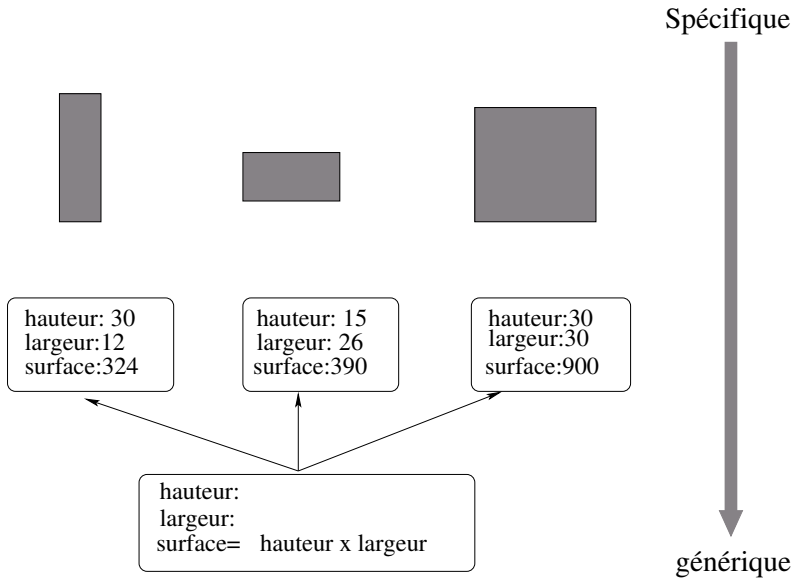
Notion d'abstraction (2)

Exemple : Rectangles

- la notion d'*« objet rectangle »* n'est intéressante que si l'on peut lui associer des propriétés et/ou mécanismes généraux
  - ☞ propriétés et mécanismes **valables pour l'ensemble** des rectangles et non pas pour un rectangle particulier.
- Les notions de largeur et hauteur sont des propriétés générales des rectangles (**attributs**),
- Le mécanisme permettant de calculer la surface d'un rectangle (surface = largeur $\times$ hauteur) est commun à tous les rectangles (**méthodes**)

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Abstraction (exemple)



Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Abstraction et Encaspuslation

Un intérêt de l'encapsulation est que cela permet d'**abstraire** :

En plus du regroupement des données et des traitements relatifs à une entité, l'encapsulation permet en effet de définir deux **niveaux de perception** :

- Le niveau externe : partie « visible » (par les programmeurs-utilisateurs) de l'objet :
  - **prototype** des méthodes et attributs accessibles hors de l'objet
  - ☞ c'est l'**interface** de l'objet avec l'extérieur résultat du processus d'*abstration*
- Le niveau interne : (détails d')**implémentation** de l'objet
  - méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou l'intérieur d'objets similaires)
  - **définition** de l'ensemble des méthodes de l'objet
  - ☞ c'est le **corps** de l'objet

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet : données (attributs) **et** traitements (méthodes)
2. isolement et dissimulation des détails d'implémentation
   Interface = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser
   ☞ Concentration sur les attributs/méthodes concernant l'objet (*abstraction*)

Exemple : L'interface d'une voiture

- Volant, accélérateur, pédale de freins, etc.
- Tout ce qu'il faut savoir pour la conduire (mais pas la réparer ! ni comprendre comment ça marche)
- L'interface ne change pas, même si l'on change de moteur... ...et même si on change de voiture (dans une certaine mesure) : abstraction de la notion de voiture (en tant qu'« objet à conduire »)

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

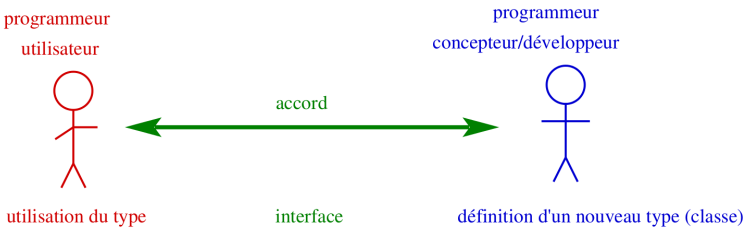
Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet : données (attributs) **et** traitements (méthodes)
2. isolement et dissimulation des détails d'implémentation
   Interface = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser
   ☞ Concentration sur les attributs et les méthodes concernant l'objet (*abstraction*)

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Les « 3 facettes » d'une classe



Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Pourquoi abstraire/encapsuler ?

1. L'intérêt de regrouper les traitements et les données conceptuellement reliées est de permettre une *meilleure visibilité* et une meilleure cohérence au programme, d'offrir une plus grande modularité.
   Comparez :

```
double largeur = 3.0;
double hauteur = 4.0;

System.out.print("Surface : ");
System.out.println(surface(largeur,hauteur));
```

   avec :

```
Rectangle rect = new Rectangle(3.0, 4.0);
System.out.print("Surface : ");
System.out.println(rect.surface()):
```

   Note : ne pas se préoccuper à ce stade des détails de syntaxe de la version du bas. C'est juste l'esprit qui compte.

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Pourquoi abstraire/encapsuler ? (2)

2. L'intérêt de séparer les niveaux *interne* et *externe* est de donner un **cadre** plus **rigoureux** à l'utilisation des objets utilisés dans un programme

   Les objets ne peuvent être utilisés qu'au travers de leur interfaces (niveau externe)
   et donc les éventuelles **modifications** de la structure interne restent **invisibles** à l'extérieur

Règle : les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur, mais uniquement par des méthodes.

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

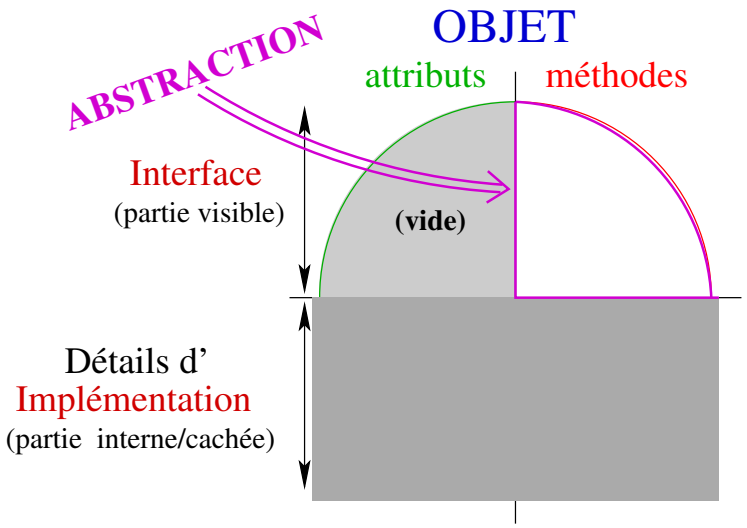
Comment bien abstraire/encapsuler ?

- Il doit y avoir **indépendance entre interface et implémentation (réalisation)** : le but ultime est qu'on puisse changer complètement l'implémentation tout en gardant exactement la même interface - et qu'ainsi le code "utilisateur" ne change pas.
  - ☞ Le but de l'encapsulation n'est pas de cacher des attributs, mais de **cacher l'encodage des données** (la manière de les représenter).
- L'interface d'utilisation doit permettre **de manipuler les données** plutôt que d'y accéder
  - ☞ elle doit exposer le comportement des données plutôt que leur état (voir exemple dans l'annexe).
- Les méthodes manipulant les données doivent s'assurer de leur validité.

Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Encapsulation / Abstraction : Résumé

MIEUX :



Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

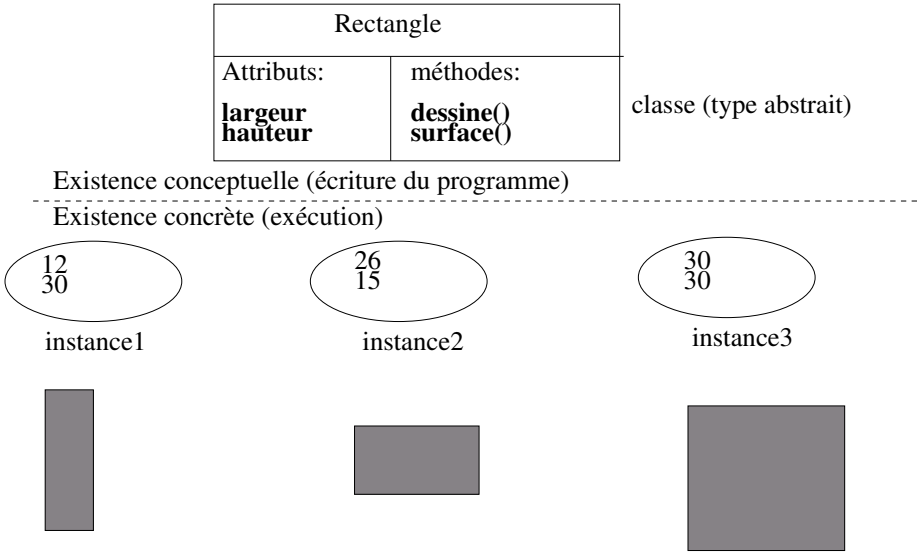
Classes et Types

En programmation Objet :

- le résultat du processus d'abstraction s'appelle une **classe**
  classe = catégorie d'objets
- une classe définit un **type** (au sens du langage de programmation)
- une réalisation particulière d'une classe s'appelle une **instance**
  instance = objet

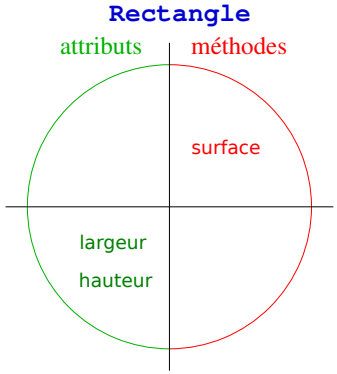
Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Classes v.s. instances

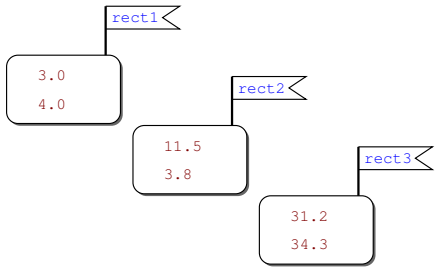


Objectifs
En cours
Encapsulation
Abstraction
Interface
Synthèse
Classes, Objets, Instances
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Classes et Instances, Types et Variables (illustration)



classe
type (abstraction)
existence conceptuelle
(écriture du programme)



objets/instances
variables en mémoire
existence concrète
(exécution du programme)

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Les classes en Java

En Java une classe se déclare par le mot-clé `class`.
Exemple :

```
class Rectangle
{
        //...
}
```

Par convention, un nom de classe commence par une majuscule.

La déclaration d'une instance d'une classe se fait de la façon similaire à la déclaration d'une variable classique :

```
nom_classe nom_instance ;
```

Exemple :

```
Rectangle rect1;
```

déclare une instance `rect1` de la classe `Rectangle`.

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Où déclarer les classes ?

Deux possibilités. . .

1. Déclaration de plusieurs classes dans le même fichier :
   - La classe `Rectangle` et le programme qui l'utilise `Dessin` déclarées dans `Dessin.java`

```
class Dessin
{
   public static void main (String[] args)
      {
         Rectangle r;
         // ....
      }
}
class Rectangle
{
   double largeur;
   double hauteur;
}
```

   - Le compilateur crée un fichier `.class` pour chaque classe :

```
$>$ javac Dessin.java
$>$ ls
Dessin.java
Dessin.class  Rectangle.class
```

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Où déclarer les classes ? (2)

2. Déclaration de chaque classe dans un fichier à part :
   - `Rectangle` déclarée dans `Rectangle.java`
   - `Dessin` déclarée dans `Dessin.java`
   - Compilation de *chaque* fichier nécessaire

```
$>$ javac Rectangle.java
$>$ javac Dessin.java
$>$ ls
Rectangle.java  Rectangle.class
Dessin.java  Dessin.class
$>$ java Rectangle
Exception in thread "main"
          java.lang.NoSuchMethodError: main
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Déclaration des attributs

Les attributs d'une classe, aussi appelé ses **variables d'instance** se déclarent selon la syntaxe suivante :

```
type nom_attribut ;
```

Exemple : les attributs `hauteur` et `largeur`, de type `double`, de la classe `Rectangle` pourront être déclarés par :

```
class Rectangle
{
        double hauteur;
        double largeur;
}
```

Les attributs en Java se déclarent au **début de la classe**.

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Accès aux attributs

L'accès aux valeurs des attributs d'une instance de nom `nom_instance` se fait par la notation pointée

```
nom_instance.nom_attribut
```

Exemple : la valeur de l'attribut `hauteur` d'une instance `rect1` de la classe `Rectangle` sera référencée par l'expression :

```
rect1.hauteur
```

`rect1` est une instance de `Rectangle` :

```
Rectangle rect1 = ...;
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Déclaration-intialisation d'une instance

L'instruction :

```
nomClasse instance = new nomClasse();
```

crée une instance de type `nomClasse` et initialise tous ses attributs avec des valeurs par défaut :

`int`	`0`
`double`	`0.0`
`boolean`	`false`
objets	`null`

Exemple :

```
Rectangle rect = new Rectangle();
```

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Notre programme (1/3)

```
class Exemple
{
  public static void main (String[] args)
  {
    Rectangle rect1 = new Rectangle();

    rect1.hauteur = 3.0;
    rect1.largeur = 4.0;

    System.out.println("hauteur : " + rect1.hauteur);
  }
}
class Rectangle
{
  double hauteur;
  double largeur;
}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Déclaration des méthodes

La syntaxe de la définition des méthodes d'une classe est similaire à celle des méthodes auxiliaires **sans le mot-clé `static`** :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) {
  // corps de la méthode
  ...
}
```

De plus, a **pas besoin de passer les attributs** de la classe comme arguments aux méthodes de cette classe
(on y revient dans une minute)

Exemple : une méthode `surface()` de la classe `Rectangle` pourrait être définie par :

```
class Rectangle {
    //...
    double surface() {
      return (hauteur * largeur);
    }
}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Portée des attributs

Les attributs d'une classe constituent des variables directement accessibles dans toutes les méthodes de la classes (*i.e.* des variables globales à la classe). Il n'est donc **pas nécessaire de les passer comme arguments des méthodes**.

Par exemple, dans toutes les méthodes de la classe `Rectangle`, l'identificateur `hauteur` (resp. `largeur`) fait donc a priori référence à la valeur de l'attribut `hauteur` (resp. `largeur`) de la classe.

Exemple : définition d'une méthode `surface` pour la classe `Rectangle`

```
class Rectangle {
   double hauteur;
   double largeur;
   double surface() {
      // PAS BESOIN DE METTRE largeur et hauteur
      // en argument de la methode
     return (hauteur * largeur);
   }
}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Déclaration des méthodes

Les méthodes sont donc :

- des fonctions propres à la classe
- qui ont donc accès aux attributs de la classe

☞ Il ne faut donc **pas** passer les attributs comme arguments aux méthodes de la classe !

Exemple :

```
class Rectangle {
  //...
  double surface() {
    return hauteur * largeur;
  }
}
```

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Paramètre des méthodes

A noter que ce n'est pas parce qu'on n'a pas besoin de passer les valeurs des attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont jamais d'arguments.

Les méthodes **peuvent très bien avoir des arguments** : ceux qui sont nécessaires (et donc **extérieurs à l'instance**) pour exécuter la méthode en question !

```
class Couleur { ... };

class FigureColoree
{
        //...
        void colorie(Couleur c) { ... }
}

FigureColoree une_figure;
Couleur rouge;
//...
une_figure.colorie(rouge); // On revient sur cette notation
//...
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Appel aux méthodes

L'appel aux méthodes définies pour une instance de nom `nom_instance` se fait à l'aide d'expressions de la forme :

```
nom_instance.nom_methode(val_arg1,...)
```

Exemple : la méthode

```
void surface()
{
        //...
}
```

définie pour la classe `Rectangle` peut être appelée pour une instance `rect1` de cette classe par :

```
rect1.surface()
```

Autre exemple (vu précédemment) :

```
figure.colorie("Rouge");
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

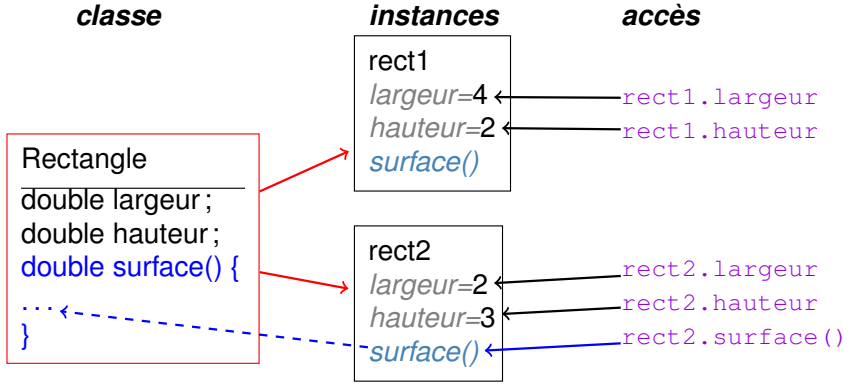
Notre programme (2/3)

```
class Exemple
{
  public static void main (String[] args)
  {
    Rectangle rect1 = new Rectangle();

    rect1.hauteur = 3.0;
    rect1.largeur = 4.0;

    System.out.println("surface : " + rect1.surface());
  }
}
class Rectangle
{
  double hauteur;
  double largeur;
  double surface() {
    return hauteur * largeur;
  }
}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Résumé : Accès aux attributs et méthodes

Chaque instance a ses propres attributs : aucun risque de confusion d'une instance à une autre.



Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Encapsulation et interface

Tout ce qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé `private` :

```
class Rectangle {
   // on affine l'encapsulation
   // utilisation du mot-cle : private
   private double hauteur;
   private double largeur;
        // ...
}
```

Attribut d'instance **privée** = inaccessible depuis l'extérieur de la classe. C'est également valable pour les méthodes.

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Encapsulation et interface (2)

Il y a erreur de compilation si l'on essaie d'accéder à un(e) attribut/méthode d'instance privée :

```
class Rectangle {
   private double hauteur;
   //..
}
//..
class Test
{
   public static void main(String[] args) {
      Rectangle unRectangle = ...;

      System.out.println(unRectangle.hauteur);
   }

}
```

```
error: hauteur has private access in Rectangle
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Encapsulation et interface (3)

À l'inverse, l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé `public`

```
class Rectangle {
        // ...
    // public =  accessible en dehors de
        // la classe Rectangle,
        // fait partie de l'interface d'utilisation
    public double surface() { ... }
}
```

Dans la plupart des cas :

- **Privé** :
  - Tous les attributs
  - La plupart des méthodes
- **Publique** :
  - Quelques méthodes bien choisies (interface)

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Droit d'accès `public` et droit par défaut

Les programmes Java sont généralement organisés au moyen de la notion de paquetage (`package`) → semestre de printemps
[Si vous ne spécifiez rien, vous travaillez dans le `package` **par défaut**. C'est le cas en TP]

Soit `C` une classe définie dans un paquetage `P` et `m`, un membre de `C` : si vous ne spécifiez aucun droit d'accès pour `m`, ce dernier est publiquement accessible par toutes les autres classes de `P` mais **pas en dehors de `P` !** (alors qu'il le serait si il était explicitement déclarée comme publique)

Attention : les exemples du cours omettent parfois de mettre explicitement `public` pour cette raison (et pour la concision des descriptions)

☞ Il est recommandé de mettre explicitement `public` devant tout membre que vous estimez devoir appartenir à l'interface de la classe

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Méthodes «get» et «set» (« accesseurs » et « manipulateurs »)

- Tous les attributs sont privés ?
  - Et si on a besoin de les utiliser depuis l'extérieur de la classe ?!
- Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...
  1. Manipulateurs (« méthodes set ») :
     - Modification
     - Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h;}

void setLargeur(double l) { largeur = l;}
```

  2. Accesseurs (« méthodes get ») :
     - Consultation
     - Retour de la valeur d'une variable d'instance précise

```
double getHauteur() { return hauteur;}

double getLargeur() { return largeur;}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Notre programme (3/3)

```
class Exemple
{
  public static void main (String[] args)
  {
    Rectangle rect1 = new Rectangle();

    rect1.setHauteur(3.0);
    rect1.setLargeur(4.0);

    System.out.println("hauteur : "
                       + rect1.getHauteur());
  }
}
```

☞ suite sur le transparent suivant

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Notre programme (3/3)

```
class Rectangle
{
  public double surface()
    { return hauteur * largeur; }

  public double getHauteur()
    { return hauteur; }
  public double getLargeur()
    { return largeur; }

  public void setHauteur(double h)
    { hauteur = h; }
  public void setLargeur(double l)
    { largeur = l; }

  private double hauteur;
  private double largeur;
}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Getters/setters et encapsulation

Lassant d'écrire des getters/setters alors que l'on pourrait tout laisser en `public` ?

```
class Personne
{
   public int age;
   public String nom;
}
```

Oui mais dans ce cas ...

```
Personne toto = new Personne();
toto.age = -36;   // OH NON !
System.out.print(toto.nom.length()); //et si le programmeur de
                                     //Personne decide que nom
                                     //n'est plus une String?
                                     //-> length() n'existe
                                     //plus forcement !
```

Si par contre `age` et `nom` sont bien mis (comme il se doit) en `private` :

☞ une méthode `setAge` peut être programmée de sorte à contrôler que la valeur de l'âge soit plausible !

☞ l'utilisateur de la classe `Personne` n'a plus le droit d'utiliser `nom` comme une `String`. Ceci laisse au programmeur de `Personne` le loisir de modifier ses choix d'implémentation sans porter préjudice à ses utilisateurs.

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Getters/setters : Mise en garde

- Un code contenant trop d'accesseurs (méthodes get et set) est cependant souvent mal conçu.
  - ☞ Les méthodes de l'interface ne devraient rendre visible que ce qui utile à l'utilisateur (voir l'exemple du jeu de Morpion, un peu plus loin).
- Une utilisation abusive des accesseurs peut "casser" l'encapsulation (voir aussi l'exemple du Morpion).

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Masquage (shadowing)

Masquage = un identificateur « cache » un autre identificateur

Situation typique : le nom d'un paramètre cache le nom d'un attribut

```
void setHauteur(double hauteur) {
    hauteur = hauteur;  //Hmm... pas terrible !
}
```


Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Masquage (2)

Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé `this`.

`this` est un **pointeur sur l'instance courante**

`this` ≃ « mon adresse »

Syntaxe pour spécifier un attribut en cas d'ambiguïté :

```
this.nom_attribut
```

Exemple :

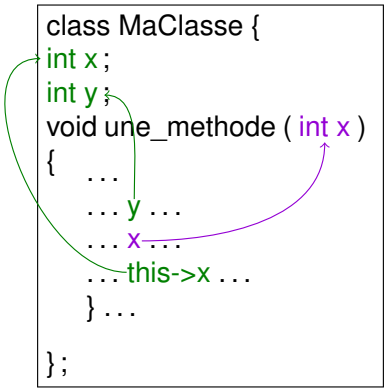
```
void setHauteur(double hauteur) {
    this.hauteur = hauteur;  // fonctionne !
}
```

L'utilisation de `this` est obligatoire dans les situations de **masquage**
(mais évitez ces situations !)

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Portée des attributs (résumé)

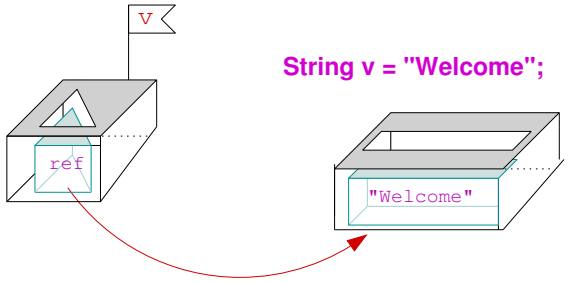
La portée des attributs dans la définition des méthodes peut être résumée dans le schéma suivant :



Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Objets en mémoire

Attention : Comme nous l'avons déjà vu pour la classe prédéfinie `String` (et pour les tableaux), les objets, contrairement aux entités de types élémentaires, sont manipulés via des **références** :



Il est impératif de s'en rappeler lorsque l'on :

- *compare* deux objets
- *affecte* un objet à un autre
- *affiche* un objet

Objectifs
En cours
En Java
class
Attributs
Méthodes
Encapsulation et interface
Accesseurs/Manipulateurs
Masquage et this
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

La constante `null`

La constante prédéfinie `null` peut être affectée à n'importe quel objet de n'importe quelle classe.

Affectée à une variable, elle indique que celle-ci ne référence aucun objet :

```
Rectangle  rect = null; // la variable rect
                        // ne reference aucun objet
```

Avant de commencer à utiliser un objet, il est souvent judicieux de tester s'il existe vraiment, ce qui peut se faire par des tournures telles que :

```
if (rect == null) {...}
if (rect != null) {...}
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Où en est-on ?

Nous savons maintenant déclarer une classe d'objets.

et comment déclarer une instance de la classe `Rectangle` :

```
Rectangle rect;
```

Une fois que l'on a fait ce genre de déclaration, on dispose d'une variable de type `Rectangle`. Mais comment donner une valeur à cette variable ?

☞ Comment réserver de l'espace pour les attributs et les comment initialiser (i.e. leur attribuer des valeurs) ?

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Initialisation des attributs

Première solution : **affecter individuellement une valeur** à chaque attribut

```
//....
Rectangle rect;
double lu;
System.out.print("Quelle hauteur? ");
lu = keyb.nextDouble();
rect.setHauteur(lu);
System.out.print("Quelle largeur? ")
lu = keyb.nextDouble();
rect.setLargeur(lu);
```

Ceci est une *mauvaise solution* dans le cas général :

- si le nombre d'attributs est élevé, ou si le nombre d'objets créés est grand (rectangles `r[0]`, `r[1]`, ..., `r[99]`) elle est fastidieuse et source d'erreurs (oublis)
- elle implique que tous les attributs fassent partie de l'interface (`public`) ou soient assortis d'un manipulateur (`set`)

Initialisation des attributs (2)

Deuxième solution : définir une **méthode dédiée à l'initialisation** des attributs

```
class Rectangle {
   private double hauteur;
   private double largeur;

   // Methode d'initialisation !
   public void init(double h, double l) {
      hauteur = h; largeur = l;
   }
}
```

En fait, Java fait déjà le travail pour vous en fournissant des méthodes particulières appelées **constructeurs**

Un constructeur réalise toutes les opérations requises en « début de vie » de l'objet.

Les constructeurs

Un constructeur est une méthode :

- invoquée au moyen du mot-clé `new` lors de la déclaration d'un objet (instanciation d'une classe)
- assurant *l'initialisation des attributs*.

Syntaxe de base :

```
NomClasse(liste_arguments)
{
 /* initialisation des attributs
    utilisant  liste_arguments */
}
```

Exemple :

```
Rectangle(double h, double l) {
   hauteur = h;
   largeur = l;
 }
```


Les constructeurs (2)

Les constructeurs sont donc des méthodes (presque) comme les autres

- que l'on peut surcharger
  Une classe peut donc avoir **plusieurs constructeurs**, pour peu que leur liste d'arguments soient différentes ;
  (exemple dans la suite).

Ce qu'il faut noter de particulier sur la syntaxe des constructeurs :

- pas d'indication de type de retour (pas même `void`)
- doit avoir le même nom que la classe pour laquelle ils sont définis

Initialisation par constructeur

La *déclaration avec initialisation* d'un objet se fait selon la syntaxe suivante :

Syntaxe :

```
NomClasse instance = new NomClasse(valarg1, ..., valargN);
```

où `valarg1, ..., valargN` sont les valeurs des arguments du constructeur.

Exemple :

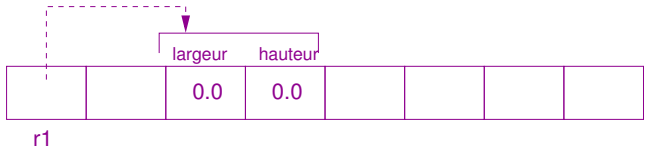
```
// new Rectangle(...) :  invocation du constructeur
Rectangle r1 = new Rectangle(18.0, 5.3);
```

Construction d'un objet

1. `Rectangle r1;` :
   - Réservation de mémoire pour une référence vers un objet de type `Rectangle`
2. `new` :
   - Réservation de mémoire pour les variables d'instance
   - Affectation de valeurs par défaut :

`int`	`0`
`double`	`0.0`
`boolean`	`false`

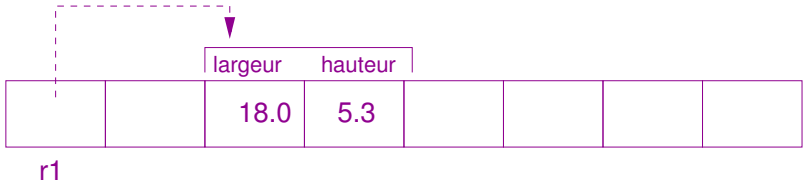
Situation en mémoire :



Construction d'un objet (2)

3. `Rectangle(18.0, 5.3)` :
   - Appel à la méthode constructeur
4. Exécution de la méthode constructeur :
   - Toute instruction possible
   - Typiquement affectations aux variables d'instance

Situation en mémoire :



Constructeur par défaut

Un constructeur par défaut est un constructeur qui **n'a pas d'argument**.

Exemples :

```
// Le constructeur par defaut
Rectangle() { hauteur = 1.0; largeur = 2.0;}
// 2eme constructeur (pas par defaut)
Rectangle(double c) { hauteur = c; largeur = 2.0*c;}
// 3eme constructeur (pas par defaut)
Rectangle(double h, double l) { hauteur = h; largeur = l;}
```


Constructeur par défaut par défaut

Si aucun constructeur n'est spécifié, une **version minimale du constructeur par défaut** est *générée automatiquement*
(c'est donc un constructeur par défaut par défaut) qui :

- initialise les attributs avec les valeurs par défaut.

Attention : Dès qu'**au moins un constructeur a été spécifié**, ce constructeur par défaut par défaut *n'est plus fourni*.

A :

```
class Rectangle {
  private double
hauteur;
  private double
largeur;
  // suite ...
}
```

B :

```
class Rectangle {
  private double
hauteur;
  private double
largeur;

  public
Rectangle()
  {
    hauteur = 0.0;
    largeur = 0.0;
  }
  // suite ...
}
```

C :

```
class Rectangle {
  private double hauteur;
  private double largeur;

  public Rectangle(double h,
                   double l)
  {
    hauteur = h;
    largeur = l;
  }
  // suite ...
}
```


Constructeur par défaut : exemples

	constructeur par défaut	`Rectangle r1 = new Rectangle();`	`Rectangle r2 = new Rectangle(1.0,2.0);`
(A)	constructeur par défaut par défaut	0.0 | 0.0	Illicite !
(B)	constructeur par défaut explicitement déclaré	0.0 | 0.0	Illicite !

Constructeur par défaut : exemples

	constructeur par défaut	`Rectangle r1 = new Rectangle();`	`Rectangle r2 = new Rectangle(1.0,2.0);`
(A)	constructeur par défaut par défaut	0.0 | 0.0	Illicite !
(B)	constructeur par défaut explicitement déclaré	0.0 | 0.0	Illicite !
(C)	pas de constructeur par défaut	Illicite !	1.0 | 2.0

Constructeur par défaut : exemples

	constructeur par défaut	`Rectangle r1 = new Rectangle();`	`Rectangle r2 = new Rectangle(1.0,2.0);`
(A)	constructeur par défaut par défaut	0.0 | 0.0	Illicite !

```
class Rectangle {
}
```

Appel aux autres constructeurs

Java autorise les constructeurs d'une classe à appeler n'importe quel autre constructeur de cette même classe

Syntaxe : `this(...);`

```
class Rectangle {
  private double hauteur;
  private double largeur;

  public Rectangle(double h, double l)
  {
      hauteur = h;
      largeur = l;
  }
  public Rectangle() {
    // appel du constructeur a deux arguments
    this(0.0, 0.0);
  }
  // suite ...
```


Valeur par défaut pour les attributs

Il est également possible de donner une valeur par défaut aux attributs à l'endroit de leur déclaration

`Type nomAttribut = valeur;`

Exemple

```
class Rectangle {
   private double largeur = 10.0;
   private double hauteur = 10.0;
   public Rectangle(){// rien
   }
   public Rectangle (double l, double h)
      {
         largeur = l;
         hauteur = h;
      }
}
```

- ☞ Les attributs non intialisés explicitement par un constructeur auront alors ces valeurs.
- ☞ Si le constructeur par défaut par défaut a disparu, il est recommandé dans ce cas de définir un constructeur par défaut, même avec un corps vide.

Constructeur de copie

Java vous fournit également un moyen de créer la copie d'une instance : le *constructeur de copie*

Ce constructeur permet d'initialiser une instance en utilisant les **attributs d'une autre instance** du même type.

Syntaxe :

```
class UneClasse {
    // Constructeur de copie:
    public UneClasse(UneClasse obj) {
    //...
   }
}
```

Il a pour rôle de créer un **objet distinct** de celui passé en paramètre mais ayant exactement les mêmes valeurs pour ses attributs.

Constructeur de copie (2)

```
class Rectangle {
    private double hauteur;
    private double largeur;
        //...
    public Rectangle(Rectangle autreRectangle){
      hauteur = autreRectangle.hauteur;
      largeur = autreRectangle.largeur;
    }
}

// ...
Rectangle r1 = new Rectangle(2.0, 5.5);
Rectangle r2 = new Rectangle(r1);
```

En Java, il n'y a pas de constructeur de copie par défaut (automatiquement généré).

Note : Le constructeur de copie n'est pas la seule façon de créer une copie d'objet en Java. Le moyen (recommandé) est d'avoir recours à la méthode `clone` que nous verrons un peu plus tard.

Portée de classe / Portée d'instance

Au fait dans le code qui a précédé, pourquoi peut-on accéder à l'attribut privé `hauteur` de `autreRectangle` sans passer par un getter :

```
class Rectangle {
    private double hauteur;
    private double largeur;
   //...
    public Rectangle(Rectangle autreRectangle){
      hauteur = autreRectangle.hauteur; //???
      largeur = autreRectangle.largeur;
    }
}
```

En Java, une classe a accès aux membres privés de **toutes ses propres instances** (aussi bien l'objet `this` que **d'autres objets** de la classe

☞ **Portée de classe** (par opposition à la "portée d'instance")

Fin de vie d'un objet

Un objet est en fin de vie lorsque le programme n'en a plus besoin

☞ la référence qui lui est associée n'est plus utilisée nulle part

Exemple

```
class Exemple
{
   public static void main(String[] args) {
         // autres traitements
         afficherUnRectangle();
         //...
   }

   static void afficherUnRectangle() {

         Rectangle r = new Rectangle(3.0, 4.0);
         System.out.println(r);
   }
}
```

☞ Le rectangle `r` n'est plus référencé nulle part une fois que la méthode `afficherUnRectangle` a fini son exécution !

☞ Récupération de la mémoire associée à l'objet

Garbage Collection

- **Garbage collection** = processus qui récupère la mémoire occupée par les objets qui ne sont plus référencés par aucune variable
  - Egalement appelé ramasse-miettes
  - Nécessaire pour éviter les fuites de mémoire : indisponibilité de zones mémoires qui ne sont plus utilisées
- Le garbage collector est lancé régulièrement pendant l'exécution d'un programme Java
- D'autres langages de programmation (par exemple C++) n'ont pas de ramasse-miettes

☞ De façon générale, le programmeur Java n'a pas à libérer explicitement la mémoire utilisée. Ceci ne veut pas dire qu'il doit en faire un usage inconsidéré !

☞ La bonne gestion de la mémoire et le rôle du garbage collector sont des sujets dépassant le cadre de ce cours

Durée de vie et portée

En Java : **la durée de vie peut dépasser la portée**

```
class A {/*...*/}
class B {
   A monA; // indirection (reference)
           // vers un objet A
}
//....
B unB = new ...; // creation d'un objet B
{ // bloc
   A unA ...;       // creation d'un objet A
   unB.monA = unA; // stockage de la reference
                   // (adresse)  de unA dans
                   //unB.monA
} // fin du bloc
```

après le bloc :

- `unA` n'est plus accessible (fin de portée)
- l'objet qui était associé à `unA` continue d'exister car référencé par `unB.monA`

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Durée de vie et portée (2)

En C++ : la **durée de vie** est strictement **égale à la portée**

```
class A {/*...*/}
class B {
   A* monA; // indirection (pointeur)
            // vers un objet A
}
//....
B unB ...; // creation d'un objet B

{ // BLOC
   A unA ...;        // creation d'un objet A
   unB.monA = &unA; // stockage de l'adresse
                     // de unA dans  unB.monA
} // fin du bloc
```

après le bloc :

- `unA` n'est plus accessible (fin de portée)
- l'objet qui était associé à `unA` est détruit
- `unB.monA` pointe vers un objet invalide

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Objets en mémoire

Attention : Comme nous l'avons déjà vu pour la classe prédéfinie `String` (et pour les tableaux), les objets, contrairement aux entités de types élémentaires, sont manipulés via des **références** :

- Variable de type élémentaire :

```
int i = 45;
double d = 7.12;
boolean b = true;
```

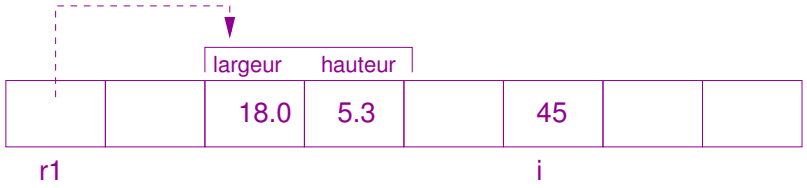
  - Réservation de mémoire pour la valeur

- Variable de type objet (2 étapes) :
  - Réservation de mémoire pour une **référence** vers les variables d'instance
  - Réservation de mémoire pour les variables d'instance

```
Rectangle rect = new Rectangle(6.0, 4.5);
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Objets en mémoire (2)




```
System.out.println(i); // 45
System.out.println(rect); // 80ca178
System.out.println(rect.largeur); // 6.0
```

En java les objets sont toujours manipulés via des **références**
(ceci n'est pas forcément le cas dans d'autres langages).
Il est impératif de s'en rappeler :

- Quand on compare deux objets ;
- Quand on affecte la valeur d'une variable à une autre variable ;
- Quand on modifie un paramètre ;
- Quand on affiche un objet

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Variables de type élémentaire vs Objets

Type élémentaire

```
int i = 1;
int j = 2;
System.out.println(j); // 2
j = i;                 // affectation d'une valeur
System.out.println(j); // 1
i = 3;
System.out.println(j); // 1
```

Objet

```
Rectangle r1 = new Rectangle(1.0, 2.0);
Rectangle r2= new Rectangle(3.0, 4.0);
System.out.println(r2.largeur); // 3.0
r2 = r1; // copie reference
System.out.println(r2.largeur); // 1.0
r1.largeur = 5.0;
System.out.println(r2.largeur); // 5.0 !!!
```

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Affectation/copie d'objets

Supposons que l'on souhaite :

- créer un objet `b` à partir d'un autre objet `a` du même type ;
- assurer que `a` et `b` soient deux objets **distincts** en mémoire

Première tentative :

```
Joueur joueurA = new Joueur(...);
Joueur joueurB = joueurA; // Ceci ne fait pas
                          // l'affaire!
```

`joueurB` et `joueurA` référencent ici le **même objet** et non pas deux copies distinctes du même objet : toute modification via `joueurB` sera visible via `joueurA`.

Si l'on veut que l'objet référencé par `joueurB` soit une copie distincte de celui référencé par `joueurA`, il ne faut pas utiliser l'opérateur `=`.

On peut avoir recours à la place au **constructeur de copie** (ou mieux à la méthode `clone` qui sera vue plus tard).

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Affichage d'objets

La portion de code suivante :

```
UneClasse c1 = new UneClasse(...);
System.out.println(c1);
```

afficherait la valeur de la référence (une sorte d'adresse bizzare).

Que faire si l'on souhaite faire afficher le contenu de l'objet en utilisant exactement le même code ?

Réponse : Java prévoit que vous fournissiez une méthode qui retourne une représentation de l'instance sous forme d'un `String`.

Il est prévu que vous donniez une entête particulière à cette méthode :

```
String toString ()
```

(La méthode `toString` est invoquée automatiquement sur l'objet par `System.out.println`)

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Affichage d'objets : la méthode `toString`

Exemple :

```
class Joueur {
   private String pseudo;
   private double score;
   public Joueur(String unPseudo){
      pseudo = unPseudo;
      score = 0.0;
   }
   public String toString(){
      return  pseudo + " is the Best";
   }
}

class Exemple {
   public static void main(String[] args){
      Joueur unJoueur = new Joueur("Darkvador99");
      System.out.println(unJoueur );
     // affiche "Darkvador99 is the Best"
   }
}
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Affichage d'objets : la méthode `toString` (2)

Vous souvenez-vous de l'exemple du cours 4 ?

```
String chaine = "Welcome";
System.out.print(chaine);
```

Vous disposez maintenant des outils pour comprendre pourquoi ce code affiche `Welcome` au lieu d'afficher une adresse.

☞ `String` est une classe, et cette classe définit une méthode `toString`

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Comparaison d'objets

L'opérateur `==` appliqué à deux objets compare les références de ces objets (nous en avons déjà parlé pour les `String`)

Que faire si l'on souhaite comparer le contenu de deux objets de type `UneClasse` ?

Réponse : il faut fournir une méthode qui fasse le test selon les critères qui semblent sensés pour les objets de type `UneClasse`.

Java prévoit que vous donniez une entête particulière à cette méthode : `boolean equals (UneClasse arg)`

(oui, `equals`, exactement pareil que pour les `String`. En toute rigueur l'entête attendue est légèrement différente. Nous y reviendrons au cours prochain.)

Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Comparaison d'objets : méthode `equals`

Exemple :

```
class Joueur {
   private String pseudo;
   private double score;
   public Joueur(String unPseudo){
      pseudo = unPseudo;
      score = 0.0;
   }
   //Comment comparer deux joueurs
   public boolean equals(Joueur autreJoueur){
      if (autreJoueur == null) {
         return false;
      }
      else
        return (pseudo.equals(autreJoueur.pseudo)
               && score == autreJoueur.score);
   }
}
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Affectation
Affichage
Comparaison
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Comparaison d'objets : méthode `equals` (2)

Exemple (suite) :

```
// Par exemple dans main:
Joueur j1 = new Joueur("Bladerunner69");
Joueur j2 = new Joueur("Bladerunner69");
System.out.println(j1 == j2); //affiche false
System.out.println(j1.equals(j2));//affiche true
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Exemple : jeu de Morpion (1)

On veut coder une classe permettant de représenter le plateau 3x3 d'un jeu de Morpion (tic-tac-toe) :

```
class JeuMorpion {
   private int[] grille;
   public void initialise() {
      grille = new int[9];
   }
   public int[] getGrille() {
      return grille;
   }
}
```

Exemple : jeu de Morpion (2)

```
class JeuMorpion {
   private int[] grille;
   public void initialise() {
      grille = new int[9];
   }
   public int[] getGrille() {
      return grille;
   }
}
```

Le joueur rond coche la case en haut à gauche :

```
JeuMorpion jeu = new JeuMorpion();
jeu.initialise();
jeu.getGrille()[0] = 1;
```

Convention : 1 représente un rond, 2 une croix et 0 une case vide

Exemple : jeu de Morpion (3)

Ce code est parfaitement fonctionnel mais ... pose **beaucoup** de problèmes :

- L'utilisateur de la classe `JeuMorpion` doit savoir que les cases sont stockées sous forme d'entiers dans un tableau 1D, ligne par ligne (et non colonne par colonne)
- Il doit savoir que la valeur entière 0 correspond à une case non cochée, que 1 correspond à un rond, et que la valeur 2 correspond à une croix.

☞ L'utilisateur doit connaître « le codage » des données

```
JeuMorpion jeu = new JeuMorpion();
jeu.initialise();
jeu.getGrille()[0] = 1;
```


Exemple : jeu de Morpion (4)

```
JeuMorpion jeu = new JeuMorpion();
jeu.initialise();
jeu.getGrille()[0] = 1;
```

- Le code est complètement cryptique pour une personne qui n'est pas intime avec les entrailles du programme. 0, 1, 2 ? Que cela signifie-t-il ? Impossible de le deviner juste en lisant ce code. Il faut aller lire le code de la classe `JeuMorpion` (ce qui devrait être inutile), et en plus ici `JeuMorpion` n'est même pas documentée !
- Le code n'est pas encapsulé : on a un accesseur public vers une variable privée, donc... on ne peut pas la modifier, non ? Malheureusement si : c'est un tableau, donc on peut directement modifier son contenu ce qui viole l'encapsulation.
- Que se passerait-il si pour représenter le plateau de jeu, on décidait de changer et d'utiliser un tableau 2D ? Ou 9 variables entières ?
  - ☞ Le code écrit par l'utilisateur de la classe `JeuMorpion` serait à réécrire !

Exemple : jeu de Morpion (5)

- Si l'utilisateur s'avisait de faire `jeu.getGrille()[23] = 1;` il aurait un message d'erreur (un `ArrayIndexOutOfBoundsException`)
- Si l'utilisateur avait envie de mettre la valeur 3 ou 11 ou 42 dans le tableau, rien ne l'en empêche - mais d'autres méthodes, comme par exemple `getJoueurGagnant()`, qui s'attendent uniquement aux valeurs 1 et 2 ne fonctionneront plus du tout !
- Si l'utilisateur avait envie de tricher et de remplacer un rond par une croix ? Il suffit d'écraser la valeur de la case avec la valeur 2 !

☞ Les méthodes choisies ici donnent un accès non contrôlé aux données et n'effectuent aucune **validation** des données

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Jeu de Morpion : bien encapsuler (1)

```
class JeuMorpion {
   private final static int VIDE  = 0;
   private final static int ROND  = 1;
   private final static int CROIX = 2;

   private int[][] grille;

   public void initialise() {
      grille = new int[3][3];
      for (int i=0; i < grille.length; ++i) {
         for (int j=0; j < grille[i].length; ++j)
         {
            grille[i][j] = VIDE;
         }
      }
   }
//...
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Jeu de Morpion : bien encapsuler (2)

```
/**
 * Place un coup sur le plateau.
 * @param ligne La ligne 0, 1, ou 2
 * @param colonne La colonne 0, 1, ou 2
 * @param coup Le coup a placer
 */
private boolean placerCoup(int ligne, int colonne, int coup) {
        if (ligne < 0 || ligne >= grille.length
        || colonne < 0 || colonne >= grille[ligne].length) {
     // traitement de l'erreur ici
        }
        if(grille[ligne][colonne] == VIDE) {
                // case vide, on peut placer le coup
                grille[ligne][colonne] = coup;
                return true;
        } else {
      // case deja prise, on signale une erreur
      //...
          return false;
        } // suite
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Jeu de Morpion : bien encapsuler (3)

```
public boolean placerRond(int ligne, int colonne) {
   return placerCoup(ligne, colonne, ROND);
}

public boolean placerCroix(int ligne, int colonne) {
   return placerCoup(ligne, colonne, CROIX);
}

// ici on peut rajouter une methode getJoueurGagnant()

} // fin de la classe JeuMorpion
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Jeu de Morpion : bien encapsuler (4)

Comment faire maintenant pour faire un rond sur la case du milieu ?

```
JeuMorpion jeu = new JeuMorpion();
jeu.initialise();
valide = jeu.placerRond(0, 0); //boolean d'eclar'e plus haut
```

Et pour faire une croix sur la 1re ligne, 2e colonne ?

```
valide = jeu.placerCroix(0, 1);
```

On aurait pu également décider d'appeler les colonnes 1, 2, 3 au lieu de 0, 1, 2 : c'est une question de convention. C'est justement ce sur quoi il faut se mettre d'accord quand on définit une interface.

Jeu de Morpion encapsulé : avantages

- **Validation** : il est impossible de mettre une valeur invalide dans le tableau (autre que 0, 1, ou 2)
- **Validation** : il est impossible de cocher une case déjà cochée.
- **Séparation des soucis** : le programmeur-utilisateur n'a pas besoin de savoir comment le plateau est stocké, ni qu'il utilise des entiers, ni quelles valeurs correspondent à quoi.
- Le code est compréhensible même par un profane – le nom des méthodes exprime clairement ce qu'elles font et s'explique de lui-même.
- Si on essaie de faire une opération invalide (cocher deux fois la même case, ou une case en dehors du tableau), on obtient un message compréhensible.

Problème 1

- un joueur à un *pseudonyme* et un *temps de jeu* ;
- il peut gagner des *points* ;
- son score se calcule comme son nombre de points divisé par son temps de jeu

Objectif : Ecrire un petit programme calculant le score d'un joueur

Solution au problème 1

```
import java.util.Scanner;

class Score {
      public static void main(String[] args) {
      Scanner keyb = new Scanner(System.in);
      System.out.println("Nom du joueur : ");
      String pseudo = keyb.nextLine();
      System.out.println("Temps de jeu : ");
      int temps = keyb.nextInt();
      System.out.println("Nombre de points");
      int points = keyb.nextInt();

      if ((temps >= 0) && (points>= 0)) {
         int score = 0;
         if (temps > 0)
            score  points / temps;
         System.out.println("Score de " + pseudo
                     + " :" + score);}
      else {
         System.out.println
            ("Erreur: temps de jeu ou points n'egatifs");}
   }
}
```


...en version modularisée

Modularisation des traitements : `main` + méthodes auxiliaires

```
class Score {
   public static void main (String[] args) {
         Scanner keyb = new Scanner(System.in);
      System.out.println("Nom du joueur : ");
      String pseudo = keyb.nextLine();
      int temps = lireEntier("Temps de jeu: ");
      int points = lireEntier("Nombre de points :");
      if (donneesOK(temps, points)) {
         afficherScore(pseudo, temps, points);
      }
      else {
         afficherErreur();
      }
   }
         //....
```

... avec les méthodes auxiliaires :

```
//...
   static int lireEntier(String texte) {
      Scanner keyb = new Scanner(System.in);
      System.out.println(texte);
      return keyb.nextInt();
   }

   static boolean donneesOK(int temps, int points) {
      return ((temps >= 0) && (points >= 0));
   }
   static void afficherScore(String pseudo, int temps, int points) {
      System.out.println ("Score de " + pseudo +
                          " :" + score(temps,points));

   }
   static int score (int temps, int points) {
         if (temps > 0)
            return (int)(points/temps);
         else return 0;
   }
   static void afficherErreur () {
      System.out.println
         ("Erreur: temps ou points n'egatifs" );
   }
}
```


Solution au problème 1 - Modularisation

Comparons un peu les deux options :

- `Classe Score non modularisée :`
  - 1 méthode
  - environ 15 lignes
- Classe `Score` modularisée :
  - 6 méthodes
  - environ 30 lignes

A quoi cela a-t-il servi ?

- Meilleure lisibilité
- Facilité d'entretien et de correction ou amélioration
- Réutilisabilité

Problème 2

Les données du problème sont les mêmes que précédemment.

Objectif : Ecrire un programme qui calcule le score moyen de 2 joueurs (ou plus ...)

Simplifications :

- Données spécifiées dans le code
- Pas de tests de validité

Solution possible :

- `main` + 2 méthodes auxiliaires principales

Score moyen : tentative

La méthode `main` :

```
class ScoreMoyen {
   public static void main(String[] args ) {
      // donn'ees du premier joueur
      String joueur1 = "Darkvador99";
      int  temps1 = 186;  // minutes de jeu
      int points1 = 1584; // points gagn'es

      // donn'ees du second joueur
      String joueur2 = "Bladerunner69";
      int  temps2 = 275;
      int  points2 = 1426;

      double  score =
         scoreMoyen(temps1, points1, temps2, points2);
      afficherScoreMoyen(joueur1, joueur2, score);
   }
   //...
```

Score moyen : tentative

. . . puis les méthodes auxiliaires :

```
//...
   static double scoreMoyen (int temps1, int points1,
                             int temps2, int points2) {
      return  (score(temps1, points1) +
               score(temps2, points2)) / 2;
   }

   static int score (int  temps, int points) {
      if (temps > 0)
         return (int)(points/temps);
      else return 0;
   }

   static void afficherScoreMoyen (String joueur1,
                                   String joueur2, double score) {
      System.out.println ("Joueurs : " + joueur1 + ", " +
                          joueur2);
      System.out.println ("Score moyen : " + score);
   }
}
```


Analyse

Instructions bien modularisées :

- 3 petites méthodes auxiliaires
- méthodes courtes faciles à lire et comprendre

Qu'en est-il des variables ? :

- pour chaque joueur : une variable pour son temps de jeu, une variable pour son nombre de points et une variable pour son pseudonyme
  - ☞ déjà **6 variables** dans la méthode `main` !
- le lien sémantique entre le temps, le nombre de points et le joueur n'est pas clairement établi

.. et s'il y avait 100 joueurs ?

Score moyen : un peu d'abstraction . . .

En fait, tous les joueurs sont caractérisés par les mêmes attributs :

- `temps` : un `int`
- `points` : un `int`
- `pseudonyme` : un `String`

On pourrait donc définir un type `Joueur` au moyen d'une classe :

```
class Joueur {
   int temps;    // temps de jeu du joueur
   int points; // nombre de points gagn'es
   String pseudo; // son pseudonyme

   // Pour initialiser un joueur
   // (ce n'est qu'une maniere possible)
   Joueur(String unPseudo, int unTemps, int desPoints) {
         pseudo = unPseudo;
         temps = unTemps;
         points = desPoints;
      }
}
```


Score moyen : les objets entrent en scène

Voici donc une première version objet de `ScoreMoyen` qui utilise le type d'objets `Joueur` :

```
class ScoreMoyenOO {
   public static void main(String[] args) {
      Joueur j1 = new Joueur("Darkvador99", 186, 1584);
      Joueur j2 = new Joueur("Bladerunner69", 275, 1426);
      double score = scoreMoyen(j1, j2);
      afficherScoreMoyen(j1, j2, score);
   }
   static double scoreMoyen(Joueur j1, Joueur j2) {
      return (score(j1) + score(j2)) / 2;
   }
   static int score(Joueur j) {
      if  (j.temps > 0)
         return (int)(j.points/j.temps);
      else return 0;
   }
   static void afficherScoreMoyen(Joueur j1, Joueur j2,
                                  double score) {
       System.out.println ("Joueurs : " + j1.pseudo + ", " +
                           j2.pseudo);
       System.out.println ("Score moyen : " + score);
      }
}
```

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Analyse

Transformation de `ScoreMoyen` en `ScoreMoyenOO` :

- Type particulier pour les joueurs sous forme d'une classe

Avantages :

- Méthode `main` plus courte, plus lisible
- Lien sémantique établi entre les données : temps de jeu, points gagné et "joueur"

On peut faire encore mieux :

- Si une méthode est spécifique à un objet, on peut l'attribuer à l'objet comme **méthode d'instance** : ici par exemple la méthode calculant le score du joueur !

☞ **encapsulation** des données et des traitements

Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Score moyen : encapsulation (1)

La méthode `score` devient spécifique au `Joueur` :

```
class Joueur
{
   int temps;
   int points;
   String pseudo;
   Joueur(String unPseudo, int unTemps,
        int desPoints) {
        // comme avant
      }
   // Methode specifique au joueur
   int score() {
         if (temps > 0)
            return (int)(points/temps);
         else return 0;
      }
}
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Score moyen : encapsulation (2)

La méthode `score` est maintenant encapsulée dans la classe `Joueur` :

```
class ScoreMoyenOO {
   public static void main(String[] args) {
     Joueur j1 = new Joueur("Darkvador99", 186, 1584);
    Joueur j2 = new Joueur("Bladerunner69", 275, 1426);
    double score = scoreMoyen(j1, j2);
    afficherScoreMoyen(j1, j2, score);
   }
  static double scoreMoyen(Joueur j1, Joueur j2) {
     // le calcul du score est maintenant une methode
     // de la classe Joueur
      return (j1.score() + j2.score()) / 2;
   }
   static void afficherScoreMoyen(Joueur j1, Joueur j2,
                          double score) {
      // comme avant
   }
}
```


Objectifs
En cours
En Java
Constructeurs
Objets en java
Bien encapsuler (annexe)
Bases de la POO : exemple (annexe)
Conclusion

Score moyen : encapsulons encore . . .

On peut faire encore mieux ! !

- Un bon programme OO **cache** les détails d'implémentation d'une classe : les modalités de calcul du score doivent pouvoir changer sans que cela n'influe sur les utilisateurs de la classe `Joueur`
- Tout ce qu'il n'est pas nécessaire de connaitre depuis l'extérieur d'un objet devient **`private`**.
  - ☞ Voir le transparent suivant

Score moyen : encapsulons encore . . .

```
class Joueur
{
   // PLUS D'ACCES DIRECT DEPUIS L'EXTERIEUR !!!
   private int temps;
   private int points;
   private String pseudo;

   // LES METHODES DE L'INTERFACE
   public Joueur(String unPseudo, int unTemps,
                                  int desPoints) {
           // comme avant
      }
   public int score() {
       // comme avant
      }
   // pour avoir acces au pseudonyme
   public getPseudo() {
         return pseudo;
      }
}
```


Score moyen : code final

```
class ScoreMoyenOO1 {
   public static void main(String[] args) {
     Joueur j1 = new Joueur("Darkvador99", 186, 1584);
     Joueur j2 = new Joueur("Bladerunner69", 275, 1426);
     double score = scoreMoyen(j1, j2);
     afficherScoreMoyen(j1, j2, score);
   }
  static double scoreMoyen(Joueur j1, Joueur j2) {
     // comme avant
   }
   // ON DOIT UTILISER LE GETTER
   static void afficherScoreMoyen(Joueur j1, Joueur j2,
                                  double score) {
      System.out.println ("Joueurs : " + j1.getPseudo()
                          + ", " + j2.getPseudo());
      System.out.println ("Score moyen : " + score);
   }
}
```


Analyse

Transformation de `ScoreMoyen` en `ScoreMoyenOO` : Une classe très spécifique est remplacée par deux classes plus générales, modularisées et potentiellement plus robustes aux erreurs et aux changements :

- on peut ajouter aisément autant de joueurs que l'on veut
- le lien sémantique entre le nombre de points, le temps de jeu, le score et la notion de joueur est clairement établie (facilité de lecture)
- on peut utiliser la notion de `Joueur` dans des contextes (programmes) différents (modularité)
- le constructeur de la classe `Joueur` peut-être complété de sorte à ne pas permettre des initialisations "farfelues" (nombre de points ou temps négatifs)
  - ☞ conséquence de l'encapsulation : pas d'acès direct aux attributs
- l'utilisateur de la classe `Joueur` ne "voit" d'elle que son calcul de score (et sa méthode de construction) :
  - ☞ abstraction de la notion de joueur
  - ☞ les modalités de calcul du score peuvent changer sans que cela n'ait d'incidence sur les utilisateurs de la classe `Joueur`.

Objets et Classes en Java

Déclarer une classe : `class MaClasse { ... }`
Déclarer une instance (un objet) de la classe `MaClasse obj1;`

Les attributs d'une classe se déclarent dans la classe :
`class MaClasse { ... type attribut; ... }`

Les méthodes d'une classe se déclarent aussi dans la classe et sans `static` :
`class MaClasse { ... type methode(type1 arg1, ...); ... }`

Encapsulation et interface :

```
class MaClasse {
private ..
// attributs et methodes privees
...
}
```

l'objet particulier `this` est une référence à l'instance courante de la classe. Exemple d'utilisation : `this.monattribut`

Ce que j'ai appris aujourd'hui

- Que l'on peut encapsuler données et traitements en utilisant des classes d'objets
- Que les éléments de base de la POO en Java sont :
  - Variables et méthodes d'instance
  - Méthode constructeur
  - L'instruction `new`
  - Notation à point
  - L'objet `this` et shadowing
  - Encapsulation et interface
  - `private`, `public`
- Comment sont gérés les objets en mémoire
- Comment copier, comparer et faire afficher des objets

☞ Je peux maintenant commencer à concevoir des *programmes orientés objet* en Java

Pour préparer le prochain cours

- Vidéos et quiz du MOOC 2 semaine 3 :
  - Héritage : concepts [14 :26]
  - Héritage : droit d'accès protected [8 :34]
  - Héritage : masquage [9 :44]
  - Héritage : constructeurs [11 :57]
  - Polymorphisme : introduction [8 :26]
- Le prochain cours :
  - de 10h15 à 11h : résumé et quelques approfondissements