

Programmation et Algorithmique II

Ch.2 – Classes et Objets

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Chapitre 2

- **Objectifs du Chapitre**

- Principes de **programmation orienté-objet** (P.O.O.)
- Notions P.O.O. en Java
 - **classe**, **attribut**, **méthode**, **constructeur**, instance, ...
 - références, déréférencement, aliasing
- Différences entre variables / méthodes de classe ou d'instance
- Bonnes pratiques d'**encapsulation**

Table des Matières

1. Introduction
2. Classe
3. Constructeur et instance
4. Opérateur de dé-référencement
5. Référence « `this` »
6. Membres de classe
7. Encapsulation

Introduction

- **Programmation Orienté-Objet**
 - Il existe plusieurs paradigmes¹ de programmation.
 - **Programmation impérative ou procédurale** (celle couverte au cours de Programmation & Algorithmique I)
 - **Programmation fonctionnelle** (cf. cours de BAC3)
 - **Programmation logique** (cf. cours de BAC3)
 - **Programmation par contraintes**
 - ...
 - Ce cours a pour but d'introduire le paradigme de la **programmation orienté-objet – P.O.O.** (*object-oriented programming – O.O.P.*).

¹ méthodologie et théories de programmation

Introduction

- **Programmation Orienté-Objet**

- La **programmation orienté-objets** vise à modéliser un problème par un ensemble d'objets ayant chacun un comportement bien défini et qui interagissent entre eux.

Cours 20	
intitulé	Réseaux I
crédits	6

Cours 33	
intitulé	Compilation
crédits	6

Cours 8	
intitulé	Prog & Algo II
crédits	9

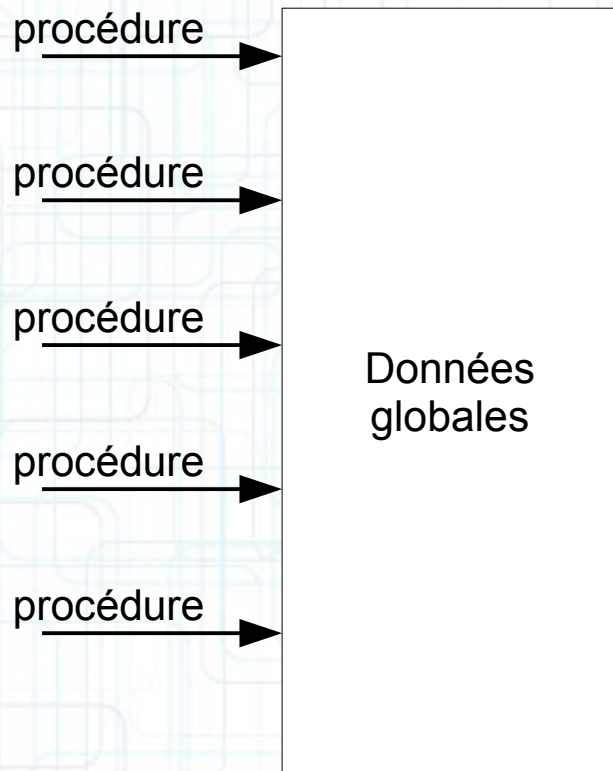
Etudiant 17	
nom	Mélot
prenom	H*****
cours []	Cours 8, ...

getMoyenne()
getMoyennePonderee()
aReussi()
inscrireCours(cours)

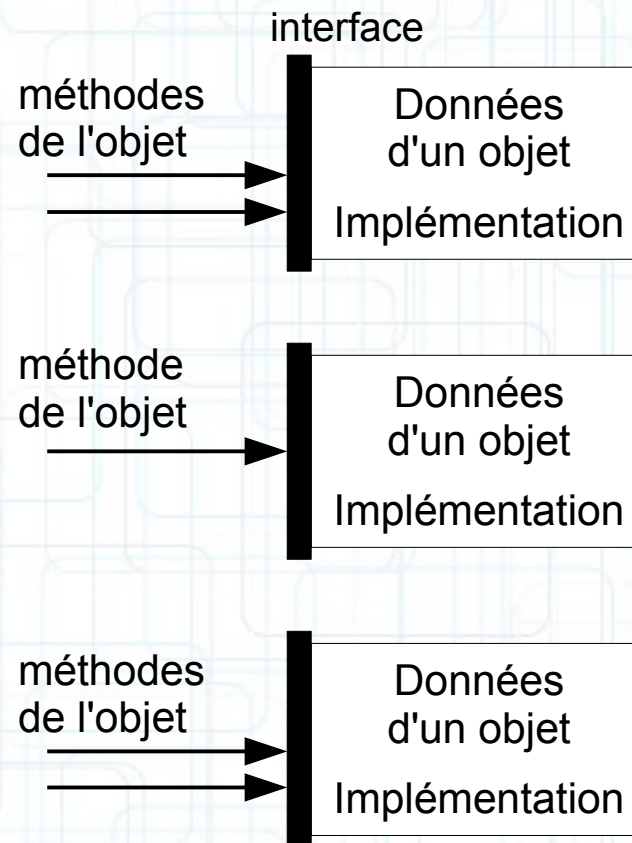
- Un **objet** modélise une entité réelle ou abstraite en regroupant
 - des **attributs** : variables modélisant l'état de l'entité
 - des **méthodes** : *fonctions* modélisant le comportement de l'entité en accédant et/ou modifiant les *attributs*.

Introduction

- **Programmation Orienté-Objet**



Paradigme procédural



Paradigme orienté-objet

Introduction

- **Programmation Orienté-Objet**
 - Exemples d'entités réelles ou abstraites qui peuvent être modélisées avec des objets
 - un étudiant, un cours, un horaire, un bulletin de notes, ...
 - une forme géométrique (par exemple un carré) dans un espace à 2 dimensions
 - un compte bancaire
 - un personnage dans un jeu vidéo
 - le profil d'un utilisateur sur un réseau social
 - un annuaire téléphonique, chaque entrée d'un annuaire
 - ...

Introduction

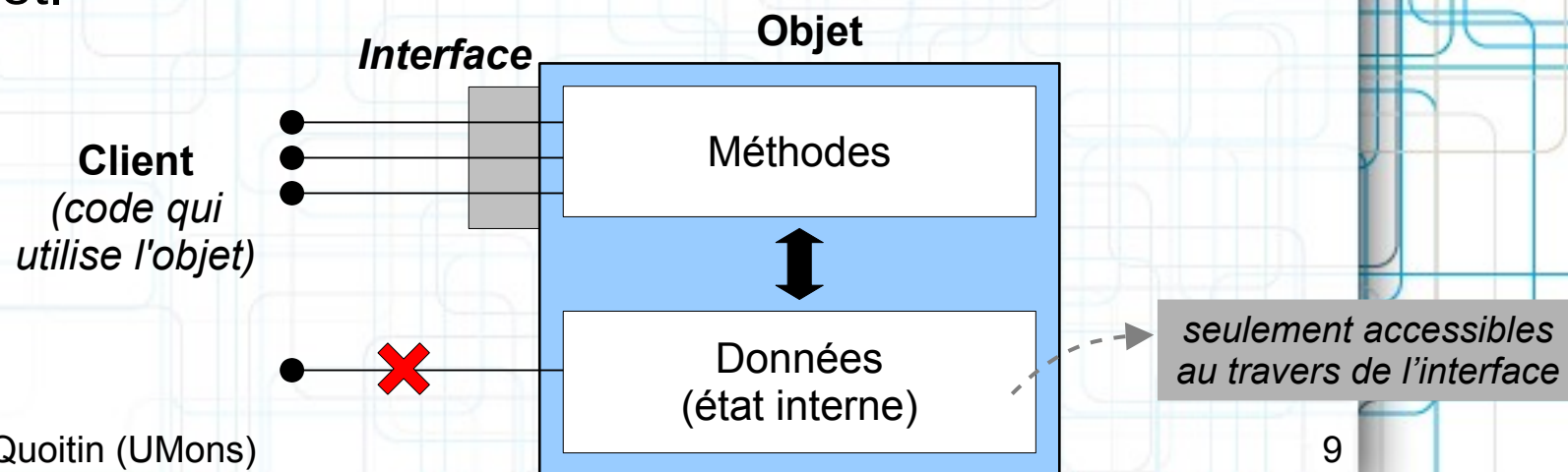
- **Principes de la P.O.O.**
 - Bien que le concept d'**objet** soit au coeur de la P.O.O., d'autres principes y sont intimement liés.
 - **Classe** : description de l'organisation et du comportement partagés par un ensemble d'objets.
 - **Abstraction** : modélisation (avec des classes) en ne conservant que ce qui est nécessaire au problème. Séparation entre l'implémentation et son contrat (interface).
 - **Encapsulation** : limitation de l'accès aux données internes à une classe, en distinguant ce qui est public ou pas.
 - **Héritage** : définition d'une classe sur base de celle d'une autre classe.
 - **Polymorphisme** : différentes implémentations possibles tout en partageant une même interface (ou un même type).

Note : les étudiants curieux sont invités à lire l'article **The Quarks of Object Oriented Development** publié par Deborah J. Armstrong dans Communications of the ACM, en Février 2006

Introduction

- **Encapsulation**

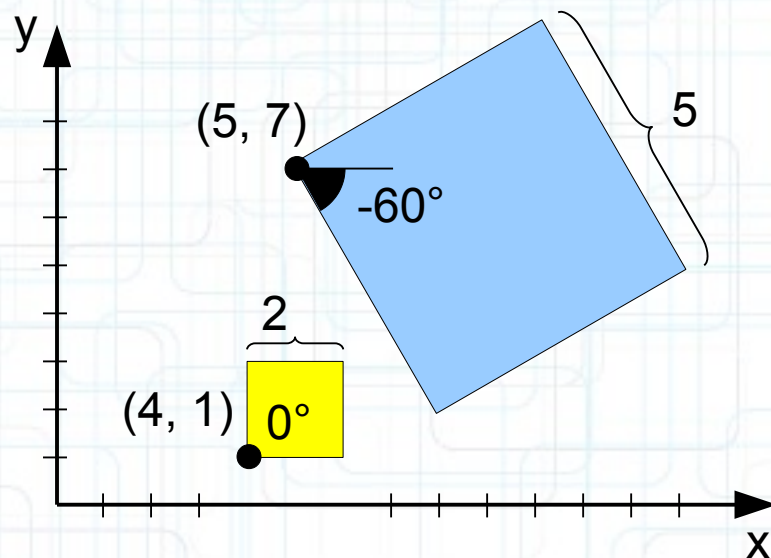
- En P.O.O, on essaye autant que possible de **séparer utilisation et implémentation** d'un objet.
- Idéalement, le client d'un objet ne doit pas connaître le fonctionnement interne d'un objet pour pouvoir l'utiliser (boîte noire).
- Par conséquent, le programmeur d'un objet (vous!) vise à **cacher l'implémentation** d'un objet en fournissant une **interface** claire et en protégeant l'accès à l'état interne d'un objet.



Introduction

- **Exemple**

- comment modéliser des carrés dans un espace à 2 dimensions ?



Modélisation
du carré bleu

- abscisse= 5
- ordonnée= 7
- angle= -60°
- longueur du côté= 5

Modélisation
du carré jaune

- abscisse= 4
- ordonnée= 1
- angle= 0°
- longueur du côté= 2

Introduction

- **Exemple : modélisation de carrés**
 - La modélisation informatique d'un carré comporte la description de son **état**. Cet état est stocké en mémoire (ensemble de variables).
 - coordonnées d'un coin de référence
 - longueur du côté
 - ...
 - Chaque variable décrivant une partie de l'état d'un objet est appelée **attribut** de l'objet.

Représentation en mémoire
de l'état du carré bleu

double	carreBleuX;	5
double	carreBleuY;	7
double	carreBleuAngle;	-60
double	carreBleuCote;	5

Représentation en mémoire
de l'état du carré jaune

double	carreJauneX;	4
double	carreJauneY;	1
double	carreJauneAngle;	0
double	carreJauneCote;	2

Introduction

- **Exemple : modélisation de carrés**
 - La modélisation informatique d'un carré peut aussi inclure ce que l'on peut faire de ce carré (~ son **comportement**).
 - Dans notre exemple, le « comportement » du carré définit comment l'état du carré peut être changé
 - déplacer un carré en modifiant les attributs **x** et **y**.
 - tourner un carré en modifiant son attribut **angle** (en s'assurant que l'angle reste compris entre 0 et 360°)
 - Le comportement d'un objet peut être modélisé par un ensemble de fonctions appelées **méthodes** qui vont utiliser ou changer les attributs de l'objet.

Introduction

- **Exemple: modélisation de carrés**
 - Exemple : fonctions propres au carré bleu

```
void carreBleuDeplacer(double deltaX, double deltaY)
{
    carreBleuX= carreBleuX + deltaX;
    carreBleuY= carreBleuY + deltaY;
}

void carreBleuTourner(double deltaAngle)
{
    carreBleuAngle=
        (carreBleuAngle + deltaAngle) % 360;
}
```

Introduction

- **Exemple: modélisation de carrés**
 - La modélisation d'un carré en programmation orientée-objet regroupe les attributs et les fonctions propres à ce carré au sein d'une même entité appelée **objet**.

Objet modélisant le carré bleu

double carreBleuX;	<input type="text" value="5"/>
double carreBleuY;	<input type="text" value="7"/>
double carreBleuAngle;	<input type="text" value="-60"/>
double carreBleuCote;	<input type="text" value="5"/>
void carreBleuDeplacer(...);	
void carreBleuTourner(...);	
void carreBleuRedimensionner(...);	

Objet modélisant le carré jaune

double carreJauneX;	<input type="text" value="4"/>
double carreJauneY;	<input type="text" value="1"/>
double carreJauneAngle;	<input type="text" value="0"/>
double carreJauneCote;	<input type="text" value="2"/>
void carreJauneDeplacer(...);	
void carreJauneTourner(...);	
void carreJauneRedimensionner(...);	

Introduction

- **Exemple: modélisation de carrés**
 - **Constatation** : Tous les modèles de carrés que nous avons définis (le carré bleu et le carré jaune) contiennent les mêmes attributs et possèdent les mêmes fonctions.
 - Ils appartiennent à la même **classe** (catégorie) de modèles : ce sont tous des modèles de carrés.

Classe Carré

Modélisation d'un carré quelconque

```
double x;  
double y;  
double angle;  
double cote;  
void deplacer(...);  
void tourner(...);  
void redimensionner(...);
```

Instance

Modélisation du carré bleu

x;	5
y;	7
angle;	-60
cote;	5

Instance

Modélisation du carré jaune

x;	4
y;	1
angle;	0
cote;	2

« se comporte
comme défini
par »

Introduction

- **Terminologie P.O.O.**

- **Objet** : modèle d'une entité particulière / individuelle.
 - Ex. : modèle du carré bleu, modèle du carré jaune.
- **Classe** : modèle d'un ensemble d'entités ayant la même structure et la même implémentation.
 - Ex. : la classe Carré définit comment modéliser un carré quelconque.
- **Instance** d'une classe *C* : objet construit à partir de *C*.
 - Ex. : les carrés bleu et jaune sont des instances de la classe Carré.
- **Attributs (champs)** : variables d'une classe ou d'un objet
- **Méthodes** : fonctions propres à une classe ou à un objet.
- **Membres** : attributs et méthodes d'une classe ou d'un objets.

Table des Matières

1. Introduction
- 2. Classe**
3. Constructeur et instance
4. Opérateur de dé-référencement
5. Référence « `this` »
6. Membres de classe
7. Encapsulation

Classe

- **Définition d'une classe**

- En Java, une **classe** est une structure de données qui regroupe des **attributs** (des variables) et des **méthodes**.
- La définition d'une classe comprend

- le **nom** de la classe (identifiant)
- la définition des **attributs** et des **méthodes** de la classe
- un **spécificateur d'accès** optionnel, indiquant quelles autres classes ont accès à celle-ci.

- Syntaxe

```
[ spécificateurAccess ] class nomClasse  
{  
    membres  
}
```

Dans la suite du cours, la notation [x] désignera un élément optionnel.

Classe

- **Nommage des Classes**

- Un nom de classe est un identifiant. Il suit donc les règles de formation des identifiants.
- Par convention,
 - le nom **commence par une lettre majuscule**.
 - le nom suit la **forme « chameau »** lorsqu'il est composé de plusieurs mots.
- Exemples
 - Carre, Compte, ArrayList, ...

Classe

- **Définition d'un attribut**

- Un **attribut** est défini de la même façon qu'une variable est déclarée, en spécifiant

- le **nom** de l'attribut (identifiant)
- le **type** de l'attribut
- un **spécificateur d'accès** optionnel
- un ou plusieurs **modificateurs** optionnels
- l'affectation optionnelle d'une **valeur initiale**

- Un attribut est défini avec la syntaxe suivante

```
[ specificateurAcces ] ( modificateur ) * nomType nomAttribut [ = expression ] ;
```

Dans la suite du cours, la notation (x) * désignera un élément optionnel qui peut être répété.

Classe

- **Portée et durée de vie d'un attribut**

- Un attribut se distingue d'une variable locale par sa **portée** et sa **durée de vie**.

- **Variable locale**

- portée limitée à la méthode où elle est définie.
- durée de vie limitée à l'exécution de la méthode.

- **Attribut**

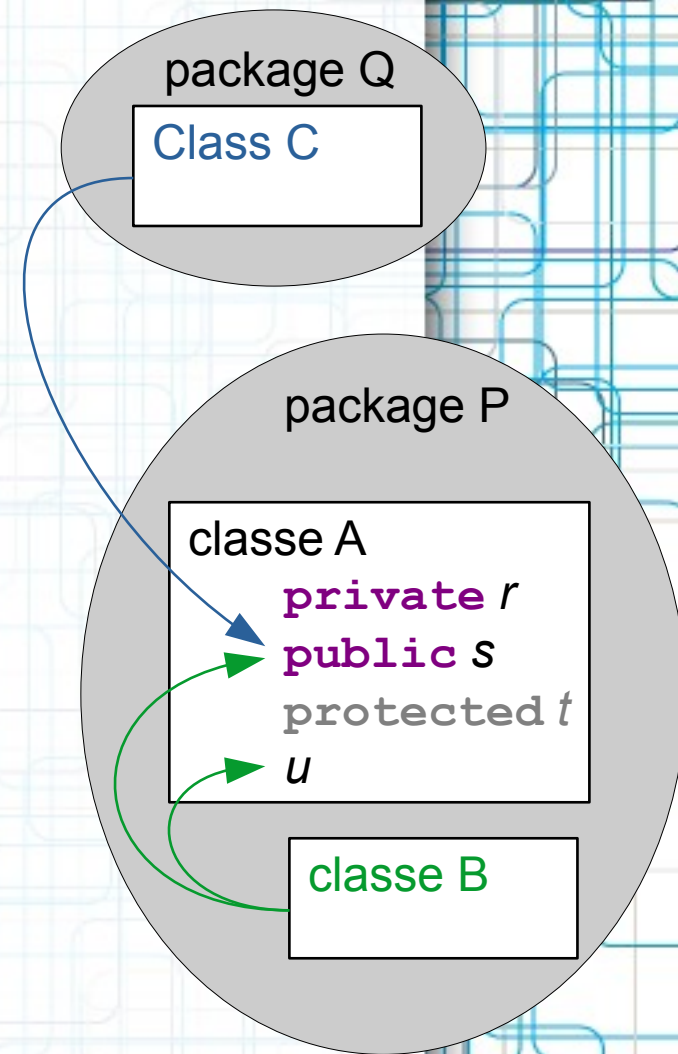
- portée étendue à l'ensemble de la classe.
- durée de vie identique à celle de la classe (ou de l'instance).

```
public class Classe {  
    public double attr;  
  
    public void meth1 ()  
    { ..... }  
  
    public void meth2 ()  
    {  
        accès varLoc  
        accès attr  
    }  
}
```


Classe

- **Spécificateurs d'accès et Modificateurs**

- **Spécificateurs d'Accès** : indiquent qui peut avoir accès à un attribut.
 - **public**, **private**, **protected** et *package* (absence de spécificateur)
 - A ce stade, nous utilisons exclusivement le spécificateur d'accès **public** qui permet l'accès à partir d'autres classes.
- **Modificateurs** : altèrent le comportement d'un attribut.
 - **static**, **final**, **native**, **synchronized**
 - Le modificateur **final** empêche qu'un attribut reçoive une valeur plusieurs fois.



Classe

- **Modificateur `final`**

- Le modificateur **`final`** permet d'empêcher qu'une variable se voie affecter une valeur plus d'une fois. Il peut être utilisé lors de déclaration de variables locales et d'attributs.
- Ce mot clé est aussi utilisé pour définir des constantes (cf. [Chapitre I](#)).
- Exemple

```
public class VariablesFinales {  
    public static final double PI= 3.14159265;  
    public static void main(String [] args) {  
        final int x= 7;  
        x= 2;  
        PI= 2;  
    }  
}
```

→ **Erreur**, le compilateur refuse ces affectations.

Classe

- **Déclaration d'une méthode**

- La définition d'une **méthode** comprend

- un **nom** (identifiant)
- une liste de **paramètres**
- le **type de la valeur de retour**
- un ou plusieurs **modificateurs** optionnels
- un **spécificateur d'accès** optionnel

- Une méthode est définie avec la syntaxe suivante

```
[ specificateurAcces ] ( modificateur ) * typeRetour  
nomMethode ( arguments )  
{  
    corps de la méthode  
}
```

Classe

- Définition d'une classe

```
public class Carre
{
    public double x, y;
    public double angle;
    public double longueurCote;
    public void deplacer(double deltaX, double deltaY)
    {
        x = x + deltaX;
        y = y + deltaY;
    }
    public void tourner(double deltaAngle)
    {
        angle = (angle + deltaAngle) % 360;
    }
}
```

Au contraire des méthodes présentées au Chapitre 1, cette méthode n'est pas définie avec le modificateur **static**.

Spécificateurs d'accès.

Attributs.

Méthode.

x, **y** et **angle** ne sont pas des variables locales, mais des **attributs de la classe**. Leur portée s'étend à l'ensemble de la classe.

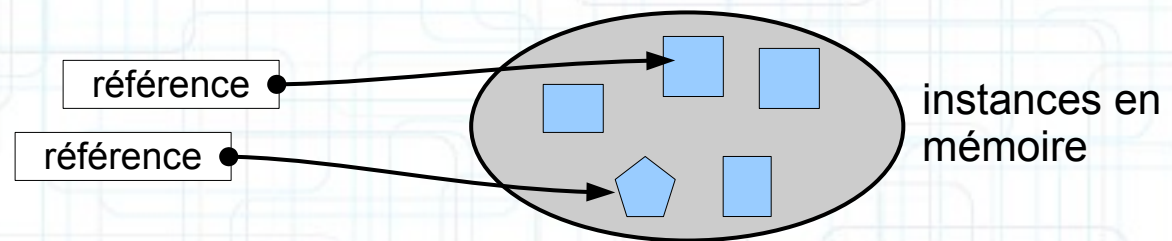
Table des Matières

1. Introduction
2. Classe
- 3. Constructeur et instance**
4. Opérateur de dé-référencement
5. Référence « `this` »
6. Membres de classe
7. Encapsulation

Constructeur

- **Construction d'une instance**

- Une classe définit la structure et le comportement d'un ensemble d'objets. Il est nécessaire de créer une **instance** de la classe pour chaque objet à modéliser.
- Création d'une instance
 - 1). allouer l'espace mémoire nécessaire à l'instance ;
 - 2). initialiser l'instance via le **constructeur** de la classe ;
 - 3). obtenir une **référence**⁽¹⁾ vers l'instance créée.



(1) Une **référence** est un identifiant utilisé par la machine virtuelle pour trouver un objet (instance) en mémoire. Une référence n'est pas l'adresse en mémoire de l'objet car celui-ci peut être déplacé en mémoire sans que la référence ne change.

Constructeur

- **Construction d'une instance**

- La création d'une instance repose sur l'opérateur **new** dont la syntaxe est

```
new nomClasse ()
```

- Exemple

```
Carre carreBleu= new Carre();  
Carre carreJaune= new Carre();
```

Variables de type Carre
(espace mémoire)

carreBleu	1234
carreJaune	5678

Les variables de type
objet contiennent des
références

liens entre références
et instances
(gérés par la JVM)

:Carre	
x	0
y	0
angle	0
longueurCote	0

:Carre	
x	0
y	0
angle	0
longueurCote	0

Référence

- **Référence nulle**

- Une variable de type objet non initialisée contient la valeur **null**.

- Exemple

```
Carre autreCarre;
```

Variable de type Carre
(pas d'instance référencée)

autreCarre **null**

- Assigner la valeur **null** à une variable de type objet lui fait « perdre » la référence connue précédemment.

- Exemple

```
Carre carreBleu= new Carre();  
carreBleu = null;
```

Variable de type Carre

carreBleu ~~1234~~
null

:Carre

x	0
y	0
angle	0
longueurCote	0

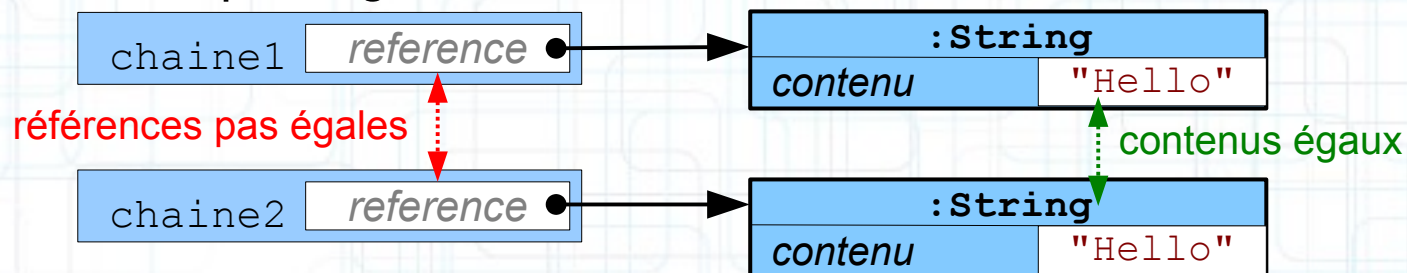
Référence

- **Comparaison d'objets**

- Au chapitre 1, nous avons vu que tester l'égalité entre deux variables `String` avec l'opérateur `==` ne fonctionne pas.

```
String chaine1= "Hello";  
String chaine2= "Hello";  
System.out.println(chaine1 == chaine2); /* false */
```

- En voici l'explication : les deux variables ne sont que des références vers des instances de la classe `String`.
 - Le test d'égalité doit réussir si le contenu des deux instances est identique. Or tester l'égalité des variables ne teste que l'égalité des références.



Référence

- **Type d'une référence**

- Une variable de type objet ne peut contenir que des références vers des instances de type compatible
- A ce stade, nous considérons que **deux types objets sont compatibles s'ils sont égaux.**
- Exemple

```
Carre carreBleu= new Carre();  
Carre carreJaune= new Livre();  
Carre autreCarre= "Carre";
```

Correct car le type de `carreBleu` et de l'instance est `Carre`.

Erroné car le type de `carreJaune` est `Carre` alors que le type de l'instance est `Livre`

Erroné car le type de `autreCarre` est `Carre` alors que le type de l'instance est `String`

Constructeur

- **Constructeur par défaut**

- Lors de la création d'une instance avec **new**, le contenu de la classe est initialisé. C'est le rôle du **constructeur** de la classe.
- Par défaut, le constructeur initialise les attributs de l'instance à des valeurs par défaut

Type d'attribut	Valeur par défaut
byte, short, int, long, float, double	0
char	'\u0000'
boolean	false
référence d'objet	null

- Il est possible de définir des constructeurs personnalisés.

Constructeur

- **Constructeur personnalisé**
 - La plupart du temps, les valeurs par défaut assignées aux attributs par le **constructeur par défaut** ne conviennent pas. C'est pourquoi, Java permet la définition de **constructeurs personnalisés**.
 - Syntaxe : la définition d'un constructeur ressemble à celle d'une méthode.
 - le nom d'un constructeur est identique à celui de la classe.
 - un constructeur ne retourne pas de résultat.

```
[ specificateurAcces ] nomClasse ( arguments )  
{  
    corps du constructeur  
}
```

Constructeur

- Constructeur personnalisé

```
public class Carre {  
    public double x, y;  
    public double angle;  
    public double longueurCote;  
  
    public Carre(double _x, double _y,  
                 double _angle, double _longueurCote) {  
        x= _x;  
        y= _y;  
        angle= _angle;  
        longueurCote= _longueurCote;  
    }  
}
```

Le nom du constructeur et celui de la classe doivent être identiques.

Dans cet exemple, le constructeur initialise les attributs avec les valeurs passées en paramètres.

```
Carre carreBleu= new Carre(5, 7, -60, 5);  
Carre carreJaune= new Carre(4, 1, 0, 2);
```

Variables
(espace mémoire)

carreBleu	référence
carreJaune	référence

:Carre	
x	5
y	7
angle	-60
longueurCote	5

:Carre	
x	4
y	1
angle	0
longueurCote	2

Constructeur

- **Constructeurs multiples**

- Il est permis de définir **plusieurs constructeurs** dans une même classe (surcharge). Ces constructeurs doivent avoir des signatures différentes⁽¹⁾.
- Exemple

```
public Carre(double _x, double _y,  
             double _angle, double _longueurCote) {  
    x= _x;  
    y= _y;  
    angle= _angle;  
    longueurCote= _longueurCote;  
}  
  
public Carre(double _x, double _y, double _longueurCote) {  
    x= _x;  
    y= _y;  
    longueurCote= _longueurCote;  
    angle= 0;  
}
```

⁽¹⁾ de la même façon qu'avec la surcharge des méthodes (cf. Ch. 1).

Constructeur

- **Constructeurs multiples**

- Le mot-clé **this** permet d'invoquer un constructeur surchargé dans un autre constructeur de la même classe.
- L'exemple précédent peut ainsi s'écrire

```
public Carre(double _x, double _y,  
             double _angle, double _longueurCote) {  
    x= _x;  
    y= _y;  
    angle= _angle;  
    longueurCote= _longueurCote;  
}  
  
public Carre(double _x, double _y, double _longueurCote) {  
    this(_x, _y, 0, _longueurCote);  
}
```


Table des Matières

1. Introduction
2. Classe
3. Constructeur et instance
- 4. Opérateur de dé-référencement**
5. Référence « `this` »
6. Membres de classe
7. Encapsulation

Dé-référencement

- **Introduction**

- L'**opérateur de dé-référencement** permet d'accéder aux membres (attributs et méthodes) d'une instance à partir de la référence vers l'instance.
- L'opérateur de dé-référencement, noté « . » (point), est un opérateur binaire.
 - Premier opérande : référence d'une instance
 - Second opérande : nom d'un membre (attribut ou méthode)
- Syntaxe

referenceInstance . *nomMembre*

Dé-référencement

- Accès à un attribut d'instance

- Exemple

```
Carre carreBleu= new Carre(5, 7, -60, 5);  
carreBleu.x= 19;
```



L'opérateur de dé-référencement peut être vu comme le moyen de « suivre la flèche ».

- 1^{er} opérande : référence contenue dans la variable carreBleu
- 2nd opérande : membre « x »

Dé-référencement

- Appel d'une méthode d'instance

- Lors du dé-référencement d'une méthode, il est nécessaire d'ajouter les valeurs de ses arguments. La syntaxe devient

referenceInstance . *nomSymbole* (*valeursArguments*)

- Exemple

```
Carre carreBleu= new Carre();  
carreBleu.deplacer(5, 7);
```

Carre	
double	x, y
double	angle
double	longueurCote
void	deplacer(double, double)
void	tourner(double)

Variable

carreBleu

référence

carreBleu . deplacer

:Carre	
x	5
y	7
angle	-60
longueurCote	5

Dé-référencement

- **Vérifications lors du dé-référencement**

- **Membre existant.** Il s'agit d'une vérification statique effectuée par le compilateur : la classe de l'instance définit-elle le membre ?

```
Carre carreBleu= new Carre();  
carreBleu.profondeur= 16;
```

```
bash-3.2$ javac UnknownSym.java  
UnknownSym.java:9: cannot find symbol  
symbol  : variable profondeur  
location: class UnknownSym  
                carreBleu.profondeur= 16;
```

- **Référence valide.** Il s'agit d'une vérification dynamique effectuée par la JVM : la référence vaut-elle **null** ?

```
Carre carreVert= null;  
carreVert.x= -9;
```

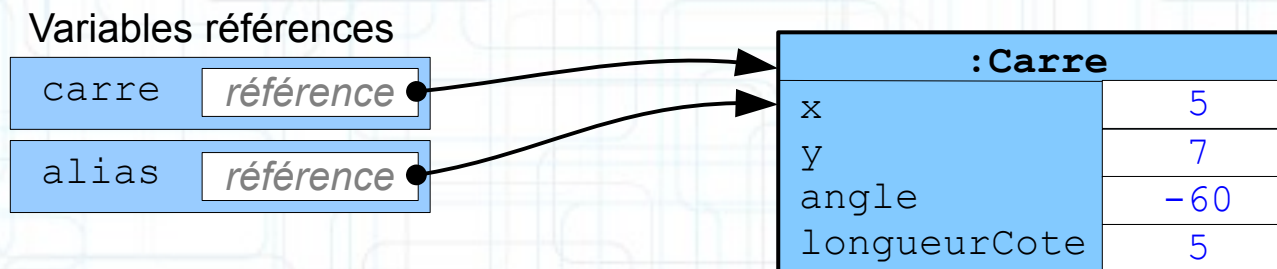
```
bash-3.2$ java InvalidRef  
Exception in thread "main" java.lang.NullPointerException  
    at InvalidRef.main(InvalidRef.java:11)
```

Aliasing

- **Introduction**

- Une variable de type objet est une cellule mémoire qui contient une référence vers un objet.
- **Que se passe-t-il si on copie une variable de type objet vers une autre variable ?** Les deux variables contiennent la même référence. Elles désignent la même instance. On appelle cela de l'*aliasing*.
- Exemple

```
Carre carre= new Carre(5, 7, -60, 5);  
Carre alias= carre;
```



Aliasing

- Problème de l'aliasing

- Copier une référence n'est pas suffisant pour obtenir une copie de l'instance !
- Une **erreur fréquente** consiste à croire que l'instance a été copiée. Or, toute modification de l'instance « copiée » (l'alias) aura aussi un impact sur l'instance originale.



- Exemple

```
Carre carre1= new Carre(5, 7, -60, 5);  
Carre carre2= carre1;  
carre2.deplacer(3, 3);  
System.out.println(carre1.x + "," + carre1.y);  
System.out.println(carre2.x + "," + carre2.y);
```

A red arrow points from the word "copie" to the variable `carre2` in the second line of code.

```
8,10  
8,10
```

Si c'était bien une copie, seule la deuxième instance (`carre2`) aurait dû changer.

En réalité, il n'y a qu'une instance et
`carre1 == carre2`

Table des Matières

1. Introduction
2. Classe
3. Constructeur et instance
4. Opérateur de dé-référencement
- 5. Référence « this »**
6. Membres de classe
7. Encapsulation

Référence « this »

- Appel de méthodes d'une instance

- **IMPORTANT** : l'appel de méthodes d'une instance est différent de l'appel de méthodes d'une classe (Chapitre I).
- A l'intérieur d'une méthode d'instance, la **référence à cette instance est disponible implicitement**. Il est ainsi possible d'accéder aux attributs et d'appeler des méthodes de l'instance courante, sans la mentionner explicitement.
- Exemple

```
public void deplacer(double deltaX, double deltaY)
{
    x = x + deltaX;
    y = y + deltaY;
}
```

Dans la méthode `deplacer`, les variables `x` et `y` sont des **attributs de l'instance** sur laquelle la méthode est appelée.

Les attributs peuvent être accédés sans l'utilisation de la référence de l'instance et sans l'opérateur de dé-référencement.

Référence « this »

- Appel de méthodes d'une instance


- Imaginons que la méthode `deplacer` soit appelée sur l'instance `carreBleu`.

```
Carre carreBleu= new Carre();  
carreBleu.deplacer(5, 7);
```

- Dans la méthode `deplacer`, tout se passe *comme si* l'instance `carreBleu` était passée en argument et mentionnée explicitement à l'opérateur de dé-référencement pour l'accès aux attributs `x` et `y`.

```
public void deplacer(Carre instance,  
                    double deltaX, double deltaY) {  
    instance.x= instance.x + deltaX;  
    instance.y= instance.y + deltaY;  
}
```

```
Carre carreBleu= new Carre();  
carreBleu.deplacer(carreBleu, 5, 7);
```



Référence « this »

- Référence « this »

- Le mot-clé **this** disponible dans les méthodes est la référence à l'instance sur laquelle la méthode est invoquée.
- Ce mot-clé est typiquement utilisé pour
 - 1). accéder aux membres d'une instance (p.ex. en cas de masquage)
 - 2). passer la référence de l'instance à une autre méthode
 - 3). comparer la référence de l'objet à une autre référence
- Exemple

```
public void deplacer(double deltaX, double deltaY)
{
    this.x= this.x + deltaX;
    this.y= this.y + deltaY;
}
```

La référence **this** permet d'indiquer explicitement qu'on accède aux membres de l'instance.

Note : dans cet exemple, **this** peut être omis (il est implicite).

Référence « this »

- **Masquage d'attributs**

- Le **masquage d'attributs** (*shadowing*) est un phénomène qui peut se produire dans une méthode à cause de
 - la **déclaration d'arguments** de la méthode
 - la **déclaration de variables locales** dans la méthode

- Exemple

```
public class Carre {  
    public double x;  
    public void setX(double x)  
    {  
        x = x;  
    }  
}
```

Les deux « **x** » correspondent à l'argument **x** de la méthode.

Le paramètre **x** de la méthode masque l'attribut **x** à l'intérieur de la méthode.

Solution

```
public class Carre {  
    public double x;  
    public void setX(double x)  
    {  
        this.x = x;  
    }  
}
```

Désigne l'argument « **x** » de la méthode.

Désigne l'attribut « **x** » de l'instance.

Exercice

- **Division euclidienne**
 - implémentation algorithme simple vu en Algo I
 - type de retour ? (quotient, reste)
 - définir classe `CoupleInt`
 - compilation, exécution plusieurs classes ?

Table des Matières

1. Introduction
2. Classe
3. Constructeur et instance
4. Opérateur de dé-référencement
5. Référence « `this` »
- 6. Membres de classe**
7. Encapsulation

Mot-clé `static`

- **Membres de classe ou d'instance**
 - Cette section revient sur l'usage du mot-clé `static` utilisé au [Chapitre I](#) lors de la définition de méthodes.
 - Il permet de définir les membres d'une classe (attributs et méthodes) comme étant liés à la classe ou liés à une instance.
 - **Membres d'instance**
 - absence du modificateur `static`
 - membres liés à une instance
 - nécessitent l'existence d'une instance
 - **Membres de classe** (`static`)
 - usage du modificateur `static`
 - membres utilisables sans l'existence d'une instance

Variable d'instance / de classe

- **Principe**

- **Variable d'instance**

- alloué lors de la création d'une instance.
 - un exemplaire propre à chaque instance
 - durée de vie = durée de vie de l'instance

- **Variable de classe** (**static**)

- défini avec le modificateur **static**
 - un seul exemplaire
 - alloué avant toute instance de la classe
 - durée de vie = durée de vie de la classe

- Exemple

```
public class DeClasseEtDInstance {  
    public static int count= 0;  
    public double x, y;  
}
```

Variable de classe.

Variables d'instance.

Variable d'instance / de classe

- Exemple

```
public class Carre {  
    public static int count= 0;  
    public double x, y ;  
    public Carre(double x, double y) {  
        count= count + 1;  
        this.x= x;  
        this.y= y;  
    }  
}
```

```
Carre carreBleu= new Carre(5, 7);  
Carre carreJaune= new Carre(4, 1);  
System.out.println(carreBleu.count);  
System.out.println(Carre.count);
```

La variable `count` compte le nombre d'instance créées. Elle est incrémentée à chaque appel du constructeur.

Carre	
count: int	2
x, y: double	

variable de classe
exemplaire unique

Variables

carreBleu	référence
carreJaune	référence

:Carre	
x	5
y	7

:Carre	
x	4
y	1

variable d'instance
1 exemplaire /
instance

Variable d'instance / de classe

- **Accès à une variable de classe**

- L'accès à une **variable de classe** s'effectue en spécifiant le nom de la classe suivi de l'opérateur de dé-référencement « . » suivi du nom de la variable.

```
public class A {  
    public static int x= 0;  
    public static void meth1 () {  
        x= 7;  
    }  
    public static void meth2 () {  
        int x;  
        A.x= 7;  
    }  
}
```

Implicitement
(A.x)

La variable locale **x** masque
l'attribut. ⇒ Il est nécessaire de
spécifier explicitement **A.x**

```
public class B {  
    public static void meth3 () {  
        System.out.println(A.x);  
    }  
    A inst= new A();  
    inst.x= 5;  
}
```

Lors de l'accès à partir d'une
autre classe, il est nécessaire
de spécifier explicitement **A.x**

Accès possible via la référence
d'une instance. **A éviter si
possible !**

Méthode d'instance / de classe

- **Principe**

- **Méthode d'instance**

- Prend un paramètre implicite, la référence de l'instance. Ce paramètre est disponible via la référence **this**.
 - Nécessite existence d'une instance pour être appelée.
 - Peut accéder aux membres d'instance (p.ex. variables).

- **Méthode de classe** (**static**)

- Déclarée avec le modificateur **static**.
 - Pas de référence implicite à une instance. La référence **this** n'est **pas disponible**.
 - Peut être appelée en l'absence d'instance.
 - **Ne peut pas accéder aux membres d'instance !**

Méthode de classe

- **Restriction**

- Une méthode de classe ne peut accéder implicitement à une variable ou à une méthode d'instance. En effet, la référence **this** n'y est pas définie.

- Exemple

```
public class Carre {  
    public double x, y;  
    public void printCoordinates() {  
        System.out.println("(" + x + ", " + y + ")");  
    }  
    public static double getX() {  
        return x;  
    }  
    public static void main(String [] args) {  
        printCoordinates();  
    }  
}
```

Incorrect : x est une variable d'instance. Or la méthode de classe getX peut être appelée sans l'existence d'une instance !

Incorrect : printCoordinates a besoin de la référence d'une instance pour accéder à x et y.

Méthode de classe

- **Utilité des méthodes de classe**

- **Fonction pure**

- Une méthode qui n'a **pas besoin d'accéder à l'état interne** d'un objet car tous ses paramètres sont fournis explicitement. Par exemple

```
Math.pow(x, 3);
```

- **Accède uniquement aux membres de classe**

- Lorsque cette méthode **n'accède qu'à des membres de classe** (**static**).

```
public class Carre {  
    private static int count= 0;  
    public Carre() {  
        count= count + 1;  
    }  
    public static int getInstanceCount() {  
        return count;  
    }  
}
```


Méthode de classe

- Appel d'une méthode de classe

- L'appel d'une méthode de classe s'effectue en spécifiant le nom de la classe suivi de l'opérateur de dé-référencement « . » suivi du nom de la méthode.

```
public class A {  
    public static void meth() {  
        System.out.println("*");  
    }  
    public static void meth2() {  
        meth();  
        A.meth();  
    }  
}
```

Classe spécifiée explicitement.

Classe spécifiée implicitement (A.meth).

```
public class B {  
    public static void meth3() {  
        A.meth();  
        A inst= new A();  
        inst.meth();  
    }  
}
```

Lors de l'accès à partir d'une autre classe, il est nécessaire de spécifier explicitement A.meth

Accès possible via la référence d'une instance. **A éviter si possible !**

Jeu des erreurs

```
public class Carre {  
    public static int count= 0;  
    public double x, y;  
    public static double angle;  
  
    public Carre() {  
        count= count + 1;  
    }  
    public static void deplacer(double dX, double dY) {  
        x= x + dX;  
        y= y + dY;  
    }  
  
    public void tourner(double dAngle) {  
        angle= (angle + dAngle) % 360;  
    }  
}
```

méthode de
classe
n'a pas
accès aux
membres
d'instance !

angle devrait être un
membre d'instance

```
Carre c= new Carre();  
System.out.println(Carre.count);  
System.out.println(Carre.x);  
System.out.println(c.x);  
c.deplacer(5, 7);  
tourner(90);
```

x est un membre d'instance
⇒ nécessite la référence de
son instance !

tourner est un membre
d'instance ⇒ nécessite la
référence de son instance !

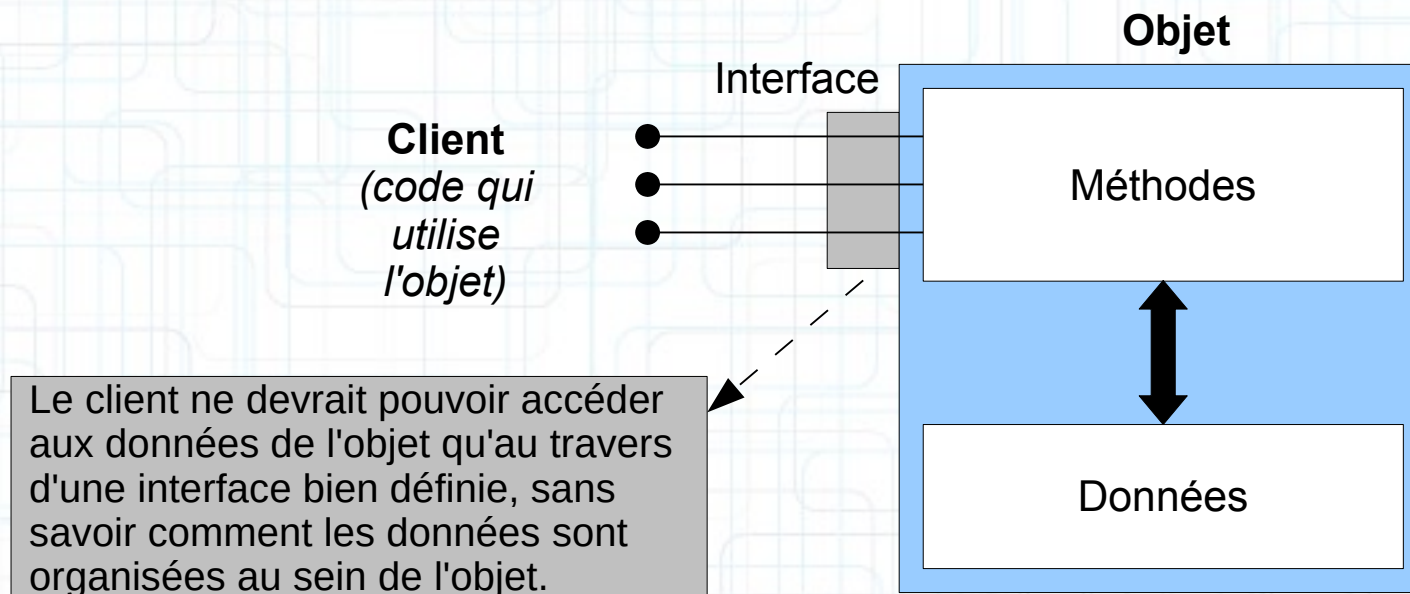
Table des Matières

1. Introduction
2. Classe
3. Constructeur et instance
4. Opérateur de dé-référencement
5. Référence « `this` »
6. Membres de classe
- 7. Encapsulation**

Encapsulation

- **Principe**

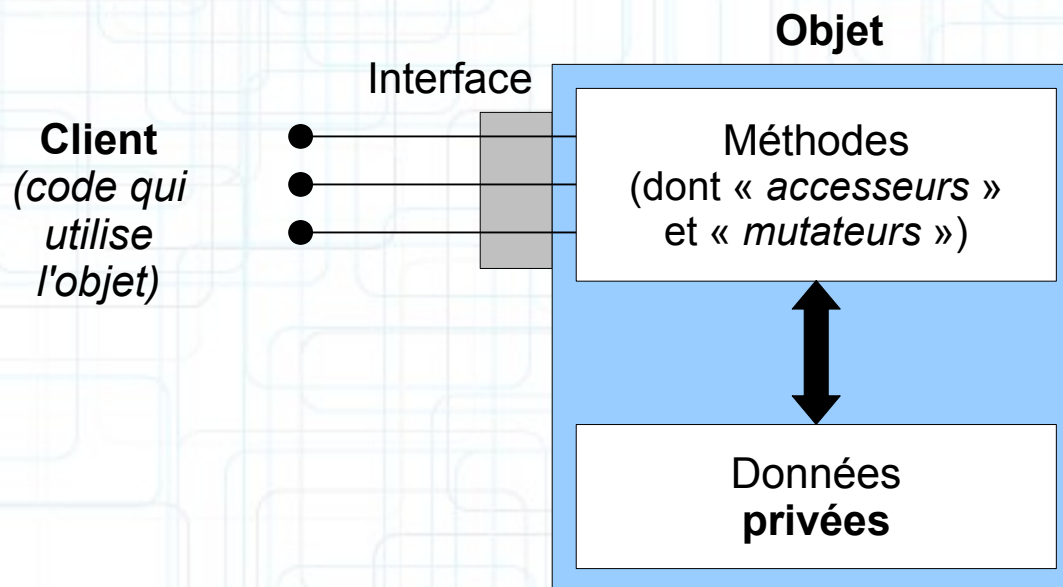
- L'**encapsulation** est le principe de la programmation orientée-objet qui vise à regrouper des données et les méthodes qui y accèdent au sein d'un même objet.
- Ce principe veut aussi que l'implémentation et la structure interne de l'objet soient le plus possible séparées de l'**interface** de l'objet \Rightarrow l'objet est une « *boîte noire* ».



Encapsulation

- **Principe**

- Pour réaliser une « **bonne encapsulation** », il est nécessaire de respecter certaines règles lorsque l'on conçoit des classes d'objets.
 - les données d'un objet sont privées ou non-modifiables
 - des méthodes spécifiques permettent de lire (**accesseurs**) et de modifier (**mutateurs**) les données de l'objet



Encapsulation

- **Utilité de l'encapsulation**

- L'encapsulation ne solutionne pas un problème de sécurité. Il s'agit plutôt de faciliter l'évolution de la classe dans le futur.
- Supposons que la structure d'une classe doive être modifiée. Considérons deux situations
 - Accès direct : les clients de la classe accèdent directement à un résultat dans une variable d'instance. Un changement dans la structure de la classe va probablement nécessiter de modifier tous les clients !
 - Accesseurs / mutateurs : les clients sont forcés d'utiliser des accesseurs et/ou mutateurs. Un changement de la structure interne de la classe peut avoir moins (ou pas) d'impact sur les clients.

Encapsulation

- 1^{er} principe : données privées

- Le spécificateur d'accès **private** empêche que des méthodes extérieures à la classe accèdent à un membre.
- Exemple

```
public class Carre {  
    private double x, y;  
    public Carre(double x,  
                 double y) {  
        this.x= x;  
        this.y= y;  
    }  
    public void deplacer(double dX,  
                        double dY) {  
        x= x + dX;  
        y= y + dY;  
    }  
}
```

L'accès aux membres **private** est autorisé au sein de la classe.

```
public class Client {  
    public static void main(String [] args) {  
        Carre carre= new Carre();  
        carre.x= 3;  
        System.out.println(carre.y);  
    }  
}
```

Erreur, l'accès aux membres **private** est empêché par le compilateur.

Encapsulation

- **1^{er} principe : données non modifiables**

- Une autre façon de protéger les données d'un objet est de les définir comme **non-modifiables**, sans les rendre privées. Cette approche est compatible avec le principe d'encapsulation dans le sens où un client de l'objet ne pourra pas modifier la donnée.

- Exemple

```
public class Carre {  
    public final double x, y;  
    public Carre(double x,  
                 double y) {  
        this.x= x;  
        this.y= y;  
    }  
    /* ... */  
}
```

Note : les Carres
sont devenus
immuables (voir plus
loin).

```
public class Client {  
    public static void main(String [] args) {  
        Carre carre= new Carre();  
        carre.x= 3;  
        System.out.println(carre.y);  
    }  
}
```

Erreur, la
modification du
membre n'est pas
autorisée.

La lecture est autorisée
(ce qui ne serait pas le
cas avec un **private**).

Encapsulation

- **2^{ème} principe : accesseurs et mutateurs**
 - Forcer le client à utiliser des méthodes spécifiques pour accéder et/ou modifier l'état interne \Rightarrow garder le contrôle.
 - **Accesseur** (*accessor*) : méthode qui calcule et/ou retourne un résultat sans modifier l'objet.
 - **Mutateur** (*mutator*) : méthode qui modifie l'objet.
 - Exemple

Convention de nommage

- accesseurs : `getX`
 - mutateurs : `setX`
- où X est dérivé du nom de l'attribut.

```
public class Carre {  
    private double x, y;  
    private double angle;  
  
    public double getX() {  
        return x;  
    }  
  
    public void setAngle(double angle) {  
        this.angle = angle % 360;  
    }  
}
```

Dans cet exemple, le mutateur permet de s'assurer que $\text{angle} \in [0, 360[$.

Encapsulation

- **2^{ème} principe : accesseurs et mutateurs**
 - **Note** : il n'est pas nécessaire de fournir des accesseurs et mutateurs pour toutes les variables d'instances d'une classe !
 - Seules les variables qui doivent pouvoir être consultées par un client de la classe doivent être assorties d'accesseurs.
 - D'autres variables pourraient n'être utilisées que pour le fonctionnement interne de l'objet. Ces dernières NE doivent PAS être accessibles de l'extérieur.

Encapsulation

- **Méthodes de support**
 - Dans une classe, certaines méthodes sont utilisées uniquement « en interne ». On les appelle les **méthodes de support**. Afin d'assurer une bonne encapsulation, ces méthodes ne devraient pas pouvoir être appelées de l'extérieur de l'objet .
 - Le spécificateur d'accès **private** peut être utilisé pour empêcher l'accès à ces méthodes à partir de l'extérieur de la classe.

Objets Immuables

- **Définition**

- Un objet **immuable** est un objet dont on ne peut pas modifier les attributs. Plus spécifiquement, les attributs d'un objet immuable sont
 - non-modifiables (**final**)
 - ou privés et aucune méthode ne permet de les modifier (pas de mutateur)
- Une classe immuable fournit généralement des méthodes qui permettent de générer une autre instance (de même type) à partir de son contenu.
- Les objets immuables peuvent être une alternative à l'encapsulation.

Objets Immuables

- **Exemple : classe `String`**

- Il n'est pas possible de modifier le contenu d'une instance de `String`. En revanche, la classe `String` fournit des méthodes permettant de générer d'autres instances de `String`.

- Par exemple, les méthodes `toUpperCase()` et `substring()` permettent de créer de nouvelles instances de `String` dont le contenu est dérivé de l'instance initiale.

- Illustration : 5 instances

```
String msg1= "J'aime les carrés";  
String msg2= msg1.toUpperCase();  
String msg3= msg1.substring(0, 11) + "ronds";
```

Nouvelle instance
"J'aime les "

Nouvelle instance
"J'aime les ronds"

Nouvelle instance
"J'AIME LES CARRES"

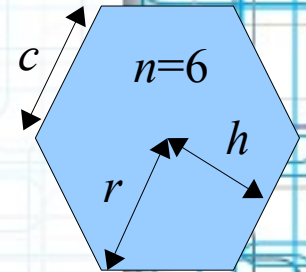
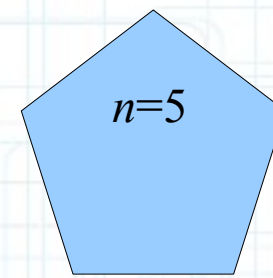
Objets Immuables

- **Discussion des `String` immuables**
 - Avantage : économie d'espace mémoire
 - Le compilateur peut partager une même instance de `String` entre plusieurs références car l'instance ne peut être modifiée (l'*aliasing* n'est pas dangereux dans ce cas). Cela permet d'économiser de la place en mémoire.
 - Inconvénient : coût de création d'instances
 - Lorsqu'un programme nécessite de fréquemment modifier une longue chaîne de caractères, la création de nouvelles instances représente un coût considérable.
 - Pour cette raison, la bibliothèque Java fournit également un support pour des chaînes de caractères non-immuables avec la classe spécialisée **`StringBuffer`**.

Table des Matières

Exercices

Exercice



- **Modélisation d'un polygone régulier**
 - Concevez une classe en Java qui permette de modéliser un polygone régulier. Cette classe doit notamment permettre de calculer la surface et le périmètre du polygone.
 - Quels sont les attributs d'un tel polygone ?
 - Comment assurer une bonne encapsulation ?
 - Peut-on fournir plusieurs constructeurs ?
 - Le premier prend le nombre de côtés et la longueur d'un côté ;
 - Le second prend le nombre de côtés et le rayon ;
 - Le troisième prend le nombre de côtés et la longueur de l'apothème.
 - Rappel : la surface d'un polygone régulier de côté c et d'ordre n est égale à

$$\frac{n \cdot c^2}{4 \tan\left(\frac{\pi}{n}\right)}$$

Exercice

- **Gestion de dates**

- Concevoir une classe `Date` qui modélise une date à l'aide d'un triplet d'entiers (jour du mois, mois, année)
 - afficher la date au format « JJ/MM/AAAA »
 - déterminer si l'année courante est bissextile
 - déterminer le jour de la semaine (lundi, ..., dimanche)
- Conseils
 - déterminer le nombre de jours d'un mois pour une année donnée
 - fournir une classe `TestDate` qui vérifie le fonctionnement des méthodes de `Date` dans des cas bien choisis

