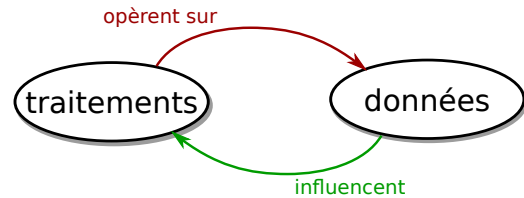


Programmation impérative/procédurale (rappel)

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 :



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);  
    }  
}
```

Objets : quatre concepts de base

Un des objectifs principaux de la notion d'**objet** :

organiser des programmes complexes

grâce aux notions :

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

Notions d'encapsulation

Principe d'encapsulation :

regrouper dans le même objet informatique («concept»), les données et les traitements qui lui sont *spécifiques* :

- ▶ **attributs** : les données incluses dans un objet
- ▶ **méthodes** : les fonctions (= traitements) définies dans un objet

☞ Les objets sont définis par leurs attributs et leurs méthodes.

Notion d'abstraction

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.

Notion d'abstraction : exemple

Exemple : Rectangles

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

Abstraction et Encapsulation

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

- ▶ niveau *externe* : partie « *visible* » (par les programmeurs-utilisateurs) :
 - ▶ l'**interface** : *entête* de quelques méthodes bien choisies
- 📌 résultat du processus d'*abstraction*
- ▶ niveau *interne* : (détails d')**implémentation**
 - ▶ **corps** :
 - ▶ méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou d'objets similaires)
 - ▶ définition de toutes les méthodes de l'objet

Exemple d'interface

L'interface d'une voiture

- ▶ Volant, accélérateur, pédale de frein, 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 »)

Encapsulation et Interface

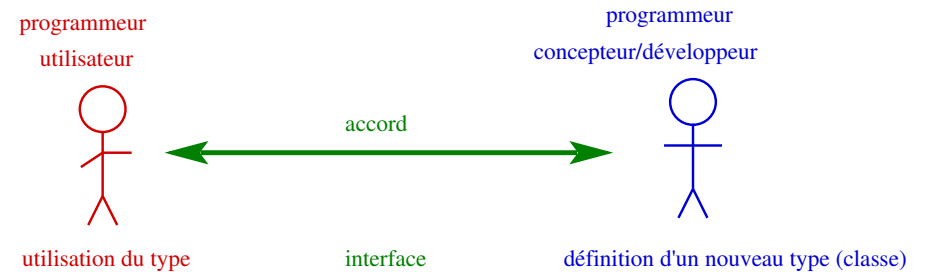
Il y a donc deux facettes à l'encapsulation :

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*)

Les « 3 facettes » d'une classe



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.

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

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

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

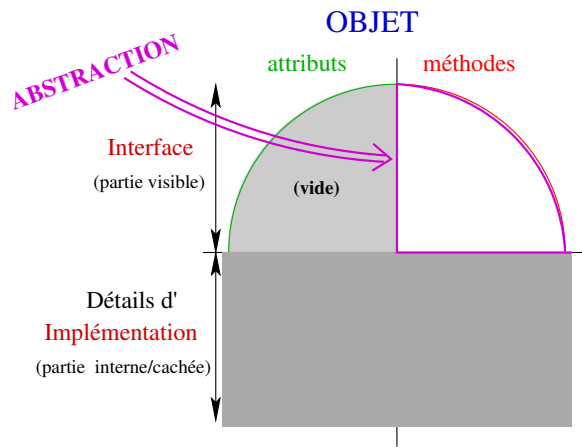
Pourquoi abstraire/encapsuler ? (2)

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 leurs 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.

Encapsulation / Abstraction : Résumé

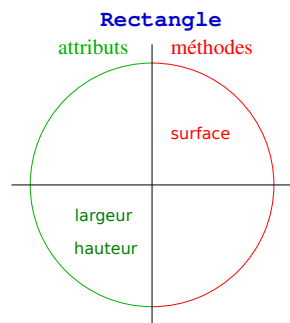


Classes et Instances, Types et Variables

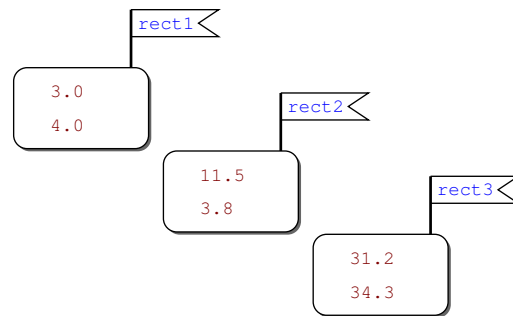
En programmation Objet :

- ▶ le résultat des processus d'encapsulation et d'abstraction s'appelle une **classe**
classe = catégorie d'objets
- ▶ une classe définit un **type**
- ▶ une réalisation particulière d'une classe s'appelle une **instance**
instance = **objet**
- ▶ un objet est une **variable**

Classes et Instances, Types et Variables (illustration)



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

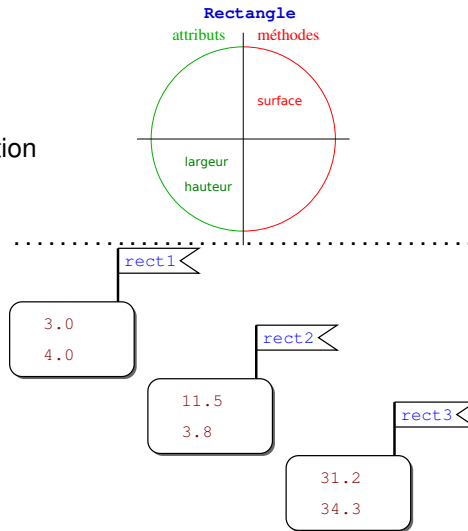


objets/instances
variables en mémoire
existence concrète
(exécution du programme)

Classes et Instances, Types et Variables

En programmation Objet :

- ▶ le résultat des processus d'encapsulation et d'abstraction s'appelle une **classe**
classe = catégorie d'objets
- ▶ une classe définit un **type**
- ▶ une réalisation particulière d'une classe s'appelle une **instance**
instance = **objet**
- ▶ un objet est une **variable**



Les classes en Java

En Java une **classe** se déclare par le mot-clé **class**.

Exemple : `class Rectangle { ... }`

Ceci définit un nouveau **type** du langage.

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

`nomClasse nomInstance;`

Exemple :

`Rectangle rect1;`

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

Où déclarer les classes ?

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

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

> `javac Dessin.java`

produit les fichiers :

`Dessin.class`
`Rectangle.class`

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

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`

produit les fichiers:

`Rectangle.class`
`Dessin.class`

Seule la classe contenant `main` est exécutable:

> `java Rectangle`
Exception in thread "main" java.lang.NoSuchMethodError: main

Déclaration des attributs

La syntaxe de la déclaration des attributs est la suivante :

type nomAttribut;

Exemple :

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

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

Accès aux attributs

L'accès aux valeurs des attributs d'une instance de nom `nomInstance` se fait via la notation pointée :

nomInstance.nomAttribut

Exemple :

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

`rect1.hauteur`

Notre programme (2/4)

```
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;
}
```

Déclaration-initialisation 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 :

<code>int</code>	<code>0</code>
<code>double</code>	<code>0.0</code>
<code>boolean</code>	<code>false</code>
<code>objets</code>	<code>null</code>

Exemple :

`Rectangle rect = new Rectangle();`

Déclaration des méthodes

La syntaxe de la définition des méthodes d'une classe est la syntaxe normale de définition des fonctions :

```
typeRetour nomMethode (typeParam1 nomParam1, ...)
{
    // corps de la méthode
    ...
}
```

mais elles sont simplement mises *dans* la classe elle-même.

Exemple : la méthode `surface()` de la classe `Rectangle` :

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

mais où sont passés les paramètres ?

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

Portée des attributs

Les attributs d'une classe constituent des variables *directement accessibles* dans toutes les méthodes de la classe (*i.e.* des « variables *globales à la classe* »).

On parle de « *portée de 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 *a priori* référence à la valeur de l'attribut `hauteur` (resp. `largeur`) de l'instance concernée (par l'appel de la méthode en question)

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;
    }
}
```

Paramètres des méthodes

Mais ce n'est pas parce qu'on n'a pas besoin de passer les attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont *jamais* de paramètres.

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

Exemple :

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

FigureColoree uneFigure;
Couleur rouge;
// ...
uneFigure.colorie(rouge);
// ...
```

Appels aux méthodes

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

`nomInstance.nomMethode(valArg1, ...)`

Exemple : la méthode

`void surface();`

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

`rect1.surface()`

Autres exemples :

`uneFigure.colorie(ROUGE);`

`i < tableau.size()`

Notre programme (3/4)

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

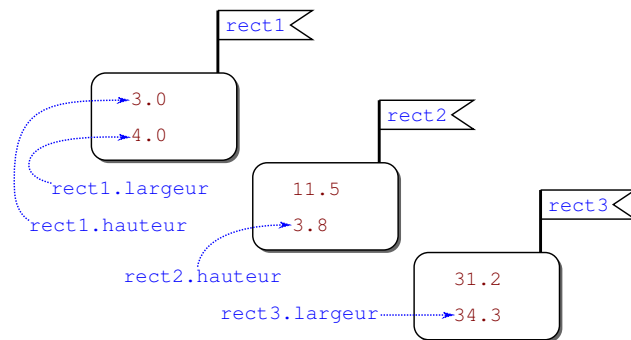
        rect1.hauteur = 3.0;
        rect1.largeur = 4.0;

        System.out.println("surface : " + rect1.surface());
    }
}

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

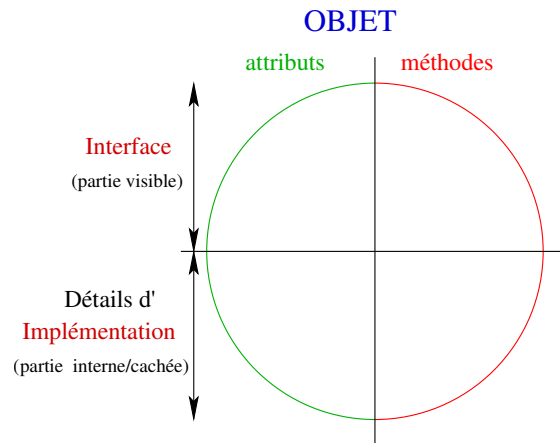
Résumé : Accès aux attributs et méthodes

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



`rect1.surface()` : méthode `surface` de la classe `Rectangle` s'appliquant à `rect1`

Encapsulation / Abstraction



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 {  
    private double hauteur;  
    private double largeur;  
    double surface();  
}
```

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

Erreur de compilation si référence à un(e) attribut/méthode d'instance privée :

`error: hauteur has private access in Rectangle`

Note : Si aucun droit d'accès n'est précisé il s'agit des droits d'accès par défaut («friendly»).

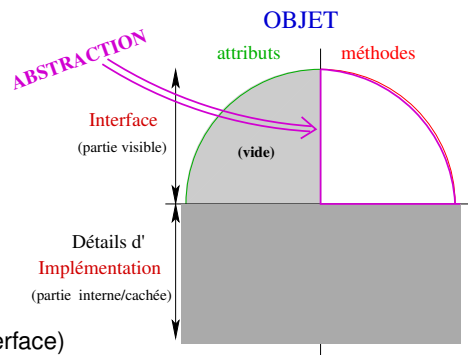
Encapsulation et interface (2)

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

```
class Rectangle {  
    public double surface() { ... }  
    ...  
}
```

Dans la plupart des cas :

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



Droit d'accès public et droit par défaut

Les programmes Java sont habituellement organisés au moyen de la notion de paquetage (`package`) (voir les compléments)

[Si vous ne spécifiez rien, vous travaillez dans le `package` **par défaut**.]

Si aucun droit d'accès n'est spécifié, alors l'attribut/la méthode est publiquement accessible par toutes les autres classes **du même package**, mais *pas en dehors* de celui-ci !

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

« Accesseurs » et « manipulateurs »

Tous les attributs sont privés ?

- ▶ Et si on a besoin de les utiliser depuis l'extérieur de la classe ?!

Par exemple, comment « manipuler » la largeur et la hauteur d'un rectangle ?

« Accesseurs » et « manipulateurs »

Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...

1. Accesseurs (« méthodes get » ou « getters ») :

- ▶ Consultation
- ▶ Retour de la valeur d'une variable d'instance précise

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

2. Manipulateurs (« méthodes set » ou « setters ») :

- ▶ Modification (i.e. « action »)
- ▶ Affectation de l'argument à une variable d'instance précise

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

Notre programme (4/4)

```
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());  
    }  
}
```

```
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;  
}
```

« Accesseurs », « manipulateurs » et encapsulation

Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle  
{  
    public double largeur;  
    public double hauteur;  
    public String label;  
}
```

mais dans ce cas ...

```
Rectangle rect = new Rectangle();  
rect.hauteur = -36;  
System.out.print(rect.label.length());
```

Masquage (shadowing)

masquage = un identificateur « cache » un autre identificateur

Situation typique en POO : un paramètre cache un attribut

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

Masquage et `this`

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 une **référence à l'instance courante**

`this` \simeq « moi »

Syntaxe pour spécifier un attribut *en cas d'ambiguïté* :

`this.nomAttribut`

Exemple :

```
void setHauteur(double hauteur) {  
    this.hauteur = hauteur; // Ah, là ça marche !  
}
```

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

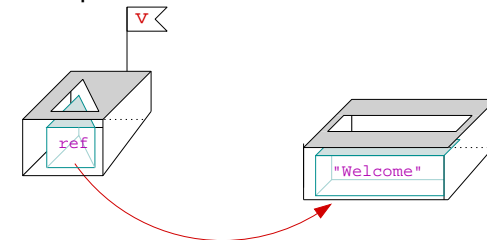
Portée des attributs (résumé)

La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {  
    private int x;  
    private int y;  
  
    public void uneMethode( int x ) {  
        ... y ...  
        ... x ...  
        ... this.x ...  
    }  
}
```

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

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) {...}
```

Notion de classe

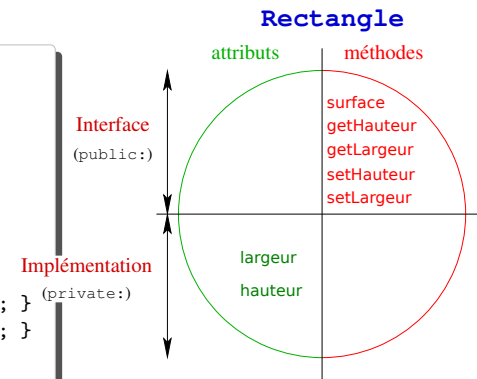
Pour résumer à ce stade, une classe permet de définir un nouveau type caractérisé par :

- ▶ des attributs (des données spécifiques)
- ▶ des méthodes (« fonctions »)
- ▶ dont certains attributs et méthodes (internes) peuvent être cachés (**private**)
- ▶ et dont d'autres constituent l'interface (**public**)

Un exemple complet de classe (1/2)

```
// définition de la classe
class Rectangle {
    // déclaration des attributs
    private double hauteur;
    private double largeur;

    // définition des méthodes
    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; }
}
```



Un exemple complet de classe (2/2)

```
//utilisation de la classe
class Geometrie
{
    private final static Scanner CLAVIER = new Scanner(System.in);

    public static void main(String[] args)
    {
        Rectangle rect = new Rectangle();
        double lu;
        System.out.print("Quelle hauteur? ");
        lu = CLAVIER.nextDouble();
        rect.setHauteur(lu);
        System.out.print("Quelle largeur? ");
        lu = CLAVIER.nextDouble();
        rect.setLargeur(lu);

        System.out.println("surface = " + rect.surface());
    }
}
```

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

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 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;
            }
        }
        //...
    }
}
```

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 à 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 déjà prise, on signale une erreur
        //...
        return false;
    } // suite
}
```

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
```

Jeu de Morpion : bien encapsuler (4)

Comment faire maintenant pour faire un rond sur la case en haut à gauche ?

```
JeuMorpion jeu = new JeuMorpion();
jeu.initialise();
valide = jeu.placerRond(0, 0); //boolean déclaré plus haut
```

Et pour faire une croix sur la 1^{re} ligne, 2^e 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.