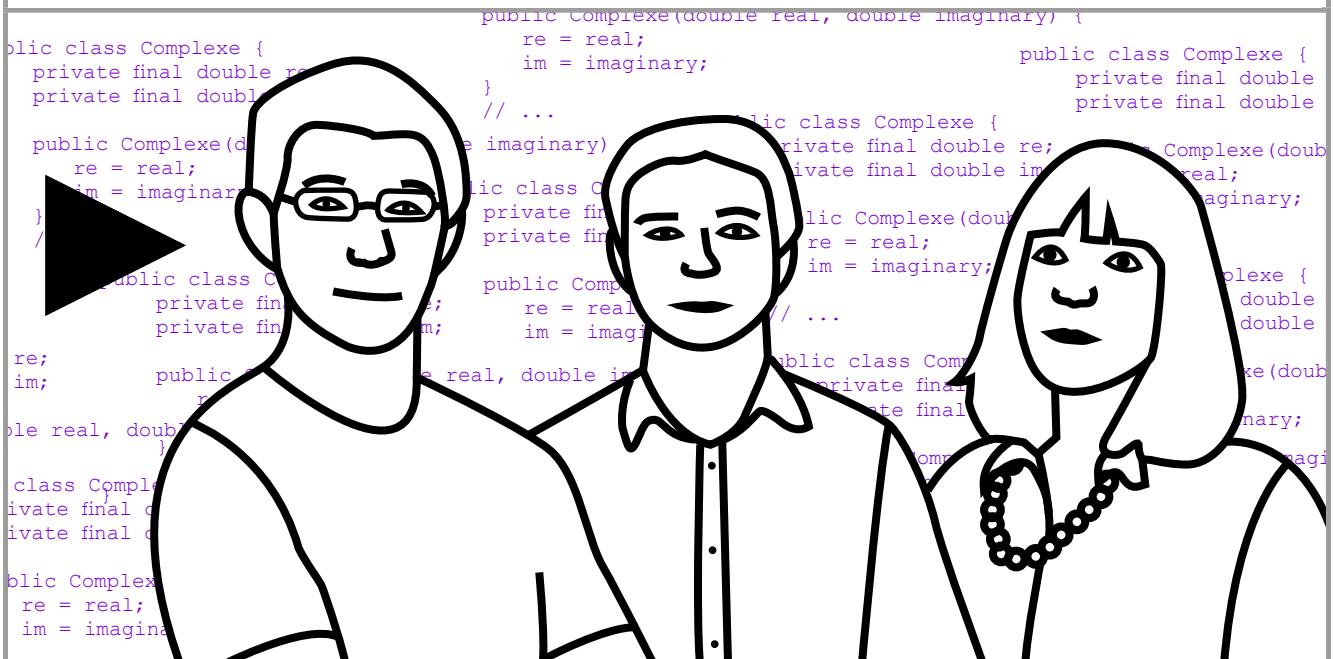
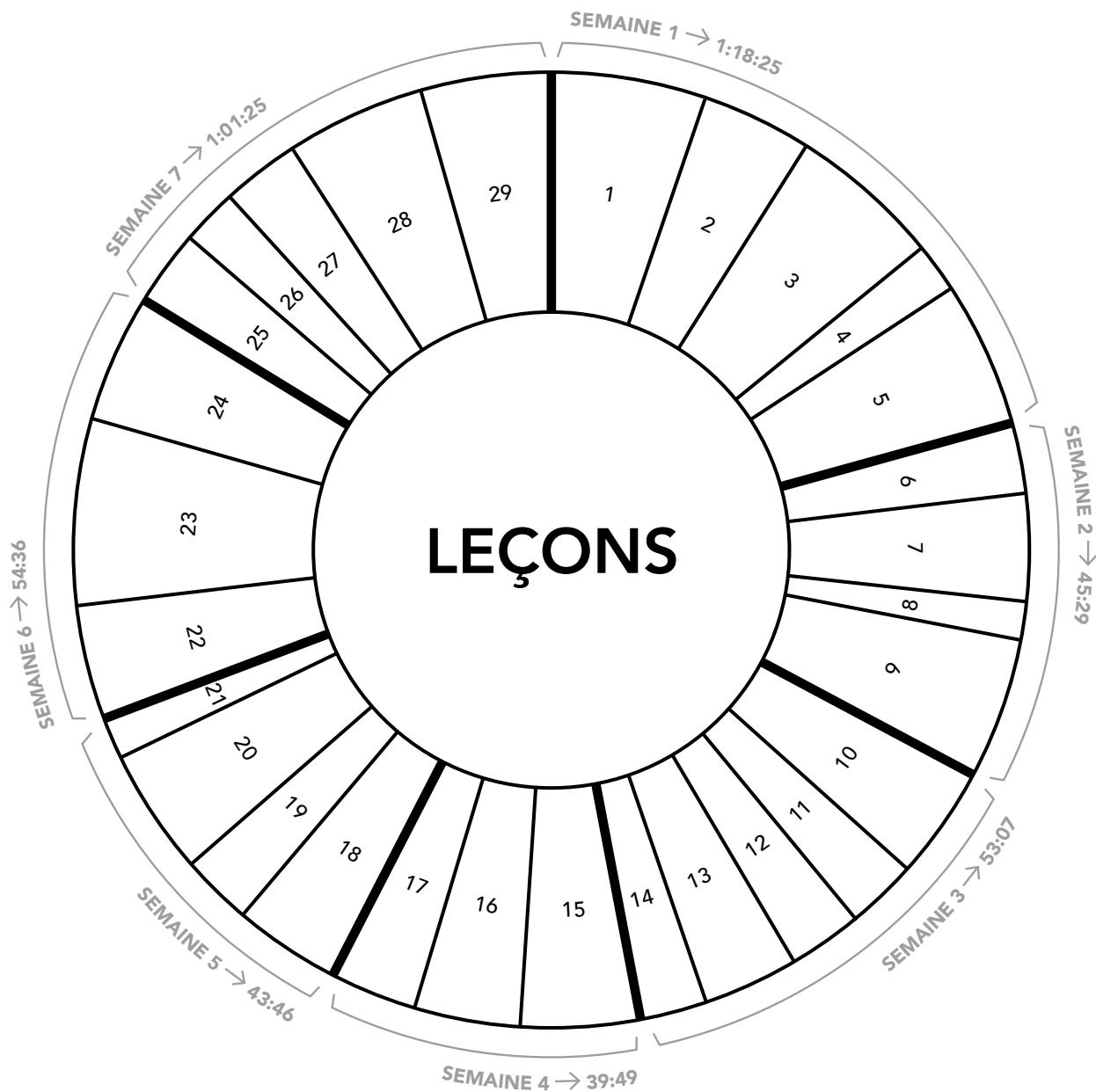


# INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET (EN JAVA)



**Jamila Sam,  
Jean-Cédric Chappelier  
et Vincent Lepetit**





# CONTENU

## SEMAINE 1: BASES DE POO

1. Introduction	4
2. Classes, objets, attributs et méthodes en Java	7
3. public et private	10
4. Encapsulation et abstraction : résumé	12
5. Encapsulation et abstraction : étude de cas	14

## SEMAINE 2: CONSTRUCTEURS

6. Constructeurs : introduction	17
7. Constructeurs par défaut en Java	19
8. Constructeur de copie	22
9. Fin de vie, affectation, affichage et comparaison d'objets	23

## SEMAINE 3: HÉRITAGE

10. Héritage : concepts	25
11. Héritage : droit d'accès protected	27
12. Héritage : masquage	28
13. Héritage : constructeurs	29
14. Polymorphisme : introduction	31

## SEMAINE 4: POLYMORPHISME

15. Classes et méthodes abstraites	32
16. Héritage et polymorphisme : compléments	34
17. Le modificateur final	36

## SEMAINE 5: INTERFACES

18. Attributs statiques	38
19. Méthodes statiques	40
20. Interfaces	42
21. Interfaces en Java 8	45

## SEMAINE 6: GESTION DES EXCEPTIONS

22. Gestion des exceptions : introduction	48
23. Gestion des exceptions : syntaxe	50
24. Gestion des exceptions : compléments	53

## SEMAINE 7: ÉTUDE DE CAS

25. Étude de cas : présentation et modélisation du problème	56
26. Étude de cas : affichage polymorphe	59
27. Étude de cas : première version	60
28. Études de cas : modélisation et mécanismes	63
29. Étude de cas : copie profonde	66



# 1. INTRODUCTION

## PROGRAMMATION ORIENTÉE OBJET

En programmation procédurale (aussi dite impérative), les données et les traitements apparaissent comme des entités séparées dans un programme. Par exemple, dans un programme qui permet de calculer la surface d'un rectangle, un style procédural déclarerait deux variables `largeur` et `hauteur`, puis offrirait une fonction `surface` prenant deux paramètres pour représenter la largeur et la hauteur (fig. 1). Il n'existe alors aucun lien sémantique entre les données, stockées ici dans les variables, et les traitements comme la fonction `surface`. Le lien qui devrait notamment unir la largeur et la hauteur, à savoir qu'il s'agit d'attributs du rectangle, est difficile à établir. De même, en dehors du nom des entités, il n'est pas évident que notre fonction `surface` réalise un calcul de surface de rectangle et non un produit quelconque.

```
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);
    }
}
```

FIGURE 1

1:50

20:03

Calcul de la surface d'un rectangle en programmation «procédurale».

La **programmation orientée objet** permet le regroupement des traitements et des données en une seule et même entité. Dans l'exemple précédent, on préférera regrouper en une seule et même entité la notion de rectangle avec ses données caractéristiques ainsi que les traitements qui lui sont spécifiques, comme dans le deuxième code de la figure 2. Un intérêt fondamental de la programmation orientée objet est donc d'assurer une meilleure lisibilité et une meilleure cohérence au programme, puisque l'on crée un lien explicite entre les données et les traitements. Ce style, qui n'est pas spécifique au langage Java, donne davantage de robustesse, de modularité et de lisibilité aux programmes. Il repose sur quatre concepts centraux: l'encapsulation, l'abstraction, l'héritage et le polymorphisme.

```
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());
```

FIGURE 2

15:30

20:03

Comparaison entre programmation procédurale (à gauche) et orientée objet (à droite), plus lisible.



## ENCAPSULATION ET ABSTRACTION

L'encapsulation consiste à regrouper dans une seule et même entité des données et des traitements qui agissent sur celles-ci. On désignera ces données par le terme « **attribut** » et les traitements par le terme « **méthode** ». Le tout sera défini au sein d'une **classe**, qui constituera un nouveau type de données, une catégorie d'objets. Une variable de ce type sera aussi désignée par le terme d'instance.

Le processus d'abstraction identifie des caractéristiques et des mécanismes communs aux entités utilisées, permettant une description générique, abstraite de l'ensemble. Dans notre exemple, si nous désirons utiliser plusieurs rectangles, tous seraient caractérisés par une largeur et une hauteur sur lesquelles on pourrait effectuer un même calcul de surface. Dans une approche procédurale, on aurait plutôt tendance à déclarer séparément chacun d'entre eux, ce qui serait fastidieux et source d'erreur.

L'encapsulation et l'abstraction consistent également à dissimuler des détails d'implémentation à l'utilisateur, qui n'utilisera plus que l'**interface d'utilisation** de l'objet, c'est-à-dire les différentes méthodes agissant sur un objet. Il y a, en effet, deux niveaux de perception d'un objet: le niveau externe, visible, utile au programmeur utilisateur, celui qui utilise l'objet et voudrait, par exemple, effectuer un calcul de surface. Cet utilisateur n'a pas besoin de connaître les détails techniques des différentes méthodes, tout comme nous n'avons pas besoin de savoir comment un moteur fonctionne pour pouvoir conduire une voiture. Le second niveau est le niveau interne, qui concerne le programmeur concepteur, celui qui se charge des détails d'implémentation, comme définir la façon de calculer la surface d'un rectangle.

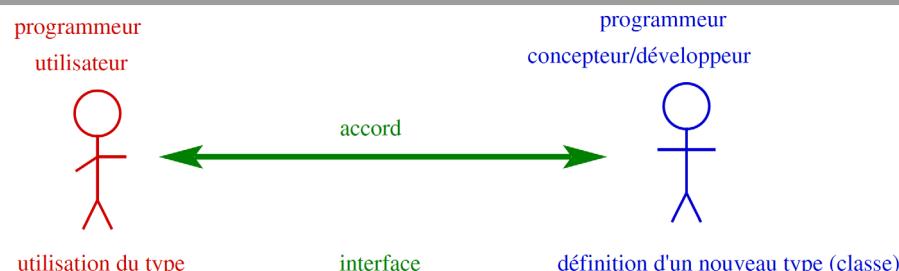


FIGURE 3

14:20

20:03

Les différentes facettes d'une classe.

Le programmeur concepteur définit différents **niveaux de visibilité** et d'accès de la classe, ce qui donne une grande robustesse au programme face aux changements et aux erreurs de manipulation. Par exemple, même si le moteur change selon le modèle des voitures, l'interface reste la même: l'action de conduire n'en est pas affectée. En orienté objet, toute modification de la structure interne d'un objet reste invisible de l'extérieur. De plus, l'accès limité de l'utilisateur à l'objet définit un cadre d'utilisation plus rigoureux, qui protège le programme des erreurs. On considère donc toujours les attributs comme des détails d'implémentation inaccessibles puisqu'ils reposent sur des choix techniques de modélisation et doivent être manipulés avec attention: l'interface se limite à quelques méthodes bien choisies.

Pour résumer, lorsque l'on définit un nouveau type d'objets au travers d'une classe, on définit des attributs communs à toutes les instances de cette classe ainsi que des méthodes spécifiques à ce type d'objets, puis on détermine ce qui est visible, l'interface d'utilisation ; et ce qui ne l'est pas, les détails d'implémentation.

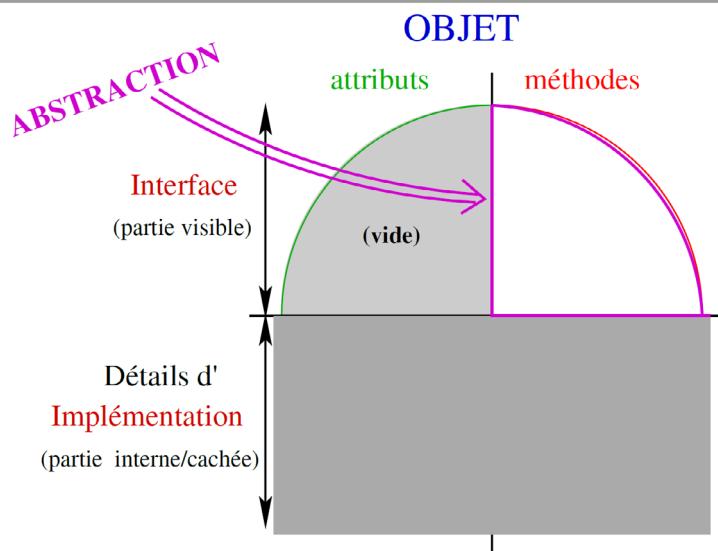


FIGURE 4

18:15

20:03

Résultat du processus d'encapsulation et d'abstraction : l'interface d'utilisation se limite à quelques méthodes tandis que les détails d'implémentation restent invisibles.



## 2. CLASSES, OBJETS, ATTRIBUTS ET MÉTHODES EN JAVA

### CLASSES

Une classe est un nouveau **type** de données dont les instances sont des objets. Pour déclarer une nouvelle classe, il suffit d'utiliser le mot-clé `class`, suivit du nom de la classe :

```
class NomClasse { ... }
```

Une fois la nouvelle classe définie, il est possible de déclarer des variables du nouveau type, qui sont des **instances** de la classe :

```
NomClasse NomVariable;
```

La figure 1 est un exemple de déclaration d'une classe `Rectangle` et d'une variable `rect1` de ce type.

De façon similaire aux tableaux, `rect1` contient une **référence** vers une instance de type `Rectangle`.

Lorsque l'on souhaite déclarer plusieurs classes dans un même programme, il existe deux façons de faire. La première consiste à déclarer plusieurs classes dans un même fichier. Dans ce cas, un fichier « `.class` » sera créé pour chaque classe à la compilation (au moyen de la commande « `java NomDuFichier.java` »). L'autre solution consiste à déclarer une classe par fichier, puis de compiler chaque fichier séparément.

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`.

FIGURE 1

1:30

14:01

Les classes en Java.

### ATTRIBUTS

Pour déclarer des attributs, on utilise la syntaxe suivante :

```
class NomDeLaClasse {  
    type_attribut_1 nom_attribut_1;  
    type_attribut_2 nom_attribut_2;  
    ...  
}
```

L'accès aux valeurs des attributs d'une instance se fait comme ceci :

```
nom_instance.nom_attribut
```

Par exemple, pour accéder à la valeur de l'attribut `hauteur` d'une instance `rect1` de classe `Rectangle`, on écrirait :

```
rect1.hauteur
```



```

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;
}
  
```

**FIGURE 2**

5:20

14:01

Déclaration et utilisation des attributs.

## DÉCLARATION-INITIALISATION

Afin d'initialiser une instance déjà déclarée d'une classe qui ne contiendrait que des attributs (comme dans notre exemple), on utilise le mot-clé `new` de la façon suivante :

`nom_instance = new nom_classe();`

Il est également possible de déclarer et d'initialiser en une seule ligne :

`nom_classe nom_instance = new nom_classe();`

Cette ligne crée une instance de type `nom_classe` et initialise tous ses attributs avec des valeurs par défaut : les entiers sont initialisés à `0`, les doubles à `0.0`, les booléens à `false` et les objets à `null`. Nous reviendrons plus en détail sur l'initialisation des instances.

## MÉTHODES

Les méthodes sont des fonctions qui sont déclarées au sein d'une classe. L'en-tête d'une méthode indique son type de retour, son nom, puis la liste des éventuels paramètres entre parenthèses. L'en-tête d'une méthode est suivie de son corps, ou définition, entre accolades. La syntaxe est donc la suivante :

`type_retour nomMéthode (type_paramètre_1 nom_paramètre_1, ...) {corps de la méthode}`

Les paramètres d'une méthode sont des variables externes à la classe : chaque méthode d'une classe a accès aux attributs de celle-ci, qui ne doivent donc pas être passés en arguments. On parle de **portée de classe** : les attributs sont des variables globales à la classe

Dans la classe `Rectangle`, on ajoute par exemple la méthode `surface` qui retourne un double qui ne prend pas de paramètres. En effet, elle n'exploite que les attributs de sa classe, à savoir la hauteur et la largeur, auxquels elle a directement accès. Dans d'autres situations, les méthodes peuvent cependant requérir des paramètres externes à la classe s'ils sont nécessaires pour leur fonctionnement.

Enfin, l'appel à une méthode se fait de façon similaire à l'accès de la valeur d'un attribut :

`nom_instance.nom_méthode(valeur_argument_1, ...);`



```
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;
    }
}
```

FIGURE 3

11:30

14:01

Exemple d'appel à la méthode `surface()` de l'instance `rect1`.



### 3. PUBLIC ET PRIVATE

En programmation orientée objet, il est important de séparer l'interface d'utilisation des détails d'implémentation, pour notamment limiter la partie accessible d'une classe à quelques méthodes bien choisies. En Java, les mots-clés `public` et `private` permettent d'établir cette distinction.

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

**FIGURE 1**

1:00

19:27

Exemple de déclaration de membres privés d'une classe.

#### PRIVATE

Le mot-clé `private` est employé pour signaler quelle partie de la classe restera inaccessible à un utilisateur de la classe. Tout élément de la classe déclaré avec le mot-clé `private` est donc un détail d'implémentation, inaccessible depuis l'extérieur de la classe. C'est dans cette catégorie que l'on déclare toujours les attributs, comme dans la classe `Rectangle` de la figure 1, mais également certaines méthodes internes qui ne peuvent être appelées qu'au sein de la même classe. Une tentative d'accès à un élément privé d'une classe depuis l'extérieur de celle-ci produira un message d'erreur à la compilation.

#### PUBLIC

Avec le mot-clé `public`, on indique quels membres d'une classe sont accessibles, visibles et utilisables depuis l'extérieur de celle-ci. Tout élément déclaré avec ce mot-clé appartient à l'interface d'utilisation, qui doit se limiter à quelques méthodes. Dans l'exemple de la classe `Rectangle`, après avoir déclaré la méthode `surface` dans la partie publique, comme cela a été fait dans la figure 2, on pourrait faire appel à celle-ci depuis la méthode `main`. Notons que par défaut, sans spécification, tout élément d'une classe possède le droit d'accès par défaut, c'est-à-dire visible, accessible et utilisable partout à l'intérieur du même paquetage. Néanmoins, le droit d'accès par défaut est très rarement utilisé et il est recommandé de systématiquement expliciter les droits d'accès avec `public` et `private`.

#### ACCESSEURS ET MANIPULATEURS

Alors que tous les attributs sont en règle générale privés, inaccessibles à l'extérieur, il peut néanmoins être utile de les utiliser depuis l'extérieur de la classe, notamment pour modifier ou connaître leur valeur. Pour cela, on peut proposer des méthodes publiques pour accéder aux attributs, en modification ou en consultation. Cette partie de la conception est extrêmement importante: il faut bien sélectionner quels attributs seront offerts au travers d'une méthode.

Des méthodes qui retournent la valeur d'un attribut sont appelées des **accesseurs**, «méthodes get» ou «getters». Des exemples d'accesseurs sont les méthodes `getHauteur` et `getLargeur` dans le code de la figure 2, qui permettent effectivement de connaître la valeur des attributs privés.

On peut aussi avoir besoin de **manipulateurs**, appelés aussi «méthodes set» ou «setters»: ceux-ci permettent de modifier la valeur d'un attribut. De telles méthodes prennent donc en paramètre la valeur à affecter et ne retournent rien. Les méthodes `setHauteur` et `setLargeur` de la figure 2 appartiennent à cette catégorie.

Le fait de garder les attributs privés et de ne les manipuler qu'au travers des méthodes permet de prévenir les erreurs de l'utilisateur: le concepteur peut par exemple imposer qu'une hauteur de rectangle soit positive, garantissant ainsi l'intégrité des données. De plus, l'interface permet au concepteur de librement modifier sa représentation interne de la classe sans que l'utilisateur n'en soit affecté. Ces contraintes de la programmation orientée objet prennent vraiment du sens dans le contexte de gros programmes ou de partage de code.



```
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;
}
```

FIGURE 2

8:30

19:27

Exemple d'une classe avec manipulateurs et accesseurs.

```
class MaClasse {
    private int x;
    private int y;

    public void uneMethode(int x){
        ... y ...
        ... x ...
        ... this.x ...
    }
}
```

FIGURE 3

15:00 19:27

Exemple de masquage d'un attribut : dans uneMethode, x désigne le paramètre tandis que this.x désigne l'attribut de l'instance courante.

## MASQUAGE

On parle de masquage lorsqu'un identificateur en cache un autre, plus particulièrement quand une variable locale prend le pas sur une variable de portée plus large. Souvent, lorsque l'on déclare des manipulateurs, on choisit comme nom de paramètre le même nom que l'attribut à modifier: il y a donc ambiguïté de ce nom qui désigne deux entités, le paramètre et l'attribut. Pour ne pas recevoir de message d'erreur du compilateur, il est recommandé de choisir un nom différent pour le paramètre, mais on peut aussi désambiguier le nom en indiquant quand il désigne l'attribut de classe. On utilise pour cela la référence `this`, qui est une référence sur l'instance courante. On utiliserait donc la syntaxe suivante: `this.attribut`. Cette syntaxe désigne donc le champ `attribut` de l'instance courante et permet de distinguer l'attribut de la variable qui le masquerait. La figure 3 est un exemple d'utilisation de cette référence. Il reste cependant conseillé de limiter les situations de masquage.

## OBJETS EN MÉMOIRE

Tous comme c'est le cas pour la classe prédefinie `String` et pour les tableaux, les objets, contrairement aux entités de types élémentaires comme les entiers, sont manipulés via des références indiquant la zone en mémoire où sont stockées les valeurs. Il est très important de s'en rappeler lorsqu'on veut comparer deux objets, affecter un objet à un autre ou afficher un objet: en comparant deux objets au moyen de l'opérateur `==`, on ne compare effectivement que les références à ces deux objets, et de même pour l'affectation ou l'affichage, ce sont bien des références qui entrent en jeu.

Pour indiquer qu'une variable ne contient la référence à aucun objet, on peut lui affecter la constante `null`. Avant de commencer à utiliser un objet, il est donc judicieux de tester si cet objet existe vraiment au moyen de tournures telles que :

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

## 4. ENCAPSULATION ET ABSTRACTION : RÉSUMÉ

Une classe regroupe des données, stockées dans ses attributs, et des traitements, décrits dans ses méthodes. De plus, le programmeur qui la conçoit peut déterminer lesquels de ces attributs ou méthodes seront accessibles ou pas depuis l'extérieur de la classe.

### EXEMPLE DE CLASSE

La figure 1 est un exemple d'implémentation de la classe `Rectangle`, qui utilise la syntaxe de définition d'une classe. Le nom de la classe désignera ensuite un type dans le programme: par convention, on choisit usuellement de le faire commencer par une majuscule. Une fois les attributs et les méthodes établis, il faut distinguer, parmi ces éléments, les détails d'implémentation et l'interface d'utilisation. Dans la classe `Rectangle`, le concepteur offre des méthodes publiques de consultation et de manipulation des attributs ainsi que la méthode de calcul de surface, mais garde les attributs privés, comme souvent en programmation orientée objet. Cela permet notamment de protéger le code contre des erreurs, en contrôlant si les données fournies sont valables.

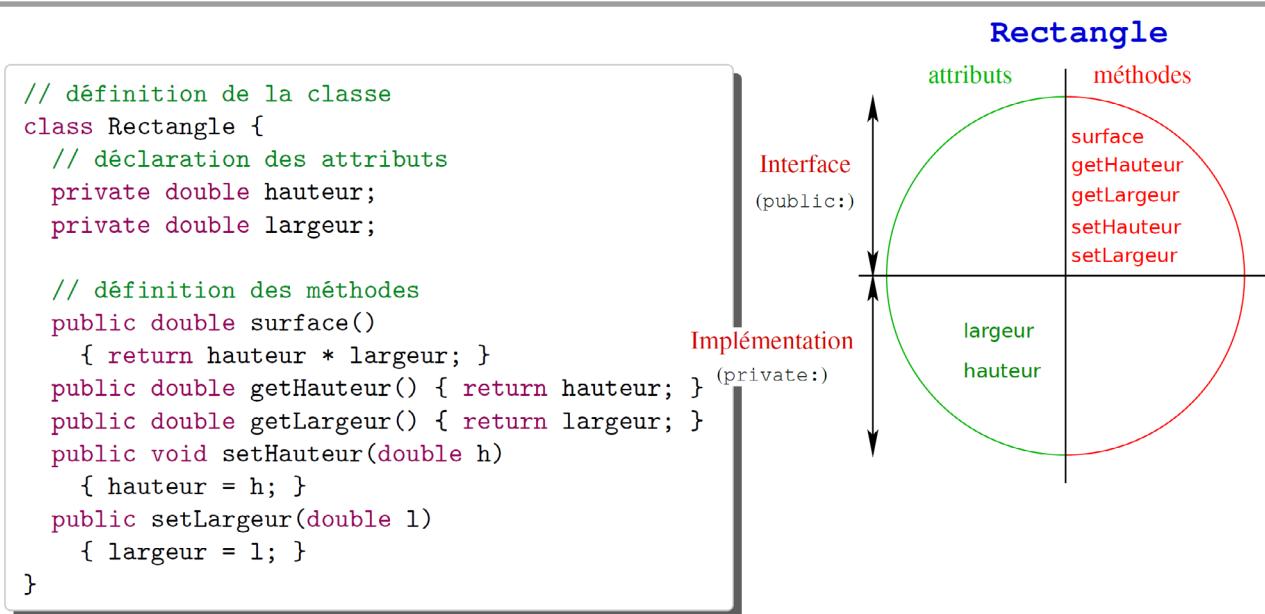


FIGURE 1

1:00

6:17

Un exemple concret de classe.

L'exemple de la classe `Rectangle` dans la figure 1 reste très basique: de manière générale, il n'est pas recommandé de systématiquement définir des accesseurs et des manipulateurs pour tous les attributs de la classe.

Une bonne encapsulation, c'est-à-dire une bonne séparation entre détails d'implémentation et interface d'utilisation, permet au programmeur concepteur de modifier plus librement la partie interne de la classe sans impact pour l'utilisateur. Par exemple, on pourrait choisir un tableau à deux éléments, au lieu de deux doubles, pour représenter la largeur et la hauteur d'un `Rectangle`, sans aucun effet pour les utilisateurs la classe, puisqu'ils n'interagissent jamais directement avec les attributs de cette classe.



Du côté du programmeur utilisateur, dont on a un aperçu avec la figure 2, `Rectangle` définit un nouveau type dont on peut déclarer une instance, un objet. Ensuite, il faut initialiser ses attributs par exemple grâce à des manipulateurs ou d'autres outils qui seront explorés plus loin. On peut alors invoquer d'autres fonctionnalités sur le rectangle comme un calcul de sa surface.

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

FIGURE 2

4:05

6:17

Une utilisation possible de la classe `Rectangle`.



## 5. ENCAPSULATION ET ABSTRACTION : ÉTUDE DE CAS

On cherche à programmer un jeu du Morpion, dans lequel deux joueurs s'affrontent sur une grille 3x3. À tour de rôle, le premier joueur pose des ronds et le deuxième des croix, par exemple, et le but pour chacun des joueurs est d'aligner trois de ses pièces.

Pour une bonne conception, l'interface d'utilisation doit limiter l'accès de l'utilisateur aux détails d'implémentation, tout en lui permettant de travailler facilement avec la classe.

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

FIGURE 1

0:45

18:37

Mauvaise interface d'utilisation pour la classe JeuMorpion.

### EXEMPLE 1

Étudions dans un premier cas un mauvais exemple d'encapsulation au travers d'une classe permettant de représenter le plateau d'un jeu de Morpion (le code de cette classe est présenté dans la figure 1). Cette classe propose une interface d'utilisation contenant deux méthodes: la méthode `initialise()` permet d'initialiser une grille vide et la méthode `getGrille()` permet d'accéder à celle-ci. Le programmeur utilisateur souhaitant créer un nouveau plateau de jeu et placer un rond dans la case en haut à gauche devrait donc écrire:

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

Ce code est parfaitement fonctionnel mais pose beaucoup de problèmes: premièrement, l'utilisateur doit connaître plusieurs détails d'implémentation de la classe `JeuMorpion`, comme le fait que les cases sont stockées sous forme d'entiers dans un tableau à une dimension, ligne par ligne, ou encore que la valeur 0 correspond à une case vide, la valeur 1 à un rond et la valeur 2 à une croix, ce qui rend le code complètement cryptique pour une personne n'ayant pas lu le code de la classe `JeuMorpion`. Puis, si l'utilisateur essayait d'accéder à un index invalide de la grille, il recevrait un message d'erreur. Troisièmement, s'il tentait d'introduire des valeurs telles que 4, 6, ou 37 dans le tableau, rien ne l'en empêcherait, et des méthodes utilisant les valeurs de la grille ne fonctionneraient plus. Finalement, l'utilisateur pourrait tricher en remplaçant simplement un rond par une croix par exemple.

Tout ceci nous montre que l'interface d'utilisation a été mal conçue. Ce code révèle une mauvaise encapsulation. L'accesseur `getGrille()` permet de directement modifier le contenu du tableau contenant les cases du plateau. Le fait d'avoir protégé l'attribut `grille` perd son intérêt. Enfin, la classe `JeuMorpion` n'offre aucune robustesse face aux changements des détails d'implémentation: si le programmeur concepteur de la classe décide de stocker les cases sous la forme d'un tableau à deux dimensions, le programmeur utilisateur devrait complètement revoir son code afin de l'adapter à ce changement.



## EXEMPLE 2

Étudions maintenant un bon exemple d'encapsulation, toujours pour le jeu de Morpion, de classe (le code de cette classe est représenté dans les figures 2, 3 et 4). Cette classe définit des constantes explicites pour représenter les cases vides, contenant un rond ou une croix, ainsi qu'un attribut contenant un tableau d'entiers à deux dimensions pour stocker les cases. Son interface d'utilisation propose une méthode `initialise()` permettant de remplir le plateau de cases vides, une méthode `placerRond()` qui se charge de placer un rond sur une case dont elle reçoit le numéro de la ligne et de la colonne, et qui retourne un booléen indiquant si le rond a pu être correctement placé, ainsi que d'une méthode `placerCroix()` analogue.

Avec ce nouveau code, afin de placer un rond sur la case en haut à gauche, l'utilisateur devrait écrire :

```
JeuMorpion jeu = new JeuMorpion();  
jeu.initialise();  
valide = jeu.placerRond(0, 0); // valide contient le booléen retourné par la méthode
```

Ce nouveau code offre de nombreux avantages : la seule convention à connaître pour l'utilisateur est que les lignes et les colonnes sont numérotées de 0 à 2 (ce qui doit être inclus dans la documentation de classe `JeuMorpion`), il est impossible de mettre une valeur invalide dans le tableau étant donné que l'on ne connaît même pas les valeurs qui y sont stockées lorsque l'on est un utilisateur de `JeuMorpion`, et enfin la méthode privée `placerCoup` se charge de vérifier que l'on ne coche pas une case déjà cochée. `placerCoup` n'est pas utile à l'utilisateur externe et ne fait donc pas partie de l'interface d'utilisation. Ce code offre également ce que l'on appelle la **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, ce qui rend le code très compréhensible : le nom des méthodes exprime clairement ce qu'elles font et ne nécessitent aucune connaissance des détails d'implémentation. Finalement, si l'on essaie de faire une opération invalide, la méthode `placerCoup` nous fournit un message d'erreur compréhensible. Une bonne encapsulation repose sur des détails d'implémentation invisibles à l'extérieur et sur une interface d'utilisation limitée et contrôlée.

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

FIGURE 2

10:30

18:37



```
/*
 * 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 ron
        grille[ligne][colonne] = coup;
        return true;
    } else {
        // case déjà prise, on signale une erreur
        //...
        return false;
    } // suite
}
```

FIGURE 3

13:00

18:37

Meilleure conception de la classe JeuMorpion (suite).

```
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 méthode getJoueurGagnant()
```

FIGURE 4

11:30

18:37

Meilleure conception de la classe JeuMorpion (suite).



## 6. CONSTRUCTEURS : INTRODUCTION

### INITIALISATION DES ATTRIBUTS

Lorsque nous déclarons un nouvel objet, comment initialiser ses attributs avec d'autres valeurs que les valeurs de base ? La solution que nous avons utilisée jusqu'à présent est d'utiliser des manipulateurs afin de modifier les valeurs des attributs après l'initialisation de l'objet. Cette méthode est toutefois mauvaise dans le cas général car elle implique qu'il y ait un manipulateur pour chaque attribut ou que ces attributs soient publics. Elle oblige également l'utilisateur à initialiser individuellement tous les attributs, au risque que certains soient oubliés.

Une deuxième solution est de définir une méthode dédiée à l'initialisation des attributs comme dans la figure 1. Cette méthode possède un paramètre par attribut et permet, par exemple, de faire des vérifications sur les arguments avant de les affecter aux attributs. Il existe en Java des méthodes particulières appelées **constructeurs** qui font exactement cela.

```
class Rectangle {  
    private double hauteur;  
    private double largeur;  
  
    public void init(double h, double l)  
    {  
        hauteur = h;  
        largeur = l;  
    }  
    //...  
}
```

FIGURE 1

2:45

8:35

Initialisation des attributs via une méthode dédiée.

### CONSTRUCTEURS

Un constructeur est une méthode à invoquer systématiquement lors de la déclaration d'un objet et qui est chargée d'effectuer toutes les opérations requises en début de vie de l'objet en question, dont l'initialisation des attributs. La syntaxe d'un constructeur est la suivante :

```
NomClasse (liste_paramètres)  
{  
// initialisation des attributs en utilisant liste_paramètres  
}
```

Les constructeurs diffèrent des méthodes traditionnelles sur quelques points : ils n'ont pas de type de retour (pas même `void`), et ils doivent avoir le même nom que la classe. Ils peuvent par contre être surchargés.



```

class Rectangle {
    private double hauteur;
    private double largeur;

    public Rectangle(double h, double l)
    {
        hauteur = h;
        largeur = l;
    }
    public double surface()
    { return hauteur * largeur; }
    // accesseurs/modificateurs si nécessaire
    // ...
}
  
```

**FIGURE 2**

5:30

8:35

Exemple de constructeur de la classe Rectangle.

La syntaxe de **déclaration avec initialisation** d'un objet est la suivante :

```
NomClasse nomInstance = new NomClasse(valeur_arg1, ..., valeur_argN);
```

Ici, *valeur\_arg1, ..., valeur\_argN* sont les valeurs des arguments du constructeur. Si la classe en question en possède plusieurs, le constructeur appelé sera celui possédant les paramètres associés aux arguments.



## 7. CONSTRUCTEURS PAR DÉFAUT EN JAVA

### CONSTRUCTEUR PAR DÉFAUT

Le **constructeur par défaut** est un constructeur qui n'a pas de paramètres. Dans la figure 1, ce constructeur est le premier proposé, et sert à initialiser les attributs avec des valeurs par défaut. L'initialisation d'un objet étant considérée comme fondamentale, si aucun constructeur n'est spécifié, le compilateur génère automatiquement une version minimale du constructeur par défaut. On parle alors de **constructeur par défaut par défaut**. C'est celui-ci qui initialise les attributs avec les valeurs par défaut: `0` pour les entiers, `0.0` pour les doubles, `false` pour les booléens et `null` pour les objets. Il faut faire attention au fait que dès qu'un constructeur est spécifié, peu importe que ce soit un constructeur par défaut ou non, ce constructeur par défaut par défaut n'est plus fourni. Ceci est toutefois considéré comme une bonne chose, puisque si le concepteur de la classe définit un constructeur sans spécifier de constructeur par défaut, l'utilisateur se voit obligé d'initialiser explicitement les objets.

```
// Le constructeur par defaut
Rectangle() { hauteur = 1.0; largeur = 2.0; }

// 2ème constructeur
Rectangle(double c) { hauteur = c; largeur = 2.0*c; }

// 3ème constructeur
Rectangle(double h, double l) { hauteur = h; largeur = l; }
```

FIGURE 1

0:30

14:04

Exemples de constructeurs.

### EXEMPLES DE CONSTRUCTEURS

La figure 2 propose trois variantes de constructeurs pour une classe `Rectangle`, la variante A utilisant un constructeur par défaut par défaut, la B un constructeur par défaut et la C un constructeur à deux paramètres. Pour chacun de ces exemples, regardons lesquels admettent un constructeur par défaut, c'est-à-dire autorisent la tournure suivante:

```
Rectangle r1 = new Rectangle();
```

et lesquels permettent celle-ci:

```
Rectangle r2 = new Rectangle(1.0, 2.0);
```

Le constructeur par défaut par défaut étant un constructeur par défaut, la première variante autorise la première tournure qui initialise les attributs à `0.0`, mais ne permet pas la seconde, puisqu'aucun constructeur prenant deux paramètres n'est spécifié. La classe `Rectangle` B est équivalente à la première, si ce n'est que le constructeur par défaut est ici explicitement déclaré, et initialise les attributs à `0.0`. La seconde tournure est illicite pour les mêmes raisons que précédemment. Enfin, la dernière variante proposant un constructeur unique explicitement déclaré et prenant deux paramètres, elle ne possède pas de constructeur par défaut, ce qui rend la première tournure illicite, mais autorise la seconde.



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 ...
}
```

FIGURE 2

7:00

14:04

Exemples de constructeurs pour la classe Rectangle.

## APPEL AUX AUTRES CONSTRUCTEURS

En Java, il est autorisé qu'un constructeur en appelle un autre, en utilisant la syntaxe montrée dans la figure 3. Ici `this(...)` reçoit les arguments correspondant aux paramètres du constructeur que l'on souhaite appeler. Deux règles s'appliquent lorsque l'on veut utiliser cette tournure : il ne peut y avoir qu'un seul appel à un autre constructeur depuis un constructeur, et cet appel doit être la toute première ligne du constructeur.

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

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

    public Rectangle()
    {
        // appel du constructeur à deux arguments
        this(0.0, 0.0);
    }
    // suite ...
}
```

FIGURE 3

10:20

14:04

Appel à un autre constructeur.



## INITIALISATION PAR DÉFAUT DES ATTRIBUTS

Java permet, comme dans la figure 4, de donner directement une valeur par défaut aux attributs. Lorsqu'une instance de cette classe est initialisée, si le constructeur appelé ne modifie pas la valeur de cet attribut, ce dernier gardera alors cette valeur par défaut. Il est toutefois conseillé d'affecter des valeurs aux attributs à l'intérieur des constructeurs plutôt que d'utiliser les valeurs par défaut, ce qui permet d'avoir une description complète des intentions du concepteur pour l'initialisation de l'objet en regardant simplement les constructeurs.

```
class Rectangle {  
    private double hauteur = 0.0;  
    private double largeur = 0.0;  
  
    public Rectangle() {}  
  
    public Rectangle(double h, double l)  
    { //...  
    }  
    //...  
}
```

FIGURE 4

11:50

14:04

Initialisation par défaut des attributs.



## 8. CONSTRUCTEUR DE COPIE

En Java, il est possible de créer une copie d'une instance au moyen de ce que l'on appelle le **constructeur de copie**. Ce constructeur s'utilise de la même façon que les autres et prend un seul argument, soit l'instance à copier :

```
NomClasse nomInstance1 = new NomClasse(liste_arguments) ;
NomClasse nomInstance2 = new NomClasse(nomInstance1) ;
```

Ici, *nomInstance1* et *nomInstance2* sont deux instances distinctes mais ayant des mêmes valeurs pour leurs attributs (au moins juste après la copie). En Java, contrairement à d'autres langages, ce constructeur n'est néanmoins pas proposé par défaut, il faut donc l'écrire avec la même syntaxe que dans la figure 1. Le principe à suivre est donc d'affecter champ à champ les valeurs des attributs de l'instance à copier aux attributs de l'instance qui contiendra la copie.

Une autre manière de créer des copies consiste à définir une méthode en charge de la copie. Usuellement, il s'agit de la méthode `clone()` en Java. Nous reviendrons sur la copie dans la leçon 29.

```
public Rectangle(Rectangle autreRectangle)
{
    hauteur = autreRectangle.hauteur;
    largeur = autreRectangle.largeur;
}
```

FIGURE 1

1:30

4:58

Exemple d'un constructeur de copie.



## 9. FIN DE VIE, AFFECTATION, AFFICHAGE ET COMPARAISON D'OBJETS

### FIN DE VIE

On dit qu'un objet est en fin de vie lorsque le programme n'en a plus besoin, c'est-à-dire que la référence vers cet objet n'est plus utilisée dans la suite. Si, par exemple, on a une méthode `afficherUnRectangle` ne retournant rien et qui s'occupe simplement de créer une instance de la classe `Rectangle`, puis de l'afficher, la référence vers cet objet est perdue à la fin de l'exécution de la méthode et donc l'objet est en fin de vie. Il est donc utile de libérer la zone mémoire qui était associée à l'instance afin de pouvoir l'utiliser pour d'autres tâches.

Dans certains langages de programmation, le programmeur doit spécifier explicitement que l'objet est en fin de vie et qu'il faut donc libérer la zone mémoire lui étant associée, mais ce n'est pas le cas en Java. En effet, il existe un programme « ramasse-miettes », ou « garbage collector » en anglais, qui s'occupe de cette tâche et qui est lancé périodiquement pendant l'exécution d'un programme Java.

### AFFECTATION ET COPIE

Supposons que l'on souhaite créer un objet `b` à partir d'un autre objet `a` du même type et que `a` et `b` soient deux objets distincts en mémoire. Une idée possible serait d'utiliser l'opérateur d'affectation :

```
Rectangle r1 = new Rectangle(5.0, 4.0);
Rectangle r2 = r1;
```

Toutefois, comme `r1` et `r2` sont deux **références** vers un objet, seule la référence `r1` est copiée dans `r2`, et on se retrouve donc avec deux références vers la même zone mémoire. Si un attribut de `r1` venait à être modifié, le changement serait également visible sur `r2`. Si l'on veut que l'objet référencé par `r2` soit une copie distincte de celui référencé par `r1`, il faut passer soit par un constructeur de copie (comme dans la figure 1), soit par une méthode de copie comme `clone` (voir leçon 29).

```
Rectangle r1 = new Rectangle(12.3, 24.5);
Rectangle r2 = new Rectangle(r1);
```

FIGURE 1

5:30

17:52

Copie d'un objet en utilisant le constructeur de copie.

### AFFICHAGE

Lorsque l'on veut afficher un objet dans la console au moyen de la méthode `println`, seule la référence vers l'objet est affichée, ce qui n'est pas très utile. Afin de remédier à ce problème, Java prévoit qu'une méthode retourne une représentation de l'instance sous forme d'une chaîne de caractères. Cette méthode doit être déclarée dans une classe de l'objet que l'on souhaite afficher et avoir la syntaxe suivante :

```
String toString() {...}
```

Cette méthode est ensuite invoquée automatiquement par la méthode `println`. Un exemple pour la classe `Rectangle` est proposé dans la figure 2.



```

class Rectangle
{
    private double hauteur;
    private double largeur;
    //...
    public String toString()
    {
        return "Rectangle " + hauteur + " x " + largeur;
    }
}
class Exemple {
    public static void main(String[] args) {
        System.out.println(new Rectangle(4.0, 5.0));
    }
}

```

**FIGURE 2**

8:50

17:52

Méthode `toString` pour la classe `Rectangle`.

## COMPARAISON

Si l'on compare deux objets au moyen de l'opérateur `==`, on ne compare que les références des objets, qui sont dans la plupart des cas différentes même si le contenu des deux instances est identique. Java prévoit donc une méthode dédiée à la comparaison d'objets : la méthode `equals`, que l'on a déjà utilisée dans le cours (Initiation à la programmation en Java) pour comparer les chaînes de caractères. Il suffit donc d'écrire une méthode `equals` telle que celle présentée dans la figure 3 dans la classe du type que l'on désire comparer.

```

class Rectangle
{
    private double hauteur;
    private double largeur;
    //...
    public boolean equals(Rectangle autre)
    {
        if (autre == null) {
            return false;
        } else {
            return ( hauteur == autre.hauteur
                    && largeur == autre.largeur);
        }
    }
}

```

**FIGURE 3**

14:40

17:52

Exemple de définition de la méthode `equals`.

En général, on commence par vérifier que l'argument n'est pas `null`, puis on compare champ à champ les attributs des deux objets. Notons qu'il existe deux entêtes possibles pour la méthode `equals`:

```

boolean equals(UneClasse c)
boolean equals(Object c)

```

Nous expliquerons le type `Object` dans les leçons sur l'héritage (leçon 10).

## 10. HÉRITAGE: CONCEPTS

Après les notions d'encapsulation et d'abstraction, la troisième notion fondamentale de la programmation orientée objet est l'**héritage**. Pour illustrer cette notion, on peut imaginer vouloir représenter les personnages d'un jeu. On pourrait créer une classe spécifique à chaque type de personnage, qui aurait comme attributs un nom, une durée de vie mais aussi des éléments propres à son identité comme une arme pour un guerrier. Ces classes partageraient également une méthode pour rencontrer d'autres personnages qui, dans le cas d'un voleur, lui permettrait de les voler. Une telle solution dupliquerait beaucoup de code et poserait des problèmes de maintenance.

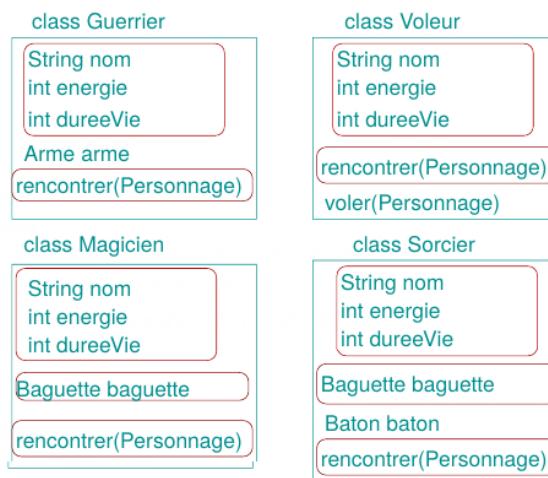


FIGURE 1

00:33

14:26

Que faire s'il y a du contenu commun à plusieurs classes pour éviter la duplication?

### HÉRITAGE

Il convient dans le cas de notre exemple d'établir une **super-classe Personnage** ayant comme attribut un nom, une durée de vie et une méthode `rencontrer(Personnage p)`. Les **sous-classes Voleur** et **Guerrier** sont des versions spécifiques de la classe **Personnage**, dont elles gardent les caractéristiques. On dit que ces sous-classes héritent de la classe **Personnage**. Les sous-classes peuvent avoir des éléments spécifiques comme une arme pour le **Guerrier** ou le fait de voler pour le **Voleur**.

En programmation, l'**héritage** permet de regrouper des caractéristiques communes dans une **super-classe** dont héritent des **sous-classes** qui en sont des versions enrichies, étendues. Cet outil permet d'établir une relation «**est-un**» : si C1 hérite de C, alors on dit que «C1 est un C», c'est-à-dire que toute instance de C1 est aussi une instance de C, comme le montre l'exemple de la figure 4. De ce fait, l'ensemble des attributs et méthodes de C (sauf les constructeurs) est disponible dans C1. C1 est **enrichie** par des attributs et méthodes supplémentaires et **spécialisée** si elle redéfinit des méthodes héritées de C (cf. fig. 2).

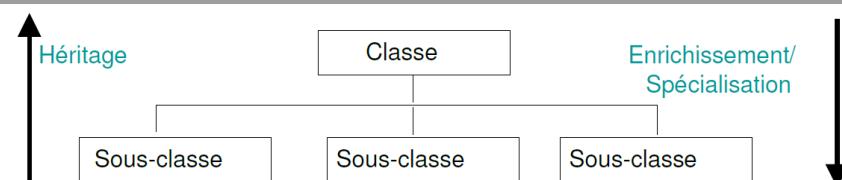


FIGURE 2

3:05

14:26

Schéma des relations d'héritage.



```
Personnage p;
Guerrier g;
// ...
p = g;
// ...
void afficher(Personnage);
// ...
afficher(g);
```

**FIGURE 3**

5:00

14:26

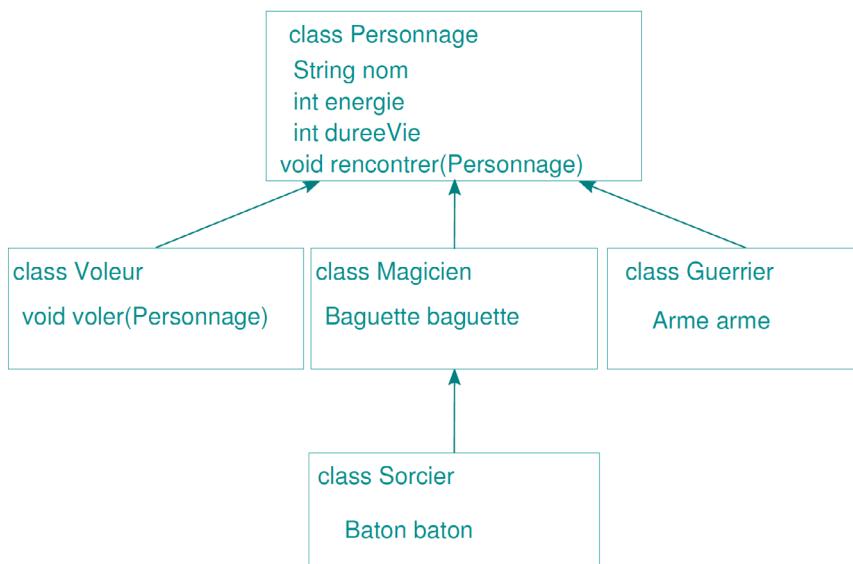
Une instance `g` de `Guerrier` est aussi une instance de `Personnage`. On peut donc copier `g` dans le `Personnage` `p`. En revanche, l'opération inverse n'est pas possible.

Dans l'exemple du jeu, les sous-classes `Voleur` et `Guerrier` sont des `Personnage` et possèdent donc toutes les caractéristiques de cette classe sans qu'il soit nécessaire de les redéfinir, mais `Guerrier` possède aussi un attribut `arme` et `Voleur` redéfinit la méthode `rencontrer(Personnage p)`.

L'héritage permet donc d'organiser et clarifier le code en explicitant les relations structurelles entre des classes, tout en limitant la redondance. L'héritage ne doit cependant jamais représenter une relation « possède-un », dans le même sens qu'une `Guerrier` possède une `arme` : ce lien doit être établi par l'encapsulation, par le fait de déclarer un attribut.

## TRANSITIVITÉ ET HIÉRARCHIE

L'héritage est transitif, c'est-à-dire que si une super-classe hérite d'une super-super-classe, alors la sous-classe possède également tous les attributs et méthodes de cette dernière. À travers l'héritage, les classes s'enrichissent progressivement et s'organisent selon un réseau de dépendances, une structure arborescente illustrée par la figure 4. Ces relations « est-un » définissent une **hiérarchie de classe** entre les classes les plus générales, classes **parentes** en haut (par exemple la classe `Personnage`) et les classes **enfant** spécialisées et enrichies, en bas.

**FIGURE 4**

2:25

14:26

Relations d'héritage (explicitées par une flèche vers la super-classe) entre différentes classes.

## DÉFINITION D'UNE SOUS-CLASSE

Pour faire hériter une classe d'une super-classe, on emploie la syntaxe suivante lors de sa déclaration :

```
class NomSousClasse extends NomSuperClasse {
    /* Déclaration des attributs et méthodes spécifiques à la sous-classe */
}
```



## 11. HÉRITAGE: DROIT D'ACCÈS PROTECTED

Dans le cadre d'une relation d'héritage, la sous-classe dispose aussi des membres privés de la super-classe mais n'y a pas accès. En effet, le droit d'accès privé limite la visibilité à l'enceinte de la classe, ce qui contraint une éventuelle sous-classe à utiliser les getters et setters prévus. Il existe un troisième type d'accès au sein d'une hiérarchie de classe: le **droit d'accès protégé**.

Le droit d'accès protégé assure la visibilité des membres d'une classe dans toutes les classes de sa descendance et se désigne par le mot-clé `protected` dans la même syntaxe d'utilisation que `public` et `private`. Le niveau d'accès protégé est une extension du niveau privé, qui accorde des droits d'accès privilégiés à toutes les sous-classes, mais également à toutes les classes du même paquetage, ce qui nuit à une bonne encapsulation. Il est donc peu recommandé d'utiliser ce droit d'accès protégé en Java.

### ACCÈS ET PORTÉE

Si la classe `Personnage` de la leçon 10 admettait l'attribut protégé `energie`, alors la classe `Guerrier` aurait accès à l'`energie`, ainsi que toute classe héritant de `Personnage` ou se trouvant dans le même paquetage. En revanche, si une classe d'un autre paquetage déclare une variable de type `Personnage` ou d'une de ses sous-classes, l'accès direct à l'attribut `energie` depuis cette variable sera refusé.

Pour conclure, alors que les membres publics sont accessibles à tous les utilisateurs d'une classe et que les membres privés ne sont accessibles qu'au programmeur de la classe, les membres protégés sont accessibles à tous les programmeurs d'extension par des sous-classes ou travaillant dans le même paquetage.

The diagram illustrates the visibility of a `protected` attribute across different packages. On the left, under 'paquetage P1', the `Personnage` class has a `protected int energie;` declaration. The `Guerrier` class, which extends `Personnage`, can access this attribute. On the right, under 'paquetage P2', the `Jeu` class cannot access the `energie` attribute of a `Guerrier` object, as indicated by a red X over the line `g.energie`.

```
class Personnage {  
    // ...  
    protected int energie;  
}  
  
class Guerrier extends Personnage {  
    // ...  
    public void frapper(Personnage lePauvre) {  
        if (energie > 0) {  
            // frapper le perso  
        }  
    }  
}  
  
paquetage P2  
class Jeu {  
    ... main (...) {  
        Guerrier g = new...;  
        g.energie  
    }  
}
```

FIGURE 1

7:55

8:34

Exemple d'accès à un attribut protégé.

## 12. HÉRITAGE: MASQUAGE

### SPÉCIALISATION

La leçon 10 annonçait la possibilité de **spécialiser**, de redéfinir des méthodes de la super-classe: par exemple, cette pratique permettrait de modifier la méthode `rencontrer()` de la super-classe `Personnage` uniquement pour la sous-classe `Guerrier`. On implémenterait donc deux versions de cette méthode, comme dans la figure 1: une dans la super-classe pour les personnages non guerriers et une autre dans cette sous-classe particulière. Il s'agit donc d'une situation de **redéfinition**: un appel à la méthode par cette sous-classe renverra à la version spécialisée, qui l'emporte sur la version héritée de la super-classe. La redéfinition désigne le partage d'un même nom méthode sur plusieurs niveaux d'une hiérarchie de classe.

Il existe aussi ce que l'on appelle le **masquage**, qui est la déclaration d'un attribut dans une sous-classe ayant le même nom qu'un attribut d'une classe plus haute dans la hiérarchie. Le masquage est souvent source de confusion et est peu courant: il consiste à nommer un attribut par le même nom (pas forcément le même type) qu'un attribut hérité, qui masquera ce dernier. En revanche, redéfinir des méthodes est courant et permet de les spécialiser. La méthode héritée est alors dite la **méthode générale**, appelée par les sous-classes qui ne la redéfinissent pas. La méthode spécifique à la sous-classe est la **méthode spécialisée**.

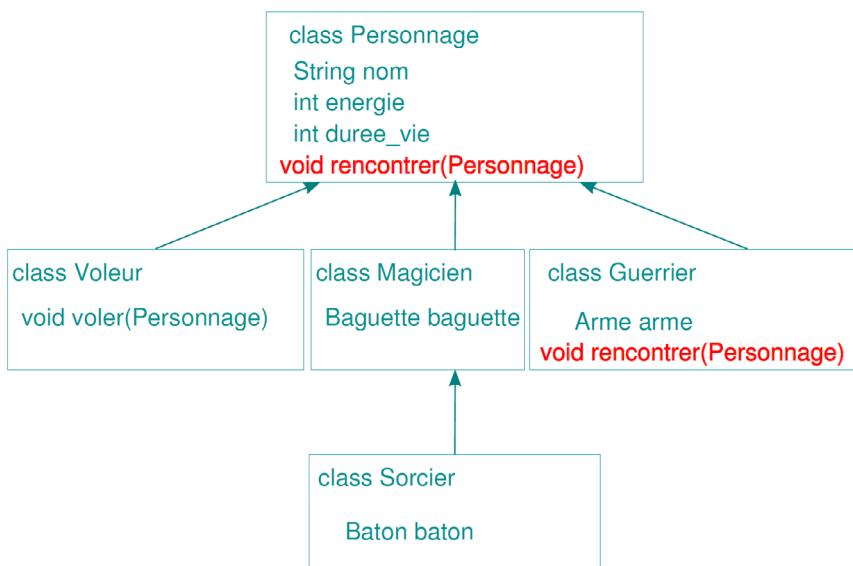


FIGURE 1

1:30

9:44

Redéfinition d'une méthode héritée dans une sous-classe.

### ACCÈS À UNE MÉTHODE MASQUÉE

Un objet dont un des membres est spécialisé dispose également du membre hérité, bien qu'il ne soit jamais appelé directement en raison des règles de résolution de portée. Dans certains cas, il est souhaitable de pouvoir désigner un membre masqué ou redéfini, par exemple pour inclure la méthode générale avant de la compléter par des actions spécifiques dans la définition de la méthode spécialisée. On utilise alors le mot réservé **super**:

`super. méthode ou attribut`

Cette syntaxe établit le lien entre le nom du membre et la classe à laquelle il appartient, et permet d'éviter des duplications de code lors de la spécialisation de méthodes.

## 13. HÉRITAGE: CONSTRUCTEURS

L'instanciation d'une classe s'accompagne d'une initialisation des attributs. Cette tâche ne peut plus être intégralement prise en charge par le constructeur d'une sous-classe : celle-ci hérite des attributs, parfois privés, de la super-classe, qu'elle ne peut pas initialiser. L'initialisation des attributs hérités doit se faire dans la classe où ils sont explicitement définis : chaque constructeur de la sous-classe fait donc appel à un constructeur de la super-classe. En Java, l'appel au constructeur de la super-classe depuis le constructeur de la sous-classe se fait au tout début du corps du constructeur au moyen du mot réservé **super** par la syntaxe suivante :

```
SousClasse(liste de paramètres)
{
    /* Arguments: liste d'arguments attendus par un des
     * constructeurs de la super-classe de SousClasse
     */
    super(Arguments),
    // initialisation des attributs de SousClasse ici
}
```

```
class FigureGeometrique {
    private Position position;
    public FigureGeometrique(double x, double y) {
        position = new Position(x,y);
    }
    // ...
}
class Rectangle extends FigureGeometrique {
    private double largeur;
    private double hauteur;
    public Rectangle(double x, double y, double l, double h) {
        super(x,y);
        largeur = l; hauteur = h;
    }
}
```

FIGURE 1

2:50

11:57

Exemple de constructeurs dans une relation d'héritage.

Le constructeur de la sous-classe `Rectangle` de la figure 1 fait par exemple appel au constructeur de sa super-classe `FigureGeometrique` au début de son constructeur, en lui donnant les arguments pour initialiser son attribut `position`.

Un autre exemple permet de souligner que les attributs supplémentaires dans une sous-classe ne sont pas nécessaires : on peut définir un carré comme une sous-classe de `Rectangle`, dont les dimensions sont égales. Cette classe `Carre` n'introduit pas d'attributs supplémentaires mais définit bien son constructeur, proposé en figure 2, et pourrait redéfinir des manipulateurs hérités comme `setHauteur()` et `setLargeur()` afin de maintenir ses deux attributs égaux. On note que son constructeur fait appel au constructeur de `Rectangle`, avec deux valeurs égales, selon la même syntaxe donnée.

```
class Carré extends Rectangle {
    public Carré(double taille) {
        super(taille, taille);
    }
    /* Et c'est tout !
    (sauf s'il y avait des manipulateurs, il
    faudrait alors sûrement aussi les redéfinir) */
}
```

FIGURE 2

6:00

11:57

Exemple de constructeurs dans une relation d'héritage qui n'ajoute pas d'attribut.

Un appel explicite au constructeur de la super-classe n'est pas nécessaire si elle possède un constructeur par défaut, puisque l'appel est alors effectué par le compilateur. Si au contraire la classe n'a pas de constructeur par défaut, l'appel doit être explicite, comme dans les exemples précédents, pour éviter des erreurs.

Pour rappel, le constructeur par défaut, qui ne prend pas d'arguments, existe par défaut si la classe n'a pas de constructeur, mais disparaît et doit être réécrit dès la création d'un constructeur. Il est conseillé de toujours déclarer au moins un constructeur pour chaque classe et d'effectuer un appel explicite à un constructeur de la super-classe, même à celui par défaut.

## ORDRE D'APPEL DE CONSTRUCTEURS

Dans une relation d'héritage, la construction d'une sous-classe appelle d'abord le constructeur de la super-classe la plus générale puis, dans l'ordre, les constructeurs des super-classes qui en héritent, avant de terminer par l'initialisation de la partie spécifique de la classe instanciée.

Soit, par exemple, une classe **C** héritant d'une classe **B** qui elle-même hérite d'une classe **A**, relation illustrée par la figure 3. La construction d'une instance de **C** appelle, explicitement ou non, le constructeur de **B** qui appelle celui de **A**. Comme ces appels sont placés en première position dans la liste d'initialisation, le constructeur de **C** va commencer par exécuter la construction de sa partie **A**, classe la plus générale dont on dérive, en donnant des valeurs à ses attributs **a1** et **a2**. Cette première étape conclue, le constructeur de **B** initialise **b1** et construit ainsi la partie **B** de l'instance. Enfin, le constructeur de **C** peut initialiser la partie spécifique à cette sous-classe, les attributs **c1** et **c2**, ce qui termine la construction de l'instance. On aura bien suivi un schéma de construction de la partie la plus générale à la plus spécifique de la classe.

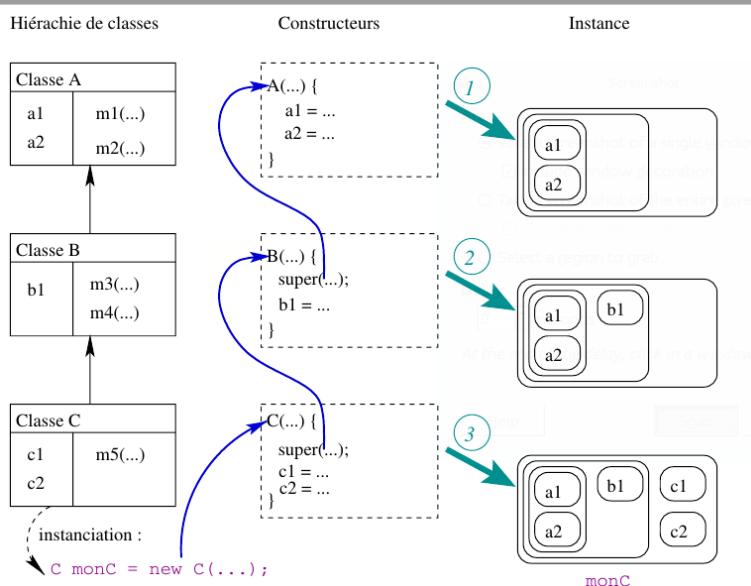


FIGURE 3

8:56

11:57

Illustration de l'ordre d'appel des constructeurs dans une relation hiérarchique.



## 14. POLYMORPHISME : INTRODUCTION

La dernière notion fondamentale de la programmation orientée objet est le **polymorphisme**. Il permet à un même code de s'adapter aux types des données auxquels il s'applique. Ainsi, il rend le code générique, écrit de façon unifiée pour différents types de données.

### RÉSOLUTION DES LIENS

Comme nous l'avons vu dans la leçon 10, un objet d'une sous-classe hérite du type de sa super-classe et cette relation est transitive: un objet peut avoir plusieurs types. Si l'on regarde le code de la figure 1, on peut se demander quelle version de la méthode `rencontrer(Personnage)` va être exécutée. Il y a en programmation deux approches à ce problème: la première consiste à regarder le type apparent, c'est-à-dire le type de la variable qui contient la référence vers l'objet. Ici, le guerrier est contenu dans une variable de type `Personnage`, et c'est donc la méthode `rencontrer` de la classe `Personnage` qui serait exécutée. Cette approche s'appelle la **résolution statique des liens**. La deuxième approche est différente car elle observe le type effectif, celui de l'objet stocké dans la variable. Comme c'est un `Guerrier` qui est stocké dans la variable `unPersonnage`, la méthode utilisée sera la version spécialisée de la classe `Guerrier`. On appelle cette approche la **résolution dynamique des liens**. En Java, c'est cette dernière qui est mise en œuvre, comme dans l'exemple de la figure 2: si l'on stocke dans le tableau `adversaires` un `Sorcier` et un `Guerrier`, la méthode `rencontrer(Personnage)` de `Personnage` est appelée pour le sorcier, car celui-ci ne spécialise pas cette méthode, mais `rencontrer(Personnage)` de `Guerrier` est appelée pour le guerrier, puisque cette classe redéfinit la méthode `rencontrer`.

Grâce à l'héritage du type et la résolution dynamique des liens, nous pouvons mettre en œuvre le dernier concept de la programmation orientée objet, appelé **polymorphisme**, qui permet qu'un même code puisse s'exécuter de façon différente selon la donnée à laquelle il s'applique.

```
// ...
Personnage unPersonnage = new Guerrier(...);
unPersonnage.rencontrer(unAutrePersonnage);
```

FIGURE 1

3:20

8:26

Choix de la méthode à exécuter.

```
class Jeu {
    private Personnage joueur;
    private Personnage[] adversaires;
    // ...
    public void tourDeJeu() {
        for (int i = 0; i < adversaires.length; ++i)
        {
            adversaires[i].rencontrer(joueur);
        }
    }
    // ...
}
```

FIGURE 2

6:50

8:26

Résolution dynamique des liens.

## 15. CLASSES ET MÉTHODES ABSTRAITES

Avec l'**héritage**, il est possible de généraliser des concepts qui sont présents dans plusieurs classes du programme en les incorporant à une super-classe. Ainsi, on construit une structure arborescente (fig. 1) reliant différents éléments à un niveau plus abstrait. Cela fait partie des mécanismes d'**abstraction** de l'orientée objet. Toutefois, au niveau le plus élevé d'une hiérarchie de classe, il est parfois impossible de définir une méthode générale qui devra pourtant exister dans toutes les sous-classes. Par exemple, calculer la surface d'une figure géométrique, `FigureFermee`, quelconque se révèle difficile, tandis qu'à un niveau plus bas on peut le mettre en œuvre. Une telle méthode `surface()` au niveau de la super-classe `FigureFermee` pourrait être nécessaire pour calculer un volume ou simplement par souci d'abstraction et d'unification des données.

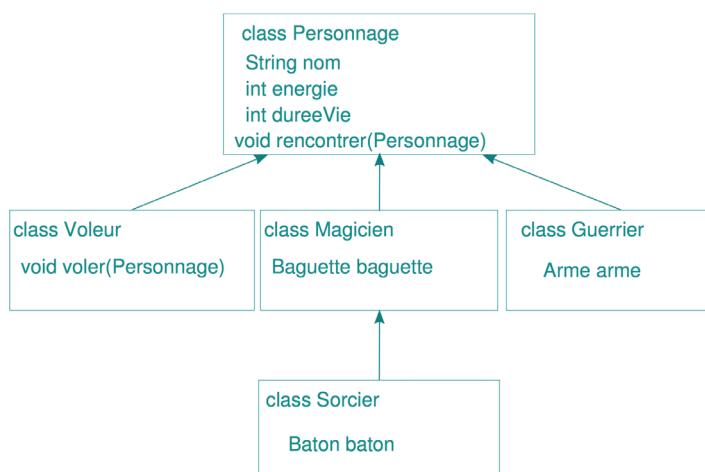


FIGURE 1

00:42

15:09

Exemple d'un arbre d'héritage.

### MÉTHODES ABSTRAITES

La solution à ce problème sont les méthodes abstraites. En Java, ces méthodes sont précédées par le mot-clé `abstract`, elles ne possèdent pas de corps et se terminent par un point-virgule.

```
public abstract double surface();
```

Ceci permet d'imposer à toutes les sous-classes que l'on souhaite instancier une redéfinition de cette méthode héritée. Cela veut dire que les sous-classes que l'on veut instancier seront obligées de spécifier une définition concrète pour la méthode.

### CLASSES ABSTRAITES

Toute classe possédant au moins une méthode abstraite est automatiquement une classe abstraite. Ces classes sont alors **non instanciables** et doivent être précédées par le mot-clé `abstract`. C'est-à-dire qu'il est impossible de créer une instance à partir d'une classe abstraite.

```
public abstract class Forme {...}
```

Ces classes permettent de définir des concepts génériques qui sont communs à toutes les sous-classes mais qui sont trop abstraits pour être codés en toute généralité.



## HÉRITAGE DE MÉTHODES ABSTRAITES

Une classe est abstraite si elle contient une méthode abstraite. Cela reste vrai pour des méthodes abstraites héritées. Plus concrètement, une sous-classe dérivant directement ou indirectement d'une classe abstraite doit redéfinir toutes les méthodes abstraites héritées pour devenir non abstraite c'est-à-dire pour que l'on puisse en créer des instances.

La figure 2 illustre cette notion: la classe `Cercle` n'est pas abstraite puisqu'elle redéfinit les méthodes abstraites héritées de son ascendance (fig. 1), alors que la classe `Polygone` le reste puisqu'elle ne donne pas de définition de `surface()`. On ne pourra donc pas déclarer d'instance de `Polygone`.

```
class Cercle extends FigureFermee {  
    private double rayon;  
  
    public double surface() {  
        return Math.PI * rayon * rayon;  
    }  
    public double perimetre() {  
        return 2.0 * Math.PI * rayon;  
    }  
}
```

```
class Polygone extends FigureFermee {  
    private ArrayList<Double> cotes;  
  
    public double perimetre() {  
        double p = 0.0;  
        for (Double cote : cotes) {  
            p += cote;  
        }  
        return p;  
    }  
}
```

FIGURE 2

13:00

15:09

Héritage des classes abstraites. `Cercle` n'est pas une classe abstraite. `Polygone` reste par contre une classe abstraite.



# 16. HÉRITAGE ET POLYMORPHISME : COMPLÉMENTS

## CONSTRUCTEURS ET POLYMORPHISME

Un constructeur a comme vocation d'initialiser l'instance courante. Il est donc par essence non polymorphique. S'il est possible d'invoquer des méthodes polymorphiques dans un constructeur, cela reste fortement déconseillé, comme le montre l'exemple de la figure 1.

En effet, l'affichage de la valeur de l'attribut `b` se fait alors que l'initialisation de l'instance à laquelle il se rapporte n'est pas finalisée. La valeur affichée pour l'attribut `b` ne correspond pas à celle qu'aura l'objet une fois construit.

```
abstract class A
{
    public abstract void m();
    public A() {
        m(); // méthode invocable de manière polymorphe
    }
}
class B extends A {
    private int b;
    public B() {
        b = 1; // A() est invoquée implicitement juste avant
    }
    public void m() { // définition de m pour la classe B
        System.out.println("b vaut : " + b);
    }
}
// .... dans le programme principal :
B b = new B();
```

affiche : b vaut 0

FIGURE 1

1:10

13:49

Appel d'une méthode polymorphe dans un constructeur.

## SUPER-CLASSE OBJECT

Nous connaissons déjà quelques méthodes qui semblent apparaître dans toutes les classes en Java, par exemple: `toString`, `equals`. Mais d'où viennent ces méthodes exactement ? En Java, tout type évolué est en réalité une sous-classe de la classe `Object`. Cela veut dire que tous les objets héritent de cette **super-classe universelle** à la racine de leur arbre d'héritage. Ce lien est automatique et nous n'avons pas à le spécifier dans le code.

Tout objet a donc aussi pour type `Object`. Cette classe contient un certain nombre de méthodes utiles comme les deux citées plus haut. Ces méthodes définissent le comportement par défaut pour une classe. Mais dans la plupart des cas, on les redéfinit pour qu'elles satisfassent mieux au cas particulier. Par exemple, la méthode `toString()` par défaut n'affiche que la référence vers l'objet sous la forme d'une adresse: `@6d06d69c`. Ce qui n'est pas très utile normalement. La méthode `equals` standard compare les références de deux objets au moyen de `==` et renvoie `true` si et seulement si elles sont égales. Là aussi, nous aimerais un comportement plus spécifique dans la majorité des cas. Pour mettre cela en œuvre, il faut redéfinir ces méthodes dans nos propres classes.

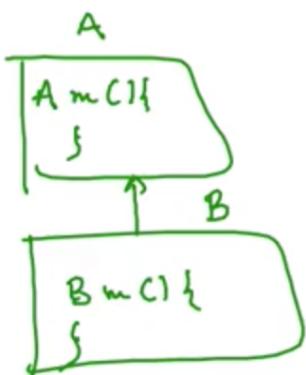


FIGURE 2

8:12

13:49

Redéfinition de méthode avec des types de retour compatibles.

## SURCHARGE ET REDÉFINITION

Il existe en Java deux façons possibles d'avoir des méthodes avec le même nom dans une seule classe : la surcharge (*overloading*) et la redéfinition (*overriding*).

S'il existe plusieurs méthodes avec le même nom dans une classe, nous parlons de **surcharge**.

La **redéfinition** n'existe que pour des méthodes héritées. Si une sous-classe définit une méthode qui est déjà spécifiée dans une super-classe en utilisant exactement le même nom, la même liste de paramètres et un type de retour compatible alors la méthode est redéfinie. Il s'agit dès lors d'une méthode polymorphe. Ce procédé a déjà été vu pour les méthodes abstraites (leçon 15).

Pour les types de base, un **type de retour compatible** est simplement ce même type. Par exemple, une méthode qui retourne un `int` ne peut qu'être redéfinie par une méthode qui retourne aussi un `int`. Pour les types évolués, il est possible de retourner un objet d'une sous-classe du type de retour de la méthode héritée. Par exemple, les types de retour de la méthode `m()` sur la figure 2 sont compatibles parce que la classe `B` hérite de la classe `A`.

## MÉTHODE EQUALS

En comparant les méthodes `equals` que nous avons vues jusqu'à présent avec celles de la classe `Object`, on peut remarquer qu'elles n'ont pas toutes des listes d'arguments identiques. Cela est dû au fait que la classe `Object` ne peut pas connaître a priori toutes ses sous-classes. Ainsi sa méthode `equals` prend en argument un objet de type `Object` pour garantir que l'on puisse comparer tout objet à tout objet. Pour sa redéfinition, il faut respecter cette contrainte.

Une méthode `equals` doit remplir quelques conditions. Elle doit être symétrique (si `a.equals(b)` alors `b.equals(a)`). Elle doit être transitive. Ainsi, si `a.equals(b)` et `b.equals(c)` alors `a.equals(c)`. La méthode se doit de ne plus retourner `false` si on lui donne l'argument `null`. Une façon de mettre tout cela en œuvre est illustrée en figure 3.

```

class Rectangle {
    //...
    public boolean equals(Object autreObjet) {
        if (autreObjet == null)
            { return false; }
        else {
            if (autreObjet.getClass() != getClass())
                { return false; }
            else {
                Rectangle r = (Rectangle)autreObjet;
                return (largeur == r.largeur &&
                        hauteur == r.hauteur);
            }
        }
    }
}
  
```

FIGURE 3

11:06

13:49

Exemple de redéfinition de la méthode `equals` pour un rectangle.

D'abord, on vérifie que l'argument n'est pas `null`. Puis, il faut s'assurer que l'argument est du même type que l'objet lui-même. Et seulement dans ce cas, on peut comparer le contenu des attributs des deux objets. Mais comme l'objet pris en argument est stocké dans une variable de type `Object`, le compilateur n'est pas sûr qu'il ait vraiment un objet de type `Rectangle`. Pour garantir la présence des attributs à comparer nous sommes donc obligés d'effectuer un transtypage vers le type `Rectangle`.



## 17. LE MODIFICATEUR FINAL

Le modificateur `final` a plusieurs fonctions. Il peut s'appliquer à des variables, des méthodes et des classes, et on l'écrit après le modificateur de droit d'accès. Pour illustration, une méthode finale :

```
public final int plus() { ... }
```

### MÉTHODES FINALES

L'intérêt d'avoir des méthodes finales est que ces méthodes ne peuvent pas être redéfinies dans des éventuelles sous-classes de la classe. Cela veut dire que l'on peut fixer le comportement de la méthode une fois pour toutes. Et si l'on essaye de redéfinir la méthode plus bas dans la hiérarchie d'héritage, il y aura un message d'erreur du compilateur. Ainsi, il nous est possible de spécifier une méthode qui se comporte de la même manière pour toutes les sous-classes.

### CLASSES FINALES

Une classe définie comme finale ne peut alors pas avoir de sous-classe, il est impossible de l'étendre par héritage. Cette contrainte paraît très stricte au début, mais elle est assez utile. On déclare finale une classe pour empêcher les programmeurs d'extensions de l'étendre. C'est par exemple le cas pour la classe `String` de l'API de Java. Si cela nous empêche d'améliorer les algorithmes de ses méthodes dans une sous-classe de `String`, il est par contre garanti que tout `String` aura un comportement bien défini et connu de tous. Les concepteurs de Java assurent ainsi que personne ne sera capable de créer une nouvelle sous-classe de `String` avec un comportement complètement différent. Ce qui aboutirait à un code très difficile à comprendre, un objet de type sous-classe de `String` pouvant être stocké dans une variable de type `String`.

```
class Personnage {
    private final int AGE_MAX = 100;
    public Personnage () {
        AGE_MAX = 900;
    }
}
```

FIGURE 1

7:09

11:18

Il n'est pas possible d'affecter deux fois une valeur à un attribut final.

### VARIABLES FINALES

Pour des variables le modificateur `final` ne signifie pas la même chose que pour les méthodes. À une variable finale, on ne peut affecter une valeur qu'une seule fois. Cette valeur ne pourra pas être modifiée par la suite. Cela est vrai pour les variables locales, les variables d'instance ainsi que pour les paramètres d'une méthode. Par exemple, si nous avons un attribut (variable d'instance) qui est déclaré `final`, nous pouvons lui affecter une valeur soit lors de son initialisation par défaut soit dans le constructeur, mais pas les deux. S'il existe plusieurs constructeurs, on doit lui affecter une valeur dans tous les constructeurs. Cela ne pose pas de problème car un seul constructeur est exécuté par objet.



## VARIABLES FINALES DE TYPE ÉVOLUÉ

Comme nous l'avons vu, Java ne stocke que la référence vers un objet dans une variable de type évolué. Cet état de fait peut engendrer des problèmes de compréhension en relation avec des variables finales qui contiennent un objet. Si une variable contenant la référence à un objet est finale, elle ne peut pas contenir la référence à un autre objet. La variable ne peut pas être modifiée. Cela ne signifie cependant pas que l'objet référencé par la variable ne puisse pas être modifié! La figure 2 montre un exemple où le paramètre d'une méthode est final. Il est indiqué dans le commentaire qu'il est impossible d'affecter une nouvelle instance à la variable `c`. Mais nous voyons qu'il est tout à fait possible de modifier cet objet via la référence `c` et la méthode `setValeur`. Il faut alors garder en tête que `final` n'est pas équivalent à non modifiable pour l'objet référencé.

```
class Conteneur {  
    private int valeur;  
    public void setValeur(int val) { valeur = val; }  
}  
  
class Test {  
    public static void main(String[] args) {  
        Conteneur c = new Conteneur();  
        c.setValeur(42);  
        modifier(c);  
    }  
    static void modifier(final Conteneur c) {  
        c.setValeur(-1); // modifie l'objet référencé !!  
        //c = new Conteneur(); //FAUX  
    }  
}
```

FIGURE 2

8:30

11:18

Un objet de type évolué est modifiable même si sa référence est finale.



## 18. ATTRIBUTS STATIQUES

Nous connaissons maintenant trois types de variables. Les variables locales qui sont déclarées dans un corps de méthode, les paramètres de méthodes et les variables d'instance. Ces dernières sont aussi appelées les attributs et elles stockent les valeurs spécifiques à une instance de la classe en question.

Prenons pour exemple une classe `Employe` représentant un employé qui prendra sa retraite à l'âge de 65 ans. Un attribut `ageRetraite` pourrait être défini dans la classe, contenant 65. Le problème est que cette valeur sera stockée dans chaque instance séparément. Si la loi change et que l'âge de retraite est élevé à 67 ans, nous sommes obligés de changer cet attribut instance par instance et ce n'est clairement pas faisable.

### VARIABLES STATIQUES

En Java, il est possible de définir des **variables de classe** ou **statiques** pour les attributs (pas pour les variables locales). On les écrit en ajoutant simplement le mot réservé `static` à la définition.

```
private static int ageRetraite = 65;
```

Les variables statiques sont des attributs qui vont être partagés par toutes les instances d'une classe. Et si leur valeur change, elle change pour toutes les instances. Ainsi, les variables statiques n'ont pas besoin d'objets pour être utilisables. Elles sont initialisées au chargement du programme et elles peuvent être appelées sans création préalable d'une instance de la classe. Un tel appel se présente ainsi :

```
Employe.ageRetraite = 67;
```

On utilise le nom de la classe suivi du nom de la variable pour accéder à un attribut statique sans devoir s'occuper d'une instance. Mais il est également possible de manipuler la variable statique comme l'attribut normal d'un objet :

```
Employe e = new Employe();
e.ageRetraite = 66;
```

### COMPARAISON STATIQUE – NON STATIQUE

Les variables statiques (variables de classe) ont toutes les propriétés des attributs non statiques (variables d'instance) : elles sont visibles dans toute la classe, elles sont déclarées en dehors des méthodes et elles sont héritées par les sous-classes. Mais il existe deux grandes différences entre ces deux sortes de variables. Les variables d'instance sont stockées dans la zone mémoire de leur instance. À chaque fois qu'un tel objet est créé, Java réserve une nouvelle zone mémoire et y introduit les valeurs. Les attributs statiques sont stockés dans une **zone de mémoire unique** liée à la classe et non aux instances. Et ils y sont initialisés au début du programme.

Cela implique plusieurs choses dans l'exemple de la classe `Employe`. D'abord, nous pouvons déjà accéder à la variable `ageRetraite` avant même d'avoir créé un employé. De plus, si nous avons beaucoup d'instances d'`Employe` et si nous désirons changer l'âge de retraite pour toutes les instances, nous pouvons simplement changer la valeur de la variable et ce changement sera visible dans tous les objets.

Par contre, si l'on change la valeur d'un attribut d'instance, elle ne change que pour son instance. En bref, si nous n'avions pas déclaré `ageRetraite` statiquement, nous serions obligés de parcourir toutes les instances afin de changer sa valeur !



## UTILISATION

Comme l'encapsulation est un des fondements de la programmation orientée objet, on peut considérer les variables statiques comme un possible contournement de cette règle. Elles peuvent en effet être employées sans le recours à un objet. La bonne manière de faire est d'utiliser les attributs statiques uniquement pour représenter des valeurs communes à tous les objets d'une classe et, typiquement, des constantes. On leur ajoute alors le mot-clé `final` (leçon 17) pour qu'elles ne changent pas après leur déclaration.

La classe `Math` de l'API fonctionne ainsi et l'on accède à la valeur `PI` de manière suivante:

`Math.PI`

```
class Planete {
    // G = constante gravitationnelle
    // Une variable G pour chaque planète :
    // Possible
    private final double G = 6.674E-8;

    // Une variable G pour toutes les planètes :
    // BEAUCOUP MIEUX !
    private final static double G = 6.674E-8;
    // ...
}
```

FIGURE 1

12:24

13:21

Exemple de déclaration d'une constante.

## EXEMPLE CONCRET

Pour illustrer cette notion, étudions une instruction souvent utilisée dans le cours jusqu'ici, mais dont nous ne connaissons pas encore le fonctionnement exact:

`System.out.println();`

`System` est clairement le nom d'une classe (lettre majuscule).`out` est appelé sur cette classe: `out` est donc forcément une variable statique de cette classe (fig. 2). Finalement, `println()` est une méthode de la classe `PrintStream` dont `out` est une instance. L'instruction ci-dessus appelle alors la méthode d'un attribut statique de la classe `System`.

```
class System {
    //...
    static PrintStream out = new PrintStream(...);
    //...
}
class PrintStream {
    void println (...)
    {...}
    //...
}
```

FIGURE 2

12:57

13:21

La classe `System` contient l'attribut statique `out`.



## 19. MÉTHODES STATIQUES

La leçon 18 a présenté les variables statiques qui sont des attributs spécifiques à une classe. La même chose existe pour les méthodes d'une classe, à savoir les **méthodes statiques**. Il suffit d'ajouter le mot-clé `static` à la définition d'une méthode pour qu'elle devienne une méthode statique ou méthode de classe. Comme les variables statiques, elles sont accessibles sans avoir besoin d'une instance de la classe en question. Effectivement, elles sont déjà chargées au début du programme et elles sont aussi invocables au moyen du nom de la classe (fig. 1). Bien sûr, nous pouvons également appeler une méthode statique sur un objet existant, néanmoins cette manière de faire n'est pas usitée.

```

class A {
    static void methode1() {
        System.out.println("Méthode 1");
    }
    void methode2() {
        System.out.println("Méthode 2");
    }
}
class ExempleMethodeStatique {
    public static void main(String[] args) {
        A.methode1(); // OK
        A.methode2(); // Non !
        A v = new A();
        v.methode1(); // OK, alternative
        v.methode2(); // OK (comme d'habitude)
    }
}
  
```

FIGURE 1

1:42

9:22

Exemple d'utilisation d'une méthode statique.

### RESTRICTIONS

Comme les méthodes statiques sont invocables indépendamment de toute instance d'une classe, il n'est pas garanti qu'un objet existe lors de l'exécution du programme. C'est pour cette raison que l'on n'a pas le droit d'utiliser les membres non statiques de la même classe dans une méthode statique. Typiquement, la référence `this` et les variables d'instances sont interdites. Par contre, il est permis d'appeler toutes les méthodes et variables statiques de la classe. Il est également possible de créer un objet dans une méthode statique et d'utiliser les méthodes non statiques de l'objet que l'on vient de créer via cet objet.



```

class A {
    int i;
    static int j;
    void methode1() {
        System.out.println(i); // OK
        System.out.println(j); // OK
        methode2();           // OK
    }
    static void methode2() {
        System.out.println(i); // Faux
        System.out.println(j); // OK
        methode1();           // Faux
        methode2();           // OK (sauf recursion infinie)
        A v = new A();
        v.methode1();         // OK
    }
}

```

FIGURE 2

4:26

9:22

Exemple d'appels depuis une méthode statique.

## UTILISATION

Les méthodes statiques sont utilisées pour déclarer des méthodes très générales qui ne sont pas vraiment spécifiques à un objet. Ce sont typiquement des **méthodes auxiliaires**, aussi appelées **méthodes outils**. Là aussi il y a risque de contourner l'approche orientée objet, ce qui est bien sûr à éviter.

En résumé, il est important de ne pas utiliser le mot-clé `static` sans une bonne raison. Normalement, les bonnes raisons sont les trois suivantes :

- définir des constantes avec le mot-clé supplémentaire `final` (courant);
- définir des valeurs communes à toutes les instances qui peuvent changer (rare);
- définir des méthodes «outils» qui ne sont pas attachées à un objet (moins courant), d'autant que les interfaces (voir leçon 20) permettent désormais aussi de regrouper un certain nombre de méthodes utilitaires.

## EXEMPLES CONCRETS

Nous avons déjà vu les constantes de la classe `Math` de l'API Java (leçon 18). Cette classe contient aussi des méthodes statiques qui représentent les fonctions mathématiques les plus connues, comme la racine carrée ou des fonctions trigonométriques :

```
Math.sqrt(25);  Math.sin(90);
```

Finalement, nous sommes aussi capables maintenant de comprendre l'en-tête de la méthode `main` qui est le point de départ pour tout programme en Java.

```
public static void main(String[] args) {...}
```

Il s'agit d'une méthode statique, ce qui explique pourquoi l'on ne peut appeler que des méthodes auxiliaires statiques dans son corps et aussi pourquoi l'utilisation de `this` y est interdite. Dans un deuxième temps, il devient également clair que la méthode `main` doit être statique puisqu'il est impossible de créer un objet avant le départ du programme. Elle doit donc être invocable sans instance.

## 20. INTERFACES

Java ne tolère que l'héritage simple; chaque classe ne pouvant avoir qu'une seule super-classe. Il s'agit là d'un choix des concepteurs du langage lié au fait que la gestion de l'héritage multiple peut être lourde; cette notion générant des ambiguïtés potentielles. L'héritage multiple est possible dans d'autres langages de programmation, comme le C++. Dans certaines situations, le seul héritage simple est insuffisant à permettre de produire une bonne conception du programme.

Prenons l'exemple de la figure 1. Il s'agit de modéliser un jeu impliquant des entités de natures diverses ([Joueur](#), [Raquette](#), [Balle](#), [Filet](#) ...). Ces entités évoluent toutes en cours de jeu, mais n'ont pas toutes les mêmes caractéristiques: certaines doivent être affichables graphiquement, d'autres sensibles aux clics de souris, etc. Dans le cadre de cet exemple, il serait très naturel d'avoir recours à de l'héritage multiple, comme le montre la figure 1.

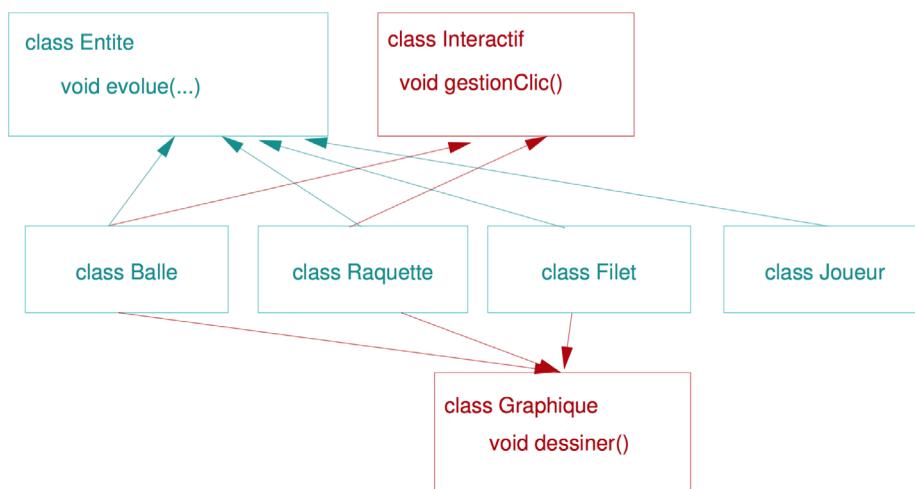


FIGURE 1

1:34

16:03

Un problème de conception où il serait naturel de recourir à l'héritage multiple.

Java ne nous le permettant pas. Quelles seraient les alternatives? Celle consistant à placer toutes les méthodes impliquées dans une seule super-classe n'est pas satisfaisante. Pourquoi doter un objet qui n'est pas affichable graphiquement d'une méthode d'affichage par exemple? Ici la seule méthode qu'il est correct de considérer comme commune à toutes les entités est la méthode `evolve()`. Les autres sont plus spécifiques et surtout sont non pertinentes pour certaines sous-classes. Nous atteignons donc ici les limites de ce que nous permet d'exprimer l'héritage simple. La notion d'**interface** pare à cette problématique. Elle permet d'imposer un contenu à des classes sans pour autant mettre en place un lien d'héritage.

```

interface Graphique {
    void dessiner();
}
interface Interactif {
    void gestionClic();
}
  
```

FIGURE 2

5:37

16:03

Exemples de définitions d'interfaces.

### DÉFINITION AVANT JAVA 8

Une interface s'écrit presque comme une classe. Au lieu du mot-clé `class` on écrit `interface`:

```
public interface Interactif { ... }
```

Les interfaces ressemblent beaucoup aux classes par leur contenu, mais elles ont été conçues initialement pour ne pouvoir comporter que des méthodes abstraites et des constantes. Cela a changé depuis Java 8. La leçon 21 sera consacrée à ces changements.



Comme les classes abstraites, les interfaces ne peuvent pas être instanciées. Elles ne possèdent pas de constructeur. Les méthodes abstraites sont celles qui doivent être imposées à certaines classes. Java ne nous oblige pas à ajouter le mot réservé `abstract` puisque toutes ces méthodes seront nécessairement abstraites, ainsi que publiques. Le mot-clé `public` n'est pas nécessaire non plus (fig. 2). Il en va de même pour les constantes d'une interface: elles sont toutes publiques. Mais comme ce sont des constantes, elles sont aussi toutes finales et statiques. Ces constantes sont initialisées en même temps qu'elles sont déclarées:

```
public interface Interactif {  
    int CONSTANTE = 3;  
}
```

Les interfaces peuvent être liées entre elles par un lien d'héritage. Les sous-interfaces héritent du contenu des super-interfaces, et le lien d'héritage est établi par le mot-clé `extends`.

```
interface Interactif { ...}  
interface GerableParSouris extends Interactif { ... }  
interface GerableParClavier extends Interactif { ... }
```

FIGURE 3

10:14

16:03

Héritage entre des interfaces.

## IMPLÉMENTATION

Afin de contraindre une classe à définir une méthode sans pour autant la faire hériter d'une classe abstraite, il faut la lier à une interface contenant la méthode en question.

```
public class Balle extends Entite implements Interactif {...}  
public class A implements B, C {...}
```

Une classe peut parfaitement implémenter plusieurs interfaces. Mais qu'entend-on par implémenter une interface? C'est le fait qu'une classe liée à une interface est forcée de donner une définition concrète de toutes les méthodes abstraites de l'interface si on veut pouvoir l'instancier. Ceci est analogue à ce que nous avons vu pour les classes abstraites (leçon 15).

Un point important est que comme une méthode d'interface est automatiquement publique, son implémentation dans une classe devra l'être aussi. Ceci est valable partout en Java: lors d'une redéfinition, il n'est permis que d'élargir les droits d'accès à une méthode et non de les restreindre.

## AMBIGUITÉS

Si une classe implémente plusieurs interfaces et que deux interfaces possèdent le même entête de méthode abstraite, il n'y a en réalité pas d'ambiguïté car ce que fait la méthode n'est pas encore défini. La méthode choisie sera de toute façon celle présente dans la classe implementant l'interface. Par contre, si deux interfaces ont une constante avec le même nom, il y a effectivement ambiguïté: quelle valeur utiliser concrètement. De telles constantes ne pourront être utilisées sans indiquer explicitement à quelle interface elles appartiennent, sinon le compilateur indiquera une erreur.



## CONCEPTS CLÉS

Nous connaissons déjà deux formes de relation entre classes. La composition/délégation est le cas de figure où une classe possède un attribut qui est l'instance d'une autre. C'est une relation « *a-un* ». L'héritage lui est par contre une relation de type « *est-un* ».

Les interfaces introduisent un nouveau type de relation ; le lien « **se comporte comme** ». C'est d'ailleurs ici, bien au-delà de la problématique de l'héritage multiple, la vocation première d'une interface, celle d'imposer un « *contrat* » de comportement aux classes qui l'implémente (une *Raquette* est une entité de *Jeu* qui se comporte comme un objet *Graphique*).

La figure 4 montre comment se présenterait une conception utilisant les interfaces pour le problème du jeu avec balles, raquettes et filets présenté en introduction.

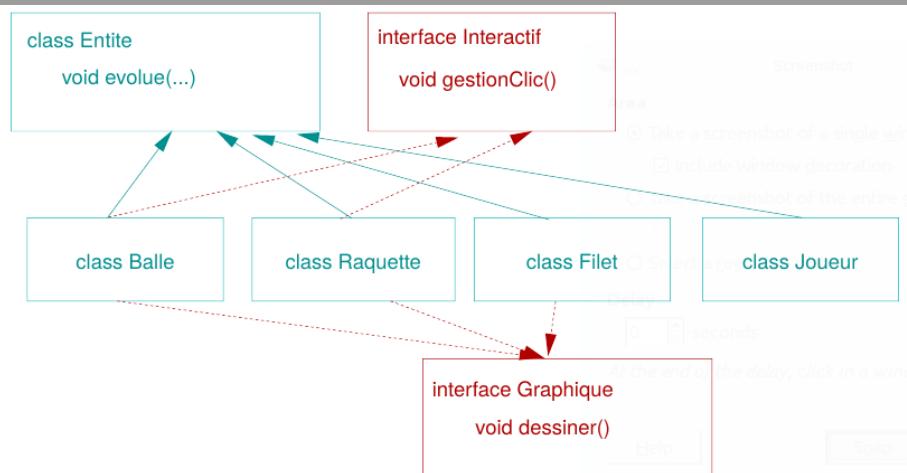


FIGURE 4

12:41

16:03

Les interfaces représentent un lien de type « *se comporte comme* » (les liens « *implements* » sont en traités bleus).

## OBJETS DE TYPE INTERFACE

Une interface définit un nouveau type dans un programme. Un objet implementant une interface *I* possède donc *I* comme type supplémentaire et il est possible de le traiter sous cette étiquette (une instance de *Raquette* peut être manipulée comme entité de type *Graphique*).

En clair, en Java, nous pouvons déclarer des variables ayant pour type celui d'une interface. La création des objets affectés à ces variables devra cependant nécessairement se faire pas le biais des constructeurs de la classe à laquelle ils appartiennent ; **il n'y a pas de constructeurs dans les interfaces**.

Dans le cas de notre exemple, il est possible de créer un tableau de *Graphique* et de le remplir, sans autres, d'objets de type *Balle*, *Raquette* et *Filet*. Assigner un *Graphique* à une *Balle* nécessiterait par contre un transtypage.

```
Graphique graphique;
Balle balle = new Balle(..);
graphique = balle;
Entite entite = new Balle(..);
graphique = (Graphique) entite; // transtypage indispensable !
```

FIGURE 5

12:34

16:03

Variables de type d'interface.

## 21. INTERFACES EN JAVA 8

Dans la leçon 20 nous avons découvert la notion d'interface en Java, jusqu'à Java 7 compris. Toutefois, en Java 8, ce concept a été substantiellement augmenté. À l'origine, une interface ne pouvait contenir que des constantes et des méthodes abstraites, c'est-à-dire des méthodes sans définition. Depuis Java 8, il est également possible de déclarer des **méthodes avec définition par défaut** et des **méthodes statiques** dans une interface.

Les méthodes statiques se comportent exactement comme des méthodes statiques dans une classe. Dans cette leçon, il s'agira de se familiariser avec les méthodes par défaut et les éventuelles ambiguïtés pouvant résulter de leur utilisation.

### MÉTHODES PAR DÉFAUT

Depuis Java 8, une méthode d'interface peut être dotée d'un corps. Il s'agit alors d'une définition par défaut qu'il convient de déclarer au moyen du mot-clé `default`:

```
default void peutDescendre() { ... }
```

À quoi cela peut-il servir concrètement ?

Prenons comme exemple la hiérarchie de classe de la figure 1. Les deux classes `Voleur` et `Guerrier` implémentent l'interface `Chevauchant`. Imaginons cependant que la méthode `peutDescendre()` ait le même comportement pour les deux classes. Jusqu'à Java 7, nous aurions dû redéfinir deux fois la même méthode : une fois dans `Voleur` et une autre fois dans `Guerrier` ; ce qui représente une duplication de code inutile et dangereuse. Par contre, en Java 8, il suffit de déclarer la méthode `peutDescendre()` comme une méthode avec définition par défaut et de lui donner le corps directement dans l'interface. Cette procédure est beaucoup plus efficace et s'apparente aux méthodes héritées d'une super-classe.

L'utilisation de méthodes par défaut est très pratique, mais elle peut induire des ambiguïtés. Imaginons une classe implementant deux interfaces qui comportent toutes deux une méthode par défaut de même nom, avec des comportements différents. Laquelle des deux méthodes sera utilisée lors d'un appel depuis la classe ? Pour lever ces ambiguïtés, **quatre règles fondamentales** régissent l'utilisation de méthodes par défaut en Java.

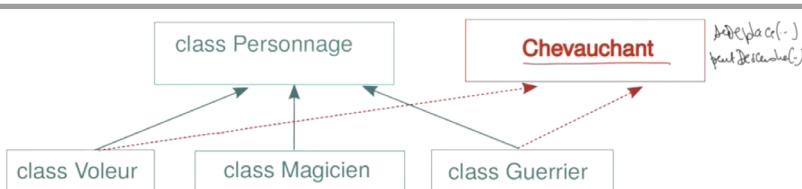


FIGURE 1

3:00

20:00

Hiérarchie de classe avec une interface (en rouge).

**RÈGLE 1*****Les définitions par défaut s'héritent.***

Les interfaces peuvent hériter d'interfaces. Nous pouvons par exemple définir une sous-interface pour l'interface `Chevauchant`, ainsi que nous le montre la figure 2. Comme pour les classes, cette sous-interface hérite des méthodes par défaut de la super-interface et elle n'est pas obligée de les redéfinir. Ces méthodes seront automatiquement incluses dans l'interface. La sous-interface a toutefois la possibilité de redéfinir une méthode pour changer le comportement d'une méthode par défaut héritée ou pour définir concrètement une méthode qui était abstraite dans la super-interface. On peut l'observer sur la figure 2 pour la méthode `seDeplace()`.

```
interface Cavalier extends Chevauchant
{
    default void seDeplace() { System.out.println("au trot"); }
}
```

☞ Inutile de redonner une définition par défaut à `peutDescendre` si on en est satisfait.

```
interface Chevauchant
{
    void seDeplace();
    default boolean peutDescendre() { return false; }
}
```

FIGURE 2

7:24

20:00

La sous-interface hérite la définition par défaut de `peutDescendre()`.

**RÈGLE 2*****Une classe implémentant une interface n'est plus obligée de redéfinir les méthodes par défaut de celle-ci.***

Habituellement, une classe doit redéfinir toutes les méthodes des interfaces qu'elle implémente pour être instanciable. C'est toujours vrai, mais seulement pour les méthodes abstraites. Les méthodes avec définition par défaut sont héritées par les classes, de la même façon qu'une méthode d'une super-classe. Si l'on applique ce principe à une classe implementant l'interface `Cavalier` de la figure 2, nous voyons qu'une telle classe n'a désormais plus besoin de redéfinir de méthode, les deux méthodes de l'interface possédant déjà une définition par défaut.

Cette règle présente un avantage fondamental. Elle permet l'ajout de méthodes à des interfaces qui sont déjà en usage. Avant Java 8, si l'on voulait ajouter une nouvelle méthode à une interface existante, il fallait ensuite retoucher toutes les classes qui implétaient l'interface. Sinon ces classes n'étaient plus instanciables parce qu'elles ne définissaient plus toutes les méthodes. Ceci rendait bien évidemment l'addition de méthodes très difficile une fois l'interface publiée. L'API de Java, qui offre un grand nombre d'interfaces prédéfinies, souffrait particulièrement de ce problème. Depuis Java 8, il est de nouveau possible de changer les interfaces existantes en leur ajoutant des méthodes avec définition par défaut.

**RÈGLE 3*****En cas d'ambiguïté, les méthodes de classes ont la préséance sur les méthodes par défaut des interfaces.***

Cette règle est appliquée dans le cas d'une classe qui, d'un côté, hérite d'une méthode depuis une super-classe, et de l'autre, implémente une interface contenant une méthode par défaut de même nom que celle héritée de la super-classe. Par défaut, la méthode de la super-classe sera appelée lors de l'invocation de la méthode sur la classe en question. Néanmoins il est également possible d'utiliser la méthode de l'interface. La figure 3 montre comment redéfinir la méthode dans la classe et appeler explicitement la méthode de l'interface.



L'appel à une méthode par défaut d'une interface doit se faire avec la syntaxe suivante : le nom de l'interface, point, le mot réservé `super`, point, le nom de la méthode. Le mot `super` aide à distinguer un tel appel d'un appel à une méthode statique qui se fait simplement avec le nom de la classe/interface, point, le nom de la méthode.

```
class Guerrier extends Personnage implements Cavalier
{
    public void seDeplace() {
        Cavalier.super.seDeplace();
    }
}
```

**FIGURE 3**

14:17

20:00

Utilisation de la méthode de l'interface.

#### RÈGLE 4

**Si une classe implémente deux interfaces proposant toutes deux une définition par défaut pour une même méthode, elle est en charge de lever l'ambiguïté.**

Cette règle est très proche de la règle précédente, sauf qu'il s'agit là d'un conflit entre deux méthodes par défaut provenant de deux interfaces différentes. Un tel conflit est présent dans la classe `MageUltime` de la figure 4 où nous voyons deux définitions par défaut pour la méthode `seDeplace()`. Pour être instanciable, la classe `MageUltime` doit soit spécifier laquelle des deux méthodes doit être utilisée, soit donner une nouvelle redéfinition de la méthode. On lève l'ambiguïté de la même manière que dans la règle trois. La classe doit redéfinir la méthode et faire un appel à l'une des deux méthodes par défaut en utilisant la syntaxe de la figure 3. Bien sûr, le choix de la méthode peut se faire dynamiquement, par exemple en fonction de l'état de la classe.

```
interface Dragonier extends Chevauchant
{
    default void seDeplace() { System.out.println("vole"); }
}

interface SeTeleporte
{
    default void seDeplace { System.out.println("plop !"); }
}

class MageUltime extends Magicien implements Dragonier, SeTeleporte {
    // conflit sur la définition de seDeplace !
}
```

**FIGURE 4**

17:11

20:00

Situation de conflit entre deux définitions par défaut d'une méthode.

#### INTERFACES VS CLASSES ABSTRAITES

Remarquons que l'introduction des méthodes abstraites rapproche beaucoup les interfaces des classes abstraites. La question se pose donc de quand utiliser une interface et quand une classe abstraite ? La principale différence entre les deux réside dans le fait que les interfaces ne peuvent pas avoir d'état – de variables d'instance. Elles ne possèdent que des constantes et elles n'ont pas de constructeur. Pour cette raison, nous utilisons des interfaces quand nous voulons modéliser un comportement ou un **lien fonctionnel** qui est indépendant d'un éventuel état de l'objet. Tandis que nous utilisons des classes abstraites quand les objets ont des états.

## 22. GESTION DES EXCEPTIONS: INTRODUCTION

Cette leçon aborde la gestion des situations d'erreurs dans un programme au moyen de ce que l'on appelle les « exceptions ».

Prenons pour exemple illustratif celui de la figure 1: nous voulons concevoir un programme qui affiche l'inverse de températures stockées dans un tableau. On presume que dans cet exemple, les températures nulles ne correspondent pas à une situation physique possible. Que se passe-t-il si une valeur nulle se trouve accidentellement dans le tableau (suite par exemple à une erreur d'appareil de mesure)? Le programme principal appelle la méthode qui affiche les inverses de chacune de ses entrées, laquelle appelle une méthode de division... qui fera une division par zéro!

Le problème auquel nous sommes confrontés est alors le suivant: la méthode qui réalise la division est ici celle qui découvre l'erreur. Elle n'a cependant aucun moyen de savoir dans quel contexte elle est appelée: est-elle appelée par un programme pour lequel la division est un problème sérieux nécessitant l'arrêt de l'exécution ou par un programme pour lequel cette situation est exceptionnelle mais réparable (redemander de faire les mesures des températures par exemple)?

Dans notre cas, c'est la méthode `main`, et non la méthode `division`, qui est capable de déterminer la gravité de l'erreur et de prendre les mesures qui s'imposent pour la réparer.

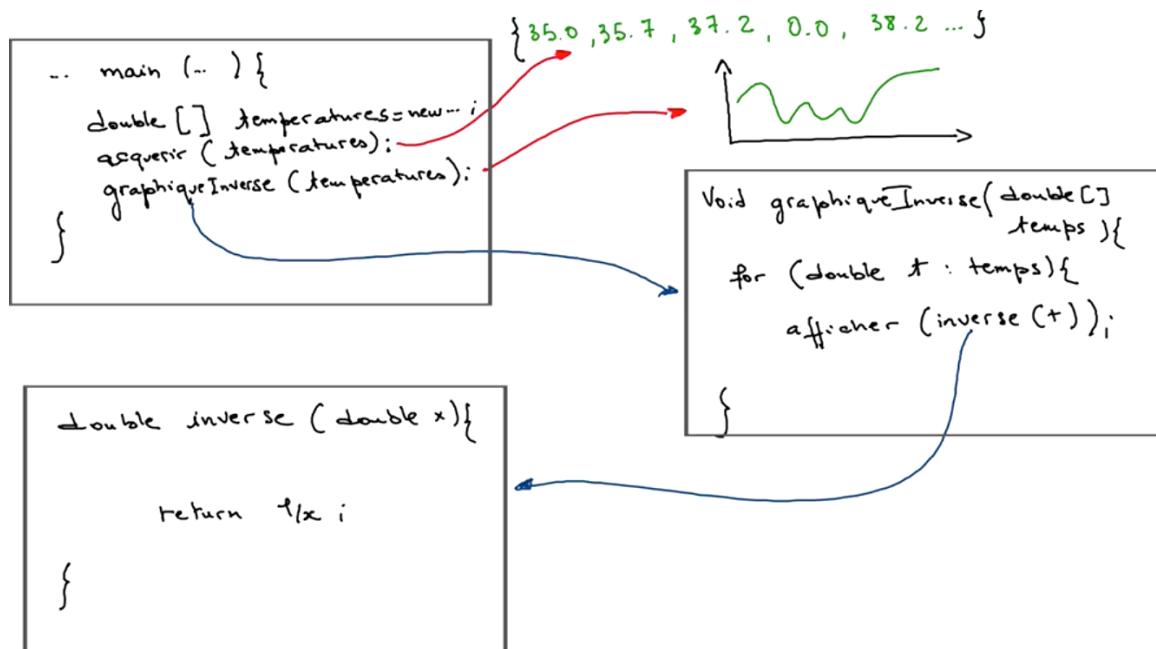


FIGURE 1

7:42

14:17

Programme avec trois méthodes.

Le système de « **gestion des exceptions** » permet de faire en sorte qu'une erreur détectée dans une partie d'un programme puisse être gérée et traitée dans une autre partie tout à fait différente de ce même programme.

## CONCEPT DE BASE

Une «exception» est un objet (du type prédéfini `Exception`) dédié à la gestion de situations d'erreur. Une méthode qui détecte une situation anormale sans savoir comment la gérer elle-même peut créer un tel objet et le «lancer», afin d'informer les autres niveaux du programme de la situation. Cet objet se propage automatiquement aux méthodes appelantes. Ces dernières peuvent alors, si elle le juge opportun, «attraper» l'exception et l'utiliser pour gérer la situation d'erreur de manière appropriée. Une exception lancée et jamais attrapée provoque simplement l'arrêt du programme.

La figure 2 schématisise la mise en œuvre de ce concept. L'erreur est détectée dans la fonction `division`, mais elle est attrapée et gérée dans la méthode `main`.

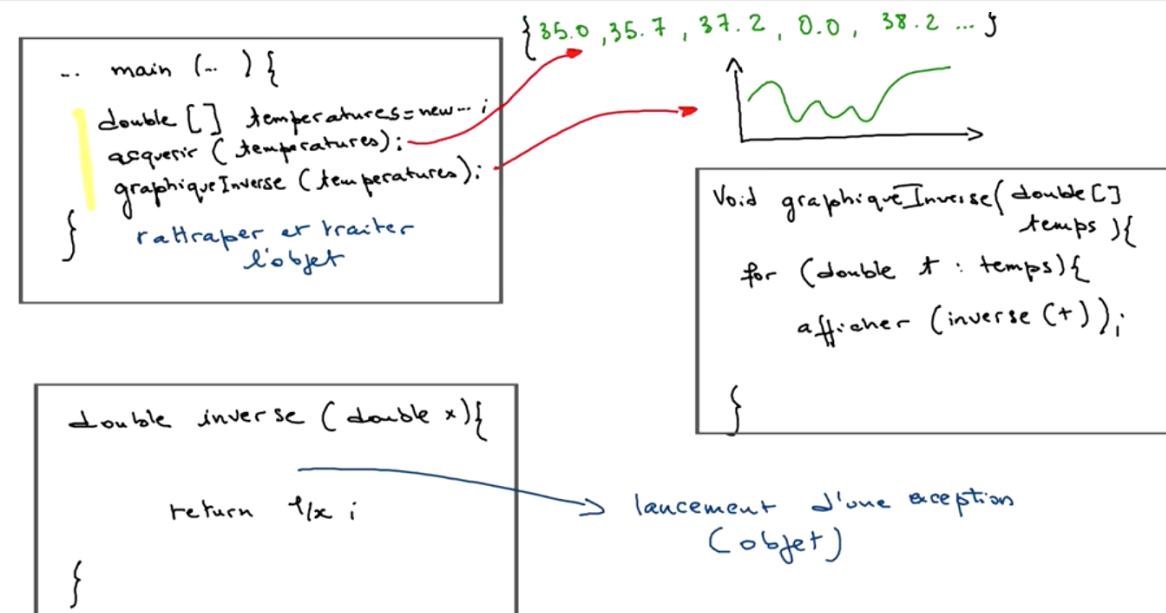


FIGURE 2

7:47

14:17

Lancement et traitement d'une exception.

## TÂCHES ET MOTS-CLÉS

La mise en œuvre de la gestion des exceptions se fait au moyen de quatre mécanismes qui sont chacun liés à un mot-clé de Java.

- Le **lancement** d'une exception dans une méthode découvrant une situation anormale sans pouvoir la traiter directement se fait au moyen du mot-clé `throw`.
- Une méthode qui juge opportun de traiter une exception lancée par un certain nombre de ses instructions, englobera ces dernières dans un bloc précédé du mot réservé `try` (elle «essaie» d'exécuter ces instructions sachant qu'elle peuvent potentiellement lancer un objet exception). Le bloc `try` signale donc un endroit **réceptif aux exceptions** lancées.
- Un bloc `try` est suivi de un ou plusieurs blocs désigné par le mot-clé `catch`. Ce sont vers ces blocs d'instructions que va se diriger l'exécution si une instruction du bloc `try` a lancé une exception. Les blocs `catch` **attrapent** donc **l'exception** pour la traiter.
- Les blocs `catch` sont parfois suivis d'un bloc associé au mot-clé `finally` qui se charge de «faire le ménage» (comme libérer certaines ressources) après qu'une exception ait été traitée.

Les deux prochaines leçons présentent l'utilisation concrète de ces mots-clés Java.

## AVANTAGES

Les exceptions permettent de traiter des situations anormales tout en préservant une écriture intuitive et lisible de programme. Elles ne modifient pas les résultats ou les comportements usuels des méthodes qui les utilisent. La robustesse et la simplicité des programmes sont améliorées grâce à la possibilité de séparer le code qui est exécuté normalement, du code qui ne l'est que pour le traitement d'erreurs.

## 23. GESTION DES EXCEPTIONS: SYNTAXE

La leçon 22 a présenté les quatre tâches fondamentales qui constituent le traitement des erreurs dans un programme Java: lancer une exception, la rattraper dans un endroit réceptif, la traiter et finalement «faire le ménage». Chacune de ces tâches est réalisée par un mot-clé Java: `throw`, `try`, `catch` et `finally`. Il s'agit maintenant d'étudier la syntaxe liée à ces tâches ainsi que leur fonctionnement.

### TYPES D'EXCEPTIONS

Le lancement d'une exception consiste à créer un objet de type `Exception`, et à déclencher un mécanisme qui rendra cet objet accessible à d'autres parties du programme. La classe `Exception` est fournie par l'API de Java et elle est une sous-classe de la classe `Throwable`. Cette dernière possède quelques méthodes très utiles. Par exemple, on peut donner une chaîne de caractères à son constructeur qui jouera le rôle de message d'erreur pour l'instance. La méthode `getMessage()` retourne ce message et la méthode `printStackTrace()` permet d'afficher la trace des appels ayant abouti au lancement de l'exception.

De la super-classe `Throwable` dérivent en fait deux sous-classes: `Error` et `Exception`. La première représente les erreurs fatales, comme les erreurs de mémoire. Ces erreurs sont trop graves pour être gérées par le programmeur. Elles nécessitent donc l'arrêt du programme. La deuxième catégorie représente les exceptions (situations anormales) que nous connaissons déjà. Il existe là aussi deux sous-catégories: les «*checked exceptions*» et les «*unchecked exceptions*». Très schématiquement:

- Les «*checked exceptions*» sont des exceptions que le programmeur est obligé de considérer: il doit les traiter ou les signaler selon des syntaxes montrées plus loin, faute de quoi le programme ne compilera pas.
- Les «*unchecked exceptions*» sont des exceptions que le programmeur n'est pas forcément obligé de traiter (typiquement parce qu'elles ne devraient pas se produire si le codage du programme avait été correct).

Nous reviendrons plus loin sur cette distinction (leçon 24).

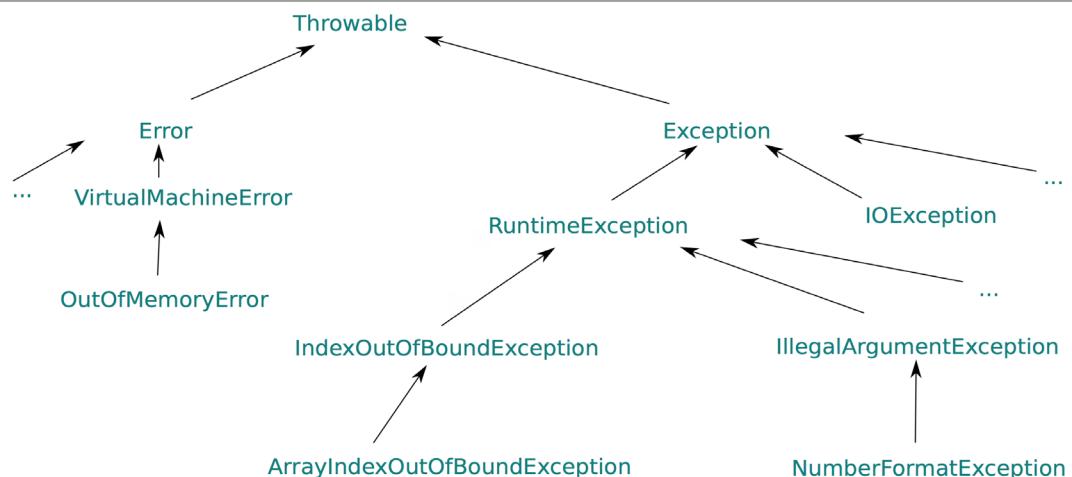


FIGURE 1

3:06

23:48

Hiérarchie de types d'exception et d'erreurs.



## LANCLEMENT D'ERREUR

La syntaxe suivante lance un objet de type Exception :

```
throw new Exception("message d'erreur");
```

Exception peut être remplacé par le nom de n'importe laquelle de ses sous-classes. Tous ces objets peuvent aussi être lancés sans le String du message d'erreur.

## ENDROITS RÉCEPTIFS

Pour que les exceptions lancées, en particulier les «checked exceptions», puissent être attrapées, il faut que le throw ait été exécuté depuis un **endroit réceptif aux erreurs** dans le programme. Un endroit réceptif est un bloc précédé du mot-clé try. Un exemple de cette syntaxe est donné dans la figure 2. Dans cet exemple, l'instruction throw se trouve directement dans le bloc try. Il est évident que le plus souvent le throw se fera plutôt dans une méthode appelée directement ou indirectement dans le bloc try.

```
try {  
    // ...  
    if (...) {  
        throw new Exception("Quelle erreur !");  
    }  
    // ...  
}  
catch (Exception e) {  
    // ...  
}
```

FIGURE 2

10:59

23:48

Syntaxe du mécanisme des exceptions en Java.

## TRAITEMENT DES EXCEPTIONS

Si l'exécution d'une instruction contenue dans un bloc try lance une exception, le programme ne passera pas à l'instruction suivante. L'exécution se poursuivra en revanche dans le bloc catch permettant de traiter le type d'exception lancée. Un bloc réceptif peut en effet avoir plusieurs catch associés. Tous utilisent la syntaxe suivante :

```
catch (E e) {...}
```

où E est le type Exception ou l'un d'une de ses quelconques sous-classes. Le bloc contient les instructions qui sont dédiées au traitement de l'exception de type E.

Il est ainsi possible d'exécuter un traitement spécifique à un type d'exception donné. Ce point est illustré dans la figure 3. Si l'exception ArithmeticException est lancée, c'est son bloc catch spécifique qui s'exécute, l'autre bloc sera ignoré. Pour tout autre type d'exception le deuxième bloc de traitement sera utilisé et le premier ignoré. Il n'y aura à chaque fois qu'un seul bloc catch exécuté. Après l'exécution d'un tel bloc, le programme continue son déroulement à l'endroit suivant le dernier bloc catch.

Si l'exécution d'un bloc try ne suscite aucun lancement d'exception, les blocs catch sont ignorés et l'exécution se poursuit aussi depuis l'endroit suivant le dernier bloc catch.

Finalement, il est très important de toujours ordonner les blocs de traitement par niveau d'exception de plus en plus général. Ainsi le type Exception lui-même devrait toujours être le dernier dans l'ordre, car le programme choisit toujours le premier bloc catch possible. Si l'on mettait Exception comme premier bloc de traitement, le programme ne compilerait pas.



```

try {
    // ...
    if (age >= 150)
        { throw new Exception("age trop grand"); }
    // ...
    if (x == 0.0)
        { throw new ArithmeticException("Division par zero"); }
    // ...
}

catch (ArithmeticException e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
}

catch (Exception e) {
    System.out.println("Qui peut vivre si vieux?");
}

```

**FIGURE 3**

8:00

23:48

Exemple d'utilisation d'un bloc réceptif aux exceptions et leur traitement.

### « FAIRE LE MÉNAGE »

La gestion de fichiers ne fait pas partie de ce cours, mais elle nous donne un contexte d'utilisation usuel du mot-clé `finally` que nous utiliserons ici en guise d'exemple. Un programme qui souhaite utiliser un fichier (par exemple pour y lire des données) doit l'ouvrir. Il doit le fermer lorsqu'il a fini de l'utiliser. Supposons qu'une exception soit lancée entre ces deux étapes. Les instructions visant à fermer le fichier ne seront pas atteintes, ce qui peut poser problème. La vocation du bloc `finally` serait de permettre la fermeture du fichier dans tous les cas, qu'une exception ait été lancée ou pas.

Comme les mots-clés précédents, `finally` désigne un bloc d'instruction spécifique (fig. 4). La particularité de ce bloc est que ses instructions seront toujours exécutées quel que soit le déroulement du programme ; qu'il y ait eu lancement d'exception ou pas. Si une exception est traitée, le bloc `finally` sera exécuté après le bloc `catch`. Il le sera aussi s'il n'y a pas eu lancement d'exception, et ce, même si le bloc `try` contient un `return`. Le bloc `finally` est aussi exécuté en cas de lancement d'exception non interceptée par un bloc `catch`.

```

class Inverse {
    public static void main (String[] args) {
        try {
            int b = Integer.parseInt(args[0]);
            int c = 100/b;
            System.out.println("Inverse * 100 = " + c);
        }
        catch (NumberFormatException e1) {
            System.out.println("Il faut un nombre entier !");
        }
        catch (ArithmeticException e2) {
            System.out.println ("Parti vers l'infini !");
        }
        finally {
            System.out.println("Passage obligé !");
        }
    }
}

```

**FIGURE 4**

18:30

23:48

Exemple avec les quatre mots-clés.



## 24. GESTION DES EXCEPTIONS: COMPLÉMENTS

Cette leçon traite de quelques aspects plus avancés liés à la gestion des exceptions.

### RELANCEMENT

Dans notre exemple introductif, la méthode `division` lançait l'exception et la méthode `main` la rattrapait pour la traiter. La méthode intermédiaire, qui affiche l'inverse des éléments d'un tableau et est appelée par `main`, ne jouait aucun rôle.

L'objet lancé par `division` ne peut en fait donner aucune information sur l'entrée du tableau concernée par l'erreur ; ce qui est dommage car cela pourrait permettre à `main` de gérer plus finement l'erreur. Il serait donc utile que la méthode intermédiaire, entre `main` et `division`, intervienne dans le processus pour produire une exception plus informée, connaissant l'indice du tableau correspondant à une valeur erronée.

Le **relancement d'exception** résout exactement ce problème. L'idée est qu'une partie intermédiaire du programme attrape l'exception par le biais d'un `catch`, lui ajoute des informations utiles et la relance ensuite avec un nouveau `throw`. Cette manière de gérer les exceptions est aussi appelée le **traitement partiel** (la méthode intermédiaire traite partiellement l'erreur avant de la renvoyer aux méthodes appelantes). Dans l'exemple de la figure 1, l'indice du tableau qui a posé un problème est affiché. Il serait également possible d'intégrer cet indice au message de l'exception relancée dans la méthode intermédiaire, par exemple.

```
for(int i = 0; i < t.size(); i++) {  
    try {  
        plot(inverse(t.get(i)));  
    } catch (ArithmetricException e) {  
        System.out.println("Problème à l'indice : " + i);  
        // RELANCEMENT  
        throw e;  
    }  
}
```

FIGURE 1

14:11

16:31

Exemple de relancement d'exception.

### TRAITER OU DÉCLARER

Comme vu précédemment, les exceptions sont soit de type « *unchecked exception* », soit de type « *checked exception* ». Dans la hiérarchie actuelle de l'API de Java seules les `RunTimeException` et les `Error` sont de type « *unchecked* ».

Supposons qu'une méthode `m_1` appelle une méthode `m_2` lançant potentiellement une exception de type `T`. Si l'exception lancée par `m_2` est une « *checked exception* » la méthode `m_1` a l'obligation de :

- la traiter ; c'est-à-dire d'englober l'appel à `m_2` dans un bloc `try` auquel est assorti un bloc `catch` traitant l'exception ;
- ou de la déclarer ; c'est-à-dire de signifier dans son entête qu'elle peut elle-même lancer une exception de type `T`.

Si elle ne fait ni l'un ni l'autre le programme ne compilera pas.



Une exception est déclarée dans l'entête de la méthode en employant le mot-clé `throws` suivi de la liste des (« checked ») exceptions , comme par exemple:

```
public void exampleMethod() throws IOException
public void autreMethod() throws IOException, Exception, ...
```

Il est aussi parfaitement possible de déclarer des « unchecked exception » selon la même syntaxe, en guise d'information.

Les `RuntimeException` sont des erreurs à l'exécution généralement difficiles à corriger, comme l'accès à un tableau avec un indice trop grand. Ce sont typiquement des erreurs que l'on imagine évitable par une programmation correcte. Il serait donc trop lourd d'en imposer le traitement systématique à l'utilisateur; qui d'ailleurs serait souvent bien emprunté pour trouver le bon moyen de les gérer.

À l'inverse, les « checked exceptions » vont typiquement modéliser des situations anormales qui ne sont pas issues d'une programmation maladroite mais plutôt de conditions d'utilisation atypiques du programme (un utilisateur qui s'acharne à entrer trop de fois des valeurs incorrectes, un nom de fichier inexistant etc.)

## EXCEPTIONS SPÉCIALISÉES

Pour la plupart des situations d'erreur susceptibles de se produire dans un programme, il existe, dans l'API Java, un type d'exception correspondant. Il est fortement recommandé de toujours utiliser l'exception la plus spécifique afin que le traitement ait accès à un maximum d'informations sur l'erreur. Toutefois, il est possible que l'on ait besoin d'un type d'exception qui n'existe pas encore. Dans ce cas-là, on peut définir ses propres sous-classes d'`Exception` ou d'une de ses sous-classes existantes. Cela se fait par simple héritage comme sur la figure 2.

```
class MonException extends Exception
{
    public MonException() {
        super("mon message par défaut");
    }
    public MonException(String message) {
        super(message);
    }
}
```

FIGURE 2

8:45

16:31

Exemple d'une nouvelle exception.

Comme il a été dit dans la leçon 23, la classe `Exception` comporte une méthode `getMessage()` qui retourne le message que l'on a éventuellement donné au constructeur lors de sa création. Pour préserver le fonctionnement usuel de cette méthode lors de la définition de classes personnalisées d'exceptions, il convient de munir ces classes de deux constructeurs:

- un constructeur par défaut, initialisant le message avec une valeur prédéfinie;
- et un second constructeur prenant en paramètre une chaîne de caractères qu'il utilisera lors de l'appel au constructeur de la super-classe (fig. 2).



Le programmeur est tout à fait libre d'ajouter autant de méthodes et d'attributs à sa classe d'exceptions qu'il le souhaite. Un second exemple d'exception personnalisée est présenté en figure 3.

```
class TropChaudException extends Exception
{
    private double temperatureAnormale;
    private String consigne;

    public TropChaudException(double uneTemperature, String uneConsigne) {
        super("Température trop élevée");
        temperatureAnormale = uneTemperature;
        consigne = uneConsigne;
    }
    public double getTemperature() {
        return temperatureAnormale;
    }
    public String getConsigne() {
        return consigne;
    }
}
```

FIGURE 3

9:06

16:31

Exception spécialisée pour la manipulation de températures.

Finissons cette leçon par une recommandation. Le mécanisme de gestion des exceptions est puissant et très utile mais a un coût. Son utilisation doit donc se limiter aux situations où il s'impose. Typiquement, si une méthode est suffisamment informée pour pouvoir réparer elle-même une erreur, elle le fera sans avoir recours aux exceptions.



## 25. ÉTUDE DE CAS : PRÉSENTATION ET MODÉLISATION DU PROBLÈME

Dans les cinq dernières leçons de ce cours nous abordons un problème de conception en programmation orientée objet qui nous permettra d'illustrer de façon relativement exhaustive tous les concepts abordés : encapsulation et classes, héritage et polymorphisme, interfaces et enfin copie profonde. Dans cette première leçon, il s'agira de traduire la donnée du problème dans un ensemble de classes en Java.

### PROBLÈME

La description du problème est donnée dans la figure 1.

- ▶ Les *montres* sont des *produits* (que l'on peut vendre : ont un prix)
- ▶ Les montres ont un *mécanisme* de base et sont constituées de différents *accessoires* (*boîtier*, *bracelet*, ...)
- ▶ Les *produits* ont un prix dont le calcul peut varier, à partir d'une valeur de base
- ▶ Tous les produits sont « affichables », chacun à sa façon
- ▶ Les *mécanismes* et *accessoires* de montre sont aussi des produits (on pourrait en acheter séparément)
- ▶ Il existe trois sortes de *mécanismes* : *analogiques*, *digitaux* et *doubles*.
- ▶ Pour les *mécanismes doubles*, on supposera ici qu'ils n'indiquent qu'une seule heure, mais se comportent sinon à la fois comme des *mécanismes analogiques* et comme des *mécanismes digitaux*

FIGURE 1

5:09

10:02

Donnée du problème avec en surligné les points qui vont guider la conception.

Cette description en français suggère une conception : les entités surlignées suggèrent les classes qui vont intervenir, le verbe **avoir** suggère une relation d'encapsulation, **être** une relation d'héritage, et enfin « **chacun à sa façon** » fait penser à l'emploi du polymorphisme.

### STRUCTURE DE CLASSES

La figure 2 donne un schéma de conception possible découlant de cette description. On notera en particulier la classe générale Produit permettant d'encapsuler les éléments communs, typiquement le prix, à tout type de produit, que ce soit une montre, accessoire ou un mécanisme.

Un point relativement compliqué est la modélisation des mécanismes double en l'absence d'héritage multiple. Il serait assez naturel en effet de faire hériter les mécanismes doubles des mécanismes analogiques et des mécanismes digitaux. En l'absence d'héritage multiple, nous nous abstiendrons. Le propre du mécanisme double reste toutefois d'afficher une heure, et il n'y a de ce fait pas de contre-indication à le voir pour le moment comme une *Mecanisme* tout court. Ce que nous faisons comme choix pour commencer. Pour une modélisation adéquate du comportement nous aurons recours aux interfaces (leçon 20).

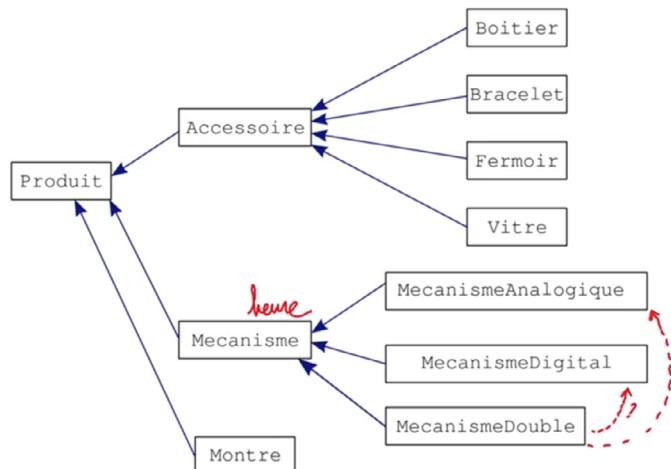


FIGURE 2

6:06

10:02

Schéma de classes.

## ÉBAUCHE DE CODE

Le schéma de classe de la figure 2 nous permet déjà d'ébaucher tous les entêtes de classes. Par exemple celui de : la classe `Accessoire`:

```
public class Accessoire extends Produit { ... }
```

Les flèches du schéma (fig. 2) représentent des liens d'héritage qui s'écrivent comme dans l'exemple précédent.

Une fois cette ébauche faite, on peut commencer à s'intéresser plus finement au contenu des classes. L'énoncé précise qu'une montre a un mécanisme et des accessoires. Nous définissons donc deux attributs dans la classe `Montre` : un attribut de type `Mecanisme` et un tableau dynamique pour représenter plusieurs accessoires (fig. 3).

```
class Montre extends Produit {  
    private Mecanisme coeur;  
    private ArrayList<Accessoire> accessoires;  
}
```

FIGURE 3

8:50

10:02

Les attributs de la classe représentant les montres.



Pour la classe `Produit`, la donnée précise que tout produit a un prix, mais que le calcul de ce prix peut varier et doit se faire à partir d'un prix de base. Ceci suggère que chaque catégorie de produit peut avoir une méthode spécifique calculant le prix à partir d'un prix de base. On s'oriente donc vers la définition d'un attribut pour le prix de base (`valeur`) et d'une méthode polymorphique `prix()`, retournant par défaut la valeur de l'attribut `valeur` dans la classe `Produit`.

Par ailleurs, chaque produit doit pouvoir s'afficher de manière spécifique. Le polymorphisme intervient à nouveau et il est naturel de redéfinir la méthode `toString()` de `Object`.

```
class Produit {
    private double valeur;

    public double prix()
    { return valeur; }

    public String toString() {
        //..
    }
}
```

FIGURE 4

8:25

10:02

Méthodes et attributs de la super-classe de la hiérarchie.



## 26. ÉTUDE DE CAS: AFFICHAGE POLYMORPHIQUE

### MÉTHODE D'AFFICHAGE

Comme évoqué précédemment, la méthode `toString()` sera redéfinie dans `Produit` et dans toutes les sous-classes pour assurer un affichage polymorphique. Dans `Produit` elle doit être capable de créer une représentation, sous forme de chaîne de caractères, des informations communes à tous les produits. En l'occurrence, il s'agit du prix. Cela étant, incorporer directement la valeur du prix de base dans la chaîne de caractère n'est pas une bonne idée. Il s'agit là d'une source d'erreurs fréquente : puisqu'**une méthode polymorphe s'adapte à la nature de l'instance traitée, les valeurs qu'elle utilise doivent le faire également**. Il est donc judicieux d'utiliser plutôt la valeur retournée par la méthode polymorphe `prix()`.

```
@Override  
public String toString() {  
    return Double.toString(prix());  
}
```

FIGURE 1

3:45

7:01

Méthode d'affichage polymorphe.

### FINALISATION DE `Produit`

Pour affiner la modélisation, nous nous posons la question de savoir si la classe `Produit` sera instanciable en tant que telle. La réponse est naturellement non car le type `Produit` n'est qu'une abstraction permettant d'encapsuler les caractéristiques communes à tous les produits concrets existants. Comme vu dans la leçon 15, il est naturel de déclarer `Produit` comme `abstract`, c'est-à-dire non instanciable.

Nous ajoutons deux constructeurs à notre classe. L'un prenant en argument la valeur du prix de base et l'affectant à l'attribut correspondant, et un constructeur par défaut `défaut` initialisant cet attribut à 0. Cette manière de faire – la surcharge de constructeurs ou de méthodes – est la façon typique d'initialiser des valeurs par défaut en Java. Ces deux constructeurs que nous venons de créer ne seront jamais appelés directement puisque la classe a été déclarée non instanciable – abstraite. Néanmoins, les sous-classes concrètes en ont besoin pour définir leurs propres constructeurs.

Si l'on souhaite assurer que la valeur par défaut du prix du produit ne puisse plus changer une fois déclarée, il suffit d'ajouter le mot-clé `final` à la déclaration de l'attribut. Ce dernier ne pourra plus changer de valeur une fois initialisé dans un constructeur.

```
abstract class Produit {  
  
    private final double valeur;  
  
    public Produit(double uneValeur) { valeur = uneValeur; }  
  
    public Produit() { valeur = 0.0; }  
  
    public double prix() { return valeur; }  
  
    public String toString() { return Double.toString(prix()); }  
}
```

FIGURE 2

5:12

7:01

La classe `Produit` finalisée.



## 27. ÉTUDE DE CAS: PREMIÈRE VERSION

Cette leçon présente une première version partielle mais fonctionnelle de notre programme.

### ACCESSOIRES D'UNE MONTRE

Dans la leçon 25, nous avons muni la classe `Montre` de deux attributs: un mécanisme et un tableau dynamique d'accessoires. Nous souhaitons maintenant pouvoir ajouter des objets de type `Accessoire` à ce tableau dynamique. Une méthode `ajouter()`, prenant en argument un `Accessoire`, remplira cette mission. Pour le moment, elle se contente d'ajouter l'accessoire tel quel à la liste. Cette façon de faire n'est pas tout à fait correcte (faille d'encapsulation), mais elle sera corrigée dans la leçon 29.

```
class Montre extends Produit {

    private Mecanisme coeur;
    private ArrayList<Accessoire> accessoires;

    public void ajouter(Accessoire accessoire) {
        accessoires.add(accessoire); // nous reviendrons sur ce point
    }
}
```

FIGURE 1

1:18

10:02

Méthode pour ajouter des accessoires.

### TYPES D'ACCESSIONS

Maintenant qu'une montre peut avoir des accessoires, nous avons besoin de quelques accessoires concrets. Par analogie avec un `Produit`, un `Accessoire` est une abstraction. Nous déclarons donc la classe comme étant abstraite et lui attribuons un constructeur qui sera utilisé lors de la création des objets des sous-classes. De plus, afin de donner un nom à chaque accessoire, la classe possède un attribut de type `String`, qui est une constante (`final`). Le constructeur doit alors initialiser le prix de la super-classe `Produit` en appelant son constructeur avec la tournure `super()` et le nom de l'accessoire (fig. 2).

```
abstract class Accessoire extends Produit {
    private final String nom;

    public Accessoire(String unNom,
                      double valeurDeBase) {
        super(valeurDeBase);
        nom = unNom;
    }

    @Override
    public String toString() {
        String result = nom + " coûtant ";
        result += super.toString();
        return result;
    }
}
```

FIGURE 2

4:31

10:02

La classe abstraite représente les accessoires en général.



Comme le prix d'un accessoire est simplement la valeur de base, il suffit d'utiliser la méthode `prix()` qui est héritée de la super-classe. Il est inutile de redéfinir cette méthode. Par contre, nous voulons un affichage spécifique. Il faut donc changer le comportement de la méthode `toString()` en la redéfinissant, ce qui se spécifie correctement avec l'annotation `@Override` (fig. 2). Pour afficher le prix de l'objet, on peut utiliser la méthode `toString()` de la super-classe car cette dernière affiche le prix d'un produit. Il faut cependant faire attention à ne pas oublier le mot-clé `super` au risque de déclencher une récursion infinie ; la méthode `toString()` dans `Accessoire` s'appellerait elle-même.

La classe `Accessoire` étant abstraite, il faut définir des sous-classes qui seront instanciées dans notre programme. Par exemple les classes `Bracelet` et `Fermoir` étendent toutes deux la super-classe `Accessoire`. Pour que ces deux classes comportent leur type dans leur nom, nous définissons des constructeurs ajoutant respectivement "bracelet" ou "fermoir" aux chaînes de caractères reçues en argument (fig. 3).

```
class Bracelet extends Accessoire {

    public Bracelet(String unNom, double valeurDeBase) {
        super("bracelet " + unNom, valeurDeBase);
    }
}

//-----

class Fermoir extends Accessoire {
    public Fermoir(String unNom, double valeurDeBase) {
        super("fermoir " + unNom, valeurDeBase);
    }
}

//...
```

FIGURE 3

4:54

10:02

Les sous-classes des accessoires.

## CLASSE Montre

Pour obtenir un programme fonctionnel, il nous manque encore les montres. La méthode qui ajoute les accessoires à une montre a déjà été définie. Le constructeur de `Montre` peut donc se contenter de créer le tableau dynamique vide des accessoires. Ce sera son constructeur par défaut.

Ensuite nous redéfinissons la méthode calculant le prix d'une montre. Le prix d'une montre est la valeur de base plus le prix de tous ses accessoires. On parcourt alors la liste des attributs et l'on ajoute tous les prix au prix de base qui est récupéré au moyen de la méthode `super.prix()` de la super-classe.

```
// ...
@Override
public double prix() {
    // Au départ, le prix est le prix de base
    double prixFinal = super.prix();

    for (Accessoire acc : accessoires) {
        prixFinal += acc.prix();
    }
    return prixFinal;
}
//...
```

FIGURE 4

7:11

10:02

La méthode qui calcule le prix d'une montre.



L'algorithme permettant l'affichage d'une montre ressemble beaucoup à celui du calcul de son prix (fig. 5).

```
//...
public void afficher () {
    System.out.print ("Une montre ");
    System.out.println("composée de :");

    for (Accessoire acc : accessoires) {
        System.out.println(" * " + acc);
    }
    System.out.print ("==> Prix total : ");
    System.out.println(prix());
}
//...
```

FIGURE 5

7:54

10:02

L'affichage d'une montre utilisant plusieurs méthodes polymorphiques.

La figure 6 propose de tester le programme avec la méthode `main()`. Ce programme crée une montre, lui donne des accessoires et l'affiche.

```
public static void main(String[] args)
{
    Montre m = new Montre();

    m.ajouter(new Bracelet("cuir" , 54.0));
    m.ajouter(new Fermoir ("acier" , 12.5));
    m.ajouter(new Boitier ("acier" , 36.6));
    m.ajouter(new Vitre ("quartz", 44.8));

    System.out.println('\n' +
                      "Montre m :");

    m.afficher();
}
```

```
Montre m :
Une montre composée de :
* bracelet cuir coutant 54
* fermoir acier coutant 12.5
* boitier acier coutant 36.6
* vitre quartz coutant 44.8
==> Prix total : 147.9
```

FIGURE 6

8:54

10:02

Programme principal pour tester.

Le code complet (à ce stade) peut être téléchargé [ici](#).

## 28. ÉTUDES DE CAS: MODÉLISATION ET MÉCANISMES

Il reste à implémenter les différents types de mécanismes. A ce stade nous disposons d'une super-classe `Mecanisme` qui a une heure en guise d'attribut. Ce type est utilisé pour définir l'attribut `cœur` de la classe `Montre`.

Nous voulons affiner la représentation des mécanismes en définissant des mécanismes analogiques, digitaux et doubles, ces derniers afficheront l'heure à la fois comme un mécanisme digital et comme un mécanisme analogique ; ce qui est potentiellement problématique vu l'absence d'héritage multiple en Java.

### HÉRITAGE ET INTERFACE

Pour ne pas dupliquer tout le code des deux types de mécanisme dans la classe `MecanismeDouble`, nous en choisissons un comme super-classe. Dans cet exemple c'est le mécanisme analogique. Il manque alors le lien entre le mécanisme double et le mécanisme digital. Là également, on pourrait dupliquer le code spécifiquement digital et le mettre dans le mécanisme double, mais il existe un meilleur moyen : les **interfaces**. Comme il s'agit ici d'un lien fonctionnel – nous souhaitons que la classe ait les mêmes fonctions que le mécanisme digital – nous pouvons déclarer une interface qui impose l'implémentation de ces méthodes aux deux classes. Non seulement cela explicite le lien entre les deux classes et met en évidence la conception des classes, mais cela permet en plus d'éviter partiellement la duplication de code en utilisant des méthodes par défaut dans l'interface. Cela nous amène à une nouvelle hiérarchie de classe visible sur la figure 1.

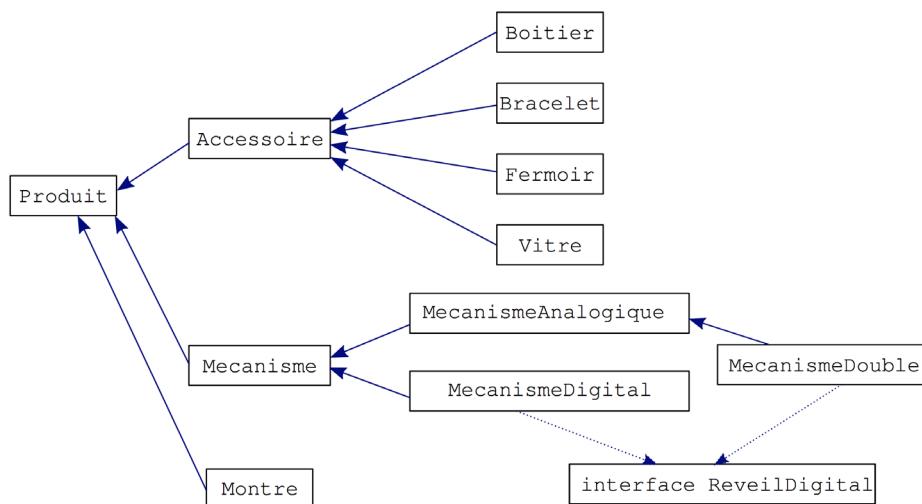


FIGURE 1

14:19

17:48

Hiérarchie de classe avec utilisation d'une interface.



## CONSTRUCTION DES MÉCANISMES

Le constructeur de la classe `Mecanisme` doit évidemment initialiser l'attribut `heure` qui lui est spécifique. Comme un mécanisme est aussi un produit, il doit également initialiser l'attribut `prix` hérité de `Produit`. La classe aura alors un constructeur prenant en argument des valeurs pour ces deux données. Le prix est passé à la super-classe avec un appel à `super()`. Finalement, on aimerait construire des mécanismes avec une valeur par défaut pour l'heure. C'est rendu possible grâce à la **surcharge du constructeur** – comme toujours en Java pour les valeurs par défaut.

```
class Mecanisme extends Produit {
    private String heure;

    public Mecanisme(double valeurDeBase, String uneHeure) {
        super(valeurDeBase);
        heure = uneHeure;
    }
    public Mecanisme(double valeurDeBase){
        super(valeurDeBase);
        heure = "12:00";
    }
}
```

FIGURE 2

3:56

17:48

Constructeurs du mécanisme.

On procède de façon analogue pour définir les constructeurs de la classe `MecanismeDouble`, en prenant soin de gérer proprement les valeurs par défaut (fig. 3).

```
class MecanismeDouble extends MecanismeAnalogique {

    public MecanismeDouble(double valeurDeBase, String uneHeure, int uneDate,
                           String heureReveil) {
        super(valeurDeBase, uneHeure, uneDate);
        reveil = heureReveil;
    }

    // gestion propre de la valeur par défaut de l'heure (super-classe)
    public MecanismeDouble(double valeurDeBase, int uneDate,
                           String heureReveil) {
        super(valeurDeBase, uneDate);
        reveil = heureReveil;
    }
}
```

FIGURE 3

6:48

17:48

Constructeurs de la classe `MecanismeDouble`.

## AFFICHAGE DES MÉCANISMES

Nous souhaitons un schéma d'affichage identique et fixe pour tous les mécanismes. Ce schéma doit cependant intégrer proprement des éléments polymorphiques, comme l'affichage du type du mécanisme ou celui du prix. Ceci peut se traduire par la création d'une méthode «`final`» dans `Mecanisme`, qui fixera le schéma d'affichage une fois pour toutes dans la super-classe et empêchera sa redéfinition dans les sous-classes. Cette méthode appellera cependant d'autres fonctions qui, quant à elles, s'adapteront au type de leur objet et seront *polymorphiques* (fig. 4). L'affichage du type de mécanisme est typiquement une méthode polymorphe abstraite.



```
abstract class Mecanisme extends Produit {
    //...
    // Tous les mécanismes DOIVENT s'afficher comme ceci
    public final String toString() {
        String result = "mécanisme ";
        result += toStringType();
        result += " (affichage : ";
        result += toStringCadran();
        result += "), prix : ";
        result += super.toString();
        return result;
    }
    // on veut offrir la version par défaut aux sous-classes et aux classes
    // du même paquetage. Par défaut, on affiche juste l'heure.
    protected String toStringCadran() {
        return heure;
    }
    // Un mécanisme, ici à ce niveau, est abstrait (= classe abstraite)
    protected abstract String toStringType();
}
```

FIGURE 4

10:31

17:48

Méthodes responsables de l'affichage du mécanisme.

Notons que la classe `Mecanisme` est maintenant devenue abstraite puisqu'elle contient une méthode abstraite.

C'est maintenant aux sous-classes d'implémenter leur version des méthodes d'affichage spécifiques. Par exemple, la classe `MecanismeDigital` va certainement redéfinir la méthode d'affichage pour le cadran puisqu'elle a plus de fonctionnalité à offrir que de seulement donner l'heure. Ainsi, sa redéfinition utilise la méthode de la super-classe `super.toStringCadran()`, mais elle utilise aussi sa propre méthode affichant le contenu du réveil. Elle est en plus obligée de redéfinir la méthode abstraite `toStringType()` pour rester instanciable.

```
class MecanismeDigital extends Mecanisme {
    //...
    @Override
    protected String toStringType() {
        return "digital";
    }

    @Override
    protected String toStringCadran() {
        // On affiche l'heure (façon de base)...
        // ...et en plus l'heure de réveil.
        return super.toStringCadran() + ", " + toStringReveil();
    }

    protected String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

FIGURE 5

13:52

17:48

Méthodes d'affichage du mécanisme digital.

Pour la classe `MecanismeDouble`, on peut écrire presque les mêmes méthodes que pour le mécanisme digital. Il faut cependant se souvenir que l'idée était de lier ces deux classes par le biais d'une interface. Il faut donc ajouter le mot-clé « `implements` » suivi du nom de l'interface à la déclaration des deux classes. Les définitions par défaut ne s'appliquent pas dans le cas de cet exemple. Le seul désavantage de cette manière de procéder est l'obligation d'ajuster les droits d'accès à `public` puisque les interfaces n'admettent que les méthodes publiques.

Comme toujours, il est conseillé de tester le code produit au moyen d'un programme principal qui crée des objets et qui les affiche.

Le code complet de cette partie est téléchargeable [ici](#).

## 29. ÉTUDE DE CAS: COPIE PROFONDE

L'étude de cas se conclut avec un sujet important en programmation: la copie d'objets. Lorsque l'on a un objet et que l'on aimerait le dupliquer à l'identique, il ne suffit pas d'affecter la valeur de cet objet à une deuxième variable. Comme ces variables ne contiennent que les références vers les objets, elles référenceront le même objet en mémoire; un changement sur l'un des deux objets aura le même impact sur les deux variables.

### COPIE PROFONDE

Pour réellement copier un objet, il faut en créer un nouveau et donner à ses attributs des valeurs identiques à celles de l'objet de départ. Cette tâche est habituellement accomplie par un **constructeur de copie**. Mais si ce dernier ne fait rien d'autre que de copier «champ par champ» tous les attributs et que ces attributs sont des types évolués, on retrouve exactement le même problème: les affectations ne suffisent pas. Une telle copie, qui copie les références sans dupliquer les objets référencés, est appelée **copie de surface** (fig. 1). Les attributs (de type évolué) des deux objets réfèrent aux mêmes objets.

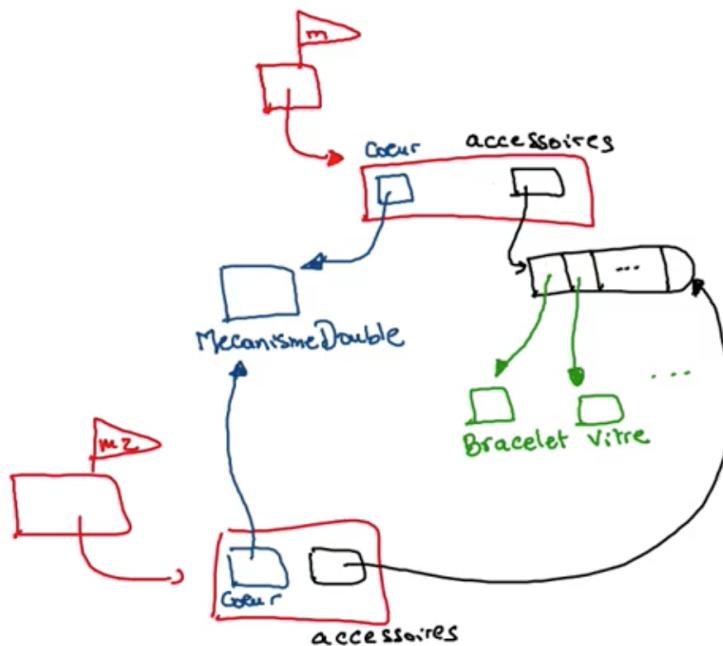


FIGURE 1

3:02

16:37

Représentation d'une copie de surface.

Pour avoir deux objets identiques mais complètement distincts, il faut copier tous les attributs à leur tour, jusqu'au niveau le plus profond. C'est ce que l'on appelle la **copie profonde**. Mais cela presuppose que tous les objets impliqués offrent justement un moyen de les copier.

## COPIE POLYMORPHIQUE

Un deuxième problème apparaît lorsque l'on essaie d'écrire le constructeur de la classe `Montre`. Ce dernier doit copier deux éléments: un objet de type `Mecanisme`, et une `ArrayList<Accessoire>` (et tous les `Accessoire` qu'elle contient!). Rappelons que les classes `Mecanisme` et `Accessoire` sont des classes abstraites. Ce qui veut dire que les objets auront forcément des types de sous-classes. Or, les constructeurs ne sont pas polymorphiques en Java. Si le mécanisme de la montre à copier est en réalité un `MecanismeDigital`, et qu'on le copie au moyen d'un constructeur de copie, il perdra toutes les spécificités digitales. Pour remédier à ce problème, il suffit de déclarer une nouvelle méthode – polymorphe – qui se charge d'effectuer les copies. Cette méthode peut par contre parfaitement utiliser le constructeur de copie de sa propre classe. Ainsi, le constructeur de copie de la classe `Montre` peut s'écrire comme en figure 2.

```
public Montre(Montre autre) {  
    super(autre);  
    coeur = autre.coeur.copie();  
    accessoires = new ArrayList<Accessoire>();  
    for (Accessoire acc : autre.accessoires) {  
        accessoires.add(acc.copie());  
    }  
}
```

FIGURE 2

6:56

16:37

Constructeur de copie profonde.

La classe `Accessoire` étant abstraite, elle ne peut pas être copiée non plus. La méthode `copie()` est déclarée comme une méthode abstraite. Par contre dans une sous-classe comme `Bracelet`, la méthode est définie et fait appel au constructeur de copie de sa classe. Notons que les types de retour ne sont pas les mêmes pour les deux méthodes. Mais comme `Bracelet` est une sous-classe d'`Accessoire`, ce sont des types compatibles et il s'agit d'une redéfinition de la méthode héritée, que l'on appelle aussi **covariance de type de retour**.

```
abstract class Accessoire extends Produit {  
    //...  
    // copie polymorphe d'Accessoire  
    public abstract Accessoire copie();  
    //..  
}  
-----  
class Bracelet extends Accessoire {  
    //..  
    public Bracelet(Bracelet autre) { super(autre); }  
    // copie polymorphe de Bracelet  
    @Override  
    public Bracelet copie(){  
        return new Bracelet(this);  
    }  
}
```

FIGURE 3

8:25

16:37

Méthode polymorphe de copie.



Dans un constructeur de copie, il est très important de toujours appeler le constructeur de copie de la super-classe. Sinon, ce sera le constructeur par défaut qui s'exécutera : il ne copiera pas les attributs définis dans la super-classe, mais reprendra les valeurs par défaut.

Pour que cette règle soit respectée, il faut que toutes les super-classes dans la hiérarchie aient un constructeur de copie.

### FAILLES D'ENCAPSULATION

La copie profonde permet d'éviter les failles d'encapsulation. Ces failles existent dans notre programme dans toutes les méthodes où nous utilisons directement une référence passée en paramètre d'une méthode pour la stocker dans un attribut.

La méthode `ajouter()` par exemple, qui ajoute un accessoire à une montre est problématique si elle stocke directement dans le tableau la référence à l'accessoire passé en paramètre. La montre contient alors des accessoires qui pourraient être modifiés par une autre partie du programme.

Pour éviter ces **failles d'encapsulation**, il faut toujours copier les objets fournis comme arguments de méthodes avant de les stocker dans un attribut. On parle de **copie défensive** qui permet d'éviter des effets de bords indésirables (fig. 4). La copie profonde permet de réaliser la copie de façon appropriée lorsque le partage de données n'est pas souhaitable (chaque montre a ici ses propres accessoires qu'elle ne partage pas avec une autre montre par exemple).

```
class Montre extends Produit {
    //...
    public Montre(Mecanisme depart)
    {
        coeur = depart.copie();
        accessoires = new ArrayList<Accessoire>();
    }
    //...
}
```

FIGURE 4

11:39

16:37

Copie défensive du mécanisme de l'extérieur.

Le code complet de l'étude de cas est fourni [ici](#). Cette leçon conclut l'étude de cas et termine ce cours d'introduction à la programmation orientée objet.



## IMPRESSIONUM

© EPFL Press, 2016.  
Tous droits réservés.

Graphisme:  
Emphase Sàrl, Lausanne

Résumé: Frank Schmutz  
et Yann Bolliger

Développés par EPFL Press, les BOOCs (Book and Open Online Courses) sont le support compagnon des MOOCs proposés par l'École polytechnique fédérale de Lausanne. Valeur ajoutée aux MOOCs, ils rassemblent l'essentiel à retenir pour l'obtention du certificat et constituent un atout pédagogique. Learn faster, learn better. Bonne révision!

ISBN 978-2-88914-398-6