

[Fiche 1 : Introduction aux classes et objets](#)[Classes et objets](#)[Créer un objet](#)[Les classes](#)[Types primitifs et objets en Java](#)[Conventions d'écriture \(alias Coding style\)](#)[Visibilité et encapsulation](#)[Accesseurs, mutateurs](#)[Construire, détruire](#)[Construire](#)[Valeurs par défaut](#)[Détruire ?](#)[Les tableaux en Java](#)[Les types énumérés en Java](#)[Éléments particuliers: point d'entrée, affichage, comparaison, ...](#)[Méthode principale en Java](#)[La méthode toString\(\)](#)[Comparaison de référence, comparaison sémantique](#)[Attributs et méthodes de classe](#)

Fiche 1 : Introduction aux classes et objets

L'objectif de cette première fiche est d'introduire deux des notions fondamentales de la programmation orientée objet, l'**objet** et la **classe**. Cette fiche résume également les notions connexes telles que l'**encapsulation**, la gestion de la mémoire en Java, les tableaux en Java et la notion de membre de classe.

Remarque

Même si le langage support utilisé est Java et que certaines choses introduites ici sont spécifiques à ce langage, la plupart des notions abordées sont néanmoins générales à la Programmation Orientée Objet et sont simplement instanciées de manière différentes dans d'autres langages.

Classes et objets

L'**objet** est la notion centrale en POO. Un objet est caractérisé par :

- un état (les valeurs de ses *attributs*) ;
- un comportement (les *méthodes* pouvant lui être appliquées), des services que l'objet peut nous rendre ;
- une identité (par exemple une adresse en mémoire).

Un objet n'a de « réalité » qu'à l'exécution du programme. On accède à un objet par l'intermédiaire d'une référence (en Anglais, *handle*, ce qui signifie littéralement « poignée »), qui a beaucoup de points communs avec un pointeur C. L'instruction Java `new` sert à créer un objet. Cette instruction réserve l'espace mémoire nécessaire, initialise les attributs de l'objet créé et renvoie la référence.

Créer un objet

Prenons un exemple. Un pangolin est un objet qui peut être caractérisé de la manière suivante :

Pangolin ? ▼

- état : un nom, un nombre d'écailles, une position spatiale (x, y)
- comportement : traduire, crier

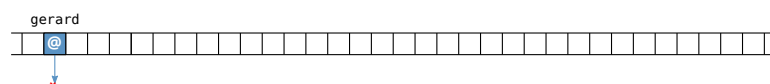
Pour créer un pangolin (et pouvoir l'utiliser après), il faut déclarer une référence de type `Pangolin` et utiliser l'opérateur `new` pour créer effectivement l'objet.

```
Pangolin gerard;           // déclare une référence de type Pangolin
gerard = new Pangolin();    // réserve la zone mémoire nécessaire et l'affecte à la référence
```

La valeur par défaut d'une référence est la valeur `null`, qui signifie approximativement que la référence ne pointe sur rien du tout.

Voici schématiquement (et de manière très imagée) ce qui se passe dans la mémoire à l'exécution des instructions précédentes.

1. Création de la référence (1ère instruction), initialement de valeur `null`



2. Création de l'objet et affectation de cet objet à la référence.



Les classes

À quel endroit les caractéristiques des objets (ce qui définit leur état et leur comportement) sont-elles définies ? Dans les **classes**. La définition d'une classe comporte trois éléments :

- son nom ;
- la liste de ses attributs (caractérisant l'état) ;
- la liste de ses méthodes (les services que tout objet de cette classe peut nous rendre).

Voici la syntaxe de déclaration d'une classe, illustrée sur un exemple.

```
class Pangolin {
    // Ci-dessous la déclaration des attributs de la classe
    double x;
    double y;
    String name;
    int nbEcailles;

    // La déclaration des méthodes avec leur code
    void translate(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }

    void crier() {
        System.out.println("Gwark Rhââgn Bwwikk"); // Cri du pangolin
    }
}
```

La définition d'une classe permet :

- de typer des références (une classe définit un *type de données*) ;
- de créer des objets (appelés *instances de la classe*) avec l'opérateur `new` ;
- d'accéder à l'état interne avec le `.` (par exemple `gerard.x = 2;`) ;
- d'appeler (invoquer) des méthodes avec le `.` (par exemple `gerard.crier();`).

Remarque

On peut remarquer dans le code ci-dessus la présence de la variable `this`. Cette variable contient une référence à l'objet sur lequel la méthode est invoquée. Il s'agit d'un paramètre implicite passé lorsque l'on appelle une méthode sur un objet. Par exemple, lors de l'appel `gerard.translate(0, 1);`, `this` contient une référence vers le même objet que `gerard` ; c'est son état qui sera modifié par la méthode.

Types primitifs et objets en Java

En Java, il existe deux catégories de types :

- les types primitifs à la C : `int`, `float`, `char`, `double`, `boolean`, ...
- les types objets, qui sont des références (on ne manipule pas directement la valeur, mais on le fait par l'intermédiaire d'un « pointeur »).

Notez que les types primitifs s'écrivent en minuscule alors que les noms de classes commencent toujours par une majuscule (si vous suivez les [conventions de codage](#), ce qui est fortement recommandé!). Ainsi votre classe doit être nommée `Pangolin` et non `pangolin`, et son attribut `name` est de type `String` - qui est donc une classe!

Le mécanisme de JVM est fondamental pour la portabilité, le codage des nombres étant ainsi indépendant de l'architecture des machines!

Le tableau ci-dessous résume les différents types primitifs.

	Taille en octets	Valeur minimale (inclue)	Valeur maximale (inclue)
entiers			

	Taille en octets	Valeur minimale (inclue)	Valeur maximale (inclue)
byte	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long	8	-9223372036854775808	9223372036854775807
caractères			
char	2	'\u0000' = 0	'\uffff' = 65535
flottants			
float	4	-3.4028235 x 10 ³⁸	3.4028235 x 10 ³⁸
double	8	-1.7976931348623157 x 10 ³⁰⁸	1.7976931348623157 x 10 ³⁰⁸

Seul le type primitif `boolean` n'est pas dans le tableau. Tout ce que l'on sait de `boolean` c'est qu'il a deux valeurs : `true` et `false`. On ne peut pas l'utiliser comme un nombre (`false` n'est pas 0, `true` n'est pas 1) et sa taille en mémoire n'est pas spécifiée.

Une variable locale non initialisée (`int x;` ou `Pangolin gerard;`) ne peut pas être utilisée directement (par exemple `x++;` ou `gerard.crier();`). Mais vous avez de la chance, le compilateur devrait vous tenir au courant ("*The local variable xxx may not have been initialized.*")

En Java, la plupart des variables manipulées sont du second type (objet). Attention donc, on manipule des **références** ! Regardez par exemple le code suivant.

```
Pangolin gerard = new Pangolin();
Pangolin toto = gerard;
toto.translater(1, 0); // Que vaut alors gerard.x ?
```

Conventions d'écriture (alias *Coding style*)

Plusieurs règles d'écriture sont en vigueur dans la communauté des développeurs Java. On désigne ces règles sous la terminologie de *Coding style*. Voici par exemple celles portant sur les noms :

- Un nom de variable, d'attribut ou de méthode commence par une minuscule et est écrit en *Camel Case*. Exemple : `maBelleVariable`

Camel Case ? ▼

- Un nom de classe commence par une majuscule et est écrit en *Camel Case*. Exemple : `MaBelleClasse`
- Un nom de constante est intégralement en lettres capitales, mots séparés par des tirets bas. Exemple : `MA_BELLE_CONSTANTE`

Retenez aussi notamment les indentations de taille 4 et les accolades ouvrantes en fin de ligne. Regardez les exemples de ce cours, qui naturellement respectent ces règles.

Vous trouverez au lien suivant le [Coding style Java](#). Officiellement il n'est plus maintenu par Oracle, mais est toujours en vigueur.

Attention

Il y a peu de langages dans lesquels les conventions d'écriture sont aussi respectées. Libre à vous de ne pas les suivre, mais alors, un grand malheur s'abattra sur vous et votre équipe de développement pendant sept générations.

Visibilité et encapsulation

Quel est le problème du code suivant ?

```
Pangolin gerard = new Pangolin();
gerard.nbEcailles = -1;
```

C'est assez évident : en ayant accès à tous les attributs internes de la classe Pangolin, l'utilisateur de cette classe peut mettre n'importe quel objet dans un état incohérent (par exemple un état dans lequel le nombre d'écailles est négatif, ce qui n'a pas de sens).

Une manière de résoudre ce problème est de restreindre la **visibilité** des attributs et méthodes à l'aide de modificateurs. C'est l'*encapsulation*. En Java, il existe quatre niveaux de visibilité pour les attributs et méthodes, dont les effets sont résumés dans le tableau ci-dessous (les notions de sous-classes et de paquetages seront précisés plus tard).

	Classes des autres paquetages	Sous-classes	Autres classes, même paquetage	Intérieur de la classe
public (+)	✓	✓	✓	✓
protected (#)	✗	✓	✓	✓
défaut ()	✗	✗	✓	✓
private (-)	✗	✗	✗	✓

- ✓ : accès possible (lecture et écriture)
- ✗ : accès interdit (lecture et écriture)

Reprenons la classe Pangolin développée ci-dessus, et précisons les visibilités pour chacun des attributs et des méthodes.

```
class Pangolin {
    // Ci-dessous la déclaration des attributs de la classe
    private double x;
    private double y;
    private String name;
    private int nbEcailles;

    // La déclaration des méthodes avec leur code
    public void translate(double dx, double dy) {
        x += dx;
        y += dy;
    }

    public void crier() {
        System.out.println("Gwark Rhâagn Bwikk"); // Cri du pangolin
    }
}
```

Avec une telle déclaration, le code suivant :

```
Pangolin gerard = new Pangolin();
gerard.nbEcailles = -1;
```

est incorrect s'il est écrit dans une autre classe que Pangolin. L'attribut `nbEcailles` (comme tous les autres attributs) est invisible de l'extérieur de la classe.

Il existe un principe fondamental en Programmation Orientée Objet : le **principe d'encapsulation**. Ce principe stipule que la représentation de l'état interne d'une classe d'objets n'est connue et accessible que de cette classe, et qu'en particulier aucune autre classe ne peut modifier directement l'état interne d'un objet. Cela se traduit de la manière (simplifiée) suivante.

Principe d'encapsulation (simplifié)

Tout attribut d'une classe doit être déclaré privé.

Accesseurs, mutateurs

Il existe cependant de nombreuses situations où l'on a besoin d'accéder à des valeurs d'attribut d'un objet, soit pour les consulter, soit pour les modifier. Dans ce cas-là, une solution est d'utiliser des méthodes particulières appelées accesseurs (*accessors*) et mutateurs (*modifiers*). Exemple sur Pangolin.

```
class Pangolin {
    // Ici le reste de classe Pangolin (comme précédemment)
    // [...]
    public int getNbEcailles() {
        return this.nbEcailles;
    }

    public void setNbEcailles(int nb) {
        if (nb <= 0) {
            throw new IllegalArgumentException("Le nombre d'écailles doit être strictement positif");
        }
        this.nbEcailles = nb;
    }
}
```

L'avantage d'utiliser des accesseurs et mutateurs est que l'on peut finement contrôler l'accès à l'état interne d'un objet, et toujours garantir son intégrité.

Il est important de **ne pas écrire systématiquement** les set/get, mais seulement lorsque c'est nécessaire. Une classe a très souvent des attributs privés auxquels on ne doit pas accéder et/ou qui ne doivent pas être modifiés depuis l'extérieur.

Construire, détruire

Construire

La visibilité permet de résoudre le problème d'incohérence lié à un accès non contrôlé aux attributs. En revanche, elle ne permet pas de résoudre les problèmes potentiels d'incohérence liés à l'initialisation des attributs aux valeurs par défaut. Considérons par exemple le code suivant.

```
Pangolin gerard = new Pangolin();
gerard.setNbEcailles(1542);
```

Entre les lignes 1 et 2, notre pangolin est dans un état incohérent car son nombre d'écailles est nul. Pour éviter ce problème, il faut rendre *atomique* la création d'un objet (la réservation de la zone mémoire) et l'initialisation de ses attributs. C'est exactement le rôle des *constructeurs*.

Un constructeur est une « méthode » spécifique d'une classe, portant le nom de cette classe, *sans type de retour*, et qui exécute du code au moment de la création de l'objet. Voici un exemple.

```
class Pangolin {
    // Ici le reste de classe Pangolin (comme précédemment)
    // [...]
    public Pangolin(String nom, double xInit, double yInit, int nbEcailles) {
        this.nom = nom;
        this.x = xInit;
        this.y = yInit;
        this.setNbEcailles(nbEcailles);
    }
}
```

Le constructeur est appelé à la création de l'objet, c'est-à-dire lors de l'utilisation de l'opérateur `new`. Ainsi, par exemple, le code suivant construit un pangolin en appelant le constructeur déclaré ci-dessus, c'est-à-dire en initialisant les attributs aux valeurs passées en paramètre.

```
Pangolin gerard = new Pangolin("G  rard", 0, 0, 1542);
```

Ainsi, plus de probl  me d'incoh  rence    l'initialisation.

Une classe peut contenir plusieurs constructeurs. Ces constructeurs peuvent m  me faire appel l'un    l'autre. Exemple :

```
class Pangolin {
    // Ici le reste de classe Pangolin (comme pr  c  demment)
    // [...]
    public Pangolin(String nom, int nbEcailles) {
        this(nom, 0, 0, nbEcailles); // Appel au constructeur    quatre param  tres
    }
}
```

Valeurs par d  faut

Lors de l'appel    `new`, plusieurs op  rations sont en fait r  alis  es:

1. allocation de la m  moire pour l'objet cr   ;
2. initialisation de tous les attributs, avec une valeur par d  faut ou celle sp  cifi  e;
3. ex  cution du constructeur (s'il y en a plusieurs, choisi selon les types des arguments).

Il est donc possible de sp  cifier la valeur initiale d'un attribut directement lors de sa d  claration :

```
class Toto {
    private int a = 17;
    private String s = new String("zut");
    // [...]
}
```

Si aucune valeur n'est sp  cifi  e, les entiers seront initialis  s    `0`, les flottants    `0.0`, les bool  ens    `false` et toutes les r  f  rences    `null`. Le constructeur n'est ex  cut   qu'*apr  s* ces initialisations.

Constructeur par d  faut

Lorsqu'il n'y a aucun constructeur explicitement d  fini dans une classe, Java en synth  tise un, qui ne prend aucun param  tre et... ne fait rien. Toutes les attributs sont donc initialis  s    leur valeur par d  faut. C'est ce constructeur (appel   *constructeur par d  faut*) qui   tait appel   dans les tout premiers programmes de cette fiche lorsque l'on faisait un `new Pangolin()`.

Attention, d  s qu'un constructeur est explicitement d  fini, ce constructeur par d  faut dispara  t (et dans ce cas-l  , l'instruction `new Pangolin()` devient invalide s'il n'existe pas de constructeur sans param  tre dans la classe `Pangolin`).

D  truire ?

En Java, on ne peut jamais d  truire explicitement un objet, donc on ne peut pas lib  rer explicitement une zone de la m  moire occup  e par un objet. Il arrive malgr   tout que des objets cr   s deviennent inutiles au cours du fonctionnement du programme. Comment   viter la saturation de la m  moire, dans ce cas ?

En fait, c'est le r  le du *ramasse-miettes* (*garbage collector*) de la machine virtuelle Java. Ce ramasse-miettes est lanc   de mani  re p  riodique par la JVM et se charge de rep  rer les objets n'ayant plus aucune r  f  rence pointant sur eux. Ces objets sont par d  finition inutiles, car inaccessibles par le programme. La zone m  moire qu'ils occupent peut donc   tre marqu  e libre.

Les tableaux en Java

En Java, un tableau d  signe un ensemble d'objets de m  me type,    acc  s direct par indice, et de taille fixe (non modifiable). Un tableau a toutes les caract  ristiques d'un objet (en particulier, un tableau se manipule par l'interm  diaire d'une r  f  rence). La manipulation de tableaux passe par une syntaxe sp  cifique inspir  e du C : les crochets `[]`.

Voici la syntaxe, via quelques exemples :

```
// 1. Un premier tableau d'entiers
int[] tab;           // déclaration, tab n'est pas encore initialisé
tab = new int[5];    // allocation;
                    // => les indices vont de 0 à tab.length - 1, ici 4
                    // => les valeurs sont initialisées à 0
tab[0] = 2;          // accès via l'opérateur []

// 2. tableau 1D d'objets
int n = 42;
Pangolin[] pangolins = new Pangolin[n]; // attention, ne contient que des références nulles

for (int i = 0; i < pangolins.length; i++) {
    pangolins[i] = new Pangolin(...); // allocation objet par objet
}

// autre parcours, de type "for each"
// (pour appliquer un traitement à *tous* les éléments)
for (Pangolin p : pangolins) {
    System.out.println(p.toString());
}

// 3. Tableau 2D
int[][] tab2D;
tab2D = new int[5][]; // tab de 5 références nulles
for (int i = 0; i < tab2D.length; i++) {
    // allocation case par case. Dimensions par forcément identiques
    tab2D[i] = new int[i + 1];
}
// et maintenant, tout simplement :
tab2D[1][0] = 45;
```

Remarques

Un tableau étant paramétré par n'importe quel type Java, on peut bien entendu créer des tableaux de tableaux, ce qui permet de simuler par exemple des matrices (par exemple : `Pangolin[][] mat = new Pangolin[10][10];` // 100 pangolins !).

On peut accéder à la longueur d'un tableau grâce au pseudo-attribut `length` (par exemple : `int l = tab1.length`)

Les indices sont numérotés de 0 à `length - 1`. Un débordement d'indice provoque une exception de type `ArrayIndexOutOfBoundsException`.

Les types énumérés en Java

Un type énuméré est une classe particulière qui ne possède qu'un nombre restreint d'objets (qui sont en fait des constantes). Voici un exemple minimal :

```
enum Couleur {
    ROUGE, VERT, BLEU;
}
```

Dans l'exemple ci-dessus, `Couleur` est donc un type qui ne contient que trois valeurs possibles : `ROUGE`, `VERT` et `BLEU`.

On peut donc utiliser cette énumération pour typer n'importe quelle variable, ce qui peut être utile partout où l'on a besoin de raisonner avec des constantes particulières (utile par exemple pour faire un `switch` ou utiliser des conditionnelles s'appuyant sur la valeur particulière d'une certaine couleur).

On peut également itérer à travers toutes les valeurs d'un type énuméré en utilisant la méthode `values()` (pour avoir une idée de la syntaxe, jetez un œil à la section sur les itérateurs, dans la [fiche sur les collections](#)).

Bon, tout cela est très intéressant, mais notre type énuméré `Couleur` est assez basique pour le moment. Ça tombe bien, on peut le complexifier un peu en lui ajoutant des attributs et des méthodes, à la manière d'une classe. Jugez plutôt :

```
enum Couleur {
    ROUGE(255, 0, 0), VERT(0, 255, 0), BLEU(0, 0, 255);

    private int rouge, vert, bleu;

    Couleur(int rouge, int vert, int bleu) {
        this.rouge = rouge;
        this.vert = vert;
        this.bleu = bleu;
    }

    public String getHtmlCode() {
        return "#" + String.format("%02x", this.rouge)
            + String.format("%02x", this.vert)
            + String.format("%02x", this.bleu);
    }
}
```

Ainsi, par exemple, on pourra écrire `Couleur c = ROUGE; System.out.println(c.getHtmlCode());` pour récupérer le code HTML associé à la couleur rouge.

Voilà, vous savez tout.

Éléments particuliers: point d'entrée, affichage, comparaison, ...

Méthode principale en Java

En Java, pour lancer un programme, il faut définir une méthode principale dans une classe, en utilisant la syntaxe suivante :

```
// Le nom de la classe est à votre convenance.
// L'usage est d'appeler TestXXX une classe qui contient le programme principal
public class TestPangolin {
    public static void main(String[] args) {
        // Le tableau args, de taille args.length, contient les arguments du program
        // Ici définissez les instructions de votre programme principal
    }
}
```

La classe doit être *publique*, via le mot-clé `public` devant `class` ! Cette notion sera vue dans la fiche sur [les paquetages](#).

Il convient ensuite de compiler ce fichier et d'interpréter le code binaire correspondant pour exécuter le programme: `java TestPangolin`. Tout ceci est expliqué en détails ici: [compilation et exécution](#).

Dans un même projet, plusieurs classes peuvent définir une méthode `main`. Le point d'entrée de l'exécution dépend simplement de la manière dont elle est lancée: `java Classe1` ou `java Classe2` invoque la méthode `main` de la classe spécifiée.

La méthode `toString()`

Dans de nombreux cas, il peut être utile d'avoir une description textuelle d'un objet (pour l'afficher à l'écran par exemple). Pour cela, l'usage est de définir une méthode s'appelant `toString()` et renvoyant une chaîne de caractères. Exemple sur Pangolin :

```
class Pangolin {
    // Ici le reste de classe Pangolin (comme précédemment)
    // [...]

    @Override
    public String toString() {
        return "Le Pangolin " + this.nom + "(" + this.nbEcailles + " écailles)"
    }
}
```

Un intérêt majeur de cette méthode est que la JVM l'appelle implicitement à chaque endroit où elle a

besoin d'une représentation textuelle d'un objet. Ainsi, par exemple, le code `System.out.println(gerard);` devrait être invalide car la méthode `System.out.println` attend une chaîne de caractères en paramètre. Cependant, elle est valide en pratique, car la JVM ajoute implicitement l'appel à la méthode `toString()` sans que l'on n'ait rien demandé: `System.out.println(gerard.toString());`.

C'est un peu magique non? Comment le compilateur sait que la méthode `toString()` existe dans `Pangolin`? D'ailleurs existe-t-elle?

Pour le savoir, restez avec nous pour les prochains cours !

Comparaison de référence, comparaison sémantique

Avancé !

N'ayez pas peur nous reviendrons dessus, mais retenez que l'opérateur `==` ne compare que des références, pas des objets. Danger !

Il existe un opérateur de comparaison en Java : `==`. Cet opérateur renvoie `true` uniquement lorsque les deux références comparées pointent sur le même objet en mémoire.

```
Pangolin gerard = new Pangolin("G  rard", 0, 0, 1452);
Pangolin gerard2 = new Pangolin("G  rard", 0, 0, 1452);
System.out.println(gerard == gerard2); //   crit "false"
gerard2 = gerard;
System.out.println(gerard == gerard2); //   crit "true"
```

Souvent, ce n'est pas ce que l'on cherche lorsque l'on veut comparer deux objets. Prenons par exemple deux cha  nes de caract  res (contrairement    ce que l'on pourrait penser, le type cha  nes de caract  res n'est pas un type primitif en Java, mais un type objet `String`). Le recours    l'op  rateur `==` ne donnera pas le r  sultat attendu :

```
String s1 = new String("abc");
String s2 = new String("abc");
System.out.println(s1 == s2); //   crit "false"
```

C'est compl  tement normal. M  me si `s1` et `s2` contiennent la m  me cha  ne de caract  res, il s'agit bien de deux objets diff  rents (dont le tableau de caract  res dont ils sont constitu  s est le m  me). Si l'on veut faire une comparaison *s  mantique* des cha  nes de caract  res (c'est-  -dire comparer les caract  res dont elles sont constitu  es), il faut utiliser la m  thode `equals()`.

```
String s1 = new String("abc");
String s2 = new String("abc");
System.out.println(s1.equals(s2)); //   crit "true"
```

Le fonctionnement de cette comparaison s  mantique n'est pas magique. Java ne peut pas deviner tout seul ce que veut dire le fait d'  tre   gal pour des objets d'une certaine classe. Cette m  thode `equals()` doit donc   tre   crite explicitement pour chaque classe pour laquelle on a besoin d'une comparaison s  mantique (elle l'est dans la classe `String()` par exemple). Si l'on veut par exemple sp  cifier que deux pangolins sont   gaux si et seulement s'ils ont le m  me nom, il faudra d  finir cette m  thode dans la classe `Pangolin`, de la mani  re suivante.

```
class Pangolin {
    // Ici le reste de classe Pangolin (comme pr  c  demment)
    // [...]

    @Override
    public boolean equals(Object other) {
        if (other instanceof Pangolin) {
            return ((Pangolin) other).nom.equals(this.nom);
        }
        return false;
    }
}
```

Remarque

On peut remarquer la présence de l'opérateur Java `instanceof` qui permet de tester si un objet est l'instance d'une certaine classe. Attention au type du paramètre de `equals` qui est bien `Object`. Quant à l'annotation `@Override`, son utilisation sera détaillée plus tard.

Remarque 2

En fait, la gestion des objets `String` est un peu plus compliquée que ce qui a été exposé plus haut et peut résulter en un comportement surprenant de l'opérateur d'égalité. Si vous voulez en savoir un peu plus (et que vous comprenez les risques que vous prenez en accédant à tant de savoir nouveau et avancé), cliquez sur le bouton ci-dessous :

String pooling (info. avancée) ▼

Attributs et méthodes de classe

Les attributs et méthodes dont nous avons parlé jusqu'ici sont relatifs à un objet particulier (une instance). Ainsi, par exemple, la valeur de l'attribut `nbEcailles` dépend de l'objet duquel on parle, et pourra être différente selon qu'il s'agisse de `gerard` ou `toto`. De même, lorsque l'on appelle la méthode `translater()`, c'est bien un pangolin particulier que l'on va translater. On parle donc d'*attributs d'instance* et de *méthodes d'instance*.

Il n'en est pas de même pour tous les attributs et méthodes. Ainsi, par exemple, imaginons un attribut, dans la classe `Pangolin`, qui stocke le nombre d'instances de pangolins créés jusqu'ici. La valeur de cet attribut ne varie évidemment pas selon les objets. Elle est au contraire commune à tous les objets de la classe. L'attribut n'est pas relatif aux objets, mais à la classe.

Un tel attribut est dit *de classe*, ou encore *statique*. Un attribut de classe est déclarée avec le mot-clef `static` et on peut y accéder avec le nom de la classe suivi de la notation usuelle `..`. Exemple avec la classe `Pangolin` :

```
class Pangolin {
    // Ici le reste de classe Pangolin (comme précédemment)
    // [...]

    // Un attribut de classe, partagé par tous les pangolins
    private static int nbPangolins = 0;

    // Une méthode de classe, indépendante de toute instance particulière
    public static int getNbPangolins() {
        return nbPangolins; // ou return Pangolin.nbPangolin ;
    }

    public Pangolin(String nom, double xInit, double yInit, int nbEcailles) {
        nbPangolins++; // Un Pangolin de plus a été créé ! On peut aussi écrire Pangoli
        this.nom = nom;
        this.x = xInit;
        this.y = yInit;
        this.setNbEcailles(nbEcailles);
    }
}
```

L'accès à un attribut de classe et l'invocation d'une méthode de classe peuvent se faire, comme d'habitude, au moyen d'une référence vers une instance. Mais, puisqu'ils ne sont liés à aucune instance particulière, il est aussi possible d'utiliser le nom de la classe, et non pas un objet :

```
Pangolin gerard = new Pangolin() ;
// récupère le nb de pangolins créés jusque-là
System.out.println( gerard.getNbPangolins() ) ; // Bon, ça marche... Affiche 1.

// Mais on peut aussi accéder à un membre de classe directement avec le nom de la c
// Et c'est d'ailleurs ce qu'on fait en général avec les membres de classe !
System.out.println( Pangolin.getNbPangolins() ) ; // Affiche 1 aussi.
```

Quizz : Que se passe-t-il si vous utilisez la référence `this` dans une méthode de classe?

[Fiche 3 : Héritage](#)[Problématique](#)[Héritage](#)[Terminologie](#)[Hiérarchies de classes](#)[Héritage simple \(et pas multiple\)](#)[Redéfinition de méthodes](#)[Redéfinition avec réutilisation](#)[Redéfinition vs surcharge](#)[Construction](#)[Constructeur par défaut](#)[Châinage ascendant des constructeurs](#)[Visibilités](#)[La super-classe Object](#)[Pour finir: final](#)

Fiche 3 : Héritage

L'objectif de cette fiche est d'introduire une notion fondamentale de la Programmation Orientée Objet : l'**héritage** (*inheritance*). Ce mécanisme permet de définir une nouvelle classe à partir d'une classe existante, en *héritant* de toutes les caractéristiques (attributs et méthodes) de la classe d'origine, tout en pouvant en ajouter de nouvelles. L'héritage est une notion puissante de la POO, qui permet à la fois la réutilisation de code, une meilleure mise en facteur du code, un haut niveau d'abstraction de vos programmes et une très forte capacité d'évolution.

Problématique

Dans la [fiche #1](#), vous avez (re)découvert cet animal merveilleux qu'est le Pangolin. Pour rappel, voici une classe permettant de le représenter :

```
class Pangolin {
    private String nom;
    private int nbEcailles;

    public Pangolin(String nom, int nbEcailles) {
        this.nom = nom;
        this.setNbEcailles(nbEcailles);
    }

    public int getNbEcailles() {
        return nbEcailles;
    }

    public void setNbEcailles(int nb) {
        if (nb <= 0) {
            throw new IllegalArgumentException("Le nombre d'écailles doit être strictement positif");
        }
        this.nbEcailles = nb;
    }

    public void crier() {
        System.out.println("Gwark Rhââgn Bwwikk"); // Cri du pangolin
    }
}
```

Une espèce particulière de Pangolin est le « Pangolin à longue queue », appendice qu'il utilise pour saisir des branches (véridique). Comparons ces deux animaux:

- Tous deux possèdent un grand nombre de **propriétés communes**: un nom, un nombre d'écailles et le fait de crier ;
- Comme son nom l'indique, le Pangolin à longue queue possède **en plus** une queue ;
- Même si leur cri est très proche, l'expert saura reconnaître celui **spécifique** de l'animal à queue.

Naturellement, nous brûlons d'envie d'écrire une classe `PangolinALongueQueue` ! Pour commencer, voici deux solutions naïves:

1. Une première solution serait de copier la classe `Pangolin`, en y ajoutant un attribut `longueurQueue` et en adaptant le constructeur et la méthode `crier()`. Par pudeur cette classe est ici masquée, mais elle ressemblerait à ça:

Solution naïve 1 ▼

Comme vous pouvez le constater, ce code est très redondant ce qui est loin d'être satisfaisant. De plus, imaginez le travail (et la maintenance !) si vous devez un jour ajouter une nouvelle sorte de Pangolin ou préciser la couleur des écailles de ces animaux... Enfin, cette solution naïve ne rend pas compte du fait qu'un Pangolin à longue queue n'est, somme toute, qu'un cas particulier de Pangolin. Si par exemple il vous prend l'envie d'écrire un traitement qui prend un Pangolin quelconque en paramètre, eh bien il vous faudrait écrire en fait deux méthodes : une qui prend un `Pangolin` en paramètre et une autre qui prend un `PangolinALongueQueue` en paramètre !

2. Une deuxième solution serait de ne garder que la classe `Pangolin` d'origine, et de lui ajouter un attribut `longueurQueue`, qui ne serait pertinent que pour certains Pangolins. Mais alors, que signifie `getLongueurQueue()` pour les autres? De plus, il faudrait ajouter un attribut pour savoir si l'animal a une queue, et un test dans la méthode `crier()` pour faire crier l'animal correctement selon son type.

Bien entendu, rien de tout ceci n'est acceptable. La solution va donc venir du principe d'**héritage**.

Une notion essentielle

« L'**héritage** constitue la deuxième caractéristique essentielle d'un langage orienté objet, avec l'**encapsulation** (abstraction des données) et avant le **polymorphisme** ». Bruce Eckel, *Thinking in JAVA*.

Héritage

La raison fondamentale qui permet d'avoir recours à l'héritage dans notre petit exemple est qu'un Pangolin à longue queue - **EST** - avant tout un Pangolin ! Nous allons définir une nouvelle classe `PangolinALongueQueue` qui **hérite** de la classe `Pangolin`.

Avec l'héritage :

- La nouvelle classe `PangolinALongueQueue` possédera les **mêmes propriétés**, attributs et méthodes, que la classe `Pangolin` : ces propriétés sont *héritées*.
- Il est possible de lui **ajouter** des attributs ou des méthodes supplémentaires.

En JAVA, le mot clé traduisant l'héritage entre classes est `extends`. Voici donc une première version de la classe `PangolinALongueQueue` :

```
class PangolinALongueQueue extends Pangolin {
    private int longueurQueue;

    public PangolinALongueQueue(String nom, int nbEcailles, int longueurQueue) {
        // [À COMPLÉTER...]
        this.longueurQueue = longueurQueue;
    }

    public int getLongueurQueue() {
        return longueurQueue;
    }
}
```

Nous pouvons maintenant créer notre premier objet à longue queue :

```
class Test {
    public static void main(String [] args) {
        Pangolin gerard = new Pangolin("Gérard", 1542);
        PangolinALongueQueue rantanplan = new PangolinALongueQueue("Rantanplan", 1966,

        System.out.println(gerard.getNom() + " a " + gerard.getNbEcailles() + " écaille
        System.out.println(rantanplan.getNom() + " a lui " + rantanplan.getNbEcailles()
        + " écailles, et une queue de " + rantanplan.getLongueurQueue() + " cm");

        // Ceci affiche:
        //   Gérard a 1542 écailles
        //   Rantanplan a lui 1966 écailles, et une queue de 28 cm

        // Crions un peu !
        gerard.crier();           // Gwwark Rhââgn Bwwikk
        rantanplan.crier();       // Gwwark Rhââgn Bwwikk
    }
}
```

Plusieurs choses à remarquer :

- Du fait de l'héritage entre classes, l'objet `rantanplan` possède bien un nom et un nombre d'écailles, deux méthodes accesseurs sur ces attributs et une méthode `crier` ; Ces attributs et ces méthodes sont *hérités*.
- Il a en plus une queue ;

Par contre, même s'il peut bien crier, `rantanplan` crie toujours comme `gérard` ! En effet, il hérite simplement la méthode définie dans la classe `Pangolin`. Tout n'est donc pas encore parfait. Mais une nouvelle classe a été créée à moindre coût, en réutilisant au maximum du code existant.

Terminologie

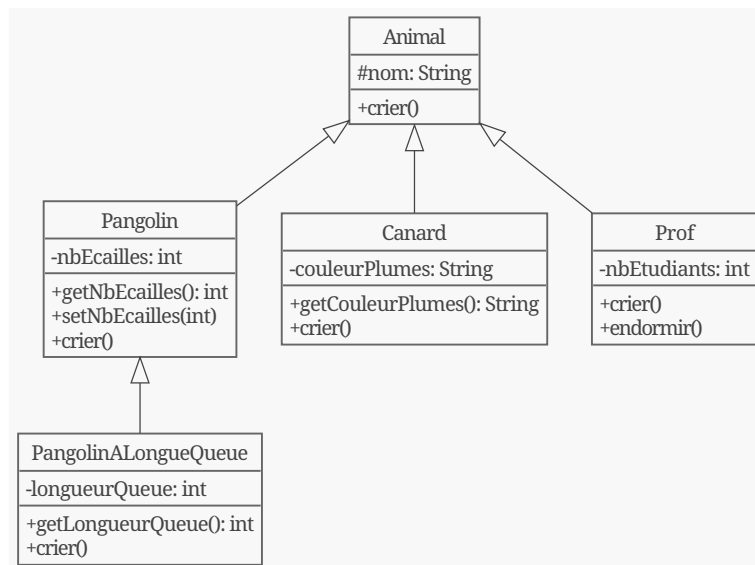
Avant de continuer, introduisons un peu de terminologie :

- L'ensemble des classes liées par une relation d'héritage s'appelle une **hiérarchie de classes** ;
- Une **classe mère** ou **super-classe** est une classe dont on hérite. Réciproquement, une **classe fille** ou **sous-classe** est une classe qui hérite d'une autre classe. Ici, `Pangolin` est la mère de `PangolinALongueQueue`, qui est sa fille.
- Une classe fille **spécialise** ou **étend** sa classe mère. Un `PangolinALongueQueue` est un cas *particulier* de `Pangolin`. Mais, important, c'EST avant tout un `Pangolin` !
- Une classe mère **généralise** les notions communes à sa/ses classe(s) fille(s). Ici `Pangolin` regroupe les caractéristiques partagées par les Pangolins « communs », ceux à longue queue, ceux à écailles tricuspidés, etc.

Hiérarchies de classes

Une classe peut avoir plusieurs filles, c'est-à-dire être héritée par plusieurs autres classes. De plus ces filles peuvent elles-mêmes avoir des filles. Se forment alors des *hiérarchies de classes* permettant de représenter des familles d'objets ayant des caractéristiques communes (en haut de l'arbre) et des spécificités (en bas).

La figure ci-dessous donne l'exemple d'une hiérarchie de classes pour quelques animaux. La notation utilisée est ici UML (*Unified Modeling Language*).



Il n'y a pas de limitation du nombre de niveaux dans la hiérarchie d'héritage. Les attributs et méthodes sont hérités au travers de tous les niveaux. Ainsi, une méthode définie dans une classe est héritée par ses petites-petites-petites filles.

Et qu'y a-t-il au-dessus ?

En fait, en Java, toute classe qui n'a pas explicitement de classe mère hérite *implicitement* d'une super-classe nommée `Object`. Elle est donc la racine commune de TOUTES les (hiérarchies de) classes JAVA existantes.

Récréation

Qu'on se le dise, l'Homme est en fait cousin du `Pangolin` (si, si...).

La fin de cette fiche reviendra plus en profondeur sur cette [super-classe Object](#).

Héritage simple (et pas multiple)

Conceptuellement, il est assez courant de pouvoir considérer un objet suivant différents point de vue. Par exemple un Pangolin est un Animal, mais pourrait aussi être vu comme un cas particulier de Peluche ou d'ObjetDeDerision. Il serait donc parfois pertinent d'hériter de *plusieurs* classes, pour hériter de leurs différentes caractéristiques. On parle alors d'**héritage multiple**. Pourtant ce n'est pas un problème simple, ni en conception ni sur le plan technique.

- En langage de modélisation Orientée Objet UML, l'héritage multiple est parfaitement défini ;
- Certains langages de programmation l'autorisent, comme C++ ou Python. Il faut alors mettre en place des mécanismes techniques pas toujours simples en pratique ; mais il n'y a pas de solution évidente et n'ayant que des avantages. Ceux qui veulent s'en persuader pourront se documenter sur le « problème du diamant » (*diamond problem*).
- D'autres langages ont choisi de ne faire que de l'héritage simple. C'est le cas de Java, d'ObjectiveC ou de C# par exemple. Le principe est alors de revoir la conception des diagrammes de classes pour contourner le problème.

Nous retiendrons donc qu'en JAVA, il n'y a qu'une seule Maman :

En JAVA, l'héritage multiple n'est *simplement pas possible*. Une classe ne peut hériter que d'une seule classe !

Redéfinition de méthodes

Une sous-classe peut ajouter de nouvelles caractéristiques (méthodes et attributs) à celles héritées de sa classe mère. Mais elle peut aussi redéfinir des méthodes héritées pour adapter – spécialiser – leur comportement.

- **Redéfinir** une méthode signifie écrire dans la sous classe une méthode dont la *signature* (nom, type de retour et paramètres) est identique à celle d'une des méthodes héritées ;
- Lorsqu'une méthode est redéfinie, elle *remplace* la méthode héritée. C'est la nouvelle définition qui est exécutée et pas celle de la classe mère.

Pour nos chers animaux, observant qu'un Pangolin à longue queue ne crie pas comme une Pangolin sans queue, nous allons redéfinir la méthode `crier()` dans la classe `PangolinALongueQueue` :

```
class PangolinALongueQueue extends Pangolin {
    [...] // Même code que précédemment

    @Override
    public void crier() {
        System.out.println("Qeyyyoooouuu Gwark Rhââgn Bwikk");
    }
}
```

Observons :

```
System.out.println(gerard.getNbEcailles());    // 1542
System.out.println(rantanplan.getNbEcailles()); // 1966

gerard.crier();                               // Gwark Rhââgn Bwikk
rantanplan.crier();                           // Qeyyyoooouuu Gwark Rhââgn Bwikk
```

- La méthode `getNbEcailles()` invoquée sur l'objet `rantanplan` est celle héritée depuis la classe `Pangolin` ; donc la même que pour `gerard`, définie dans la classe `Pangolin`.
- Par contre, la méthode `crier()` a été redéfinie dans la classe fille. C'est donc cette nouvelle définition qui est invoquée sur `rantanplan`. Le comportement est maintenant bien spécifique !

Redéfinition avec réutilisation

Lors d'une redéfinition, il est possible d'appeler la méthode héritée que l'on est en train de réécrire au moyen du mot clé `super` :

```
@Override
public void crier() {
    System.out.print("Qeyyyouuuu "); // partie spécifique
    super.crier();                    // partie commune (réutilisation)
}
```

De la même manière que `this` est la référence à l'instance sur laquelle la méthode est invoquée, `super` permet de désigner la classe mère. L'usage de `super` pour désigner un champ de la super-classe peut être fait à n'importe quel endroit dans le code de la sous-classe.

Redéfinition vs surcharge

Attention

Ne pas confondre **redéfinition** (*override*; réécriture d'une méthode héritée) et **surcharge** (*overload*; ajout d'une nouvelle méthode de même nom) !

Rappel :

- **Surcharger** une méthode (dans une classe, ou dans une sous classe) signifie écrire une *nouvelle* méthode qui a le même nom qu'une méthode déjà existante, mais dont le nombre et/ou le type des paramètres diffère. Surcharger *ajoute* une nouvelle méthode à la classe.
- **Redéfinir** une méthode dans une sous-classe signifie écrire une méthode qui a exactement la même signature qu'une méthode héritée. La méthode redéfinie *remplace*, dans la classe fille, la méthode héritée.

Détails sur la différence entre surcharge et redéfinition ▼

Annotations

L'annotation `@Override` précédant la redéfinition d'une méthode permet de notifier au compilateur qu'il s'agit bien de la redéfinition d'une méthode héritée.

Ceci évite de faire une surcharge par inadvertance (le programme ne compile plus), ce qui est une erreur assez commune mais potentiellement longue à détecter. Il est conseillé de toujours utiliser les annotations.

Remarque sur le type de retour pour la surcharge

Il n'est pas possible de surcharger une méthode en ne changeant que son type de retour. Les paramètres doivent être différents, en nombre et/ou en type. Par exemple `public void crier(boolean)` ou `public String crier(float, float)` sont valides, mais pas `public String crier()`.

Remarque sur le type de retour pour la redéfinition

Lors d'une redéfinition, la nouvelle méthode doit avoir le même nom et exactement les mêmes paramètres que la méthode redéfinie. Pour être précis, signalons que, depuis Java 1.5, le type de retour de la redéfinition peut être une *sous-classe* du type de retour de la méthode redéfinie.

Exemple ▼

Remarque sur la visibilité lors d'une redéfinition

La visibilité d'une méthode redéfinie doit être **égale ou moins restrictive** que celle de la méthode héritée :

- La redéfinition d'une méthode `public` ne peut être que `public` ;
- Celle d'une méthode `protected` peut être `protected` ou `public` ;
- Celle d'une méthode sans visibilité (paquetage) peut être `paquetage`, `protected` ou `public` ;
- Enfin une méthode `private` ne peut pas être redéfinie puisqu'elle n'est pas accessible aux filles ! (implicitement, elle est `final`)

Par exemple, vous aurez peut-être rencontré le problème en essayant de redéfinir `String`

toString() (sans visibilité). Elle doit être public !

Construction

Les constructeurs d'une classe servent à initialiser ses attributs lors de la création d'un objet. Avec l'héritage, il faut aussi initialiser les attributs hérités. Ceci se fait en invoquant les constructeurs de la classe mère, avec les paramètres adéquats, au moyen du mot-clé `super()` :

```
class Pangolin {
    private String nom;
    private int nbEcailles;

    public Pangolin(String nom, int nbEcailles) {
        this.nom = nom;
        this.setNbEcailles(nbEcailles);
    }

    [...] // reste de la classe
}

class PangolinALongueQueue extends Pangolin {
    private int longueurQueue;

    public PangolinALongueQueue(String nom, int nbEcailles, int longueurQueue) {
        super(nom, nbEcailles); // appel au constructeur de la mère
        this.longueurQueue = longueurQueue; // initialisation(s) spécifique(s)
    }

    [...] // reste de la classe
}
```

Il faut donc initialiser les attributs hérités de la mère, puis ceux spécifiques à la classe.

- L'appel au constructeur de la mère se fait avec `super(...)`. Si la super classe possède plusieurs constructeurs, le nombre et type des arguments déterminent lequel est invoqué.
- L'appel de `super(...)` doit toujours être la **première instruction** dans le corps du constructeur.
- Si ce n'est pas le cas, alors le compilateur insère implicitement en 1ère ligne l'appel `super()` au constructeur par défaut (sans paramètres) de la mère. Si celui-ci n'existe pas, il y a une erreur.

Il est toujours recommandé d'initialiser les attributs hérités via `super(...)` et jamais directement. Outre la réutilisation du code existant, ceci permet d'assurer l'intégrité des paramètres qui est garantie par la mère.

En fait, le principe d'**encapsulation** doit être respecté, même entre classes d'une même hiérarchie !

Constructeur par défaut

Jusque-là nous avons vu que si une classe ne définit pas explicitement de constructeur, un *constructeur par défaut* est implicitement ajouté, qui ne fait rien. En réalité ce n'est pas tout à fait vrai: il appelle le constructeur par défaut de sa classe mère !

```
class Pangolin {
    // Si AUCUN constructeur n'est défini, le constructeur par défaut est implicitement
    public Pangolin() {
        super();
    }

    [...] // reste de la classe
}
```

Si la classe mère n'a elle-même pas de constructeur par défaut, il y aura une erreur de compilation.

Comme toujours, le constructeur par défaut n'est généré que si aucun autre constructeur n'est défini dans la classe. Dès qu'au moins un constructeur explicite existe, c'est à vous d'écrire le constructeur sans paramètre si vous en avez besoin.

Chaînage ascendant des constructeurs

Lors de la création d'un objet, les constructeurs d'une hiérarchie de classes sont *appelés* du bas vers le haut et *exécutés* du haut vers le bas. Avec la hiérarchie ci-dessous :

- l'ordre des appels est : PangolinALongueQueue, Pangolin puis Animal.
- l'ordre d'exécution (des initialisations spécifiques) est donc : Animal, Pangolin puis PangolinALongueQueue.

```
class Animal {
    private String nom;

    // 3ème APPEL
    public Animal(String nom) {
        // super(); --> IMPLICITE!           // invoque le constructeur de la mère
        this.nom = nom;                     // 1ère EXECUTION
    }
    [...]
}

class Pangolin extends Animal {
    private int nbEcailles;

    // 2ème APPEL
    public Pangolin(String nom, int nbEcailles) {
        super(nom);                         // invoque le constructeur de la mère
        this.nbEcailles = nbEcailles;       // 2ème EXECUTION
    }
    [...]
}

class PangolinALongueQueue extends Pangolin {
    private int longueurQueue;

    // 1er APPEL
    public PangolinALongueQueue(String nom, int nbEcailles, int longueurQueue) {
        super(nom, nbEcailles);             // invoque le constructeur de la mère
        this.longueurQueue = longueurQueue; // 3ème EXECUTION
    }
    [...]
}

[...]

PangolinALongueQueue rantanplan = new PangolinALongueQueue("Rantanplan", 1966, 28);
```

Visibilités

Il est possible de contrôler la *visibilité* des membres d'une classe (attributs et méthodes). Avec l'héritage, un nouveau qualificateur de visibilité apparaît pour contrôler ces droits au sein d'une hiérarchie de classe (à destinations des filles) : le mot clé `protected`.

Ce qui suit fait un point définitif sur les qualificateurs de visibilité en Java :

- `public` : accessible depuis le code de n'importe quel classe ;
- `private` : accessible uniquement à l'intérieur de la classe (mais pas par les filles);
- `protected` : accessible dans la classe et **dans toutes ses sous-classes**. (En JAVA, l'accès est en fait autorisé à toutes les classes du même paquetage. Ce n'est pas le cas général en POO) ;
- Si aucune visibilité n'est spécifiée, l'accessibilité par défaut est : accessible uniquement aux classes du même [paquetage](#). En particulier, la visibilité par défaut interdit l'accès depuis le code d'une classe fille si elle n'est pas définie dans le même paquetage que sa mère !

Dans tous les exemples ci-dessus, vous aurez remarqué que les attributs étaient toujours de visibilité `private`. Ceci veut dire que, si un objet `PangolinALongueQueue` **possède bien** un attribut `nbEcailles`, le code de la classe fille n'y a par contre **pas accès directement** !

```
[...]
```

```
rantanplan.nbEcailles;      // INTERDIT! (private dans la mère)
rantanplan.getNbEcailles() // Autorisé, la méthode est publique
```

Cela peut paraître peu pratique. Mais en fait déclarer `private` les attributs, même dans une classe mère, est en général une bonne pratique.

Il s'agit encore et toujours du principe d'**encapsulation** des données, qu'il convient d'appliquer aux sous-classes comme aux classes extérieures à une hiérarchie. Selon ce principe, les données propres à un objet ne sont accessibles qu'au travers de méthodes. Rappelons deux avantages :

- **Sécurité des données** : passer par des méthodes pour modifier l'état d'un objet permet de garantir l'intégrité de ce nouvel état. En cas de problème, la méthode peut générer une erreur (en Java, typiquement, en levant une exception).
- **Masquer l'implémentation** : la structure de donnée de la classe peut être modifiée sans remettre en cause le code qui utilise cette classe.

La super-classe Object

Vous l'avez compris, une hiérarchie de classes est en fait un arbre dont la racine est une première classe « qui n'a pas de mère ». Dans les exemples précédents, c'était `Pangolin` (ou `Animal`).

En réalité, une classe qui ne définit pas de clause `extends` hérite *implicitement* d'une classe nommée `Object` :

```
class Tatou extends Object { // IMPLICITE quand on écrit juste : class Tatou
    // ça change des Pangolins non?
    [...]
}
```

En conséquence TOUTE classe hérite, directement ou indirectement, de `Object`. De ce fait, les méthodes définies dans la classe `Object` peuvent être invoquées sur tous les objets d'un programme.

Quelques-unes des principales méthodes de la classe `Object` sont :

- `public String toString()` : retourne la représentation d'un objet sous forme de chaîne de caractères ;
- `public boolean equals(Object obj)` : permet de comparer deux objets *sémantiquement* (selon leur état, leurs attributs, et pas uniquement égalité des références) ;
- `public int hashCode()` : retourne un identifiant entier unique associé à un objet. Ceci est surtout utilisé pour l'insertion dans des structures de données de type table de hachage.
- `protected void finalize()` : méthode exécutée par le ramasse-miettes au moment de la destruction d'un objet qui n'est plus référencé. Peut être redéfinie afin de bien libérer certaines ressources externes (comme un fichier) même si le programmeur a oublié de le faire. Attention, `finalize` peut poser des problèmes de performances et causer des bugs, c'est pourquoi elle est dépréciée depuis Java 9. D'autres mécanismes comme les [références fantômes](#) permettent de réagir à la destruction d'un objet.

Reprenons l'exemple de `public String toString()`. Vous l'avez vu, cette méthode peut être invoquée implicitement dans différents contextes. Par exemple `System.out.println(unObjet)` est en fait `System.out.println(unObjet.toString())`.

Ceci n'est possible que du fait que tout objet possède bien cette méthode, puisque qu'elle est définie tout en haut de la hiérarchie ! Par défaut, elle retourne une chaîne composée du nom de la classe (type) de l'objet et son adresse en mémoire, par exemple : `Pangolin@13f0c45f`. Bien entendu, il est possible, et même recommandé, de **redéfinir** la méthode `toString()` dans vos classes, pour avoir un comportement plus pertinent.

Pour finir: final

Le mot clé `final` sur une méthode interdit toute redéfinition dans une classe fille :

```
class Pangolin {  
    [...]  
  
    // Si nous avons écrit ceci, le Pangolin à longue queue ne pourrait  
    // pas être connu pour son cri si caractéristique. Quelle tristesse...  
    public final void crier() {  
        System.out.println("Gwwark Rhââgn Bwwikk"); // Cri du pangolin  
    }  
}
```

Enfin, le même mot-clé `final` sur une classe interdit cette fois l'écriture de toute sous-classe ! (Par exemple `String` est finale.)

```
// Ici il aurait été simplement impossible de définir la classe PangolinALongueQueue !  
// (ce qui aurait été une catastrophe pour ce cours et pour la biodiversité en général)  
final class Pangolin {  
    [...]  
}
```

Attention à l'utilisation de cette notion. Il n'est pas toujours évident d'imaginer les évolutions possibles d'un code, en particulier les sous-classes qui pourraient être nécessaires pour un utilisateur futur.

Pour aller plus loin sur l'héritage :

Une introduction au polymorphisme ▼

Ensimag POO by des profs de POO passés et présents.

licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/) 

Documentation built with [MkDocs](https://mkdocs.org/).

[Fiche 4 : Exceptions en Java](#)[Un exemple introductif](#)[Exceptions : la théorie](#)[Les exceptions en Java](#)[La hiérarchie des exceptions](#)[Exceptions sous contrôle / hors-contrôle](#)[Définir sa propre exception](#)[Lever, propager, rattraper une exception](#)[Lever une exception](#)[Contrainte catch or specify](#)[Résoudre localement une erreur : try / catch](#)[Rattraper plusieurs exceptions à la fois](#)[Gestionnaires et hiérarchie d'exceptions](#)[Clause finally](#)[Déléguer le traitement de l'erreur : throws](#)[Exemple complet](#)

Fiche 4 : Exceptions en Java

L'objectif de cette fiche est de présenter le mécanisme d'**exceptions** en Java. Ce mécanisme permet de traiter de manière fine les erreurs et autres situations anormales en déléguant ce traitement à des gestionnaires d'erreur appropriés.

Un exemple introductif

Ah, le pangolin... Quel animal merveilleux. Malgré son apparence ridicule, il permet, à lui seul, d'illustrer tous les concepts de la programmation orientée objet. Revenons donc à notre classe `Pangolin` définie dans les fiches précédentes, et dont voici un extrait :

```
class Pangolin {
    private String nom;
    private int nbEcailles;

    public Pangolin(String nom, int nbEcailles) {
        this.nom = nom;
        this.setNbEcailles(nbEcailles);
    }

    public void setNbEcailles(int nbEcailles) {
        this.nbEcailles = nbEcailles;
    }

    [...]
}
```

Même sans connaissance zoologique avancée, on peut aisément imaginer que le nombre d'écailles d'un pangolin ne peut être négatif. Pourtant, rien ne l'empêche dans ce code. Voici une première tentative naïve de modification de la méthode `setNbEcailles(int nb)` :

```
public void setNbEcailles(int nbEcailles) {
    if (nbEcailles >= 0) {
        this.nbEcailles = nb;
    } else {
        System.out.println("Pauvre pangolin... Il a un nombre d'écailles négatif...");
        this.nbEcailles = 42; // Un nombre d'écailles par défaut
    }
}
```

C'est un peu mieux : on ne met pas le pangolin dans un état incohérent et on prévient l'utilisateur s'il tente d'initialiser un pangolin avec un nombre d'écailles négatif. Mais ce n'est pas satisfaisant pour autant. D'une part, le traitement local de l'erreur consiste à fixer le nombre d'écailles à un nombre arbitraire sans se soucier de ce que veut vraiment l'utilisateur. D'autre part, on le prévient avec un simple message sur la sortie standard (ce qui lui fera une belle jambe s'il lance son programme sans console, par exemple lors d'une procédure de tests automatisés).

Essayons autre chose :

```
public void setNbEcailles(int nb) {
    if (nbEcailles >= 0) {
        this.nbEcailles = nb;
    } else {
        System.err.println("Tu as initialisé un pangolin avec un nombre négatif d'écaill
            + "Tu ne mérites rien d'autre que d'aller coder en assembleu
            + "en écoutant le dernier album de Francis Lalanne.\n"
            + "Et d'ailleurs, pour bien te signifier mon mépris profond
            + "je quitte le programme avec une erreur.");

        System.exit(1);
    }
}
```

Que penser de cette solution ? D'abord, on peut remarquer qu'il est probable que le développeur de cette méthode ait passé une sale journée, mais c'est un détail. D'autre part, c'est mieux que

précédemment, car cette fois-ci, on signifie bien à l'utilisateur qu'il a fait n'importe quoi lors de l'appel de `setNbEcailles`, mais d'une manière probablement un peu extrême : on quitte le programme sur une erreur, sans sommation, et sans donner l'occasion à celui qui appelle `setNbEcailles` de comprendre ce qui se passe et de résoudre son erreur.

Comment s'en sortir ? Vous l'avez deviné, nous allons utiliser le mécanisme d'**exceptions**. Grosse surprise...



ci-dessus : John Nash était lui aussi [Un Homme d'Exception](https://commons.wikimedia.org/wiki/index.php?curid=1333668), mais il n'a rien à voir, ni avec le Java, ni avec le propos de cette fiche de synthèse. [Source: Elke Wetzig [CC BY-SA 3.0](https://commons.wikimedia.org/wiki/index.php?curid=1333668) <https://commons.wikimedia.org/wiki/index.php?curid=1333668>]

Exceptions : la théorie

Le mécanisme d'exceptions permet de gérer les situations anormales (comme des erreurs) qui peuvent survenir lors de l'exécution d'un programme. Sans ce mécanisme, on traite les cas exceptionnels par une série de conditionnelles, qui rendent le code peu lisible. Formellement :

Exception

Une exception est un événement anormal interrompant le flot d'exécution d'un programme. La gestion d'une exception repose sur trois phases :

- une exception est **levée** quand une erreur survient ;
- l'exception est ensuite **propagée**, c'est-à-dire que l'exécution séquentielle du programme est interrompue et le flot de contrôle transféré aux gestionnaires d'exception ;
- l'exception est **récupérée** par un gestionnaire d'exception, qui la traite. Ensuite, l'exécution reprend avec les instructions qui suivent le gestionnaire d'exception.

Attention

Si une exception n'est récupérée par aucun gestionnaire d'exception, alors le programme s'interrompt sur une erreur, affiche la nature de l'exception ainsi que toute la pile d'appel au moment où l'exception a été levée.

Afin d'illustrer une partie de ces concepts théoriques, considérons par exemple le programme suivant :

```
public class TestException {  
    public static void m1() {  
        System.out.println("Début de m1()");  
        m2();  
        System.out.println("Fin de m1()");  
    }  
  
    public static void m2() {  
        System.out.println("Début de m2()");  
        m3();  
        System.out.println("Fin de m2()");  
    }  
  
    public static void m3() {  
        System.out.println("Début de m3()");  
        int x = 0;  
        int y = 1 / x; // Très mauvais, ça : java.lang.ArithmeticException  
        System.out.println("Fin de m3()");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Début de la méthode principale");  
        m1();  
        System.out.println("Fin de la méthode principale");  
    }  
}
```

Lorsque l'on exécute ce programme, voici ce que l'on obtient :

```
Début de la méthode principale  
Début de m1()  
Début de m2()  
Début de m3()  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at TestException.m3(TestException.java:17)  
    at TestException.m2(TestException.java:10)  
    at TestException.m1(TestException.java:4)  
    at TestException.main(TestException.java:23)
```

Le comportement du programme est clair : un événement anormal (division par zéro) se produit à la ligne 17 du programme (ce qui correspond à la ligne 3 de la méthode `m3()`), provoquant la levée d'une exception de type `java.lang.ArithmeticException`. En conséquence le flot d'exécution normal est interrompu, ce qui explique que la dernière instruction de la méthode `m3()` n'est jamais exécutée. En l'absence de gestionnaire adéquat, l'exception est propagée vers la méthode appelante `m2()`, qui fait de même, et propage à son tour l'exception vers `m1()`, puis vers `main(String[] args)`. Lors de toute cette propagation, aucun gestionnaire adéquat n'ayant été rencontré, le programme s'interrompt en affichant la nature de l'exception, ainsi que la pile d'appels.

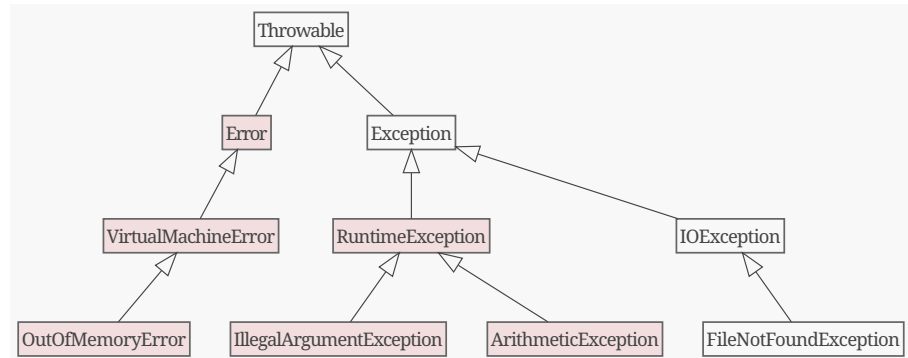
Les exceptions en Java

En Java, la notion d'exception est définie en pratique par la classe `Throwable` (que les anglicistes parmi vous auront traduit approximativement par « qui peut être lancé »). Tout objet étant une instance d'une classe qui hérite de `Throwable` sera considéré, en Java, comme une exception. Ainsi, par exemple, la classe `java.lang.ArithmeticException`, a pour arbre d'héritage : `java.lang.ArithmeticException` → `java.lang.RuntimeException` → `java.lang.Exception` → `java.lang.Throwable` (puis `java.lang.Object`, bien entendu). Allez donc voir sur [sa documentation](#) si vous ne me croyez pas.

La hiérarchie des exceptions

Comme l'exemple de la classe `ArithmeticException` ci-dessus le suggère, il existe toute une hiérarchie d'exceptions dans la bibliothèque standard Java. Cette hiérarchie a pour racine la classe `Throwable`. Cette classe `Throwable` a deux sous-classes : `Error` et `Exception`. La classe `Error` est la classe des exceptions « graves », qui ne devraient pas être récupérées (erreurs système, plus de mémoire... etc.). La classe `Exception` correspond aux exceptions qu'on peut vouloir récupérer. Celle-ci a une sous-classe `RuntimeException`. Les exceptions que vous avez déjà rencontré précédemment (`ArithmeticException`, `IllegalArgumentException`) sont des sous-classes de `RuntimeException`.

Une partie de l'arbre représentant la hiérarchie des exceptions est représentée ci-dessous.



Exceptions sous contrôle / hors-contrôle

À l'instar du [monde](#), les exceptions se divisent en deux catégories :

1. Les exceptions **hors-contrôle** (apparaissant en rouge sur le diagramme UML ci-dessus), qui sont les classes :
 - descendantes de `java.lang.Error` : ce sont typiquement des erreurs critiques qui ne peuvent pas être récupérées (plus de mémoire par exemple) ;
 - descendantes de `java.lang.RuntimeException` : il s'agit ici d'erreurs de programmation qui risquent de provoquer l'hilarité générale de vos collègues développeurs à la machine à café (par exemple la fameuse `NullPointerException`, grande classique des dîners mondains entre programmeurs).
2. Les exceptions **sous contrôle**, qui sont les classes dérivées de `java.lang.Exception`, mais pas de `java.lang.RuntimeException`. Elles correspondent typiquement aux cas d'erreurs que l'on a anticipés, et que l'on veut traiter. Ces exceptions doivent être récupérées et traitées dans le programme.

Gardez en tête ces deux catégories, nous y reviendrons un peu plus loin.

Définir sa propre exception

Même si la bibliothèque standard Java contient déjà un grand nombre d'exceptions, vous pouvez définir vos propres types d'exceptions. C'est même conseillé : ainsi, vous pourrez classer finement les situations anormales auxquelles votre programme pourrait être confronté. Pour définir son propre type d'exception, rien de plus simple : il suffit de définir une classe fille de `Exception` (ou alors `Throwable`, même si l'usage veut que l'on sous-type plutôt `Exception`).

Revenons à notre pangolin par exemple et son nombre d'écaillés négatif. Pour caractériser cette situation anormale, on pourrait tout-à-fait utiliser par exemple `IllegalArgumentException`, mais si l'on veut être plus précis dans la caractérisation de cette anomalie, on peut aussi définir notre propre type d'exception :

```

public class NbEcaillésIncorrectException extends Exception {
    public NbEcaillésIncorrectException(String message) {
        super(message);
    }
}

```

Et c'est tout... Dans le constructeur de la classe ci-dessus, le paramètre `message` correspond simplement au message qui sera affiché si l'erreur est lancée mais non rattrapée.

Lever, propager, rattraper une exception

Lever une exception

Nous avons vu ci-dessus que certaines instructions erronées pouvaient lever des exceptions, comme dans l'exemple de la méthode `m3()` déjà vu :

```
public static void m3() {
    System.out.println("Début de m3()");
    int x = 0;
    int y = 1 / x; // Très mauvais, ça : java.lang.ArithmeticException
    System.out.println("Fin de m3()");
}
```

Dans cet exemple, l'instruction `int y = 1 / x;` provoque une exception qui correspond à une division par zéro : c'est une erreur de programmation (NB : d'ailleurs, il s'agit d'une exception hors-contrôle selon la classification définie [ci-avant](#)).

Mais nous pouvons également lever explicitement nos propres exceptions, à l'aide du mot-clé `throw`. Jugez plutôt :

```
public void setNbEcailles(int nbEcailles) {
    if (nbEcailles < 0) {
        throw new NbEcaillesIncorrectException("Nombre d'écailles négatif !");
    }
    this.nbEcailles = nbEcailles;
}
```

Interlude : vous pouvez observer au passage qu'en Java, les exceptions sont **lancées** (`throw`), alors qu'en Python – par exemple –, les exceptions sont **levées** (`raise`).

Contrainte *catch or specify*

Essayons de compiler notre classe Pangolin augmentée du code précédent qui lance une exception de type `NbEcaillesIncorrectException`. Voici ce que l'on obtient :

```
Pangolin.java:27: error: unreported exception NbEcaillesIncorrectException; must be caught
        throw new NbEcaillesIncorrectException("Nombre d'écailles négatif !");
        ^
1 error
```

Que signifie cette erreur ? Elle vient de la contrainte *catch or specify* de Java :

Contrainte *catch or specify*

En Java, toute exception **sous contrôle** lancée doit être :

- soit rattrapée et traitée localement dans la méthode dans laquelle elle a été lancée ;
- soit déclarée explicitement dans la signature de la méthode comme pouvant être lancée.

Insistons bien sur le fait que cette exigence ne concerne que les exceptions **sous contrôle**. Celles qui sont hors contrôle échappent donc à cette contrainte, ce qui explique le fait que tous les exemples précédents impliquant des exceptions compilaient sans erreur.

La contrainte *catch or specify* illustre simplement le fameux petit principe suivant de survie élémentaire en entreprise. Si vous faites une erreur (oui, tout le monde en fait, même vos enseignants de POO, aussi étonnant que ça puisse paraître), alors vous avez deux attitudes possibles :

- soit vous résolvez vous-même l'erreur, et dans ce cas-là, personne ne vous dira rien ;
- soit vous déléguiez la résolution de l'erreur à la personne qui vous a mandaté pour faire ce job pourri : votre client, votre patron, etc. **Mais** (le gras indique que ce « mais » est important) vous ne pouvez faire ça que si au préalable vous aviez prévenu cette personne que ça pouvait tourner mal. Sinon, vous passez juste pour un incompetent (d'où, encore une fois, hilarité générale à la machine à café).

Nous allons maintenant voir comment implanter ces deux attitudes en Java.

(NB : il existe aussi en Java, comme dans la vraie vie, une attitude qui consiste à dissimuler l'erreur sous le tapis en espérant que personne ne la verra (voyez la note sur le gestionnaire attrape-tout plus loin). Vous serez priés de ne pas le faire.)

Résoudre localement une erreur : *try / catch*

Supposons que nous exécutions du code qui peut échouer en lançant une exception, et que nous

voulions traiter cette exception localement. Alors, il faudra entourer ce code d'un bloc `try`, suivi immédiatement par un bloc `catch` contenant le code permettant de traiter l'exception. Un bloc `catch` doit être paramétré par le type de l'exception que ce bloc doit traiter.

Jugez plutôt :

```
public void setNbEcailles(int nbEcailles) {
    try {
        if (nbEcailles < 0) {
            throw new NbEcaillesIncorrectException("Nombre d'écailles négatif !");
        }
        this.nbEcailles = nbEcailles;
    } catch (NbEcaillesIncorrectException e) {
        // Ici, on va traiter l'exception en question
        System.out.println(e);
        e.printStackTrace();
        System.exit(1);
    }
}
```

Dans l'exemple ci-dessus, si on appelle la méthode avec un paramètre négatif, une exception sera lancée par l'instruction `throw`. La ligne suivante (`this.nbEcailles = nbEcailles;`) ne sera donc pas exécutée, et le flot d'instructions sera donc directement dévié vers le gestionnaire d'exception, c'est-à-dire l'ensemble d'instructions du bloc `catch (NbEcaillesIncorrectException e)`. Ici, il n'y a qu'un seul bloc `catch`, mais on peut en mettre autant que l'on souhaite après un bloc `try`.

Vous me direz, dans ce cas, quel est l'intérêt d'alourdir le code avec des mots-clés supplémentaires, alors que finalement, l'erreur est traitée localement, et qu'un simple `if` aurait suffi ? La question est excellente et je vous remercie de me l'avoir posée. Ici, l'intérêt est double :

1. Cette syntaxe permet de séparer très clairement le code exécuté lors d'un comportement « nominal » du programme, et le code dédié au traitement d'erreur.
2. Cette syntaxe permet de classifier clairement le code dédié à la gestion de chaque type d'erreur. De même, le code dédié à un type particulier d'erreur est factorisé à un seul endroit (alors que cette erreur peut survenir à plusieurs endroits), évitant ainsi la duplication.

Pas clair ? Prenons un autre exemple. Considérons maintenant le constructeur de notre classe `Pangolin`. Pas simple de construire un pangolin. Plein de choses peuvent se passer : le nombre d'écailles peut être incorrect, le nom peut être déjà pris (chaque pangolin a un nom unique au monde), le nombre maximal de pangolins vivants peut être atteint (oui, la population est limitée...).

```

class NbEcaillesIncorrectException extends Exception {
    public NbEcaillesIncorrectException(String message) {
        super(message);
    }
}

class NbMaxPangolinException extends Exception {
    public NbMaxPangolinException() {
        super("Le nombre maximal de pangolins est atteint...");
    }
}

class NommageException extends Exception {
    public NommageException(String message) {
        super(message);
    }
}

public class Pangolin {
    private String nom;
    private int nbEcailles;
    private static java.util.Set<String> noms = new java.util.HashSet<String>();
    private static final int NOMBRE_MAX_PANGOLINS = 42; // Population mondiale de pango

    public Pangolin(String nom, int nbEcailles) {
        try {
            if (Pangolin.noms.size() >= Pangolin.NOMBRE_MAX_PANGOLINS) {
                throw new NbMaxPangolinException();
            }
            if (nbEcailles < 0) {
                throw new NbEcaillesIncorrectException("Nombre d'écailles négatif !");
            }
            if (nbEcailles == 0) {
                throw new NbEcaillesIncorrectException("Le pangolin est nu. Il va avoir");
            }
            if (nbEcailles == 13) {
                throw new NbEcaillesIncorrectException("13 épines sur un pangolin, ça p");
            }
            this.nbEcailles = nbEcailles;
            if (Pangolin.noms.contains(nom)) {
                throw new NommageException("Ce nom est déjà pris...");
            }
            if (nom.equals("Kikoolol38")) {
                throw new NommageException("Quel nom ridicule. Faut pas charrier non pl");
            }
            this.nom = nom;
            Pangolin.noms.add(nom);
            // Ici on peut mettre plein d'autre code à exécuter
            // pour créer un pangolin...
            // ...
        } catch (NommageException e) {
            System.out.println(e); // On ne fait rien d'autre que d'afficher l'exceptio
        } catch (NbEcaillesIncorrectException e) {
            System.out.println(e); // On ne fait rien d'autre que d'afficher l'exceptio
        } catch (NbMaxPangolinException e) {
            e.printStackTrace();
            System.exit(1); // Gros problème : du coup, on quitte sur une erreur.
        }
    }
}

```

Dans l'exemple ci-dessus, nous voyons donc que trois types d'exception peuvent survenir lors de la création d'un pangolin : `NbMaxPangolinException`, `NommageException` et `NbEcaillesIncorrectException`. Chaque type d'exception peut se produire à des endroits différents, pour des raisons différentes, mais le gestionnaire dédié à un type d'exception exécutera toujours la même chose, quelle que soit la cause associée à la levée de l'exception.

Dans un bloc `catch`, c'est le type de l'exception passée en paramètre qui déterminera si le gestionnaire est exécuté ou pas. Si l'exception lancée dans le bloc `try` correspond au type du bloc `catch`, alors c'est le code de ce gestionnaire qui sera exécuté.

Rattraper plusieurs exceptions à la fois

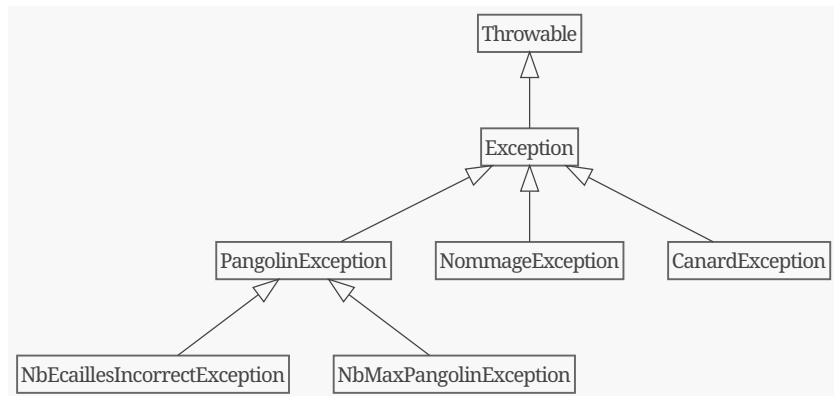
Dans l'exemple ci-dessus, vous avez pu remarquer que le traitement associé à `NommageException` et `NbEcaillesIncorrectException` était le même. Est-il possible de factoriser ce code ? Oui, il est possible de le faire depuis Java 7, en utilisant la syntaxe suivante :

```
catch (NommageException | NbEcaillesIncorrectException e) {
    e.printStackTrace(); // On ne fait rien d'autre que d'afficher la pile d'appel
}
```

Gestionnaires et hiérarchie d'exceptions

Je vous ai dit plus haut que si l'exception lancée dans le bloc `try` correspondait au type du bloc `catch`, alors c'est le code de ce gestionnaire qui serait exécuté. En fait, c'est légèrement plus compliqué que cela : pour qu'un bloc `catch` soit exécuté, il faut que l'exception lancée dans le bloc `try` ait un type **compatible** avec le type spécifié dans la clause `catch`. Ici, « compatible » signifie « être du même type ou n'importe quel sous-type ». Prenons un exemple.

Considérons la hiérarchie d'exceptions ci-dessous :



Le code suivant illustre différents cas de figure d'exceptions pouvant être rattrapées par différents gestionnaires.

```
try {
    // Ici du code qui peut lancer plusieurs types d'exception...
} catch (NommageException e) {
    // Ici, tout ce qui est de type NommageException sera traité
} catch (NbMaxPangolinException e) {
    // Ici, tout ce qui est de type NbMaxPangolinException sera traité
} catch (PangolinException e) {
    // Ici, tout ce qui est de type PangolinException sera traité
    // donc en particulier NbEcaillesIncorrectException, mais pas
    // NbMaxPangolinException, qui a déjà été traité plus haut.
} catch (Exception e) {
    // Le gestionnaire attrape-tout. Toutes les exceptions
    // qui n'ont pas été rattrapées plus haut (même les exceptions
    // hors-contrôle d'un sous-type de RuntimeException) seront
    // traitées ici
}
```

Note importante

Une exception ne peut être rattrapée que par un seul gestionnaire d'exception. Techniquement, ce sera le premier gestionnaire ayant un type compatible avec le type de l'exception. Attention donc à l'ordre de déclaration des gestionnaires. Dans l'exemple ci-dessus, si le gestionnaire `catch (PangolinException e)` avait été placé avant `catch (NbMaxPangolinException e)`, il aurait rattrapé également toutes les exceptions de type `NbMaxPangolinException`, rendant caduque la clause `catch (NbMaxPangolinException e)`. Dans les faits, le compilateur vous le signalera et refusera de compiler.

Clause *finally*

Il est parfois utile de spécifier du code qui soit exécuté quoi qu'il se passe : qu'il y ait une exception levée ou pas, et qu'elle soit traitée localement ou propagée à la méthode appelante. Ce peut être par exemple le cas si vous avez besoin de libérer une ressource (un fichier par exemple) avant de traiter l'erreur ou de quitter le programme sur un message d'insulte.

C'est le rôle de la clause `finally`, toujours placée après le dernier bloc `catch` :

```
try {
    // Ici du code qui peut lancer plusieurs types d'exception...
} catch (...) {
    ...
} catch (...) {
    ...
} finally {
    // Ici du code qui sera exécuté dans tous les cas
}
```

Déléguer le traitement de l'erreur : *throws*

Jusqu'ici nous avons vu comment traiter localement une erreur. Parfois, ce n'est pas possible ou pas souhaitable. Dans l'exemple précédent, après tout, qui est responsable si un pangolin est initialisé avec un nombre négatif d'écaïlles ? Il peut sembler naturel de dire que c'est de la responsabilité de la méthode appelante. Dans ce cas, il faut que l'exception se **propage** dans la pile d'appels de méthode, afin de déléguer le traitement de cette erreur à la méthode appelante. En gros, vous dites à la méthode appelante : « écoute ma vieille [pardonnez cette familiarité simplement destinée à vous tenir en haleine], tu as fait n'importe quoi avec ce paramètre, tu assumes. ». Mais comme nous l'avons dit ci-avant, vous ne pouvez dire ça que si vous avez prévenu avant. Prévenir, c'est simplement dire que la méthode peut lever une exception de tel type (et préciser dans la documentation dans quels cas cela arrive). Cela se fait avec le mot-clef `throws`.

Reprenons notre classe `Pangolin` :

```
public class Pangolin {
    private String nom;
    private int nbEcaïlles;
    private static java.util.Set<String> noms = new java.util.HashSet<String>();
    private static final int NOMBRE_MAX_PANGOLINS = 42; // Population mondiale de pango

    public Pangolin(String nom, int nbEcaïlles) throws NbEcaïllesIncorrectException, No
        try {
            // Le reste du code ne change pas
        } catch (NbMaxPangolinException e) {
            e.printStackTrace();
            System.exit(1); // Gros problème : du coup, on quitte sur une erreur.
        }
    }
}
```

Dans cette nouvelle version, si le code à l'intérieur du bloc `try` peut encore lever des exceptions de type `NbEcaïllesIncorrectException` et `NommageException`, ces exceptions ne seront récupérées par aucun gestionnaire d'exception, contrairement à la version précédente (NB : le fait que ce code soit à l'intérieur d'un bloc `try` ne change rien ici). Pour que le code compile, il est donc indispensable de déclarer ces deux types d'exception dans la signature de la méthode avec `throws`.

La responsabilité du traitement de ces erreurs sera donc transférée à la méthode appelante. Par exemple :

```
public static void main(String[] args) throws NbEcaïllesIncorrectException, NommageExce
    new Pangolin("Gérard", -5);
}
```

Ici, l'utilisateur a choisi de ne pas traiter localement les erreurs, donc, vu que la contrainte *catch or specify* s'applique ici, les deux exceptions doivent être déclarées dans la signature de la méthode. Les plus alertes d'entre vous ont dû voir qu'il s'agissait de la méthode `main` : ce qui se passera en pratique dans ce cas sera donc que le programme quittera sur une erreur, en affichant l'exception et sa pile

d'appel.

Attention au gestionnaire attrape-tout

Vous l'avez compris, la contrainte *catch or specify* vous oblige à traiter localement toute exception sous contrôle pouvant être lancée, ou à la déclarer dans la signature (ce qui vous obligera à la traiter dans la méthode appelante). Si vous avez du code qui peut lancer plusieurs exceptions différentes, c'est pénible et il peut être tentant de « museler » ces exceptions en utilisant un bloc `catch` attrape-tout :

```
try {  
    // Ici du code qui peut lancer plein d'exceptions  
} catch (Exception e) {}
```

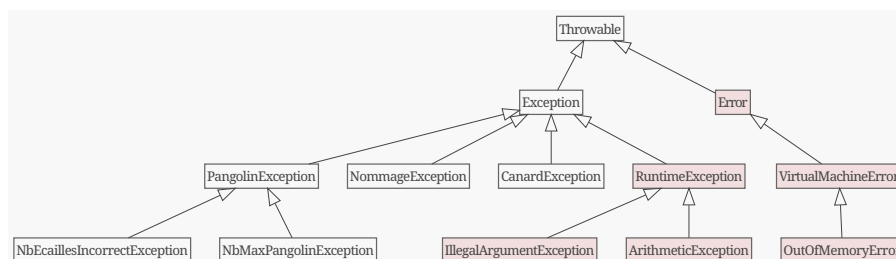
Bel exemple de stratégie consistant à dissimuler les erreurs sous le tapis. Dans le meilleur des cas il ne se passe rien et tant mieux. Mais s'il y a une erreur, le code n'est pas exécuté, l'erreur n'est pas affichée (car le gestionnaire d'exception la rattrape et ne fait rien) et provoquera probablement des problèmes à d'autres endroits du code qui n'ont rien à voir avec cette méthode. Vous pouvez passer des jours à comprendre ce qui se passe. Avec un peu de chance, cela concerne en plus un code que vous avez écrit il y a six mois et vous avez oublié cette petite entorse à toutes les règles de déontologie informatique.

Moralité : ne le faites pas. Le dernier stagiaire qui a tenté a fini pendu par les pieds au vidéoprojecteur de la salle de réunion.

Exemple complet

Pour conclure cette fiche d'exception, voici un petit exercice.

Considérons la hiérarchie d'exceptions suivante :



Considérons maintenant le code suivant :

```

public class TestException {
    private static final boolean PANGOLIN_EXCEPTION = false;
    private static final boolean NOMMAGE_EXCEPTION = false;
    private static final boolean CANARD_EXCEPTION = false;
    private static final boolean NBECAILLESINCORRECT_EXCEPTION = false;
    private static final boolean NBMAXPANGOLIN_EXCEPTION = false;
    private static final boolean ARITHMETIC_EXCEPTION = false;
    private static final boolean ILLEGAL_ARGUMENT_EXCEPTION = false;

    public static void m1() throws CanardException {
        try {
            System.out.println("Début de m1()");
            m2();
            System.out.println("Fin de m1()");
        } catch (PangolinException e) {
            System.out.println("Exception Pangolin (de m1()).");
        }
    }

    public static void m2() throws PangolinException, CanardException {
        try {
            System.out.println("Début de m2()");
            m3();
            System.out.println("Fin de m2()");
        } catch (NbMaxPangolinException e) {
            System.out.println("Exception nombre max pangolin (de m2()).");
            throw new IllegalArgumentException();
        } finally {
            System.out.println("Finally de m2()");
        }
        System.out.println("Fin de m2() après le try/catch/finally");
    }

    public static void m3() throws PangolinException, CanardException {
        try {
            System.out.println("Début de m3()");
            if (PANGOLIN_EXCEPTION)
                throw new PangolinException();
            if (NOMMAGE_EXCEPTION)
                throw new NommageException();
            if (CANARD_EXCEPTION)
                throw new CanardException();
            if (NBECAILLESINCORRECT_EXCEPTION)
                throw new NbEcaillesIncorrectException();
            if (NBMAXPANGOLIN_EXCEPTION)
                throw new NbMaxPangolinException();
            if (ARITHMETIC_EXCEPTION)
                throw new ArithmeticException();
            if (ILLEGAL_ARGUMENT_EXCEPTION)
                throw new IllegalArgumentException();
            System.out.println("Fin de m3()");
        } catch (NommageException e) {
            System.out.println("Exception Nommage (de m3()).");
        }
    }

    public static void main(String[] args) throws CanardException {
        System.out.println("Début de la méthode principale");
        m1();
        System.out.println("Fin de la méthode principale");
    }
}

```

Nous allons maintenant essayer de déterminer ce qui se passe selon les valeurs des attributs booléens déclenchant ou non l'envoi des exceptions.

Question 1 : Qu'affiche le programme tel quel ?

Cliquez pour la réponse ▼

Question 2 : Qu'affiche le programme si `NOMMAGE_EXCEPTION` passe à `true` ?

Fiche 5 : Polymorphisme et liaison dynamique

Polymorphisme ?

[Polymorphisme de méthodes](#)

[Polymorphisme d'objets](#)

[Type statique, type dynamique](#)

[Interroger le type dynamique durant l'exécution du programme](#)

[Conversion \(transtypage, coercion, cast\) de type statique entre classes](#)

[Mécanisme d'appel de méthode en Java. Liaison dynamique](#)

[Traitement à la compilation \(partie statique\)](#)

[Traitement à l'exécution \(« liaison dynamique »\)](#)

[Exemples d'appel de méthodes](#)

[Exemple : le cas de la méthode equals](#)

Fiche 5 : Polymorphisme et liaison dynamique

Cette fiche fait le point sur le concept de polymorphisme en POO/Java et sur le mécanisme de résolution d'appel de méthodes (partie statique et « liaison dynamique ») en Java.

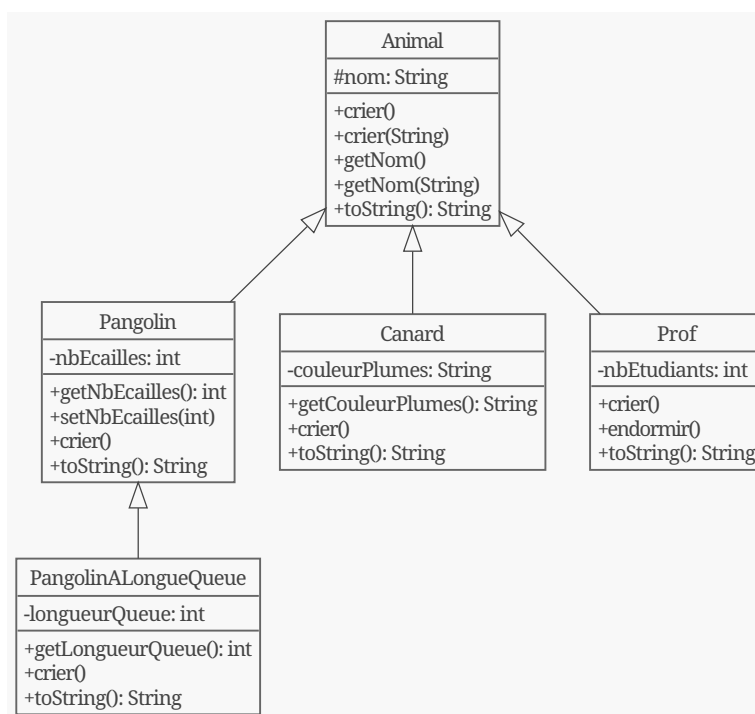
Polymorphisme ?

Une notion essentielle

Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage.

Bruce Eckel, *Thinking in JAVA*.

Pour introduire le polymorphisme (« qui peut prendre plusieurs formes »), considérons l'exemple (désormais célèbre ?) du diagramme UML de la figure suivante :



Voici un code possible pour certaines des classes de notre hiérarchie :

```
public class Animal {
    private String nom;

    public Animal(String nom) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public void crier() {
        System.out.println("Je ne sais pas comment crier !");
    }

    public void crier(String raison) {
        System.out.print("Je vais crier car " + raison);
        crier();
    }

    public String toString() {
        return "Mon nom est " + nom + ". Je suis un Animal ";
    }
}
```

```
public class Prof extends Animal {
    private int nbEtudiants;

    public Prof(String nom, int nbEtudiants) {
        super(nom);
        setNbEtudiants(nbEtudiants);
    }

    public void setNbEtudiants(int nbEtudiants) {
        if(nbEtudiants < 0) {
            throw new IllegalArgumentException("nbEtudiants négatif !");
        }
    }

    @Override
    public void crier() {
        System.out.println("Grrrrr !");
    }

    @Override
    public String toString() {
        return super.toString() + "de type Prof. " + " J'ai " + nbEtudiants + " étudiant
    }
}
```

```
public class Pangolin extends Animal {
    ...
    @Override
    public void crier() {
        System.out.println("Gwark Rhââgn Bwwikk ");
    }

    @Override
    public String toString() {
        return super.toString() + "de type Pangolin avec " + getNbEcailles() + " écaill
    }
    ...
}
```



```

public class PangolinALongueQueuee extends Pangolin {
    private int longueurQueuee;

    public PangolinALongueQueuee(String nom, int nbEcailles, int longueurQueuee) {
        super(nom, nbEcailles);
        this.longueurQueuee = longueurQueuee;
    }

    @Override
    public crier() {
        System.out.print("Qeyyyouuuu "); // partie spécifique
        super.crier(); // partie commune (réutilisation)
    }

    @Override
    public String toString() {
        return super.toString()
            + "\nMais je suis aussi un Pangolin spécial : en plus, j'ai une queue de "
            + longueurQueuee + " cm !";
    }
}

```

Polymorphisme de méthodes

Cet exemple comprend une première catégorie de polymorphisme, que vous connaissez déjà en fait : le *polymorphisme de méthodes*, lié à la surcharge et la redéfinition :

- **Surcharge** : la méthode `void crier(...)` est surchargée dans la classe `Animal`. Il en existe donc deux versions (deux « formes ») avec des signatures différentes dans cette classe (et donc aussi les classes qui héritent de `Animal`).
- **Redéfinition** : la méthode `void crier()` (sans paramètre) est redéfinie dans la plupart des sous-classes, pour tenir compte du fait que chaque animal a un cri spécifique. Elle existe donc en plusieurs versions (plusieurs « formes », mais avec cette fois-ci la même signature) réparties dans la hiérarchie de classe.

Polymorphisme d'objets

Cet exemple permet de plus d'introduire une deuxième catégorie de polymorphisme, parfois appelé *polymorphisme d'objets*.

Rappel : la relation d'héritage entre les classes `Animal` et, par exemple, `Prof` traduit en Java le fait que, « dans la vraie vie », un prof est un type particulier d'animal (si, si...).

Poussons un peu plus loin. Puisque, grâce à l'héritage, un objet instance de la classe `Prof` « est » un type particulier d'`Animal`, ne serait-il pas possible en Java de manipuler un objet `Prof` « en tant que » `Animal` ? Eh bien oui, tout simplement au moyen d'une référence de type `Animal`. Voyons ce que cela donne :

```

public class Test {
    public static void main(String args[]) {
        Prof p = new Prof("Sidonie", 1542);
        // La classe Prof dérive de la classe Animal.
        // => Un objet Prof "est" aussi un Animal
        // => On peut manipuler un objet de type Prof "en tant que" Animal !
        Animal unProfEstUnAnimal = p; // et voilà !

        // Notez qu'on pourrait même écrire directement sans problème :
        // Animal unProfEstUnAnimal = new Prof("Sidonie", 1542);

        // Désormais, unProfEstUnAnimal est une référence de type Animal
        // qui manipule (réfère) un objet dont le "vrai" type est Prof !

        // Faisons crier notre Animal... qui est un Prof... donc fait crier le prof. A
        unProfEstUnAnimal.crier(); // affiche "Grrrr"

        // affiche l'état du Prof :
        // "Mon nom est Sidonie. Je suis un Animal de type Prof. J'ai 1542 étudiants"
        System.out.println(unProfEstUnAnimal.toString());
    }
}

```

De cet exemple, on retiendra que :

Avec le polymorphisme, on peut toujours manipuler une instance d'une classe fille au moyen d'une référence du type de n'importe quelle super classe.

Plus fort, le polymorphisme permet par exemple de stocker (... manipuler) ensemble plusieurs objets de types différents si leurs classes héritent toutes d'une même classe mère, au moyen par exemple d'un tableau du type de la classe mère.

Il permet encore d'écrire des traitements (des méthodes) génériques, qui s'appliqueront indifféremment à des objets quelque soit leur type réel, pour peu que leurs classes héritent toutes de la même classe mère.

Voyons tout cela par l'exemple :

```

public class Test {
    public static void main(String args[]) {
        // On peut stocker dans un tableau
        // un mélange de Prof et de Canard... Et même des pangolins à longue queue
        Animal tab[] = new Animal[10];
        tab[0] = new Prof("Pierre", 12);
        tab[1] = new Canard("Paul", 122);
        tab[2] = new PangolinALongueQueue("Jacques", 765, 42);
        ...

        system.out.println(tabFig[2]);
        // Affiche :
        // Je m'appelle Jacques. Je suis un Animal de type Pangolin avec 765 écailles.
        // Mais je suis aussi un pangolin spécial : en plus, j'ai une queue de 42 cm !"

        // On peut appliquer un traitement générique à tous les animaux
        // du tableau. Peu importe qu'ils soient des Pangolin ou des Diplodocus !
        ajouterANom(tabFig[0], "Kiroule");
        ajouterANom(tabFig[1], "Yesterre");
        ajouterANom(tabFig[2], "Kadi");

        system.out.println(tabFig[2].getNom());
        // Affiche :
        // Jacques Kadi.

        faireCrierUneMenagerie(tabFig, "Oula, que de bruit ! : ");
        // Affiche :
        // "Oula, que de bruit !
        // Grrr
        // Coin coin
        // Qeyyyoouuuu Gwwark Rhââgn Bwwikk"
    }

    // On peut écrire des traitements génériques
    // qui s'appliqueront à des objets de types divers !
    public static void ajouterANom(Animal a, String complement) {
        String nomComplet = a.getNom() + complement;
        a.setNom(nomComplet);
    }

    system.out.println(tabFig[2]);
    // Affiche :
    // « Je m'appelle Jacques Kadi. Je suis un Animal de type Pangolin avec 765 écaille
    // Mais je suis aussi un pangolin spécial : en plus, j'ai une queue de 42 cm ! »;

    // On peut même écrire des traitements génériques
    // sur des conteneurs (ici, un tableau)
    // qui contiennent des références sur
    // des objets de types différents !
    public static void faireCrierUneMenagerie(Animal tab[], String commentaire) {
        System.out.println(commentaire);
        for(Animal a: tab) {
            a.crier();
        }
        // Notez que, suivant le type de l'objet référencé par "a"
        // (Prof, Pangolin, PangolinALongueQueue ou Canard)
        // ce ne sera en fait pas la même méthode crier()
        // qui sera in fine exécutée : chaque animal criera
        // à sa manière, précisée dans chaque redéfinition de la méthode crier() !
    }
}

```

De cet exemple, on retiendra le principe fondamental suivant :

Principe de substitution

Avec le polymorphisme, une instance d'une classe fille peut être utilisée partout où une référence du type de sa classe parent est attendue.

Type statique, type dynamique

Le polymorphisme d'objet conduit à distinguer en POO deux notions de type pour chaque objet :

- le **type dynamique** de l'objet, c'est à dire le type effectif de l'objet en mémoire. Il est déterminé une fois pour toutes au moment de la création de l'objet avec `new`.
- le **type statique** de la référence qui, à un endroit du code, est utilisé pour manipuler cet objet. Le type statique peut varier au cours du programme, en fonction du type des références qui, au fur et à mesure, sont utilisées pour manipuler l'objet.

Dans le premier exemple ci-dessus, il n'existait qu'un seul objet en mémoire, de type (dynamique) `Prof`. Mais cet unique objet est ensuite manipulé au moyen de deux références (types statiques) : la référence `p` de type `Prof`, puis la référence `unProfEstUnAnimal` de type `Animal`.

Avec les notions de type statique et de type dynamique, on peut maintenant donner une définition plus précise du polymorphisme d'objets.

Polymorphisme d'objets ▼

Interroger le type dynamique durant l'exécution du programme

Le compilateur Java connaît toujours les types statiques (les types des références) utilisés dans le code. Par contre, il n'a pas connaissance du type dynamique de l'objet référencé.

Plusieurs mécanismes existent toutefois pour consulter le type dynamique d'un objet durant l'exécution du programme. Voici, par l'exemple, les plus courants :

```
public class Test {
    public void exempleDeMecanismesTestantLeTypeDynamique(Object o) {
        // o est une référence (type statique Object)
        // sur un objet de type dynamique quelconque...
        // L'objet qui est "au bout" de la référence o
        // peut être un Animal, un Pangolin (avec ou sans longue queue)
        // une instance de la classe AvionEnPapier,
        // une instance d'Object ou tout autre chose...
        // Allez savoir...

        // Il est possible de récupérer une chaîne de caractères contenant
        // "en toutes lettres" le type dynamique d'un objet :
        System.out.println(o.getClass().getName() );

        // Test sur le type dynamique de l'objet :
        if (o.getClass() == Pangolin.class) {
            // VRAI si et seulement si le type dynamique
            // de o est Pangolin
            ...
        }

        // Vérification de la cohérence entre type avec l'opérateur instanceof
        if (o instanceof Animal) {
            // VRAI si et seulement si
            // le type dynamique de o est Animal
            // OU n'importe quel sous classe de Animal

            // Donc vrai par exemple si o réfère un objet de type
            // dynamique Prof
            ...
        }
    }
}
```

Attention toutefois à l'usage de ces mécanismes d'interrogation du type dynamique ! Dans de rares cas, comme celui de la méthode `boolean equals(Object o)` discuté plus loin dans cette fiche, ils sont indispensables. Il faut donc les connaître. Par contre, hormis ces cas, **leur usage est souvent le signe d'une mauvaise conception objet.**

Un bon développeur POO n'a (presque...) jamais besoin d'interroger le type dynamique de l'objet qui se cache « au bout » de la référence qu'on utilise pour le manipuler !

Conversion (transtypage, coercion, cast) de type statique entre classes

Transtyper (convertir, *caster*) une référence vers un type plus générique est toujours possible. Cela s'appelle un *upcast* en bon français, ou coercion ascendante. On peut toujours écrire, par exemple :

```
Pangolin p = new PangolinALongueQueue("Tartiflette", 12, 134);
Animal    a = p; // ou upcast explicite: Animal a = (Animal) p
Object    o = p ; // ca marche aussi, car toute classe dérive de Object en Java !

System.out.println(o.toString()) ;
// et il s'affiche toutes les propriétés du Pangolin, y compris la longueur de son appe
```

A l'inverse, transtyper une référence vers un sous type (ce qu'on appelle un *downcast* en mauvais anglais, ou coercion descendante) appelle quelques précautions :

- A la compilation, cela nécessite un *cast* explicite dans le code, sans quoi le compilateur provoque une erreur.
- A l'exécution, cela nécessite que le type statique cible soit compatible avec le type dynamique de l'objet. Autrement dit, il faut que le type statique cible soit plus générique que le type dynamique de l'objet. Si ce n'est pas le cas, la machine virtuelle lève une exception de type `ClassCastException`.

Tout ceci est expliqué dans les commentaires de l'exemple suivant :

```

public class Test {
    public static void main(String args[]) {
        Animal uneBebete;

        switch (new Random().nextInt(2)) { // 0 ou 1, aléatoirement
            case 0:
                // un canard à 2 plumes, pauvre bête...
                uneBebete = new Canard("Coin coin", 2);
                break;
            case 1:
                // un prof sans élève... You know what ? Is he Happy ??
                uneBebete = new Prof("Droopy", 0);
                break;
        }
        // Désormais, uneBebete réfère aléatoirement un Canard ou un Prof

        // Affichons l'état notre animal (que ce soit un Canard ou un Prof)
        System.out.println(uneBebete);

        // La ligne suivante est refusée à la COMPILATION car
        // Potiron n'est (sans doute... ???) ni une super classe,
        // ni une sous classe de Animal, mais une classe
        // dans une hiérarchie de classe tout à fait différente...
        Potiron uneCucurbitacee = (Potiron) uneBebete;

        // La ligne suivante est refusée à la COMPILATION.
        // Canard est une sous classe de Animal.
        // Il s'agit donc d'un cast vers un sous type ("downcast").
        Canard c = uneBebete;
        // Eh oui. Pour un downcast, Java impose qu'on écrive
        // un cast explicite dans la code.

        // Voilà comment on fait.
        // La ligne suivante est acceptée à la COMPILATION :
        Canard c = (Canard) uneBebete;

        // Lors de l'EXECUTION de la ligne précédente,
        // si jamais uneBebete réfère un objet de type dynamique Prof,
        // alors il vaudrait mieux que Java le détecte...
        // Eh oui, un prof est certes un animal, mais tout de même pas un canard !

        // Fort heureusement, la machine virtuelle va s'assurer que ce cast est possible
        // De deux choses l'une :
        // - Si le type dynamique de uneBebete est Canard (ou une sous classe de Canard)
        //   alors tout se passe bien et l'exécution continue.
        // - Si le type dynamique de uneBebete est Prof (ou Pangolin, ou...),
        //   alors la machine virtuelle lève une exception de type ClassCastException.
        //   Et comme dans notre cas cette exception n'est récupérée nulle part,
        //   le programme va s'arrêter immédiatement...

        c.setNbPlume(222); // voilà notre canard (si c'en est bien un...) bien remplum

        // Avec instanceof, on peut s'assurer de la validité du downcast avant de l'exécuter
        if(anim instanceof Canard) {
            // avec le test qui précède, le cast qui suit ne lèvera jamais d'exception
            Canard c = (Canard) anim;
            ...
        }
    }
}

```

Attention toutefois aux usages du *downcast*. Il faut savoir le faire mais il est souvent le signe d'une mauvaise conception objet.

Un bon développeur POO ne devrait (presque...) jamais avoir besoin de transtyper une référence vers un sous-type (*downcast*).

Mécanisme d'appel de méthode en Java. Liaison dynamique

Jusqu'ici, à chaque fois qu'on a fait crier un animal quelconque (par exemple un `PangolinALongueQueue`), même si c'est avec une référence de type `Animal`, la pauvre bestiole a toujours crié comme il faut (par exemple comme tous ses amis à longue queue) :

```
Animal a = new PangolinALongueQueue("Kirikou", 12, 120);
a.crier(); // affiche le cri des Pangolins à longue queue, même si a est de type Animal

System.out.println(a.toString());
// affiche :
// "Je m'appelle Kirikou. Je suis un Animal de type Pangolin avec 12 écailles.
// Mais je suis aussi un Pangolin spécial : en plus, j'ai une queue de 120 cm !"
```

C'est un peu magique tout ça, non ? Faisons le point...

Supposons qu'on exécute une méthode au moyen d'une référence de type donné (type statique) qui réfère un objet de type dynamique donné. Cette méthode peut avoir été *surchargée*, dans la classe du type dynamique de l'objet, ou dans une de ses super-classes. Elle peut aussi avoir été *redéfinie* à différents niveaux de la hiérarchie. **Comment Java sélectionne-t-il la méthode qui sera finalement exécutée ?**

Une manière pas trop bête de se représenter intuitivement ces mécanismes est la suivante.

Le choix de la méthode dépend du type statique et du type dynamique de l'objet, ainsi que des types statiques des paramètres. Il est fait en deux temps :

- **À la compilation, le compilateur détermine la signature précise de la méthode qui sera exécutée**, parmi toutes les surcharges. Il ne connaît pour cela que les types **statiques** de l'objet sur lequel la méthode est appelée et des paramètres passés à la méthode. Il dresse la liste des signatures de toutes les méthodes connues dans la classe du type **statique** (y compris les éventuelles méthodes héritées), puis il choisit celle dont la signature est « la plus adaptée » (on dit la **plus spécifique**) aux types statiques des paramètres effectifs utilisés lors de l'appel. On résume parfois cette étape en disant que **la surcharge est résolue à la compilation** en fonction des types statiques.
- **À l'exécution la machine virtuelle exécute la méthode de l'objet qui a la bonne signature**. Parmi toutes les redéfinitions situées au dessus du type dynamique de l'objet, ce sera celle la plus proche du type dynamique de l'objet. Ce mécanisme s'appelle **la liaison dynamique**. Pour l'intuiter, rappelons que, en POO, l'exécution d'une méthode peut-être vue comme l'« envoi d'un message » à un objet : c'est *l'objet* lui-même qui est chargé d'exécuter *sa* méthode, connaissant sa signature. Si la méthode a été redéfinie dans la hiérarchie de classe, la version de la méthode que possède l'objet est celle qui est la plus proche de la classe de notre objet, en remontant la hiérarchie des classes. C'est cette méthode qui sera exécutée. On résume parfois cela en disant que **la redéfinition est résolue à l'exécution** en fonction du type dynamique de l'objet sur lequel on invoque la méthode.

Dans l'exemple qui précède, lorsqu'on a exécuté `a.crier()` :

- le compilateur a sélectionné la signature `void crier()` sans paramètre ; il a éliminé la surcharge `void crier(String raison)`.
- la machine virtuelle a exécuté la méthode `void crier()` de l'objet. L'objet est de type `PangolinALongueQueue`. Comme cette classe a redéfini la méthode `void crier()`, c'est la version de cette classe qui était, *in fine* exécutée.

Pour ceux d'entre vous qui veulent en savoir plus, voici une explication détaillée du mécanisme d'appel de méthode en Java.

En savoir plus ▼

Exemples d'appel de méthodes

L'exemple suivant démontre quelques appels de méthodes avec vos animaux préférés.

```

public class Test {    //1
    public static void main(String args[]) {    //2
        Animal tabFig[] = new Animal[2]; //3
        tabFig[0] = new PangolinALongueQueue("Kirikou", 124, 12); //4
        tabFig[1] = new Prof("Esseur", 10); //5

        tabFig[0].setNbEcailles(1000); //6
        tabFig[0].setNom("Hubert"); //7
        System.out.println( tabFig[0].toString() ); //8

        tabFig[0].crier(); //10
        tabFig[1].crier("Les élèves dorment !"); //11
    } //12
} //13

```

Voici quelques explications :

- **Ligne 6. Le compilateur génère une erreur de compilation.** L'objet `tabFig[0]`, de type dynamique `PangolinALongueQueue`, possède bien une méthode `void setNbEcailles(int nb)` (cette méthode est héritée de `Pangolin`)... Mais le compilateur ne voit pas cette méthode ! En effet :

- le type statique de `tabFig[0]` est `Animal`
- la méthode `void setNbEcailles(int)` n'est pas définie au niveau de la classe `Animal`
- le compilateur ne trouve donc pas de méthode correspondant à l'appel.

Cela peut sembler ennuyeux, mais c'est en fait un gage de sécurité : le compilateur s'assure que l'appel de méthode pourra bien être exécutée par la machine virtuelle. On retiendra que :

Pour pouvoir appeler une méthode sur un objet, il faut que cette méthode soit déclarée dans la classe du type statique de la référence utilisée pour le manipuler.

- **Ligne 7. La méthode exécutée est la méthode `void setNom(String)` définie dans la classe `Animal`.** En effet :
 - À la compilation, la signature de la méthode la « plus spécifique » trouvée dans la classe `Animal` est `void setNom(String)`.
 - Comme cette méthode n'est pas redéfinie dans les sous-classe, c'est elle qui est exécutée.

L'attribut `nom` de notre pangolin, hérité dans la classe `PangolinALongueQueue`, est donc bien modifié. Bonjour Hubert ! - **Ligne 8. La méthode exécutée est la méthode `String toString()` définie dans `PangolinALongueQueue`.** En effet : - A la compilation, la signature de la méthode la plus spécifique trouvée dans `Animal`, est `String toString()`. - A l'exécution, c'est la version de la sous-classe `PangolinALongueQueue` qui est exécutée, puisque la méthode est redéfinie dans cette classe.

Rappel en passant : notez que le code de cette méthode réalise un appel explicite à la version de la super-classe. - **Ligne 10. La méthode exécutée est la méthode `void crier()` définie dans la classe `PangolinALongueQueue`** (mêmes raisons que précédemment). - **Ligne 11. La méthode exécutée est la méthode `void crier(String raison)` de la classe `Animal`.** En effet : - A la compilation, la signature de la méthode la plus spécifique trouvée dans `Animal`, est `void crier(String raison)`. - Cette méthode n'est pas redéfinie dans les sous-classes. C'est donc cette méthode que possède l'objet `Prof`, et c'est elle qui est exécutée.

Notez que cette méthode `String crier(String raison)` fait appel à `String crier()` (sans paramètre). Pour cet appel : - A la compilation, le type de `this` (type statique) est `Animal`, puisqu'on est dans cette classe. La classe `Animal` possède bien une méthode `void crier()` sans paramètre. Tout va bien... - A l'exécution, le type dynamique de l'objet étant `Prof`, c'est la version de `void crier()` définie dans la classe `Prof` qui est exécutée.

In fine, nous obtiendrons quelque chose comme : "Les élèves dorment ! Du coup je crie : grrrrrrrr".

Exemple : le cas de la méthode equals

Votre professeur préféré vous a sans doute déjà indiqué que, en Java, lorsqu'on veut pouvoir tester l'égalité sémantique entre deux objets (l'égalité de l'état de ces objets) il faut écrire dans la classe une

méthode de signature `public boolean equals(Object o)`.

Il faut que cette méthode prenne *toujours* bien un `Object` en paramètre, et pas autre chose. En effet, il s'agit de **redéfinir la méthode `public boolean equals(Object o)` de la classe `Object`** – et non pas de *surcharger* cette méthode. Il faut donc respecter la signature de la méthode d'origine.

Mais pourquoi cela ? On peut le comprendre maintenant que l'on connaît mieux le mécanisme d'appel de méthodes et la liaison dynamique.

Considérons deux versions d'une classe `Position`, la première avec une erreur dans la méthode `equals`, la seconde sans erreur.

```
class PositionErreur {
    public int x;
    public int y;
    ...

    // Ceci est une SURCHARGE de la méthode
    //     boolean equals(Object o)
    // héritée de la classe Object
    public boolean equals(PositionErreur other) {
        return other.x == this.x && other.y == this.y;
    }
}
```

```
class PositionOK {
    public int x;
    public int y;
    ...
    // Ceci est une REDEFINITION de la méthode
    //     boolean equals(Object o)
    // de la classe Object
    // L'annotation @Override permet de le notifier explicitement au compilateur.
    // (pas nécessaire, mais fortement conseillé pour se protéger de surcharger par err
    @Override
    public boolean equals(Object o) {
        if( ! (o instanceof PositionOk)) {
            return false ;
        }
        // ou, plus restrictif :
        // if(o.getClass() != this.getClass()) {
        //     return false ;
        // }

        // cast explicite nécessaire, car Downcast :
        PositionOk other = (PositionOk) o;
        return other.x == this.x && other.y == this.y;
    }
}
```

Considérons maintenant le code de test suivant :

```

public class Test { // 1
    public static void main(String args[]) { // 2
        PositionErreur posErr1 = new PositionErreur( 1, 1 ); // 3
        PositionErreur posErr2 = new PositionErreur( 1, 1 ); // 4
        // tout ces casts sont licites, puisque toute classe java
        // (et donc en particulier PositionErreur)
        // dérive de la classe Object
        // (éventuellement via plusieurs niveau d'héritage)
        Object objErr1 = posErr1; // 9
        Object objErr2 = posErr2; // 10

        System.out.println("posErr1 eq posErr2 : " + posErr1.equals(posErr2)); // 12
        System.out.println("posErr1 eq objErr2 : " + posErr1.equals(objErr2)); // 13
        System.out.println("objErr1 eq posErr2 : " + objErr1.equals(posErr2)); // 14
        System.out.println("objErr1 eq objErr2 : " + objErr1.equals(objErr2)); // 15

        PositionOK posOk1 = new PositionOK( 2, 2 ); // 17
        PositionOK posOk2 = new PositionOK( 2, 2 ); // 18

        Object objOk1 = posOk1; // 20
        Object objOk2 = posOk2; // 21

        System.out.println("posOk1 eq posOk2 : " + posOk1.equals(posOk2)); // 22
        System.out.println("posOk1 eq objOk2 : " + posOk1.equals(objOk2)); // 23
        System.out.println("objOk1 eq posOk2 : " + objOk1.equals(posOk2)); // 24
        System.out.println("objOk1 eq objOk2 : " + objOk1.equals(objOk2)); // 25
    }
}

```

A priori, on voudrait que ce programme affiche 8 fois "true", puisque les objets `posErr1` et `posErr2` sont « égaux » et que les objets `posOk1` et `posOk2` sont aussi « égaux ».

Or voici ce qui se passe...

- **Ligne 12:** Tout se passe bien et il s'affiche "posErr1 eq posErr2 : true".
- **Ligne 13:** il s'affiche "posErr1 eq objErr2 : false". En effet :
 - le type statique de l'objet `posErr1` sur lequel la méthode est appelée est `PositionErreur`.
 - le type statique du paramètre `objErr2` est `Object`.
 - dans la classe `PositionErreur`, la *méthode la plus spécifique* a pour signature `boolean equals(Object)`. Il s'agit de la méthode héritée de la classe `Object`.
 - c'est donc la méthode `boolean equals(Object)` de la classe `Object` qui est exécutée.
 - cette méthode compare l'égalité des références (des adresses en mémoire). Elle renvoie donc `false`, puisque les deux objets n'ont pas la même adresse !
- **Ligne 14:** il s'affiche "objErr1 eq posErr2 : false". En effet :
 - le type statique de l'objet `objErr1` sur lequel la méthode est appelée est `Object`.
 - le type statique du paramètre `posErr2` est `PositionErreur`.
 - dans la classe `Object`, la *méthode la plus spécifique* a pour signature `boolean equals(Object)` (moyennant un cast implicite de `PositionErreur` vers `Object` pour le paramètre `posErr2`).
 - c'est donc la méthode `boolean equals(Object)` de la classe `Object` qui est exécutée.
 - cette méthode compare l'égalité des références (des adresses en mémoire). Elle renvoie donc `false`, puisque les deux objets n'ont pas la même adresse.
- **Ligne 15:** il s'affiche "objErr1 eq objErr2 : false". En effet :
 - le type statique de l'objet `objErr1` sur lequel la méthode est appelée est `Object`.
 - le type statique du paramètre `objErr2` est `Object`.
 - dans la classe `Object`, la *méthode la plus spécifique* a pour signature `boolean equals(Object)`.
 - c'est donc la méthode `boolean equals(Object)` de la classe `Object` qui est exécutée.
 - cette méthode compare l'égalité des références (des adresses en mémoire). Elle renvoie donc `false`, puisque les deux objets n'ont pas la même adresse.
- **Ligne 22 à 25 :** dans tous les cas, c'est bien "true" qui s'affiche, comme attendu. En effet, la seule méthode nommée `equals` de la classe `PositionOK` a pour signature `boolean equals(Object)` et elle est en accord avec tous les types statiques utilisés. C'est donc cette méthode qui sera exécutée.

Principe

En java, la méthode qui teste l'égalité de deux objets doit *toujours* avoir pour signature `public boolean equals(Object o)`.

Ensimag POO by des profs de POO passés et présents.

licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/) 

Documentation built with [MkDocs](https://mkdocs.org/).

[Fiche 6 : Abstraction et interfaces](#)[Notion de classe et de méthode abstraite](#)[Classe abstraite en Java](#)[Méthode abstraite](#)[Mais pourquoi s'embêter à déclarer des méthodes abstraites ?](#)[Un code Java pour nos Figures](#)[Notion d'interface](#)[Une interface est un type !](#)[Résumons...](#)[Les classes anonymes](#)[Un exemple d'interface](#)

Fiche 6 : Abstraction et interfaces

L'objectif de cette fiche est d'introduire le mécanisme d'abstraction en Java, complémentaire de l'héritage, ainsi que la notion d'interface.

Notion de classe et de méthode abstraite

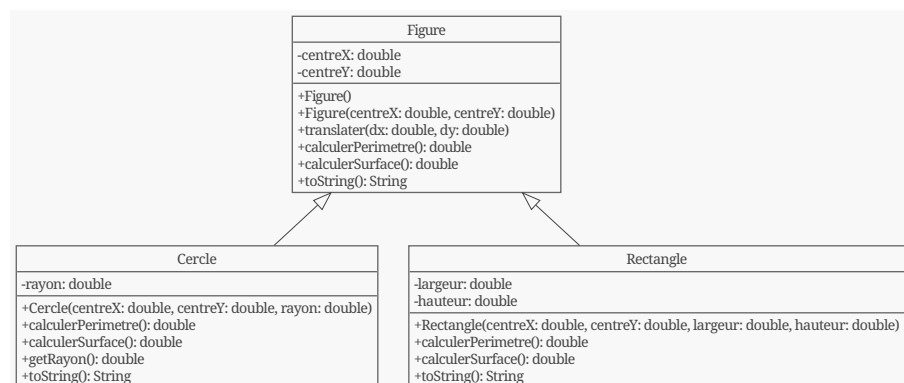
Un Pangolin, c'est très joli, mais nous avons bien assez fait crier ces petites bêtes. Qu'elles se reposent un peu.

Dans cette fiche, on s'appuie sur un grand classique de la POO : on suppose qu'on veut écrire un programme qui manipule des **figures géométriques** de plusieurs types : des cercles, des rectangles par exemple.

Chaque figure est caractérisée par un certain nombre de propriétés, qui vont se concrétiser dans le code par des attributs. Ainsi, toutes les figures ont un point central ; les cercles ont en plus un rayon ; et les rectangles ont en plus une largeur et une hauteur.

Chaque figure possède également un certain nombre de savoir-faire, qui vont se concrétiser dans le code par des méthodes. Ainsi, par exemple, nos cercles et nos rectangles (bref : toutes nos figures) peuvent être translatés, et il est possible de calculer leur périmètre et leur surface.

Le concept d'héritage conduit bien sûr à introduire une classe mère `Figure` et autant de classes filles que de types de figures :



La classe mère `Figure`, s'impose en POO dans notre contexte. Son existence offre plusieurs avantages :

- la classe `Figure` factorise le code (attributs, méthodes) commun à toutes les figures
- elle traduit en Java le fait que tous les cercles et tous les rectangles SONT des cas particuliers d'une notion plus générique : les figures.
- ce faisant, elle rend possible l'usage du [polymorphisme d'objets](#).

Par contre, à votre avis, est-il logique de pouvoir *instancier* des objets de type dynamique `Figure` ? Autrement dit, est-il légitime d'écrire quelque chose comme :

```
Figure f = new Figure(...);
```

Eh bien, non ! Dans notre petit programme, on instanciera des cercles et des rectangles, mais jamais de figure « tout court ». En effet, qu'est ce que pourrait bien être une « figure » qui ne serait ni un rectangle, ni un cercle, ni un triangle, ni... ?

Ainsi, la **notion de Figure est un concept abstrait**. En POO, on en fera de la classe `Figure` une **classe abstraite** : la classe existe, mais on ne peut pas l'instancier.

Par ailleurs, que penser du code des méthodes `calculerSurface()` et `calculerPerimetre()` dans la classe `Figure` ? Certes, chaque figure (que ce soit un cercle ou un rectangle) peut calculer son

périmètre... Mais bienheureux(se) celui ou celle qui peut écrire, par exemple, le code de la méthode `calculerPerimetre()` dans la classe `Figure` ! Car comment calculer le périmètre d'une « figure » si on ne sait pas si c'est un cercle ou un rectangle, ou ...?

Autrement dit, ces traitements sont des traitements qui existeront bien sur toutes les figures, mais auxquels on ne peut pas associer de comportement (de code) au niveau de la classe `Figure`. En POO, on on dira que ce sont des **méthodes abstraites** dans la classe `Figure` : elles sont déclarées dans la classe en tant que promesse de traitement qui existeront bien dans les sous-classes, mais sans code.

Classe abstraite en Java

Attention, cela va aller vite !

En Java, l'abstraction est dénotée par le mot clé `abstract`.

Une classe est déclarée abstraite de la façon suivante :

```
public abstract class MaClasse {  
    [...]  
}
```

Le seul effet du mot clé `abstract` appliqué à une classe est que cette classe ne peut plus être instanciée avec `new`. Il ne pourra donc jamais exister en mémoire un objet dont le type *dynamique* soit cette classe.

Ainsi, le code suivant provoquerait une erreur de compilation, car la classe `Figure` est abstraite :

```
Figure f = new Figure(...);  
// Et le compilateur râle :  
// "error: Figure is abstract; cannot be instantiated"
```

A part le fait qu'elle ne peut pas être instanciée, une classe abstraite s'écrit et se manipule comme toute autre classe. Elle peut contenir des attributs, des méthodes (dont des constructeurs), comme n'importe quelle classe.

Voici, par exemple, à quoi pourrait ressembler le début de notre classe `Figure` désormais abstraite :

```
public abstract class Figure {  
    private double centreX;  
    private double centreY;  
  
    public Figure(double x, double y) {  
        this.centreX = x;  
        this.centreY = y;  
    }  
    [...]  
  
    /** translate la figure du vecteur (dx, dy) */  
    public void translater(double dx, double dy) {  
        centreX += dx;  
        centreY += dy;  
    }  
    [...]  
}
```

Méthode abstraite

En Java, une méthode abstraite est une méthode :

- déclarée abstraite au moyen du mot clé `abstract`
- qui n'a pas de corps : on n'écrit que le prototype.

Une méthode abstraite peut être vue comme une promesse de traitement qui existera dans les sous-classes concrètes, mais dont on ne peut pas encore écrire le code dans la classe mère.

Voici à quoi ressemblerait les méthodes abstraites `calculerPerimetre()` et `calculerSurface()` dans la classe `Figure` :

```
public abstract class Figure {  
    [...] // comme précédemment  
    public abstract double calculerPerimetre(); // méthode abstraite  
    public abstract double calculerSurface(); // méthode abstraite  
}
```

Par ailleurs :

Toute classe qui déclare une (ou plusieurs) méthode(s) abstraite(s) est elle même nécessairement abstraite.

En conséquence, une classe qui a une méthode abstraite doit donc être elle-même déclarée `abstract` si vous voulez éviter que le compilateur ne râle. Rappelons qu'à l'inverse une classe peut être abstraite même si elle n'a pas de méthode abstraite.

Mais pourquoi s'embêter à déclarer des méthodes abstraites ?

Oui, tiens : pourquoi donc est-il important que la classe `Figure` ait une méthode abstraite `abstract double calculerPerimetre()` alors que cette méthode n'a pas de code ?

Voici trois explications :

- D'abord, du point de vue de la conception, la méthode `abstract double calculerPerimetre()` de la classe `Figure` traduit dans le code l'idée qu'il « est possible de calculer le périmètre de toutes les figures ». Il est donc logique qu'elle soit déclarée dans cette classe, même si on ne sait pas comment faire le calcul à ce niveau de la hiérarchie.
- Ensuite, une méthode abstraite est une promesse de traitement. Sa déclaration dans la super-classe va obliger à bien définir une version concrète dans les sous classes. Si par exemple dans la classe `Rectangle` on oublait de définir cette méthode, alors cette méthode héritée resterait abstraite dans la sous-classe `Rectangle`. Mais alors, cette sous classe `Rectangle` serait elle même abstraite et on ne pourrait plus créer de `Rectangle` !
- Enfin, l'existence des méthodes abstraites dans la classe `Figure` rend possible l'usage de ces méthodes dans le cas du polymorphisme. En effet, le code suivant :

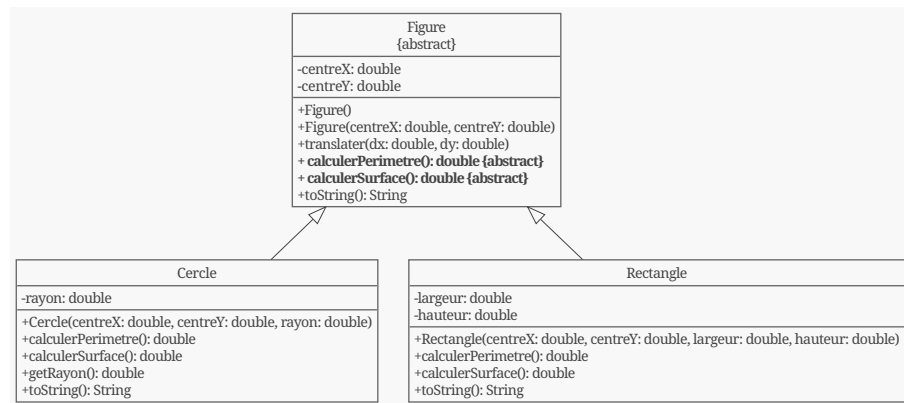
```
Figure f = new Rectangle(...);  
double perim = f.calculerPerimetre();  
...
```

ne compile et ne fonctionne que si `double calculerPerimetre()` est bien définie (même abstraite) dans la classe `Figure`. Si vous en doutez, vous pourrez avantageusement vous replonger dans [la partie statique du mécanisme d'appel de méthode](#) en Java de la fiche précédente...

Notez pour finir que dans la super-classe `Figure`, il serait illogique d'écrire une méthode `double getRayon()` (qu'elle soit abstraite ou non). En effet, la notion de « rayon » n'a de pas de sens au niveau de la super classe, mais uniquement dans la sous-classe `Cercle`, contrairement au périmètre par exemple (toute figure a un périmètre).

Un code Java pour nos Figures

Voici, avec l'abstraction, le schéma UML et un exemple de code pour nos classes (à compléter... par exemple avec des accesseurs). Notez qu'en UML l'abstraction est dénotée par la mise en italique, ou l'ajout du mot-clef `{abstract}` entre accolades.



```

public abstract class Figure {
    private double centreX;
    private double centreY;

    public Figure() {
        this.centreX = 0;
        this.centreY = 0;
    }

    public Figure(double x, double y) {
        this.centreX = x;
        this.centreY = x;
    }

    /** translate la figure du vecteur (dx, dy) */
    public void translater(double dx, double dy) {
        centreX += dx;
        centreY += dy;
    }

    public abstract double calculerPerimetre(); // méthode abstraite
    public abstract double calculerSurface(); // méthode abstraite

    @Override
    public String toString() {
        return "centre (" + x + ", " + y + ")";
    }
}

```

```
import java.math.*;
public class Cercle extends Figure {
    private double rayon;

    /** constructeur par défaut : cercle de rayon 1 centré sur l'origine */
    public Cercle() {
        super();
        this.rayon = 1;
    }

    public Cercle(double x, double y, double rayon) {
        super(x, y);
        this.rayon = rayon;
    }

    public double getRayon() {
        return rayon;
    }

    /** définition des méthodes calculerPerimetre et calculerSurface */
    @Override
    public double calculerPerimetre() {
        return 2 * Math.PI * rayon ;
    }

    @Override
    public double calculerSurface(){
        return Math.PI * rayon * rayon;
    }

    @Override
    public String toString() {
        return "Cercle " + super.toString() + " ; rayon = " + rayon;
    }
}
```

```
import java.math.*;
public class Rectangle extends Figure {
    private double largeur;
    private double hauteur;

    /** constructeur par défaut : rectangle de hauteur et largeur 1 centré sur l'origin
    public Cercle() {
        super();
        largeur = 1;
        hauteur = 1;
    }

    public Cercle(double x, double y, double hauteur, double largeur) {
        super(x, y);
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    /** définition des méthodes calculerPerimetre et calculerSurface */
    @Override
    public double calculerPerimetre() {
        return 2*(hauteur+largeur) ;
    }

    @Override
    public double calculerSurface(){
        return largeur * hauteur;
    }

    @Override
    public String toString() {
        return "Rectangle " + super.toString() + " ; largeur = " + largeur + " hauteur
    }
}
```

Et une classe de test pour finir :


```
public class Test {
    public static void main(String [] args) {
        // Figure = new Figure (1, 2);
        // => refusé à la compilation, car Figure est une classe abstraite
        // et ne peut être instanciée !

        Cercle c = new Cercle();
        Rectangle r = new Rectangle(1, 2, 10, 12);

        c.translater(2, 2);

        System.out.println(c);
        System.out.println(r);

        System.out.println("** Perimetre de c = " + c.calculerPerimetre());
        System.out.println("** Surface de r = " + r.calculerSurface());

        // Ce qui précède affiche :
        // Cercle centre (2,2) ; rayon = 1
        // Rectangle centre (1,2) ; largeur = 10 hauteur = 12
        // ** Perimetre de c = 6.283185
        // ** Surface de r = 120

        // un peu de polymorphisme pour finir...
        Figure [] tab = new Figure[2];
        tab[0] = new Cercle();
        tab[1] = new Rectangle(1, 2, 3, 4);
        for(Figure f: tab) {
            System.out.println(f);
        }
        // Affiche :
        // Cercle centre (0,0) ; rayon = 1
        // Rectangle centre (1,2) ; largeur = 3 hauteur = 4
    }
}
```

Notion d'interface

En première approche, en java et plus généralement en POO, une interface est un type (tout comme une classe) qui regroupe un ensemble de méthodes abstraites dont on ne donne que la signature (sans code).

Une interface s'écrit comme une classe, mais au moyen du mot clé `interface` en lieu et place du mot clé `class`.

Une interface ne contient aucun traitement véritable. Elle se contente de déclarer un cadre pour un ensemble de traitements qui devront être implémentés plus tard par une classe. Une interface est donc destinée à être **réalisée** (on dit aussi **implémentée**) par des classes.

Une classe déclare qu'elle implémente (ou « réalise ») une interface au moyen du mot clé `implements`.

Une classe qui déclare implémenter une interface s'engage à fournir le service (le « contrat ») spécifié par l'interface. Elle doit donc fournir une implémentation pour chacune des méthodes listées dans l'interface.

Détails ▼

Voyons un premier exemple, avec une interface `Deplacable`. Remarquez que le corps d'une interface se présente d'abord comme une classe allégée qui ne contient que des constantes et des signatures de méthodes (donc des méthodes abstraites).

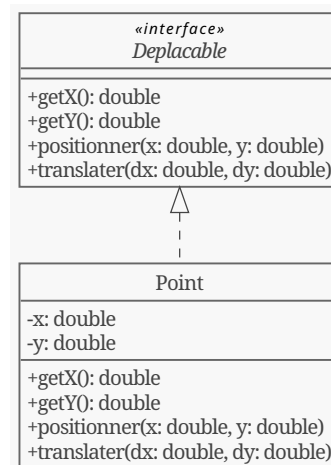
```
/**Cette interface spécifie la notion abstraite d'objet déplaçable.
Toute classe qui implémente l'interface Deplacable
doit pouvoir retourner une position (en 2D)
et fournir des services de modification de la position */
interface Deplacable {
    double getX();
    double getY();
    void positionner(double x, double y);
    void translater(double dx, double dy);
}

// Et voici une version (partielle...) de la classe Point qui réalise l'interface Depla
public class Point implements Deplacable {
    private double x;
    private double y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    // Puisque la classe Figure réalise l'interface Deplacable
    // il faut que toutes les méthodes déclarée dans l'interface
    // soit implémentées par la classe
    @Override
    public double getX() {
        return this.x;
    }
    @Override
    public double getY() {
        return this.y;
    }
    @Override
    public void positionner(double x, double y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public void translater(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }
}

// on peut bien sûr munir la classe d'autres méthodes que celles déclarées dans l'i
void symetrieOrigine() {
    x = -x;
    y = -y;
}

@Override
public toString() {
    return "Point (" + x + ", " + y + ")";
}
}
```

Et voici le diagramme UML de notre exemple :



Vous pouvez remarquer sur ce schéma la notation UML pour une interface et l'implémentation d'une interface. Ça ressemble beaucoup à l'héritage, non ? Mais notez le trait pointillé, qui se traduit par « implémente » ou « réalise ».

Une interface peut être vue :

- Comme une collection d'opérations abstraites, qui spécifie un service (un comportement) que les classes qui la réalisent doivent nécessairement implémenter. C'est pour cela que le nom d'une interface, par convention, se termine souvent par « able » (Deplacable, Cloneable, Comparable...)
- Mais aussi comme une classe « purement abstraite », qui ne contient que des méthodes abstraites et aucun attribut.

Sur ce dernier point, les notions d'interface et de classe purement abstraite sont d'ailleurs si proches que, dans certains langages objet, comme le C++ une interface n'est ni plus, ni moins qu'une classe purement abstraite.

Mais il existe une différence conceptuelle importante : alors qu'une classe (abstraite ou non) représente ce « que sont » des objets (leur « essence », leur « être »), une interface garantit juste que les classes qui la réalisent offrent le service déclaré dans l'interface, indépendamment de ce que la classe représente.

En java, il y a une autre différence importante :

En Java (et en UML), une différence essentielle entre les notions d'interface et de classe purement abstraite est qu'une classe peut implémenter plusieurs interfaces, alors qu'elle ne peut hériter que d'une unique classe (héritage simple).

On peut écrire par exemple :

```

class BonhommeDeNeige implements Deplacable, Cassable, PouvantEtreDecoreDuneCarotte {
    [...]
}
  
```

Les interfaces et le problème de l'héritage multiple en Java ▼

Une interface est un type !

Une interface définit un type, de la même manière qu'une classe.
 On peut donc déclarer des références du type d'une interface.
 Toute instance d'une classe qui implémente une interface peut être considérée comme étant du type de l'interface. Ou, dit autrement, une référence du type d'une interface (type statique) peut être utilisée pour manipuler des objets dont le type dynamique est n'importe quelle classe qui implémente cette interface.
 Une interface peut donc être utilisée, comme n'importe quelle super classe, pour le polymorphisme.

Par exemple :

```
Deplacable d = new Point(10, 12);
```

Si par exemple on écrit une interface `Dessinable` qui déclare des méthodes de dessin, et diverses classes qui réalisent cette interface, alors on peut manipuler les instances de ces classes « en tant que » `Dessinable` :

```
interface Dessinable {
    void dessiner(Graphics g);
    void setColor(Color c);
    [...]
}

class BonhommeDeNeige extends ElementDecors implements Dessinable {
    [...]
    void setColor(Color c) {
        this.color = c;
    }
    void dessiner(Graphics g) {
        g.drawOval(...);
        [...]
    }
    [...]
}

class Rectangle extends Figure implements Dessinable, Redimensionnable {
    [...]
    void setColor(Color c) {
        this.color = c;
    }
    void dessiner(Graphics g) {
        g.drawRect(...);
        [etc.]
    }
    [...]
}

class PangolinDessinable extends Pangolin implements Dessinable {
    [...]
    void setColor(Color c) {
        this.color = c;
    }
    void dessiner(Graphics g) {
        g.drawLine(...);
        [...]
    }
    [...]
}

class Test {
    public static void main(String args[]) {
        Graphics g = [...];
        // un tableau d'objets dessinable
        Dessinable[] tab = new Dessinable [3];
        // Avec ce tableau, on va pouvoir stocker des objets "en tant que dessinable" :
        // Peu importe qu'ils soient en fait des instances (type dynamique) de classes
        // (des bonhommes, des rectangles, des torchons et des serviettes...)
        // dès lors que leur classe (type dynamique) réalise l'interface Dessinable.

        tab[0] = new BonhommeDeNeige(...);
        tab[1] = new Rectangle(...);
        tab[2] = new PangolinDessinable(...);
        for(Dessinable d : tab) {
            d.dessiner(g);
        }
        // Et voilà un beau dessin avec un Pangolin posé sur un Rectangle et en émoi de
    }
}
```

Notez que les règles du polymorphisme s'appliquent avec les interfaces de la même manière que pour

les classes. En conséquence, comme d'habitude, sur une référence de type interface, seules les méthodes définies dans l'interface peuvent être exécutées :

```
Rectangle r = new Rectangle(2, 3, 12, 10);
r.translater(3, 4) ; // OK, pas de problème
Dessinable d = r; // on considère le rectangle "en tant qu'objet dessinable"
d.setColor(Color.RED) ; // OK, pas de problème : setColor(Color) est déclarée dans l'in
d.translater(3, 4) ; // erreur de compilation, même si un Rectangle a bien une méthode
// car translater() n'est pas définie dans Dessinable
```

Résumons...

On résume ci après les éléments essentiels à connaître pour les interfaces.

- Une interface est une liste de méthodes dont on donne seulement la signature.
- Une interface ne peut pas déclarer d'attribut - plus précisément, mais ce n'est pas essentiel : une interface ne peut déclarer que des attributs constants, déclarés `static final`.
- Toutes les méthodes d'une interface sont implicitement déclarées `abstract` et `public`. Il n'est pas nécessaire d'utiliser ces qualificatifs dans une interface - mais c'est possible.
- Une interface est un « contrat » que « réalise » ou « implémente » des classes (mot clé `implements`).
- Alors qu'une classe ne peut hériter que d'une unique autre classe, elle peut implémenter autant d'interface qu'elle le souhaite.
- Des classes dans des hiérarchies différentes peuvent implémenter la même interface.
- **Héritage entre interfaces** : une interface peut hériter d'un nombre **quelconque** d'interfaces, avec le mot clé `extends`. Exemple : `interface SuperDessinable extends Coloriable, Dessinable`.

À noter... ou pas... ▼

Les classes anonymes

Voici une petite digression par une construction bien pratique de Java : les classes anonymes.

Revenons sur notre interface `Dessinable` de tout-à-l'heure. Pour pouvoir l'utiliser, nous avons besoin d'écrire explicitement le code de classes qui réalisent cette interface et de créer des objets de ces classes. Reprenons le code ci-avant. Le bonhomme de neige dessinable ne sert qu'une seule fois. Il est peut-être un peu dommage de créer une classe nommée juste pour cela, alors que finalement, nous n'avons besoin que d'un objet qui soit dessinable, et dont on redéfinit précisément le comportement à la volée.

Ça tombe bien, nous pouvons définir en Java une classe **anonyme** et s'en servir pour instancier une variable. Voyez plutôt l'exemple ci-dessous.

```

interface Dessinable {
    public void dessiner(Graphics g);
    public void setColor(Color c);
    [...]
}

class Test {
    public static void main(String args[]) {
        Graphics g = [...];
        Dessinable[] tab = new Dessinable [3];

        tab[0] = new Dessinable() { // Une belle classe anonyme pour notre bonhomme de
            private Color color;

            public void setColor(Color c) {
                this.color = c;
            }
            public void dessiner(Graphics g) {
                g.drawOval(...);
                [...]
            }
        };
        tab[1] = new Rectangle(...);
        tab[2] = new PangolinDessinable(...);
        for(Dessinable d : tab) {
            d.dessiner(g);
        }
    }
}

```

Soyons clair : la syntaxe `new Dessinable()` ne signifie pas que l'on instancie directement une interface (ce qui serait contradictoire avec le fait qu'une interface ne contient que des méthodes abstraites). Il s'agit ici d'un raccourci d'écriture qui condense la définition d'une classe réalisant l'interface `Dessinable` et son instanciation en une seule ligne d'instructions. D'ailleurs, ce raccourci n'est possible que parce que l'on spécifie explicitement le code des méthodes abstraites lors de l'instanciation.

La contrepartie de ce raccourci d'écriture est que cette classe anonyme ne peut être formellement instanciée qu'une seule fois, lors de sa définition.

Un exemple d'interface

Vous découvrirez progressivement que la notion d'interface a beaucoup d'usages en POO.

L'exemple qui suit montre qu'une méthode (un algorithme) peut prendre en paramètre des références du type d'une interface. Il suffit que le corps de cette méthode n'utilise que les méthodes déclarées dans l'interface. Dès que cela est respecté, en effet, on est sûr que tout se passera bien : tout objet qui sera passé en paramètre à notre méthode sera une instance d'une classe qui implémente l'interface et qui, en conséquence, possédera nécessairement toutes les méthodes utilisées. On est donc certain, dès la compilation, que ces méthodes pourront bien être exécutées.

L'important dans cet exemple n'est pas l'algorithme du tri par insertion, mais :

- la notion d'interface et son utilisation dans le traitement générique de la méthode `trierParInsertion()` de la classe `Tri`
- l'usage du polymorphisme : des objets de types dynamiques variables sont manipulés au moyen de références (type statique) de type `InterfaceComparable`.
- le fait que, grâce au polymorphisme, l'algorithme de tri peut s'appliquer indifféremment à n'importe quel type d'objets dès lors que ces objets implémentent l'interface `InterfaceComparable`.

Ainsi, on écrit une fois pour toutes l'algorithme de tri... pour trier n'importe quels types d'objets "comparables" entre eux !

```

/** Toute classe qui implémente l'interface InterfaceComparable
 * L'interface InterfaceComparable impose donc l'existence d'une
 * relation d'ordre sur les classes qui l'implémente.
 */
interface InterfaceComparable {
    // Signature de la relation de comparaison
    // Retourne true si "this <= other"
    boolean infEgal(Object other);
}

class Tri {
    /**
     * tri par insertion d'un tableaux d'objets implantant l'interface InterfaceComparable
     * Cette méthode est capable de trier n'importe quel tableau d'objets
     * dès lors que leur classe implémente l'interface InterfaceComparable !
     */
    public static void trierParInsertion(InterfaceComparable[] t) {
        InterfaceComparable aux;
        int i;
        for (int k = 1; k < t.length; k++) {
            aux = t[k];
            i = k - 1;
            while (i >= 0 && (!t[i].infEgal(aux))) {
                t[i + 1] = t[i];
                i--;
            }
            t[i + 1] = aux;
        }
    }
}

/**
 * exemple de classe implémentant l'interface InterfaceComparable
 * La classe Entier encapsule une valeur de type int
 * et définit une relation d'ordre entre objets de type Entier
 */
class Entier implements InterfaceComparable {

    int valEntier;

    public Entier(int i) {
        valEntier = i;
    }

    @Override
    public boolean infEgal(Object x) {
        if (x.getClass() != this.getClass()) {
            return false;
        }
        // Le cast explicite est nécessaire pour le compilateur...
        // Mais, grâce au if, on est sûr qu'il ne générera pas une exception
        Entier e = (Entier) x;
        return this.valEntier <= e.valEntier;
    }

    public String toString() {
        return "Entier " + valEntier;
    }
}

/**
 * Autre exemple de classe implémentant l'interface InterfaceComparable
 * Une autre Voiture est "supérieure" à la Voiture this si sa puissance est
 * supérieure à la puissance de this.
 */
class Voiture implements InterfaceComparable {

    double puissance;

    public Voiture(double puissance) {
        this.puissance = puissance;
    }

    @Override

```

```

    public boolean infEgal(Object x) {
        if (x.getClass() != this.getClass()) {
            return false;
        }
        // Le cast explicite est nécessaire pour le compilateur...
        // Mais, grâce au if, on est sûr qu'il ne générera pas une exception
        Voiture e = (Voiture) x;
        return this.puissance <= e.puissance;
    }

    public String toString() {
        return "Voiture " + puissance;
    }
}

/** Et pourquoi ne pas "comparer" des Figures entre elles ?
 * Considérons, par exemple qu'une Figure est "inférieure" à une autre Figure si sa sur
 * intérieure à la surface de cette autre Figure...
 */
abstract class Figure implements Comparable {
    [... notre classe figure ...]

    abstract public double calculerSurface();

    @Override
    public boolean infEgal(Object other) {
        if (! other instanceof Figure) {
            return false;
        }
        return this.calculerSurface() <= ((Figure) other).calculerSurface();
    }
}

class Cercle extends Figure {
    private double rayon;
    [...]
    public double calculerSurface() {
        return Math.PI * rayon * rayon ;
    }
}

public class ExempleTri {
    public static void main(String args[]) {
        Comparable[] tab = new Comparable[5];
        tab[0] = new Entier(1);
        tab[1] = new Entier(9);
        tab[2] = new Entier(5);
        tab[3] = new Entier(10);
        tab[4] = new Entier(8);

        Tri.trierParInsertion(tab);
        for (Comparable c : tab) {
            System.out.println(c);
        }

        System.out.println("-----");

        tab = new Comparable[8];
        tab[0] = new Voiture(1.5);
        tab[1] = new Entier(9);
        tab[2] = new Entier(5);
        tab[3] = new Entier(10);
        tab[4] = new Voiture(5.1);
        tab[5] = new Voiture(8.5);
        tab[6] = new Cercle(1, 1, 10);
        tab[7] = new Cercle(2, 3, 100);

        Tri.trierParInsertion(tab);
        for (Comparable c : tab) {
            System.out.println(c);
        }
    }
}

```



```
// Ce programme affiche :  
//Entier 1  
//Entier 5  
//Entier 8  
//Entier 9  
//Entier 10  
//-----  
//Entier 5  
//Entier 9  
//Entier 10  
//Voiture 1.5  
//Voiture 5.1  
//Voiture 8.5  
//Cercle 10  
//Cercle 100
```

Notez que cet exemple a vocation pédagogique. Dans la vraie vie, Java dispose déjà d'une interface nommée `Comparable` et de méthodes qui implantent les algorithmes de tris usuels entre objets `Comparable`.

Ensimag POO by des profs de POO passés et présents.

licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/) 

Documentation built with [MkDocs](https://mkdocs.org/).

[Fiche 7 : Référence simplifiée pour la notation UML](#)[Classes](#)[Associations](#)[Associations simple](#)[Association directionnelle](#)[Agrégation et composition](#)[Héritage](#)[Interface](#)[Commentaires](#)

Fiche 7 : Référence simplifiée pour la notation UML

La maîtrise de la notation UML (*Unified Modeling Language*) ne faisant pas partie des objectifs pédagogiques de cet enseignement, vous trouverez ici une mini-fiche de référence (très simplifiée) permettant une bonne lecture des diagrammes du sujet de TPL. Seules les notations des **diagrammes de classes** sont couvertes, pas celles des autres types de diagrammes UML.

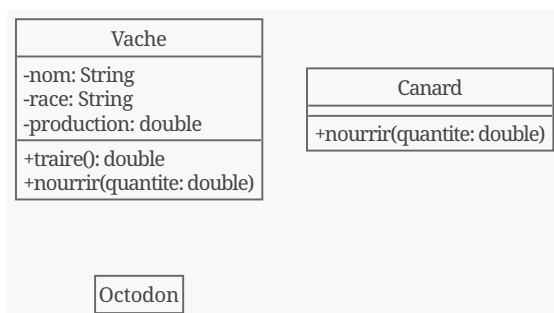
Classes

Les classes sont représentées sous leur forme la plus détaillée par un tableau en trois parties :

1. nom de la classe
2. liste des attributs (avec leur type)
3. liste des méthodes (avec leur signature).

Il est courant que tous les attributs et méthodes ne soient pas renseignés, mais seuls ceux importants dans un contexte donné. Le signe `+`/`-`/`#` est la *visibilité*, respectivement publique, privée ou protégée. Dans la forme simplifiée, on peut omettre la liste des attributs et/ou des méthodes (si les deux sont omis, il n'y a parfois qu'un rectangle avec le nom de la classe).

Syntaxe



Traduction

Vache est une classe, possédant (au moins) trois attributs privés `nom`, `race` et `production`, de types respectifs `String`, `String` et `double`, et deux méthodes publiques `traire`, sans paramètre et retournant un `double`, et `nourrir`, prenant en paramètre un `double` et ne retournant rien.

Canard est une classe dont les attributs ne sont pas détaillés, et possédant au moins une méthode `nourrir`, de visibilité non spécifiée, prenant en paramètre un `double` et ne retournant rien.

Octodon est une classe, dont le détail des attributs et méthodes n'est pas donné.

Associations

Une association est une relation structurelle forte entre deux classes. Comme pour les classes, différents niveaux de détails peuvent être affichés.

Associations simple

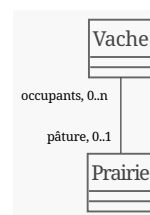
Syntaxe



Traduction

Il y a une relation entre les deux classes, mais on n'en sait pas plus. Par exemple, la classe `Vache` possède un *attribut* de type `Prairie` (ou tableau, ou une collection de `Prairie`), et vice-versa.

Syntaxe

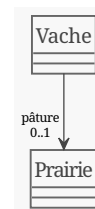


Traduction

Il y a, dans la classe `Vache`, un attribut nommé `pâture` de type `Prairie`, dont la valeur peut être `null` (multiplicité `0..1`). Réciproquement, il y a, dans la classe `Prairie`, un attribut nommé `occupants` de type tableau (ou collection) de `Vache`. Une prairie est associée à un nombre de vaches allant de 0 à n.

Association directionnelle

Syntaxe



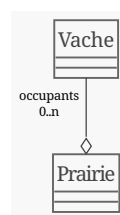
Traduction

Il y a, dans la classe `Vache`, un attribut nommé `pâture` de type `Prairie`, dont la valeur peut être `null` (multiplicité `0..1`).

Par contre, la classe `Prairie` n'a pas d'attribut de type `Vache`. Ainsi, « une vache sait dans quelle prairie elle se trouve », mais « une prairie ne sait pas quelle(s) vache(s) elle accueille »...

Agrégation et composition

Syntaxe

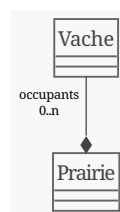


Traduction

Une `Prairie` connaît les vaches qu'elle accueille, elle possède un attribut nommé `occupants` de type tableau (ou collection) de `Vache`. Par contre, les vaches ne savent pas forcément dans quelle prairie elles se trouvent. Elles peuvent aussi ne pas être dans une prairie.

Dans une **agrégation** (losange **creux**), une instance de `Vache` peut exister même si elle n'est associée à aucune `Prairie`.

Syntaxe

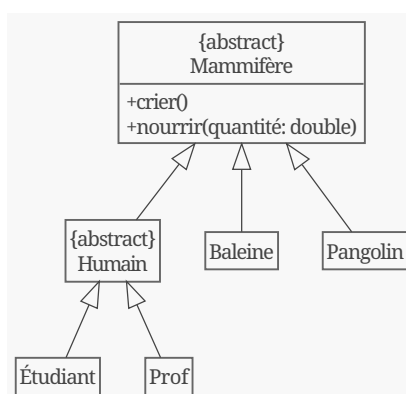


Traduction

Comme pour une agrégation, une `Prairie` connaît les vaches qu'elle accueille. Par contre une **composition** (losange **plein**) signifie qu'une vache est toujours dans une et une seule prairie. La classe `Prairie` est « propriétaire » des instances de `Vache` qui la composent; si la prairie est détruite, ses vaches le sont également.

Héritage

Syntaxe



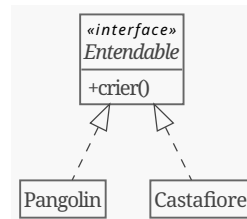
Traduction

Les classes `Baleine`, `Pangolin` et `Humain` sont des sous-classes de la classe `Mammifere`, et `Etudiant` et `Enseignant` sont des sous-classes de `Humain`.

La classe `Mammifere` est **abstraite**. Elle possède deux méthodes `crier`, qui est concrète, et `nourrir` qui est abstraite et devra être redéfinie dans les classes filles.

Interface

Syntaxe



Traduction

Entendable est une **interface** spécifiant une seule méthode crier. Les classes Pangolin et Castafiore **réalisent** cette interface, elles donc doivent (re)définir la méthode crier.

Commentaires


Syntaxe

Ceci est un commentaire...
Eh oui !

Traduction

Les commentaires sont informatifs mais n'ont aucune signification formelle (comme n'importe quel commentaire dans un programme Java).

Ensimag POO by des profs de POO passés et présents.

licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/) 

Documentation built with [MkDocs](https://mkdocs.org/).

Fiche 8 : Collections

Principe

Organisation

Parcours d'une collection : itérateur et for each

Les séquences : List

LinkedList

ArrayList

Files et piles: Queue, Deque

File à priorités

Avec l'interface Comparable

Avec une instance de Comparator

Ensembles : Set

Ensemble quelconque

Ensemble ordonné

Exemple d'ensemble ordonné

Dictionnaires: Map

Parcours

Cheat-sheet des coûts associés aux principales collections

Fiche 8 : Collections

Ce document présente le concept des **collections Java**, les principales classes et leur utilisation. Avec les exemples donnés, ceci est normalement suffisant pour la réalisation du TPL; vous n'utiliserez d'ailleurs pas tout ce qui est présenté. Cette présentation surtout technique, nous reviendrons sur les aspects plus conceptuels dans la fin du cours.

Principe

Les *collections* Java sont un ensemble de classes définissant des **structures de données** efficaces pour stocker, rechercher et manipuler des objets. De nombreuses structures existent couvrant les besoins les plus courants :

- séquences d'éléments (listes) ;
- ensembles quelconques, ensembles ordonnés ;
- queues (files, piles, files à priorité) ;
- des dictionnaires associatifs entre des clés et des valeurs ;
- et d'autres conteneurs plus spécifiques...

Le choix d'une collection dépend de l'utilisation recherchée et des coûts des opérations principalement réalisées.

Important

Avant de se lancer dans le développement éventuel de vos propres structures de données, il est toujours préférable de d'abord chercher parmi les collections du langage celles qui répondent le mieux à vos besoins. Dans la majorité des cas les structures adéquates existent déjà, qui plus est implantées de manière efficace et validées.

Organisation

Ces différents « types de données abstraits » sont spécifiés dans des *interfaces* de haut niveau, puis implantés dans différentes classes concrètes. Voir la fiche dédiée à [l'abstraction et aux interfaces Java](#).

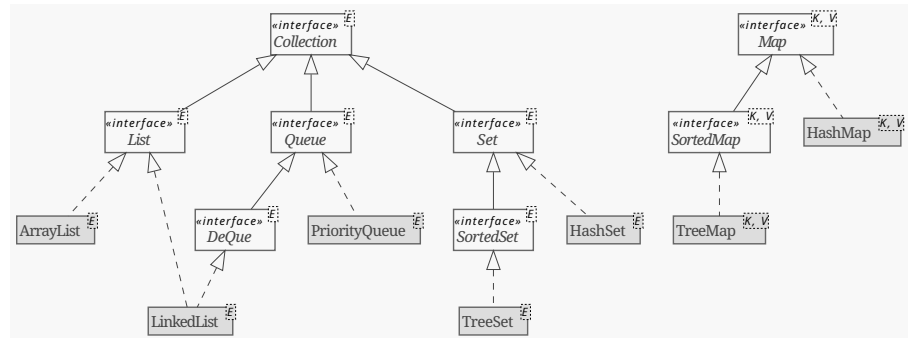
Il existe en fait deux hiérarchies de classes :

1. celle des collections proprement dites qui permettent de stocker des éléments. Elles sont filles de l'interface `Collection<E>` .
2. celle des *dictionnaires associatifs* contenant des couples (*clé*, *valeur*), dont la racine est l'interface `Map<K, V>` .

Nous ne décrivons ici que les collections usuelles principales. De nombreuses ressources sont également disponibles, notamment :

- [l'API](#) (indispensable!). Toutes les classes et interfaces discutées ici sont dans le paquetage `java.util` ;
- les [Java tutorials](#)

Ci-dessous, voici la hiérarchie des principales collections et dictionnaires JAVA. En grisé les classes concrètes, qui réalisent les interfaces abstraites de haut niveau.



Généricité

Les collections sont **génériques** : elles permettent de stocker des **objets** de tout type (`Object` , `String` , `Pangolin` , ...).

Pour stocker des données d'un type de base (ce ne sont pas des objets), il existe en fait des classes « wrapper » : `Integer` pour `int` , `Double` pour `double` , etc. La collection est déclarée avec le wrapper (par exemple `List<Integer>`), mais on peut y ajouter des données `int` sans besoin de conversion explicite (mécanisme d'*auto-boxing*).

Parcours d'une collection : itérateur et `for each`

Toutes les collections possèdent au moins un point commun : le parcours de leur contenu.

Le mécanisme d'**itération** permet de parcourir séquentiellement tous les éléments d'une collection, de manière uniforme et efficace. Il existe dans plusieurs langages, notamment Python ou C++.

En Java toute collection possède une méthode `Iterator<E> iterator()` qui retourne un *itérateur* permettant d'accéder aux éléments un par un en « avançant » dans la collection. Initialement, l'itérateur est placé « avant » le premier élément. A chaque accès, le prochain élément est retourné et l'itérateur avance à l'élément suivant. Lors du parcours complet d'une collection, chaque élément est retourné une et une seule fois, dans un ordre dépendant en fait du type de la collection.

Les deux méthodes principales d'un itérateur sont :

- `public boolean hasNext()` qui retourne `true` s'il reste des éléments dans l'itération ;
- `public E next()` qui **retourne** le prochain élément et **avance** dans l'itération. S'il n'y a plus d'élément, une `NoSuchElementException` est levée.

Le schéma d'utilisation est toujours le même, quelle que soit la collection :

```

Collection<E> coll = new ...;           // Collection existante, de type quelconque
                                        // Elle contient des éléments de type E

Iterator<E> it = coll.iterator();       // Crée un nouvel itérateur sur la collection,
                                        // initialisé AVANT le 1er élément

while (it.hasNext()) {                 // Tant qu'il reste des éléments
    E e = it.next();                   // Récupère le prochain élément et avance
    ...                                // Traiter e ici
}
  
```

Les collections, comme les tableaux, peuvent aussi être parcourues à l'aide du mécanisme de `for each` :

```

for (E e: coll) {                     // Pour tout élément e de coll
    ...                               // Traiter e ici
}
  
```

Par rapport au parcours avec un itérateur, un `for each` traite systématiquement **tous** les éléments de la collection. L'itérateur permet aussi d'enlever un élément pendant le parcours (la classe `Iterator` possède une méthode `remove()` , optionnelle, qui retire de la collection le dernier élément retourné par `next()`).

Les séquences : List

L'interface `List<E>` définit une séquence d'éléments, indexés du 1er au dernier par leur position dans la séquence : un entier de 0 à `size() - 1`. Les méthodes permettent d'insérer au début, à la fin, ou à une position précise. Lors d'une itération, les éléments sont naturellement parcourus dans l'ordre de la séquence. Les deux principales classes réalisant cette interface sont `LinkedList<E>` et `ArrayList<E>`.

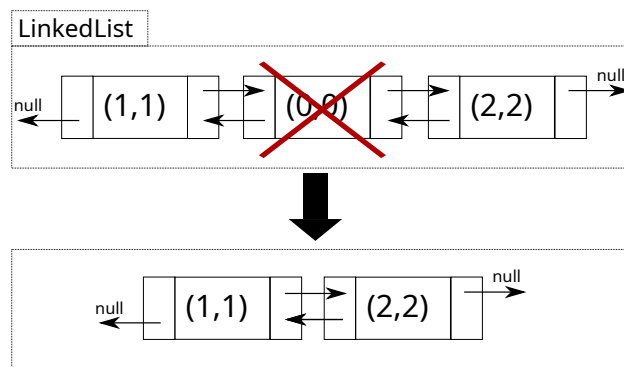
LinkedList

Fondée sur une liste doublement chaînée, cette classe est intéressante en cas d'ajouts et suppressions en début ou en fin de séquence. En ce qui concerne les ajouts et suppressions en milieu de séquence, elles se font également en temps constant, mais nécessite d'accéder à la position de la cellule à ajouter ou supprimer, ce qui prend un temps $\frac{n}{2}$ (donc un temps linéaire) si la cellule est en milieu de liste. Un tel ajout / suppression n'est donc efficace que s'il se fait *en cours de parcours* (par exemple en supprimant la cellule à laquelle on est en train d'accéder avec un itérateur).

```
// Exemple: une LinkedList de points
LinkedList<Point> points = new LinkedList<Point>();
points.add(new Point(0, 0)); // par défaut, ajout en fin
points.addFirst(new Point(1, 1));
points.add(new Point(2, 2));
System.out.println(points); // [(1,1), (0,0), (2,2)]
System.out.println(points.get(2)); // Attention : la complexité de la méthode get(i)
// de la classe LinkedList<> est en O(n) !
// Car il faut parcourir la liste
points.remove(1); // enlève la valeur en position 1
System.out.println(points); // [(1,1), (2,2)]
points.remove(0); // Suppression en tête. Coût : O(1).
System.out.println(points); // [(2,2)]
```

Voici un exemple d'utilisation de la collection `LinkedList`. Suppression de l'élément d'indice 1 (le deuxième donc) de la liste :

`points.remove(1)`



Ici, le coût est en $\Theta(n)$ pire cas car :

1. parcourir la liste pour accéder à l'élément d'indice i : $\Theta(i)$;
2. supprimer cet élément : $\Theta(1)$.

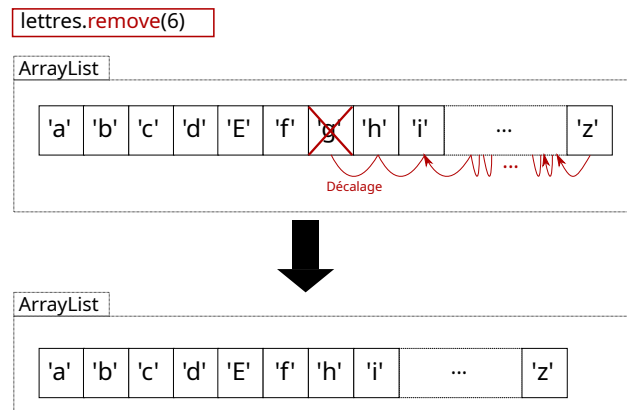
ArrayList

Fondée sur un tableau redimensionnable et donc à coût constant pour les accès direct à un élément en fonction de sa position (ième). Par contre, les coûts des insertions/suppressions en position quelconque sont linéaires (décalage des valeurs aux indices suivants).


```
// Exemple: un ArrayList de caractères
// => notez le ArrayList<Character> et pas ArrayList<char>
// Pour les types de base, il faut utiliser des classes "wrapper"
// (Integer, Float, Double ...) dans la déclaration
// => le mécanisme d'"auto-boxing" permet d'utiliser des char normalement:
// - lettres.add('a') est wrappé par lettres.add(new Character('a'))
// - lettres.remove(i) retourne lettres.remove(i).getValue()
ArrayList<Character> lettres = new ArrayList<Character>();
for (char c = 'a'; c <= 'z'; c++) {
    lettres.add(c); // ajout en fin
}
lettres.set(4, 'E'); // remplace la 5ème lettre par sa majuscule. Coût : 0
System.out.println("La 25ème lettre est " + lettres.get(24));
System.out.println(lettres.get(5)); // accès par indice en O(1) dans une ArrayList<
lettres.remove(6); // coût! (décalages)

// On affiche les 10 premières lettres
Iterator<Character> it = lettres.iterator();
int n = 0;
while (it.hasNext() && n++ < 10) {
    char c = it.next();
    System.out.print(c + " ");
} // affichage: a b c d E f h i j k
```

Voici un exemple d'utilisation de la collection `ArrayList`. Suppression d'un élément de la collection.



Ici, le coût est en $\Theta(n)$ pire cas car :

1. accéder à l'élément d'indice i : $\Theta(1)$;
2. supprimer cet élément, puis décaler des éléments qui suivent : $\Theta(n)$.

Files et piles: Queue, Deque

L'interface `Queue<E>` définit typiquement une file (FIFO, *First In First Out*), avec deux méthodes `add(E e)` qui ajoute en fin et `E remove()` qui enlève et retourne le premier élément.

L'interface `Deque<E>` (*Double Ended Queue*) spécialise une `Queue<E>` avec des méthodes d'insertion, d'accès et de retrait en tête et queue de liste : `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, `getLast`. Ces méthodes permettent notamment d'utiliser `Deque` comme une pile (LIFO, *Last In First Out*), en considérant par exemple la tête comme le sommet de la pile.

La classe de référence réalisant ces interfaces est `LinkedList<E>`. Les opérations principales sont à coût constant $O(1)$.

```
// La queue du Pangolin, bien sûr!
Queue<Pangolin> file = new LinkedList<Pangolin> ();
file.add(new Pangolin("G  rard", 1542));
file.add(new Pangolin("Pierre", 1939));
file.add(new Pangolin("Heckel", 1));
System.out.println(file.remove()); // c'est G  rard
file.add(new PangolinALongueQueue("Jeckel", 2);
while (!file.isEmpty()) {
    System.out.println(file.remove()); // Pierre, Heckel puis Jeckel
}
```

Type de haut niveau et implantation

Notez qu'ici la `file` a   t   d  clar  e (type statique) avec une type abstrait de haut niveau, l'interface `Queue`. Par contre son type dynamique ne peut   tre lui qu'une classe concr  te r  alisant l'interface, ici `new LinkedList`.

Il est recommand   d'utiliser un type statique abstrait si on n'utilise que les m  thodes d  clar  es    ce niveau. L'information est souvent plus pr  cise : ici on sait que conceptuellement l'objet `file` est utilis   comme une `Queue` (alors que la classe `LinkedList` autorise en fait beaucoup plus de choses). De plus, il est possible de modifier le type concret ult  rieurement (`ArrayList` au lieu de `LinkedList`) pour des raisons de performances selon les op  rations utilis  es, sans modifier le reste du code.

File    priorit  s

`PriorityQueue<E>` est une file    priorit   : l'ordre de sortie des   l  ments ne d  pend plus de leur date d'insertion, comme une FIFO ou LIFO, mais d'une *priorit  * entre   l  ments.

L'implantation de la classe `PriorityQueue<E>` repose sur un tas binaire. Les co  ts des op  rations d'insertion et de retrait de l'  l  ment le plus prioritaire sont en $O(\log(n))$. L'op  ration d'acc  s    l'  l  ment le plus prioritaire sans le retirer `peek()` est l'op  ration la plus efficace : elle est en $O(1)$ (ce qui fait tout l'int  r  t de cette structure par rapport    un `TreeSet`). Attention par contre, la recherche ou la suppression d'un   l  ment quelconque sont possibles mais en $\Theta(n)$.

En pratique les   l  ments doivent   tre munis d'une *relation d'ordre*, et l'  l  ment le plus prioritaire (le prochain    sortir) est le plus petit selon cet ordre.

Deux solutions sont possibles pour d  finir cette relation d'ordre.

Avec l'interface Comparable

La classe `E` des   l  ments peut r  aliser l'interface `Comparable<E>`, qui d  finit l'ordre dit « naturel » entre des instances de `E`. Elle doit donc red  finir la m  thode `public int compareTo(E e)` qui retourne une valeur n  gative/nulle/positive si `this` est plus petit/  gal/ plus grand que `e`.

```
// Exemple: une PriorityQueue avec l'implémentation de l'interface Comparable
public class Etudiant implements Comparable<Etudiant> {
    private String name;
    private int note;

    public Etudiant(String name, int note) {
        this.name = name;
        this.note = note;
    }
    @Override
    public String toString() {
        return "Je m'appelle " + name + " et j'ai eu une note de " + note;
    }
    // On implémente la méthode compareTo qui prend en paramètre
    // un autre objet Etudiant à comparer avec this
    @Override
    public int compareTo(Etudiant e) {
        // On retourne un nombre positif
        // si la note de this est supérieur à celle de e
        if(this.note > e.note) {
            return 1;
        }
        if(this.note < e.note) {
            return -1;
        }
        return 0;
    }
    // equals(Object o) doit être cohérent avec compareTo(Etudiant e),
    // comme expliqué dans la Javadoc de l'interface Comparable<E>.
    // Ici, donc, 2 étudiants doivent être égaux au sens de equals()
    // si et seulement si ils se comparent à 0 au sens de compareTo()
    @Override
    public boolean equals(Object other) {
        if(other instanceof Etudiant) {
            Etudiant et = (Etudiant) other;
            return et.note == this.note;
        }
        return false;
    }
}

public class ExemplePriorityQueueComparable {
    public static void main(String args[]) {
        // Création d'une PriorityQueue, comme on implémente l'interface Comparable
        // il n'y a pas besoin de spécifier avec quelle méthode on veut comparer,
        // Le tri va directement appeler la méthode compareTo de la classe Etudiant
        Queue<Etudiant> etudiants = new PriorityQueue<Etudiant>();
        etudiants.add(new Etudiant("Pauline", 14));
        etudiants.add(new Etudiant("Julia", 10));
        etudiants.add(new Etudiant("Théo", 16));
        etudiants.add(new Etudiant("Alphonse", 12));
        etudiants.add(new Etudiant("Pierre", 8));
        etudiants.add(new Etudiant("Isabelle", 20));
        etudiants.add(new Etudiant("Olivier", 15));

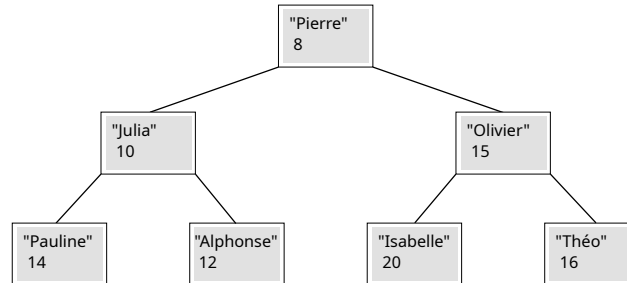
        // Remarque : puisque une PriorityQueue repose sur un tas,
        // un itérateur sur une PriorityQueue
        // ne parcourt pas dans l'ordre défini par le comparateur.

        // Par contre, principe du tas :
        // l'élément en tête est toujours le plus grand
        // (méthodes peek() et poll())

        // Utilisation de la méthode peek pour accéder à l'élément en tête de la file
        while(etudiants.peek() != null) { // on peut aussi écrire if( etudiants.size()
            // On retire l'élément en tête de la file triée
            Etudiant e = etudiants.poll();
            System.out.println(e);
            // Affiche :
            // Je m'appelle Pierre et j'ai eu une note de 8
            // Je m'appelle Julia et j'ai eu une note de 10
            // Je m'appelle Alphonse et j'ai eu une note de 12
            // Je m'appelle Pauline et j'ai eu une note de 14
            // Je m'appelle Théo et j'ai eu une note de 16
        }
    }
}
```

```
        // etc.  
        // c'est à dire les étudiants, par ordre croissant de leur note  
        // Complexité de toute la boucle boucle :  $O(n * \log(n))$   
    }  
    // maintenant, la file de priorité est vide...  
}  
}
```

Voici l'état final du tas binaire après l'ajout des éléments lors de l'exécution du code ci-dessus :



Avec une instance de Comparator

Il est aussi possible de déléguer la comparaison de deux objets `E` à une tierce classe qui réalise l'interface `Comparator<E>`. Celle-ci définit une seule méthode : `public int compare(E e1, E e2)` qui retourne une valeur négative/nulle/positive si `e1` est plus petit/égal/plus grand que `e2`. Si une classe fille de `Comparator` est donné à la `PriorityQueue`, c'est elle qui effectue la comparaison des éléments même si le type `E` réalise `Comparable<E>`.

Cette seconde approche est surtout utilisée pour pouvoir comparer des éléments selon des critères différents. Il est par exemple possible de créer deux classes qui comparent des étudiants selon leur nom ou selon leur note de POO.

```

// Exemple: une PriorityQueue avec l'implémentation de l'interface Comparator
// L'ordre choisi pour cet exemple est l'ordre alphabétique des noms.
// Il faut implémenter l'interface Comparator dans une classe tierce, nommée par exemple
class EtudiantComparator implements Comparator<Etudiant> {
    // On implémente la méthode compare(),
    // qui est la seule méthode définie dans l'interface Comparator.
    // Elle prend en entrée deux objets Etudiant e1 et e2 à comparer
    public int compare(Etudiant e1, Etudiant e2) {
        // On retourne un nombre positif
        // si le nom de e1 est plus grand alphabétiquement à celle de e2
        if(e1.getName().compareToIgnoreCase(e2.getName()) > 0) {
            return 1;
        }
        else if(e1.getName().compareToIgnoreCase(e2.getName()) < 0) {
            return -1;
        }
        else {
            return 0;
        }
    }
}

class Etudiant {
    private String name;
    private int note;

    public Etudiant(String name, int note) {
        this.name = name;
        this.note = note;
    }
    public String getName() {
        return name;
    }
    @Override
    public String toString() {
        return "Je m'appelle " + name + " et j'ai eu une note de " + note;
    }
    // equals(Object o) doit être cohérent avec compare(Etudiant e1, Etudiant e2)
    // Ici, donc 2 étudiants doivent être égaux au sens de equals()
    // si et seulement si ils se comparent à 0 au sens du comparateur
    @Override
    public boolean equals(Object other) {
        if(other instanceof Etudiant) {
            Etudiant et = (Etudiant) other;
            return et.name.equals(this.name);
        }
        return false;
    }
}

public class ExemplePriorityQueueComparator {
    public static void main(String argc[]) {
        // Création d'une PriorityQueue, comme on implémente l'interface Comparator
        // dans une classe tierce, il faut spécifier avec quelle méthode on veut comparer
        // Il faut donc donner une instance de la classe
        // qui implémente la méthode de comparaison à la PriorityQueue
        Comparator comparator = (Comparator)(new EtudiantComparator());
        Queue<Etudiant> etudiants = new PriorityQueue<Etudiant>(comparator);
        etudiants.add(new Etudiant("Pauline", 14));
        etudiants.add(new Etudiant("Julia", 10));
        etudiants.add(new Etudiant("Théo", 16));
        etudiants.add(new Etudiant("Alphonse", 12));
        etudiants.add(new Etudiant("Pierre", 8));
        etudiants.add(new Etudiant("Isabelle", 20));
        etudiants.add(new Etudiant("Olivier", 15));

        // le cout de l'ajout de tous les étudiants est en O(n * log(n))

        // Utilisation de la méthode peek pour accéder à l'élément en tête de la file triée
        while(etudiants.peek() != null) {
            // On retire l'élément en tête de la file triée
            Etudiant e = etudiants.poll();
            System.out.println(e);
        }
    }
}

```

```
// Affiche :  
// Je m'appelle Alphonse et j'ai eu une note de 12  
// Je m'appelle Isabelle et j'ai eu une note de 20  
// Je m'appelle Julia et j'ai eu une note de 10  
// Je m'appelle Olivier et j'ai eu une note de 15  
// Je m'appelle Pauline et j'ai eu une note de 14  
// Je m'appelle Pierre et j'ai eu une note de 8  
// Je m'appelle Théo et j'ai eu une note de 16  
}  
// maintenant, la file de priorité est vide...  
}  
}
```

Ensembles : Set

A la différence des séquences ou queues, un ensemble défini par l'interface `Set<E>` n'admet *pas de doublons* : un élément ne peut pas être présent deux fois dans la collection. Cette égalité entre éléments est testée via la méthode `public boolean equals(Object o)` héritée de la super classe `Object`, qui doit de ce fait être correctement redéfinie dans la classe `E` des éléments du `Set`.

Ensemble quelconque

La classe `HashSet<E>` réalise l'interface `Set<E>` avec une implémentation de type table de hachage. Le coût amorti des opérations principales (ajout, retrait, recherche) est en $O(1)$. L'itération retourne bien toutes les valeurs mais dans un ordre quelconque.

En plus de redéfinir `equals`, les éléments doivent aussi redéfinir une autre méthode de la classe `Object` : `public int hashCode()`, qui doit retourner une clé de hachage entière. Cette méthode doit être **cohérente** avec la redéfinition de l'égalité : si deux objets sont égaux au sens de `equals`, alors leur méthode `hashCode` doit retourner la même valeur.

Attention !

Même si elles ne sont pas utilisées par toutes les collections, il est recommandé de toujours redéfinir `hashCode` en même temps que `equals`, pour garantir la cohérence en cas d'utilisation d'une classe dans des collections.

Voici un exemple d'utilisation de `HashSet` :

```

class Fleur {
    private String name;
    public Fleur(String name) {
        this.name = name;
    }
    // la méthode equals à redéfinir pour que l'ajout
    // dans un HashSet se passe bien
    @Override
    public boolean equals(Object o) {
        if(o instanceof Fleur) {
            Fleur f = (Fleur)o;
            if(name.equals(f.name)) {
                return true;
            }
        }
        return false;
    }
    @Override
    public int hashCode() {
        // Dans la vraie vie, il suffirait de dire que le code de hachage d'une Fleur
        // est le code de hachage de son nom.
        // Bien sûr, la classe String redéfinit la méthode hashCode() !
        // Donc on écrirait donc :
        //     return name.hashCode();

        // Dans notre exemple, on plante une méthode de hachage un peu bête,
        // pour bien expliquer comment les fleurs vont être
        // rangées dans la table de hachage.
        // Bien sûr, cette méthode de hachage n'est pas recommandable,
        // car elle génère beaucoup de collisions.
        int hashCode = 0;
        for(int i = 0; i < name.length(); i++) {
            hashCode += (int)name.charAt(i);
        }
        return hashCode;
    }
    public String toString() {
        return name + " a un hash code qui vaut : " + hashCode();
    }
}

public class ExempleHashSet {
    public static void main(String args[]) {

        // On utilise un HashSet<Fleur>, il faut implémenter :
        // - la redéfinition de la méthode boolean equals(Object o)
        // - la redéfinition de la méthode int hashCode()

        // Capacité initiale (optionnelle) : 16 cases dans le tableau de hachage
        Set<Fleur> bouquetFleurs = new HashSet<Fleur>(16);

        bouquetFleurs.add(new Fleur("tulipe"));
        Fleur rose = new Fleur("rose");
        bouquetFleurs.add(rose);
        bouquetFleurs.add(new Fleur("coquelicot"));
        bouquetFleurs.add(new Fleur("quecolicot"));
        bouquetFleurs.add(new Fleur("lantana"));
        bouquetFleurs.add(new Fleur("litupe"));
        bouquetFleurs.add(new Fleur("lantana"));

        // le cout de l'ajout de toutes les fleurs est en O(n) seulement
        // car l'ajout d'un élément dans un HashSet est en O(1) coût amorti

        System.out.println("Taille du HashSet : " + bouquetFleurs.size());
        // Affiche :
        // Taille du HashSet : 6
        // Ah, la dernière fleur "lantana" n'a pas été ajoutée.
        // Pourquoi ?

        // Comme dans un HashSet, les éléments sont uniques (au sens de equals()),
        // l'ajout a échoué.

        // le parcours est fait dans un ordre apparemment "quelconque"
        for(Fleur fleur: bouquetFleurs) {

```

```

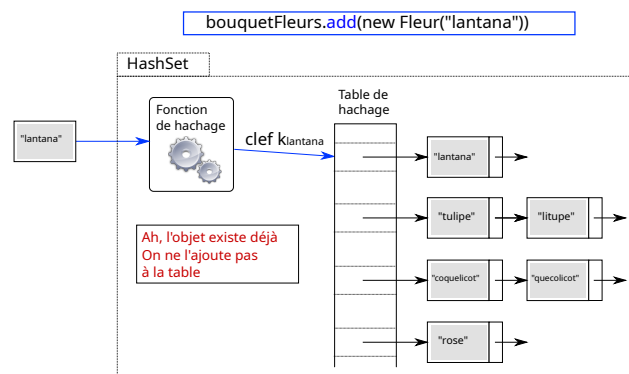
        System.out.println(fleur);
        // affiche :
        // tulipe a un hash code qui vaut : 659
        // litupe a un hash code qui vaut : 659
        // coquelicot a un hash code qui vaut : 1080
        // quecolicot a un hash code qui vaut : 1080
        // rose a un hash code qui vaut : 441
        // lantana a un hash code qui vaut : 735
    }

    // Utilisation de la méthode contains avec un objet rose en entrée
    System.out.println("Le bouquet de fleurs contient t-il une rose? : "
        + bouquetFleurs.contains(new Fleur("rose")));
    // Affiche:
    // Le bouquet de fleurs contient t-il une rose? : true
    // Coût de la méthode contains() : O(1)

    // On pourrait aussi utiliser un TreeSet<String> à la place du HashSet
    // => il faudrait munir la classe Fleur d'un comparateur
    // => le coût de l'ajout de toutes les fleurs passerait en  $O(n * \log(n))$ ;
    // => l'ordre d'affichage de mots respecterait l'ordre défini par le comparateur
    // => le coût de la méthode contains() passerait en  $O(\log(n))$ 
}
}

```

Voici comment se passe l'ajout de l'objet `Fleur` avec comme nom « `lantana` ».



Ensemble ordonné

La classe `TreeSet<E>` définit un ensemble dont les valeurs sont *ordonnées*. Il est donc nécessaire de munir les éléments d'une *relation d'ordre* pour pouvoir les comparer. Comme précédemment, deux solutions sont possibles :

- La classe `E` des éléments peut réaliser l'interface `Comparable<E>`. C'est donc la méthode `compareTo` qui est utilisée pour ordonner les éléments dans la collection.
- Un objet de type `Comparator<E>` peut être donné au `TreeSet`. C'est alors lui qui réalise la comparaison des éléments même si la classe `E` réalise `Comparable<E>`.

Quelle que soit la méthode utilisée, la comparaison doit être compatible avec l'égalité : si `e1.equals(e2)` (respectivement `!equals`) alors `e1.compareTo(e2)` et/ou `compare(e1, e2)` doivent retourner 0 (respectivement une valeur non nulle).

L'implantation de cette classe repose sur un arbre équilibré (de type rouge-noir), avec des coûts en $O(\log(n))$ pour les opérations principales.

Ordre vs égalité

Pour un ensemble ordonné ou un file à priorité, c'est la relation d'ordre (avec `Comparable` ou un `Comparator`) qui définit si un élément est déjà présent dans la collection. Le test repose sur le résultat de `compareTo` ou `compare`, s'il est nul ou non, et pas sur `equals`.

En revanche il est recommandé de toujours redéfinir `equals` (et même `hashCode`) pour pouvoir utiliser correctement un objet dans d'autres types de collections.

Exemple d'ensemble ordonné

Un exemple (long mais complet) d'ordonnement avec les célèbres frères Dalton, à retrouver dans le fichier [ExempleOrdonnement.java](#). Enjoy!

Les Dalton ▼

Dictionnaires: *Map*

L'interface `Map<K,V>` spécifie des associations entre une clé de type `K` et une valeur de type `V`. Un `Map` ne peut pas contenir des clés identiques, et chaque clé n'est associée qu'à une et une seule valeur. Les opérations principales sont l'ajout d'un couple (`put(K key, V value)`), l'accès à une valeur via sa clé (`V get(K key)`), la recherche de clé ou de valeur, la suppression d'une valeur, etc.

L'interface `SortedMap<K,V>` étend `Map<K,V>`, en rajoutant une relation d'ordre sur les clés du dictionnaire.

Deux classes principales existent:

1. `HashMap<K,V>` : une table avec hachage sur les clés ;
2. `TreeMap<K,V>` : un ensemble ordonné sur les clés (implanté là encore avec un arbre rouge-noir équilibré).

Comme pour les `Set`, les méthodes `equals`, `hashCode` ainsi qu'une relation d'ordre doivent être correctement définies, en particulier sur le type `K` des clés.

Parcours

Un `Map` peut être parcouru de différentes manières. En fait trois méthodes renvoient les clés et/ou les valeurs dans des collections, qui peuvent à leur tour être itérées.

- la méthode `Collection values()` retourne une collection (dont on ne connaît pas le type dynamique, mais ce n'est pas nécessaire) contenant toutes les valeurs.
- `Set<K> keySet()` retourne un ensemble contenant toutes les clés.
- `Set<Map.Entry<K,V>> entrySet()` retourne un ensemble de tous les couples (*clé, valeur*). Ces couples sont de type `Map.Entry<K,V>`, où `Entry<K,V>` est une classe *interne* à `Map` fournissant principalement deux méthodes `K getKey()` et `V getValue()`.

Pour un `SortedMap`, les collections ci-dessus sont ordonnées suivant l'ordre défini sur les clés.

Ci-dessous, un exemple ([ExempleMap.java](#)).

```
// ex: dictionnaire associant des étudiants (clé, une chaîne)
// à des notes (valeurs entières)
Map<String, Integer> annuaire = new HashMap<String, Integer> ();

String mc = new String("Matthieu");
String sb = new String("Sylvain");
String nc = new String("Nicolas");
annuaire.put(mc, 4);
annuaire.put(sb, 18);
annuaire.put(nc, 12);
annuaire.put(mc, 14); // pas de doublons,
                      // mais remplace l'ancienne valeur associée a mc

// affichage avec toString() : {Nicolas=12, Sylvain=18, Matthieu=14}
// (ordre indéterminé, c'est du hachage)
System.out.println("L'annuaire contient: " + annuaire);

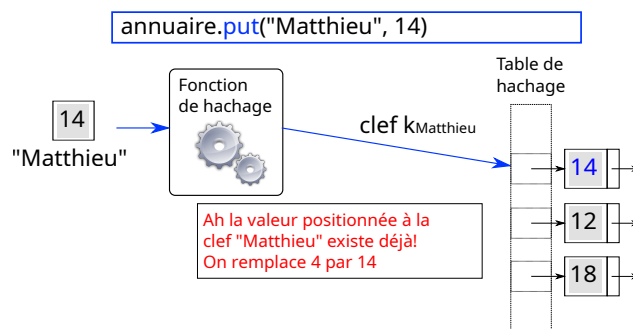
// et quelle est la note de Catherine?
Integer note = annuaire.get("Catherine");
System.out.println("La note de Catherine est: " + note); // null!

// ensemble des clés : [Nicolas, Sylvain, Matthieu]
Set<String> cles = annuaire.keySet();
System.out.println("Les clés sont: " + cles);

// collection des valeurs : [12, 18, 14]
Collection<Integer> notes = annuaire.values();
System.out.println("Les valeurs sont: " + notes);

// parcours avec un itérateur sur les couples
// (prévoir une aspirine pour la syntaxe...)
System.out.println("Les couples sont:");
Set<Map.Entry<String, Integer>> couples = annuaire.entrySet();
Iterator<Map.Entry<String, Integer>> itCouples = couples.iterator();
while (itCouples.hasNext()) {
    Map.Entry<String, Integer> couple = itCouples.next();
    System.out.println("\t" + couple.getKey()
                      + " a la note " + couple.getValue());
}
```

Exemple d'ajout de l'objet mc. L'objet est déjà présent (comparaison de nom), par conséquent, il y a remplacement de l'objet déjà existant.



Cheat-sheet des coûts associés aux principales collections

Voici un tableau récapitulatif des coûts associés aux opérations sur les collections présentées dans cette fiche :

Interface List	Add	Remove	Get	Contains	Next	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	O(1)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	O(1)	Doubly Linked List

Interface Set	Add	Remove	Contains	Next	Size	Data Structure
HashSet	O(1)	O(1)	O(1)	O(h/n)	O(1)	Hash Table
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)	O(1)	Red-black tree
Interface Queue	Offer	Peak	Poll	Remove	Size	Data Structure
PriorityQueue	O(log n)	O(1)	O(log n)	O(n)	O(1)	Priority Heap
LinkedList	O(1)	O(1)	O(1)	O(1)	O(1)	Doubly Linked List
ArrayDeque	O(1)	O(1)	O(1)	O(n)	O(1)	Array
Interface Map	Get	ContainsKey		Next	Data Structure	
HashMap	O(1)	O(1)		O(h / n)	Hash Table	
TreeMap	O(log n)	O(log n)		O(log n)	Red-black tree	

Ensimag POO by des profs de POO passés et présents.

licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/) 

Documentation built with [MkDocs](https://mkdocs.org/).