UML

Baptiste Bauer

Version v0.0.8.sip-221124091640, 2022-11-23 09:32:18

Table des matières

1. La relation d’héritage (classe mère, classe fille) . . . 1
    1.1. Comprendre et modéliser la notion d’héritage . . . 1
    1.2. Implémentation de la relation d’héritage . . . 7
    1.3. Point technique (en PHP) . . . 10
    1.4. Quelques exercices . . . 13
2. La relation abstraite (classe abstraite) . . . 17
    2.1. Comprendre le sens de l’adjectif "abstrait" . . . 17
    2.2. Représentation UML d’une classe abstraite . . . 18
    2.3. Implémentation d’une classe abstraite . . . 18
    2.4. Les méthodes aussi peuvent être abstraites . . . 19
Index . . . 22

1. La relation d’héritage (classe mère, classe fille)

1.1. Comprendre et modéliser la notion d’héritage

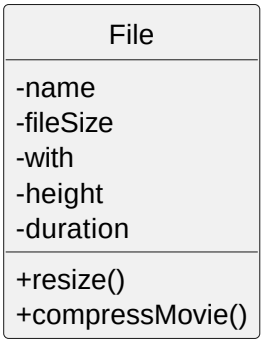
La **relation d’héritage** n’exprime plus l’idée qu’un objet A est lié à un objet B mais qu'**un objet A est un objet B**.

Imaginons que nous devions gérer des fichiers dans une application de gestion électronique des documents. La première chose est de créer la classe `File` :

Les types de fichiers à gérer sont les suivants :

- fichier texte avec un nom et une taille en Mo.
- fichier image avec un nom, une taille en Mo et des dimensions en pixels. Une image peut être redimensionnée.
- fichier vidéo avec un nom, une taille, une durée. Une vidéo peut être compressée.

Nous pouvons prendre en compte ces informations dans le diagramme suivant :

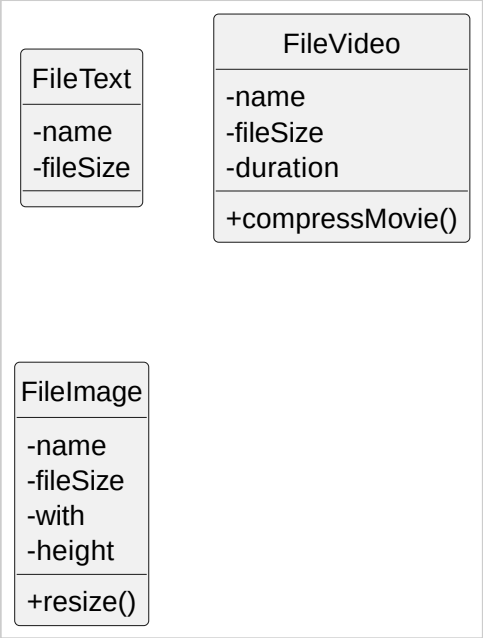


Cette classe soulève plusieurs remarques :

- Certains attributs et ou certaines méthodes sont inutiles en fonction du type de fichier. Un fichier texte ne peut pas être redimensionné ou subir une compression vidéo, une image n’a pas de durée, etc.
- Dès qu’il faut ajouter une nouvelle caractéristique (par exemple, le format d’affichage d’une image : portrait ou paysage), il faut faire évoluer la classe `File`.
  Elle va devenir de plus en plus "grosse" et poser un problème de maintenance évolutive (et peut être devenir un God object).

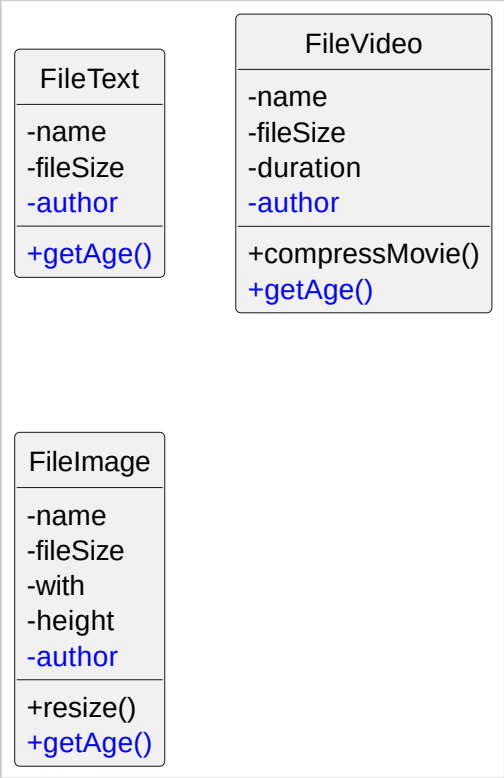
Le principe général en programmation est de "diviser pour régner". Dans notre cas, cela signifie qu’il faut que chaque type de fichier ait sa propre classe :

Voici comment illustrer ce principe :



Maintenant, s'il faut ajouter le type d'affichage (portrait ou paysage) d'une image, il est facile de cibler la classe à faire évoluer sans "polluer" les autres avec des attributs ou méthodes qui leur seraient inutiles.

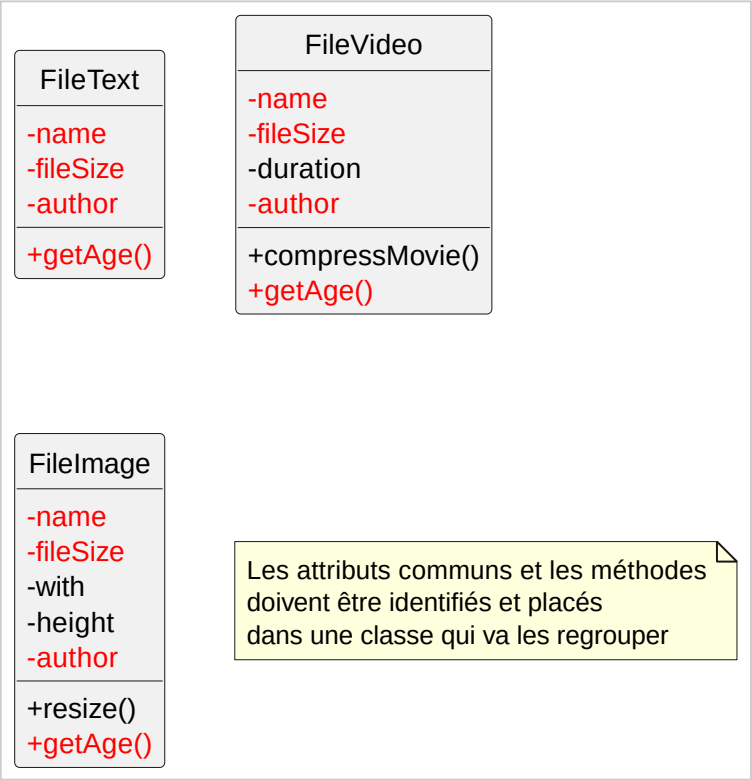
Par contre, si l'on souhaite ajouter l'auteur du fichier, il faut le faire dans les 3 classes. S'il faut ajouter une méthode qui retourne l'âge du fichier, il faut le faire dans chaque classe. Si l'implémentation de cette méthode évolue, il faudra faire la mise à jour dans les 3 classes. Tout cela va vite devenir difficile à maintenir. Toutefois, si l'on passe outre ces remarques, cela donne le diagramme suivant :



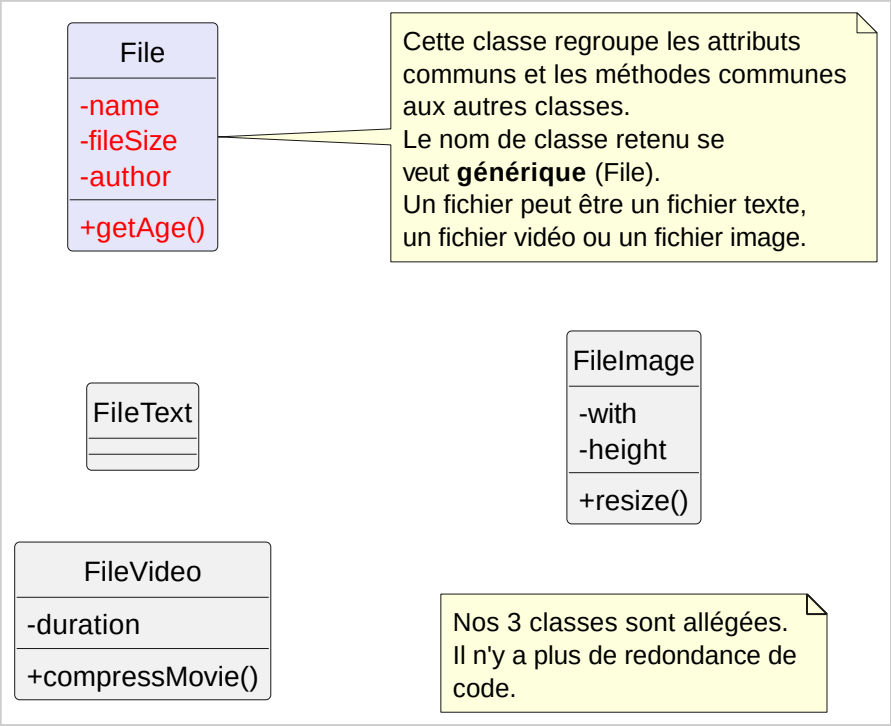
Le problème de cette approche est que nous nous répétons. Il y a un autre grand principe en programmation, c'est le principe DRY (don't repeat yourself ⇒ ne vous répéter pas). Dans le cas présent, ce principe n'est clairement pas respecté.

L'idéal serait de mettre dans une classe les attributs et les méthodes communes aux 3 classes et de laisser ce qui est spécifique dans les classes précédemment créées.

Pour cela il faut repérer tous les attributs communs et les méthodes communes :



Après regroupement :



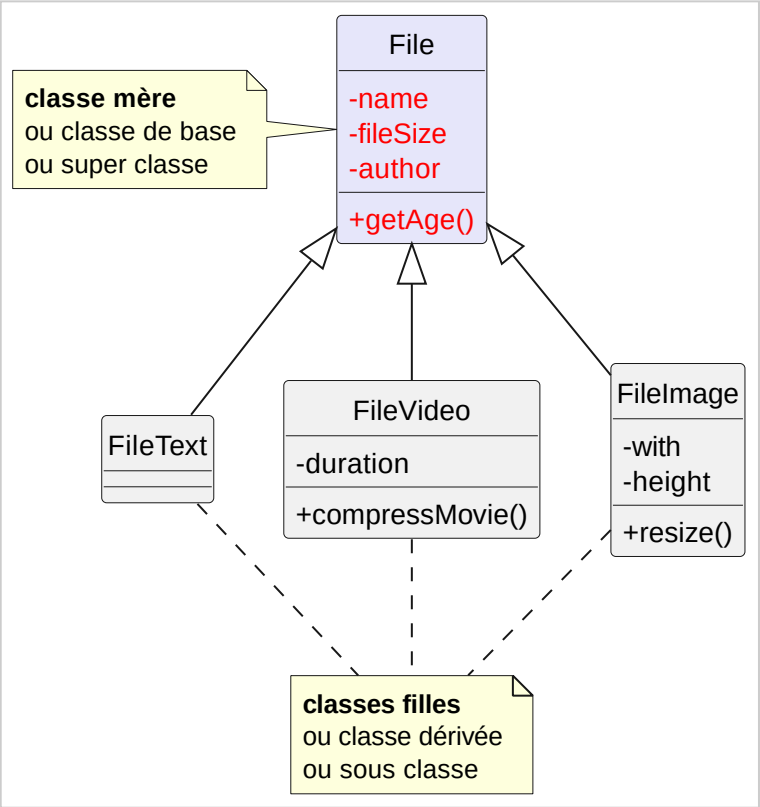
A ce stade, comment savoir que nos 3 classes possèdent les attributs et méthodes de la classe `File` ?

C'est là qu'intervient la **relation d'héritage**.

UML nous permet de dire qu'un fichier texte **est un** fichier, qu'un fichier vidéo **est un** fichier et

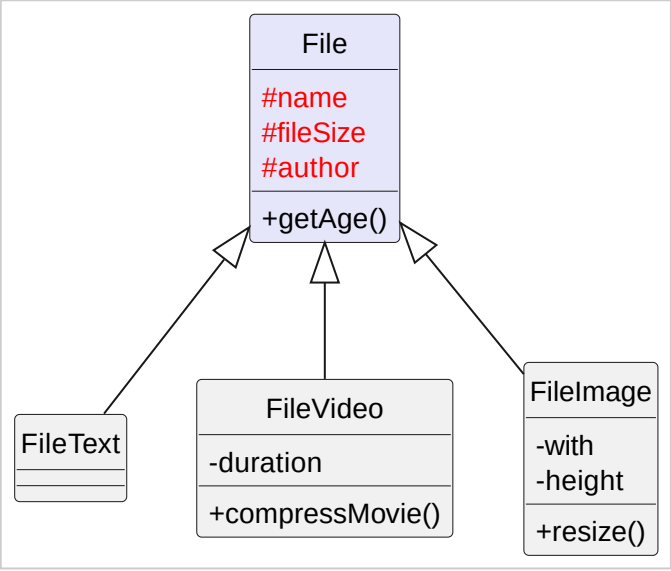
enfin qu'un fichier image **est un** fichier.

Sur le diagramme de classes, cela se traduit par une **flèche du côté de la classe qui regroupe les attributs communs et méthodes communes** :



Il y a un problème majeur dans cette modélisation. Les attributs sont déclarés comme privés. Ils ne peuvent être utilisés à l'extérieur de la classe dans laquelle ils sont déclarés. Pour permettre leur utilisation à l'extérieur, il faut les déclarer comme protégés avec le caractère #. Attention, il ne faut pas les déclarer avec une visibilité publique sinon le principe d'encapsulation n'est plus respecté.

Voici le diagramme corrigé :



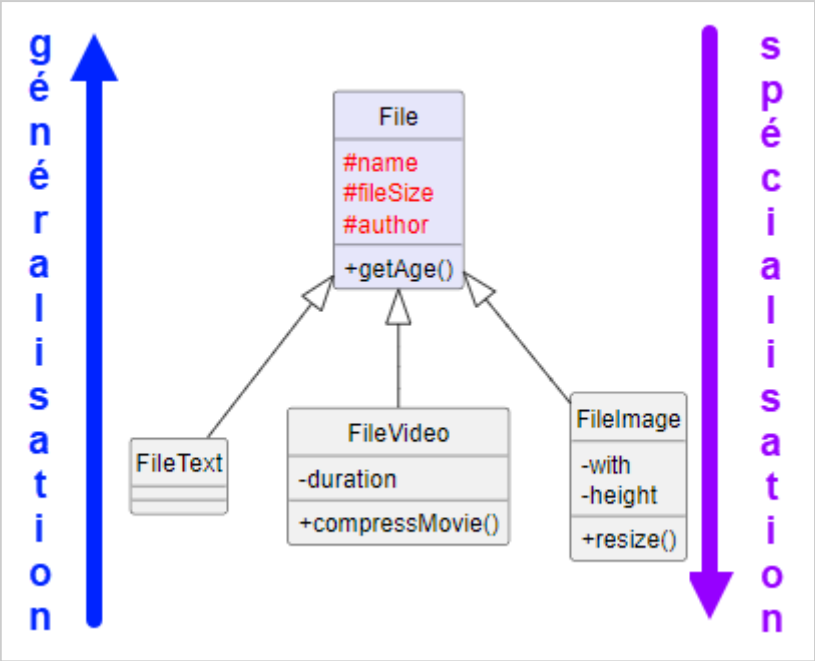
La classe qui regroupe les attributs communs et méthodes communes est appelée la **classe mère** (ou **super classe** ou **classe de base**). La ou les classes qui

contiennent des attributs spécifiques / spécialités sont des **classes filles** (ou **classes dérivées** ou **sous classes**)

Il est dit que **les classes filles héritent de la classe mère**. C'est-à-dire que les classes filles peuvent utiliser les attributs et méthodes de la classe mère si leur visibilité est au moins protégée.

Lorsque l'on remonte d'une classe fille vers la classe mère (donc que l'on factorise les attributs et les méthodes en commun), on parle de **généralisation**.

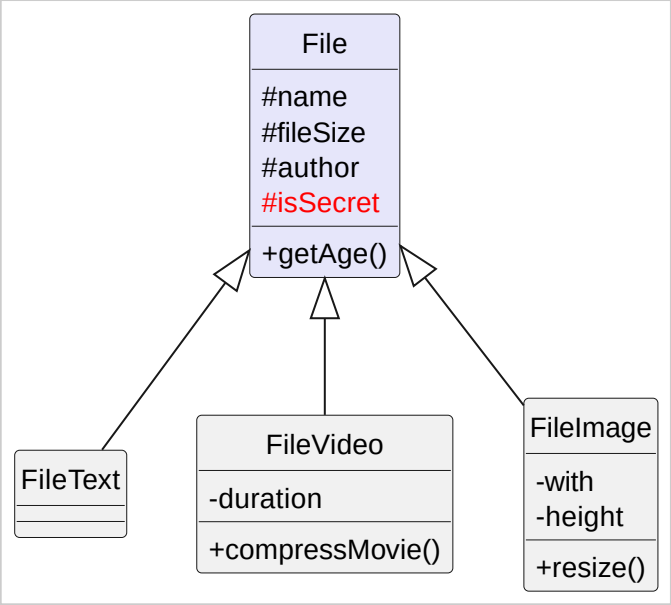
Lorsque l'on descend de la classe mère vers les classes filles (donc lorsque l'on spécialise certains attributs ou méthodes qui ne concernent qu'une classe), on parle de **spécialisation**.



Le gros avantage de l'héritage est de pouvoir faire évoluer rapidement une application.

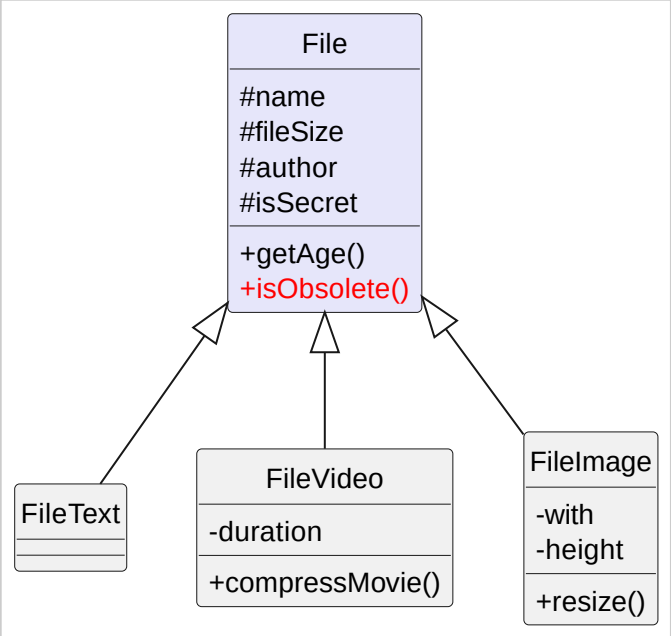
Imaginons que nous ayons besoin de savoir si un fichier est top secret ou pas. Cette caractéristique concerne tous les fichiers. Il n'y a qu'à ajouter cet attribut dans la classe mère et le travail est terminé !

Comme ceci :



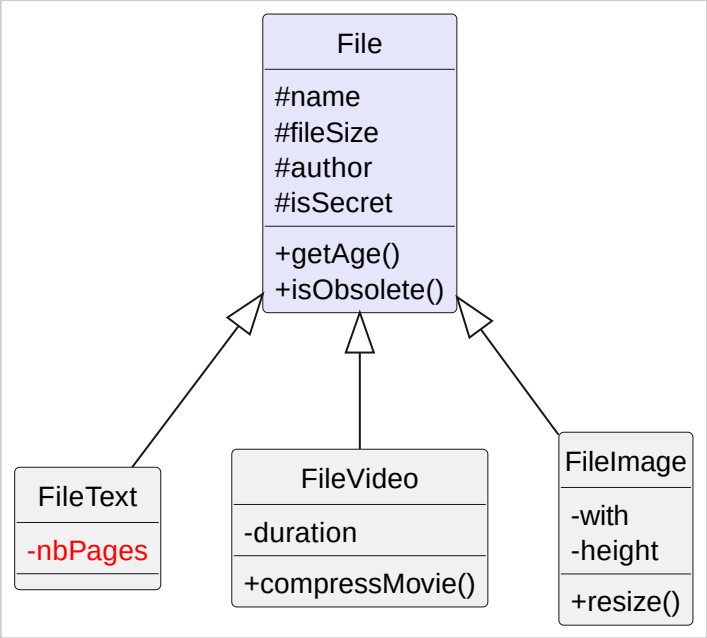
Puisque l'attribut ajoutée l'est dans la classe mère, cela signifie que l'on a fait de la **généralisation**.

Si nous devons ajouter une méthode qui indique que le fichier est peut être obsolète car il a plus de 12 mois, l'ajout d'une méthode `isObsolete()` dans la classe mère suffit pour qu'elle soit utilisable dans les classes filles :



Puisque la méthode ajoutée l'est dans la classe mère, cela signifie que l'on a fait de la **généralisation**.

Si l'on doit ajouter la possibilité qu'un fichier retourne un nombre de pages, cela ne concerne que les fichiers texte. La classe mère n'est pas impactée, seule la classe `FileText` est concernée :



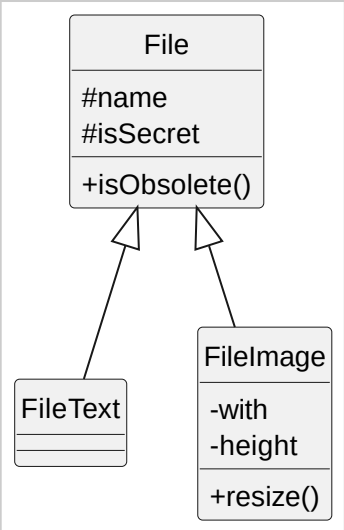
Puisque l'attribut ajouté l'est dans la classe fille, cela signifie que l'on a fait de la **spécialisation**.

1.2. Implémentation de la relation d'héritage

L'implémentation de l'héritage est très simple :

- Il faut implémenter les attributs communs et les méthodes communes aux classes filles (avec une visibilité protégée) dans la classe mère.
- Il faut implémenter chaque classe fille avec leurs attributs spécialisés et méthodes spécialisées.
- Il faut indiquer pour chaque classe fille le nom de la classe mère dont elle hérite

Je simplifie le diagramme pour avoir moins de code à implémenter :



Commençons par la classe mère `File`:

```
class File
{
```

```
    //déclaration des attributs communs
    public function __construct(
        protected string $name,
        protected bool $isSecret = false
    ) {
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }

    public function isSecret(): bool
    {
        return $this->isSecret;
    }

    public function setIsSecret(bool $isSecret): void
    {
        $this->isSecret = $isSecret;
    }


    // méthodes commune
    public function isObsolete(): bool
    {
        //code qui renvoie true si l'age du document est > à 12 mois
        return false; //on décide de renvoyer false de façon arbitraire
    }

}
```

Voici le code de la sous classe `FileText` :

```
class FileText extends File ①
{

    //la classe FileText n'a aucun attribut spécifique ni aucune méthode spécifiques

    //dans certains langages (en C# par exemple), il faut appeler le constructeur
parent depuis la classe fille. En PHP, c'est implicite. ②

    //elle hérite du code de la classe mère (c'est comme si le code de la classe mère
était copié à l'intérieur de la classe fille sauf qu'il n'est pas nécessaire de le
copier !)
```

```
}
```

① Bien indiqué que la classe hérite de `File` via le mot clé `extends` (pour PHP)

② Bien lire l'information à propos du constructeur. Il est inutile de le prévoir car celui de la classe mère sera appelé automatiquement.

Et enfin le code de la sous classe `FileImage` :

```php
class FileImage extends File ①
{

    //on n'ajoute que les attributs spécialisés
    private int $width;
    private int $height;

    public function getWidth(): int
    {
        return $this->width;
    }

    public function setWidth(int $width): void
    {
        $this->width = $width;
    }

    public function getHeight(): int
    {
        return $this->height;
    }

    public function setHeight(int $height): void
    {
        $this->height = $height;
    }

    //ajout de la méthode spécialisée
    public function resize(): bool
    {
        //ici du code qui redimensionne l'image
        return true; //on considère que tout est ok
    }
}
```

① Bien indiqué que la classe hérite de `File` via le mot clé `extends` (pour PHP)

Pour bien, comprendre, nous allons créer un fichier texte et un fichier image. Nous allons ensuite appeler soit des méthodes issues de la classe mère depuis une classe fille, soit appeler des méthodes

spécialisées.

```
//création d'un fichier texte
$fileText = new FileText("monFichierDeDonnees", false);
//nom du fichier (utilisation d'une méthode de la classe mère)
$fileName = $fileText->getName();
//le fichier texte est-il obsolète ?  (idem)
$fileTextIsObsolete = $fileText->isObsolete();

//création d'un fichier image
$fileImage = new FileImage("imageDuMarcheurBlanc", false);

//affectation des dimensions de l'image (utilisation des méthodes de la classe fille)
$fileImage->setHeight(800);
$fileImage->setWidth(600);

//l'image est-elle secrète ? (utilisation d'une méthode de la classe mère)
$imageIsSecret = $fileImage->isSecret();
//on redimensionne l'image (utilisation d'une méthode de la classe mère)
$resizeIsOk = $fileImage->resize();
```

L'autre avantage de l'héritage, c'est que si un nouveau type de fichier qui n'est pas géré par l'application doit être manipulé, on peut le faire grâce à la classe `File` car elle est générique (n'importe quel type de fichier est … un fichier).

Si nous devons gérer un fichier de type exécutable, aucune des sous classes ne correspond. Soit on crée un nouveau sous-type (mais ce doit être une volonté clairement établie), soit on utilise le type `File` qui convient à tous les types de fichiers :

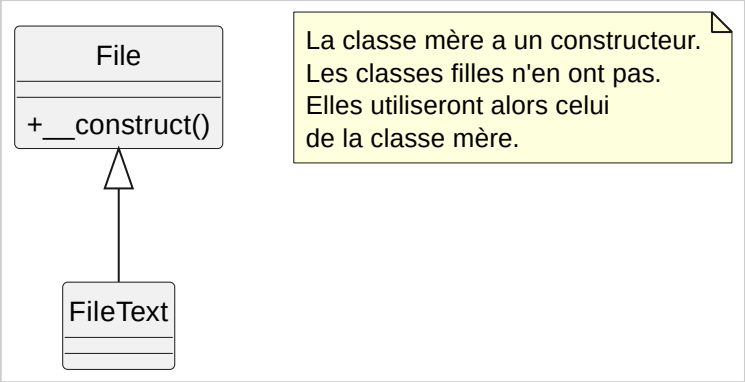
```
//Création d'un fichier exécutable (qui est avant tout un fichier)
$executableFile = new File("unVirus", true);

$fileExeAge = $executableFile->getAge();
$fileExeIsObsolete = $executableFile->isObsolete();
```

Nous voyons tout l'intérêt de pouvoir utiliser la classe mère pour gérer des sous types qui ne sont pas pris en charge par l'application.

1.3. Point technique (en PHP)

Si vous avez compris que depuis une instance d'une classe fille, il est possible d'utiliser un membre de la classe mère (attribut ou méthode), il en va de même pour le constructeur (puisque c'est aussi une méthode).

Prenons le diagramme simplifié suivant :



L'implémentation du diagramme donne le code suivant :

```php
class File
{
    public function __construct()
    {
        echo "Je suis du texte dans le constructeur de la classe File.\n";
    }
}

class FileText extends File
{

}
```

Et voici son utilisation et son rendu :

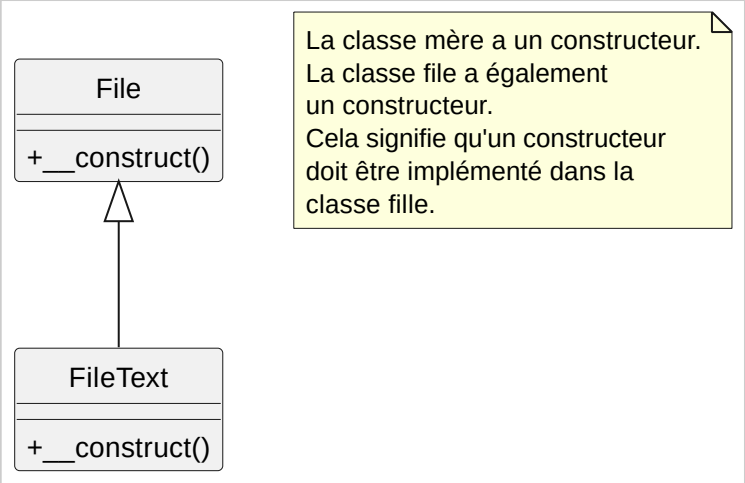
```php
$file = new File();
$fileText = new FileText();
```

La sortie :

```
Je suis du texte dans le constructeur de la classe File.
Je suis du texte dans le constructeur de la classe File.
```

Il apparaît clairement que le constructeur parent (celui de la classe mère) a été utilisé par la classe fille. Elle en a hérité.

Maintenant, ajoutons un constructeur dans la classe fille :




```php
class File
{
    public function __construct()
    {
        echo "Je suis du texte dans le constructeur de la classe File.\n";
    }
}

class FileText extends File
{

    public function __construct()
    {
        echo "Appel du constructeur de la classe FileText.\n";
    }
}
```

Procédons au même appel que tout à l'heure :

```php
$file = new File();
$fileText = new FileText();
```

La sortie :

```
Je suis du texte dans le constructeur de la classe File.
Appel du constructeur de la classe FileText.
```

Ce comportement est utile lorsque la classe fille doit prévoir un comportement différent de celui de la classe mère tout en utilisant le même nom de méthode.

Il est également possible de profiter du comportement de la classe mère et d'ajouter celui de la classe fille :

```php
class File
```

```
2 {
3     public function __construct()
4     {
5         echo "Je suis du texte dans le constructeur de la classe File.\n";
6     }
7 }
8
9 class FileText extends File
10 {
11
12     public function __construct()
13     {
14         echo "Appel du constructeur de la classe FileText.\n";
15         //appel du constructeur parent
16         parent::__construct(); ①
17     }
18 }
```

① le constructeur parent est appelé depuis le constructeur de la classe fille.

Notre exemple utilise le constructeur mais les remarques sont les mêmes pour n'importe quelle méthode. Il est possible de surcharger une méthode dans la classe fille pour remplacer celle de la classe mère. Il est possible de faire appel à la méthode de la classe mère (via `parent::nomMethode`) et de la "compléter" depuis la méthode fille.

Lorsqu'une méthode "fille" surcharge une méthode mère, il est indispensable de respecter le principe de substitution de Liskov. Pour faire simple, ce principe préconise que la signature de la méthode "fille" qui surcharge la méthode "mère" soit compatible. Vous pouvez lire la documentation au sujet des règles de compatibilité de signature. La signature d'une méthode est défini par le type de ses paramètres, leur nombre et le type de son retour.

Une classe (en PHP) ne peut hériter que d'une seule classe. L'héritage multiple n'est donc pas possible.

L'objectif n'est pas de faire un cours sur la programmation orientée objet. Effectivement, il reste bien des aspects à aborder concernant l'héritage et les aspects techniques qui en découlent.

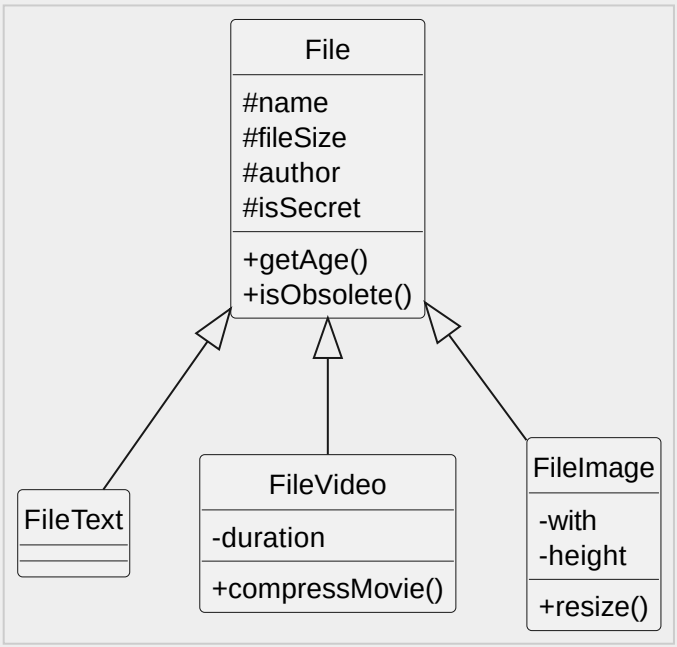
1.4. Quelques exercices

Q1) Faites évoluer le diagramme ci-dessous de façon à prendre compte les évolutions suivantes :

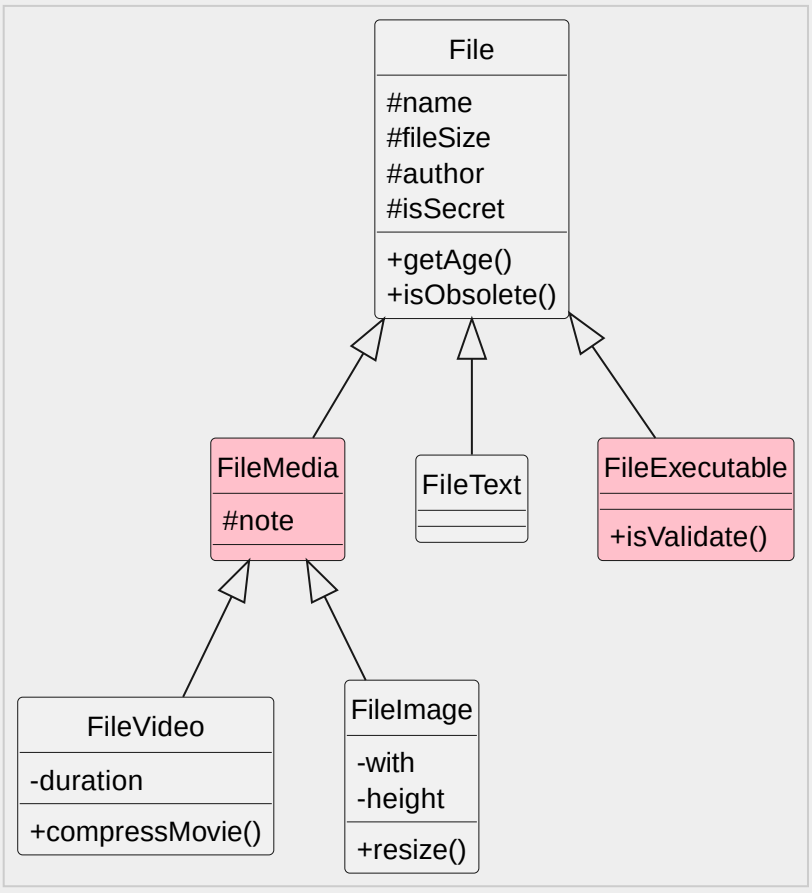
- l'application doit prendre en compte un nouveau sous-type pour les fichiers exécutables. Un fichier exécutable doit disposer d'une méthode `isValidate` qui indique qu'il ne s'agit pas d'un virus.

- l'application doit permettre de gérer des fichiers de type "média". Ces fichiers sont des fichiers vidéos, des images, des fichiers audio, etc (il n'est pas attendu de gérer un type "audio"). Un fichier media doit pouvoir être noté avec une note chiffrée.

Diagramme de départ :



Correction de Q1



Q2) Implémentez le code des classes qui apparaissent en couleur dans la correction du point précédent sans oublier les éventuelles modifications à apporter aux autres classes.

Correction de Q2

- Il faut implémenter le code des classes `FileExecutable` et `FileMedia`
- Il faut modifier la relation d'héritage des classes `FileVideo` et `FileImage`.

```
class FileExecutable extends File
{
    public function isValidate(){
        //code qui valide le fichier ou l'invalide
    }
}
```

```
class FileMedia extends File
{
    protected ?int $note = null;
}
```

```
//modification de la déclaration de la classe FileVideo
class FileVideo extends FileMedia { ①
... ②
}

//modification de la déclaration de la classe FileImage
class FileImage extends FileMedia { ①
... ②
}
```

① bien faire attention à hériter de `FileMedia` et non de `File`

② le code de la classe qui hérite ne change pas

Q3) Répondre aux questions qui suivent à partir du diagramme qui a été implémenté à la question précédente

a. Quelle classe sera instanciée pour manipuler un fichier csv ?

b. Quelle classe sera instanciée pour manipuler un fichier audio ?

c. Quelle classe sera instanciée pour manipuler un fichier de type "archive" (.zip, .rar, .jar, ...) ?

Correction de Q3

a. Pour manipuler un fichier csv (qui est un fichier texte), il faut instancier la classe `FileText`.

b. Pour manipuler un fichier audio (qui est un fichier média), il faut instancier la classe `FileMedia` car il n'existe pas de sous-type spécial "audio".

c. Pour manipuler un fichier archive, il faut instancier la classe `File` car il n'existe pas de sous-type correspondant.

2. La relation abstraite (classe abstraite)

2.1. Comprendre le sens de l'adjectif "abstrait"

La relation abstraite reprend les principes de la relation d'héritage :

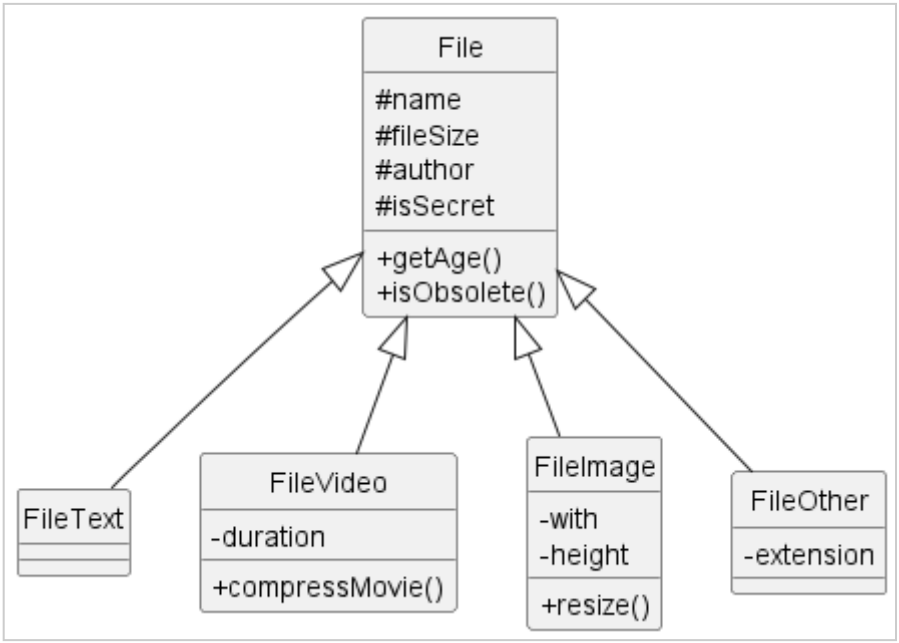
- Les membres communs (attributs et méthodes) sont généralisés dans une classe mère.
- Les membres spécifiques (attributs et méthodes) aux sous classes sont spécifiés dans ces dernières.

La difficulté majeure pour comprendre cette partie de cours est la connaissance du mot "abstrait". Comprendre ce terme aide clairement à comprendre ce que cela implique.

Une "chose" abstraite n'a pas d'existence dans la réalité.

Une classe abstraite est une classe dont les instances n'ont pas de sens dans la réalité. Cela signifie qu'il n'y a aucun intérêt à instancier la classe car les "objets" instanciés ne seront jamais utilisés. **Une classe abstraite est donc une classe qui ne peut pas être instanciée. Elle est obligatoirement héritée.**

Pour bien comprendre ce concept, nous allons partir du diagramme suivant :



La classe mère est la classe `File`. Les sous-classes héritent des membres de la classe mère (s'ils ne sont pas privés bien sûr).

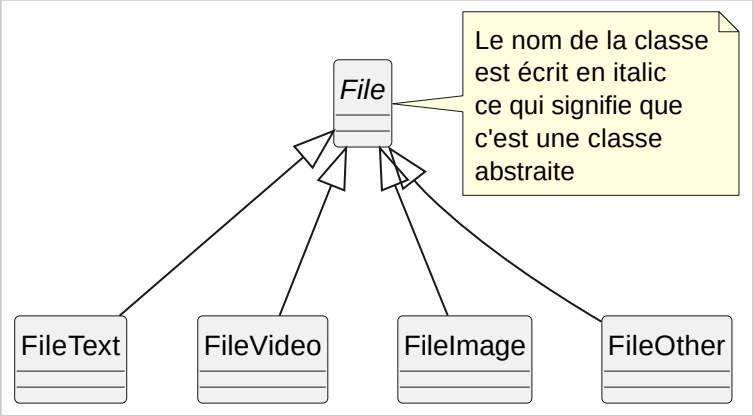
Nous avons 4 sous-types (un pour le texte, un pour la vidéo, un pour les images et un pour tous les autres types de fichiers). Cela signifie que si un fichier n'est pas du texte, une vidéo ou une image, ce sera forcément un fichier de type "autre" `FileOther`. Le type `File` est donc totalement inutile. Cette classe ne sera jamais instanciée ! Son existence n'est là que pour généraliser des membres communs aux sous-types. Puisqu'il n'y aura jamais d'instance de `File`, on peut affirmer que le type `File` ne correspond à rien de réel, c'est un type abstrait.

Lorsqu'une classe n'est pas abstraite, on parle de classe concrète.

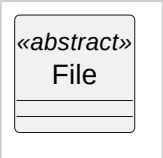
Une classe concrète est une classe qui peut être instanciée. Les instances représentent un objet du réel.

2.2. Représentation UML d'une classe abstraite

Le nom d'une classe abstraite est écrit en italique :



Il est également possible de spécifier que la classe est abstraite grâce à son stéréotype :



La flèche qui représente le lien d'héritage reste inchangée.

2.3. Implémentation d'une classe abstraite

L'implémentation d'une classe abstraite est très simple, il suffit (en PHP) de préfixer la déclaration de celle-ci avec le mot `abstract` :

```php
<?php

abstract class File { ①
    //ici les membres (attributs et méthodes)
}

class FileText extends File { ②
    //membres spécifiques au sous-type (attributs et méthodes)
}
class FileVideo extends File { ②
    //membres spécifiques au sous-type (attributs et méthodes)
}
class FileImage extends File { ②
    //membres spécifiques au sous-type (attributs et méthodes)
}
```

```
16 class FileOther extends File { ②
17     //membres spécifiques au sous-type (attributs et méthodes)
18 }
```

① Le mot `abstract` permet de rendre la classe abstraite.

② Que l'héritage provienne d'une classe concrète ou abstraite, le mot `extends` est à utiliser dans les deux cas dans les classes filles.

Comme dit précédemment, une classe abstraite ne peut pas être instanciée mais si c'est tout de même la cas ...

```
1 $file = new File();
```

... une erreur fatale sera générée :

```
Fatal error: Uncaught Error: Cannot instantiate abstract class File in ...
```

2.4. Les méthodes aussi peuvent être abstraites

L'héritage à pour objectif de factoriser des membres au sein d'une classe afin que d'autres puissent les utiliser.

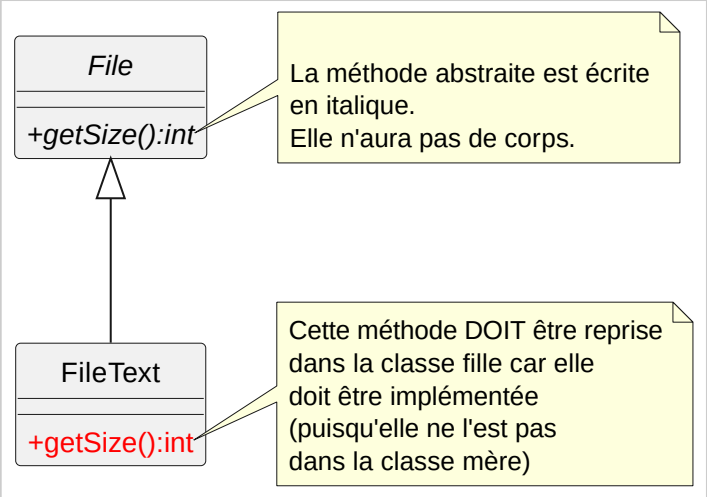
Puisqu'une classe abstraite ne peut pas être instanciée, il est tout à fait possible de prévoir une méthode qui n'a aucun corps (même pas les accolades qui marquent le début et la fin de la méthode !). Cette méthode est une méthode abstraite . Cela ne pose aucun problème puisqu'il ne sera pas possible d'appeler cette méthode puisque la classe qui la contient est abstraite.

Mais quelle utilité peut avoir une méthode sans corps me direz-vous !

N'oubliez pas que nous sommes dans un contexte d'héritage.

Une classe abstraite est forcément prévue pour être héritée (sans quoi elle n'a absolument aucune utilité). Si une classe hérite d'une classe abstraite qui contient une méthode abstraite, cela signifie que la classe fille hérite de cette méthode sans corps. Mais comme la classe fille est concrète, celle-ci doit prévoir le corps de la méthode abstraite de la mère puisqu'il sera possible de l'appeler depuis la classe fille.

Nous allons illustrer ce concept avec ce diagramme :



Voyons ce que cela donne au niveau du code :

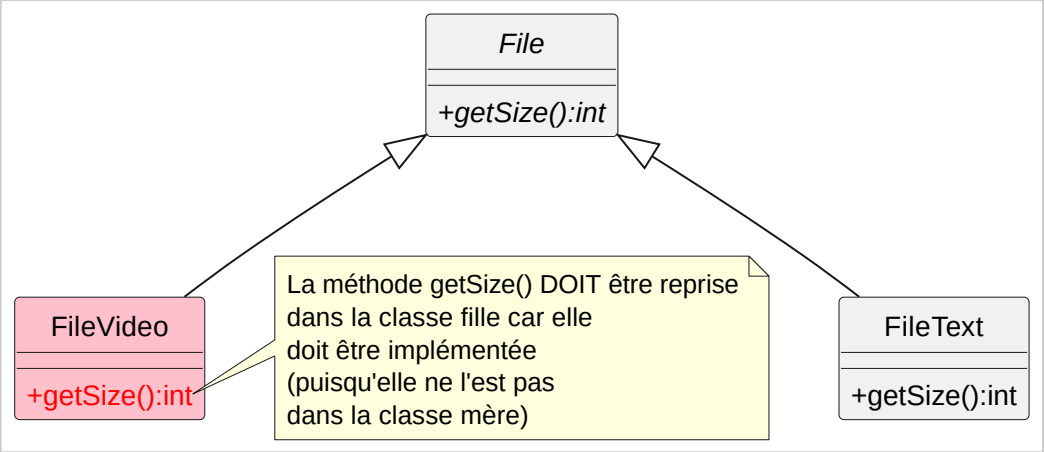
```php
abstract class File {
    //ici les membres "factorisés" de la classe

    //méthode abstraite
    abstract public function getSize():int; ①
}

class FileText extends File {
    //ici les membres spécifiques de FileText

    //l'implémentation de la méthode abstraite (qui est obligatoire)
    public function getSize():int { ②
        //ici le code qui retourne un entier.
    }
}
```

① La méthode abstraite est déclaré avec le mot `abstract` comme c'est le cas pour la classe. La méthode n'a pas de corps, donc pas d'accolade.

② La méthode abstraite est obligatoirement implémentée dans la classe fille. Elle n'a pas le choix !

Une méthode abstraite est très pratique lorsque l'on souhaite "forcer" les classes filles à implémenter une méthode particulière avec des arguments et des retours dont le type est imposé. Celle-ci permet de prévoir les contraintes à respecter par les classes filles, c'est-à-dire la signature de la méthode.

Imaginons que notre diagramme évolue et qu'une nouvelle classe hérite de la classe `File` :



Dès lors, la classe `FileVideo` doit obligatoirement prévoir la méthode `getSize()`

```php
class FileVideo extends File {
    //ici les membres spécifique à FileVideo

    //obligation d'implémenter les méthodes abstraites de la classe mère
    public function getSize():int {
        //ici l'implémentation
    }
}
```

Petite précision avant de clore cette partie :

J'ai précisé tout à l'heure qu'il était possible de prévoir une méthode abstraite dans une classe abstraite.

Il faut également faire le raisonnement inverse :
Dès lors qu'il y a une méthode abstraite dans une classe, cette dernière doit également être abstraite (sans quoi, on pourrait tenter d'utiliser la méthode abstraite ce qui n'est pas possible !)

Index

A

abstraite, 20

C

classe abstraite, 17
classe concrète, 18
classe de base, 4
classe mère, 4
classes dérivées, 5
classes filles, 5

G

généralisation, 5

H

https://fr.wikipedia.org/wiki/Principe_de_substitution_de_Liskov[principe de substitution de Liskov], 13
https://www.php.net/manual/fr/language.oop5.basic.php#language.oop.lsp[règles de compatibilité de signature], 13

L

la signature de la méthode, 20

M

méthode abstraite, 19

R

relation d'héritage, 1

S

signature d'une méthode, 13
sous classes, 5
spécialisation, 5
super classe, 4
surcharger une méthode, 13