

Les classes abstraites et les interfaces

Issu du cours Apprenez à programmer en Java, partie 5 d'OpenClassRoom <https://openclassrooms.com/courses/26832-apprenez-a-programmer-en-java/21973-les-classes-abstraites-et-les-interfaces>

Nous voilà de retour avec deux fondements du langage Java. Je vais essayer de faire simple : derrière ces deux notions se cache la manière dont Java vous permet de structurer votre programme.

Grâce aux chapitres précédents, vous vous rendez compte que vos programmes Java regorgeront de classes, avec de l'héritage, des dépendances, de la composition... Afin de bien structurer vos programmes (on parle d'*architecture logicielle*), vous allez vous creuser les méninges pour savoir où ranger des comportements d'objets :

- dans la classe mère ?
- dans la classe fille ?

Comment obtenir une structure assez souple pour pallier les problèmes de programmation les plus courants ?

La réponse est dans ce chapitre.

Les classes abstraites

Une classe **abstraite** est quasiment identique à une classe normale. Oui, identique aux classes que vous avez maintenant l'habitude de coder. Cela dit, elle a tout de même une particularité : *vous ne pouvez pas l'instancier* ! Vous avez bien lu. Imaginons que nous ayons une classe déclarée **abstraite**. Voici un code qui ne compilera pas :

```
public class Test{

    public static void main(String[] args){

        A obj = new A(); //Erreur de compilation !

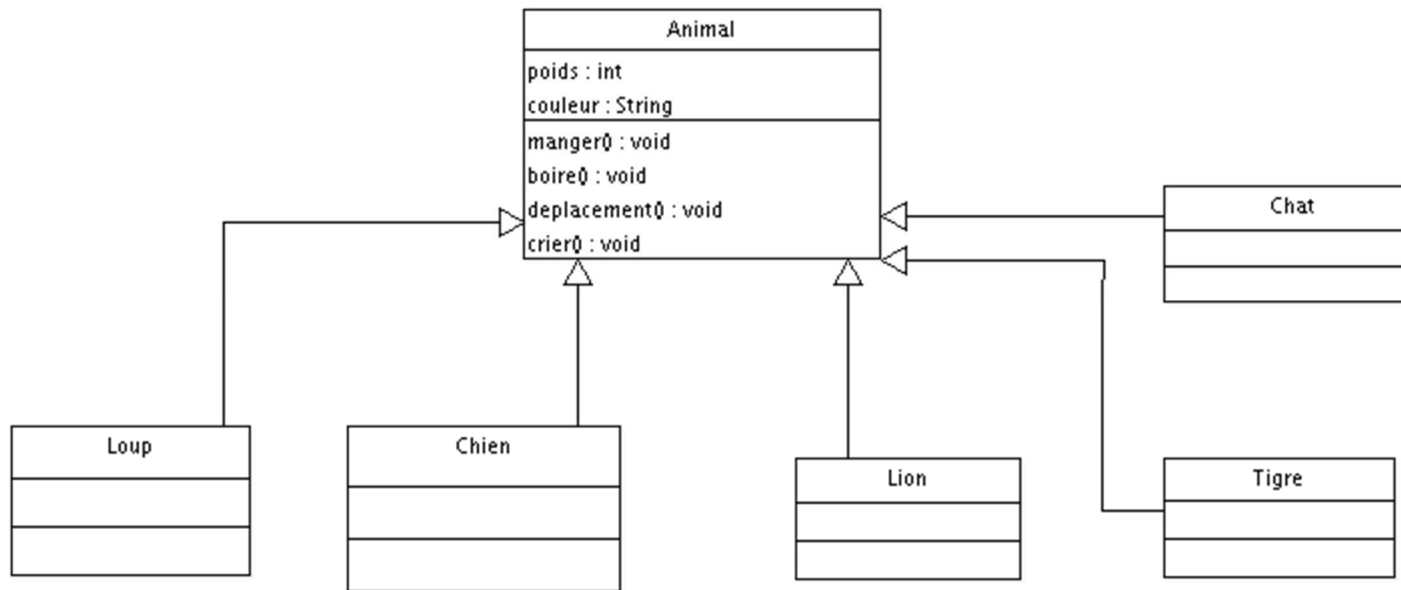
    }

}
```

Pour bien en comprendre l'utilité, il vous faut un exemple de situation (de programme, en fait) qui le requiert. Imaginez que vous êtes en train de réaliser un programme qui gère différents types d'animaux (oui, je sais : l'exemple est bête, mais il a le mérite d'être simple à comprendre).

Dans ce programme, vous aurez des loups, des chiens, des chats, des lions et des tigres. Mais vous n'allez tout de même pas faire toutes vos classes bêtement : il va de soi que tous ces animaux ont des points communs ! Et qui dit points communs dit *héritage*. Que pouvons-nous définir de commun à tous ces animaux ? Le fait qu'ils aient une couleur, un poids, un cri, une façon de se déplacer, qu'ils mangent et boivent quelque chose.

Nous pouvons donc créer une classe mère : appelons-la `Animal`. Avec ce que nous avons dégagé de commun, nous pouvons lui définir des attributs et des méthodes. La figure suivante représente nos classes.



Classe Animal

Nous avons bien notre classe mère `Animal` et nos animaux qui en héritent. À présent, laissez-moi vous poser une question. Vu que notre classe `Animal` est `public`, qu'est censé faire un objet `Animal`? Quel est son poids, sa couleur, que mange-t-il ? Je sais, cela fait plus qu'une question. 🤔

Si nous avons un morceau de code qui ressemble à ceci :

```

public class Test{

    public static void main(String[] args){

        Animal ani = new Animal();

        ((Loup)ani).manger(); //Que doit-il faire ?

    }

}
  
```

Personnellement, je ne sais pas ce que mange un objet `Animal`. Vous conviendrez que toutes les classes ne sont pas bonnes à être instanciées !

C'est là qu'entrent en jeu nos classes abstraites. En fait, ces classes servent à définir une *superclasse* : par là, vous pouvez comprendre qu'elles servent essentiellement à créer un nouveau type d'objets. Voyons maintenant comment créer une telle classe.

Une classe `Animal` très abstraite

En fait, il existe une règle pour qu'une classe soit considérée comme abstraite. Elle doit être déclarée avec le mot clé `abstract`. Voici un exemple illustrant mes dires :

```

abstract class Animal{ }
  
```

Une telle classe peut contenir la même chose qu'une classe normale. Ses enfants pourront utiliser tous ses éléments déclarés (attributs et méthodes déclarés `public` ou `protected`, nous sommes

d'accord). Cependant, ce type de classe permet de définir des méthodes abstraites qui présentent une particularité : elle n'ont pas de corps ! En voici un exemple :

```
abstract class Animal{  
  
    abstract void manger(); //Une méthode abstraite  
  
}
```

Vous voyez pourquoi on dit « méthode abstraite » : difficile de voir ce que cette méthode sait faire.

Retenez bien qu'une méthode abstraite n'est composée que de l'en-tête de la méthode suivie d'un point-virgule «;».

Il faut que vous sachiez qu'une méthode abstraite ne peut exister que dans une classe abstraite. Si, dans une classe, vous avez une méthode déclarée abstraite, vous devez déclarer cette classe comme étant abstraite.

Voyons à quoi cela peut servir. Vous avez vu les avantages de l'héritage et du polymorphisme. Eh bien nos classes enfants hériteront aussi des méthodes abstraites, mais étant donné que celles-ci n'ont pas de corps, nos classes enfants seront obligées de redéfinir ces méthodes ! Elles présentent donc des méthodes polymorphes, ce qui implique que la covariance des variables pointe à nouveau le bout de son nez :

```
public class Test{  
  
    public static void main(String args[]){  
  
        Animal loup = new Loup();  
  
        Animal chien = new Chien();  
  
        loup.manger();  
  
        chien.crier();  
  
    }  
  
}
```

Attends ! Tu nous as dit qu'on ne pouvait pas instancier de classe abstraite !

Et je maintiens mes dires : nous n'avons pas instancié notre classe abstraite. Nous avons instancié un objetLoupque nous avons mis dans un objet de typeAnimal(il en va de même pour l'instanciation de la classeChien). Vous devez vous rappeler que l'instance se crée avec le mot clénew. En aucun cas, le fait de déclarer une variable d'un type de classe donné – ici,Animal – n'est une instanciation ! Ici, nous instancions unLoupet unChien.

Vous pouvez aussi utiliser une variable de typeObjectcomme référence à un objetLoup, à un objetChienetc. Vous saviez déjà que ce code fonctionne :

```
public class Test{
```

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

    Object obj = new Loup();

    ((Loup)obj).manger();

}

}
```

En revanche, ceci pose problème :

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

    Object obj = new Loup();

    Loup l = obj; //Problème de référence

}
```

Eh oui ! Nous essayons de mettre une référence de type `Object` dans une référence de type `Loup` : pour avertir la JVM que la référence que vous voulez affecter à votre objet de type `Loup` est un `Loup`, vous devez utiliser le transtypage ! Revoyons notre code :

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

    Object obj = new Loup();

    Loup l = (Loup)obj;

    //Vous prévenez la JVM que la référence que vous passez est de type Loup.

}
```

Vous pouvez bien évidemment instancier directement un objet `Loup`, un objet `Chien`, etc.

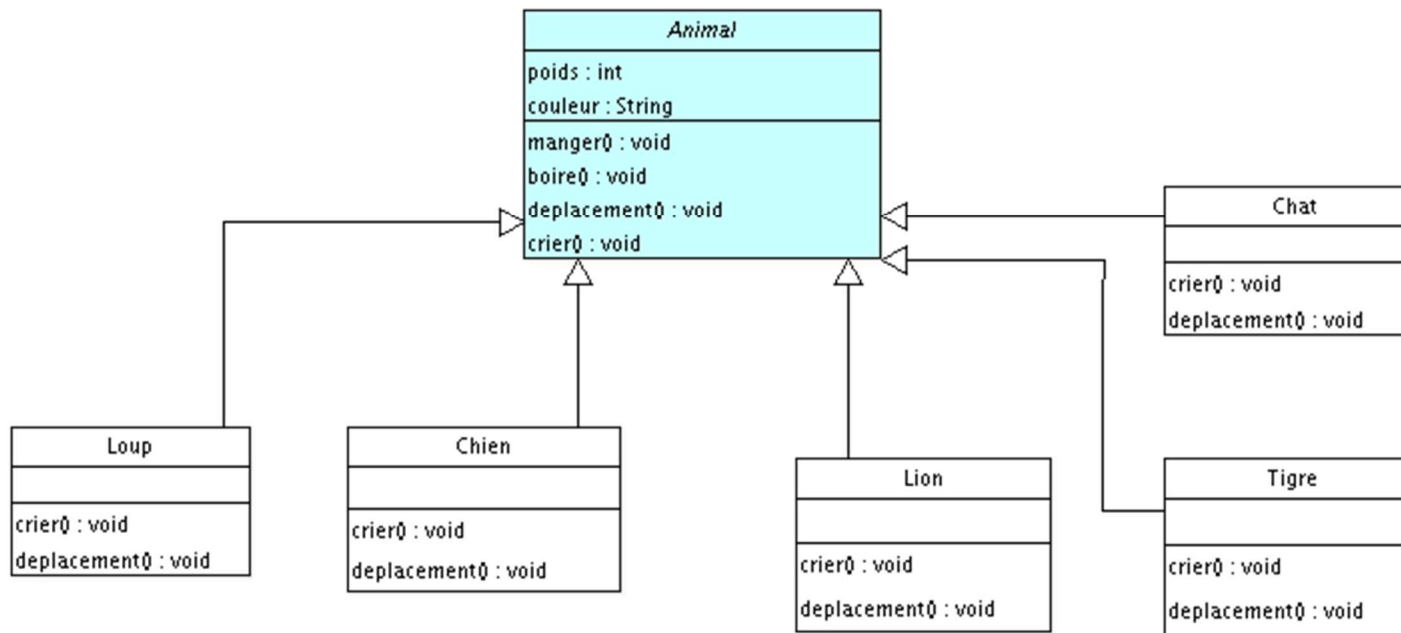
Pour le moment, nous n'avons de code dans aucune classe ! Les exemples que je vous ai fournis ne font rien du tout, mais ils fonctionneront lorsque nous aurons ajouté des morceaux de code à nos classes.

Étoffons notre exemple

Nous allons donc ajouter des morceaux de code à nos classes. Tout d'abord, établissons un bilan de ce que nous savons :

- Nos objets seront probablement tous de couleur et de poids différents. Nos classes auront donc le droit de modifier ceux-ci.
- Ici, nous partons du principe que tous nos animaux mangent de la viande. La méthode `manger()` sera donc définie dans la classe `Animal`.
- Idem pour la méthode `boire()`. Ils boiront tous de l'eau (je vous voyais venir).
- Ils ne crieront pas et ne se déplaceront pas de la même manière. Nous emploierons donc des méthodes polymorphes et déclarerons les méthodes `deplacement()` et `crier()` abstraites dans la classe `Animal`.

La figure suivante représente le diagramme des classes de nos futurs objets. Ce diagramme permet de voir si une classe est abstraite : son nom est alors en italique.

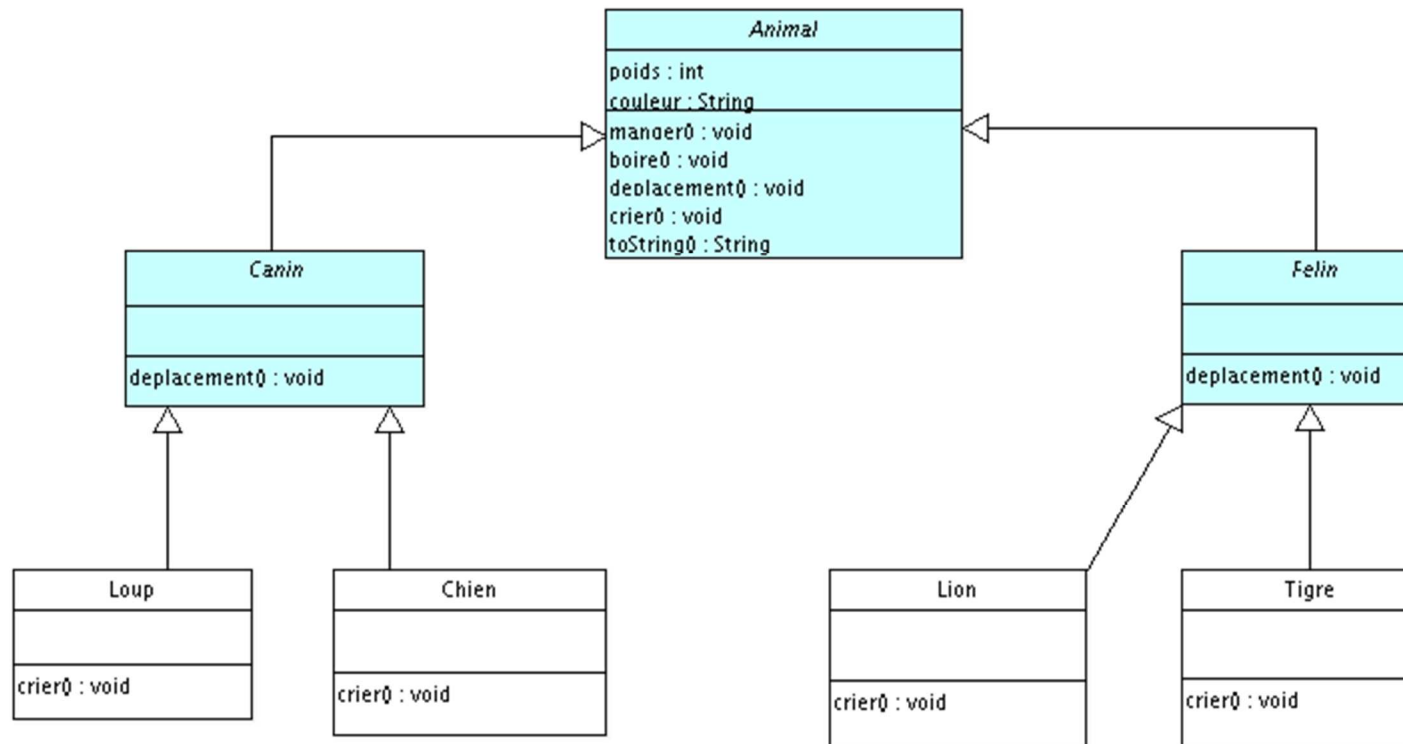


Hiérarchie de nos classes

Nous voyons bien que notre classe `Animal` est déclarée abstraite et que nos classes filles héritent de celle-ci. De plus, nos classes filles ne redéfinissent que deux méthodes sur quatre, on en conclut donc que ces deux méthodes doivent être abstraites. Nous ajouterons deux constructeurs à nos classes filles : un par défaut et un autre comprenant les deux paramètres d'initialisation. À cela, nous ajouterons aussi les accesseurs d'usage. Mais dites donc... nous pouvons améliorer un peu cette architecture, sans pour autant rentrer dans les détails !

Vu les animaux présents, nous aurions pu faire une sous-classe `Carnivore`, ou encore `AnimalDomestique` et `AnimalSauvage`... Ici, nous allons nous contenter de faire deux sous-classes : `Canin` et `Felin`, qui hériteront d'`Animal` et dont nos objets eux-mêmes hériteront !

Nous allons redéfinir la méthode `deplacement()` dans cette classe, car nous allons partir du principe que les félins se déplacent d'une certaine façon et les canins d'une autre. Avec cet exemple, nous réviserons le polymorphisme. La figure suivante correspond à notre diagramme mis à jour (vous avez remarqué ? J'ai ajouté une méthode `toString()`).



Nouvelle architecture des classes
Voici les codes Java correspondants.

Animal.java

```
abstract class Animal {

    protected String couleur;

    protected int poids;

    protected void manger(){

        System.out.println("Je mange de la viande.");

    }

    protected void boire(){

        System.out.println("Je bois de l'eau !");

    }

}
```

```

    abstract void deplacement();

    abstract void crier();

    public String toString(){

        String str = "Je suis un objet de la " + this.getClass() + ", je suis " + this.couleur
+ ", je pèse " + this.poids;

        return str;

    }

}

```

Felin.java

```

public abstract class Felin extends Animal {

    void deplacement() {

        System.out.println("Je me déplace seul !");

    }

}

```

Canin.java

```

public abstract class Canin extends Animal {

    void deplacement() {

        System.out.println("Je me déplace en meute !");

    }

}

```

Chien.java

```

public class Chien extends Canin {

    public Chien(){

```

```
}

public Chien(String couleur, int poids){

    this.couleur = couleur;

    this.poids = poids;

}

void crier() {

    System.out.println("J'aboie sans raison !");

}

}
```

Loup.java

```
public class Loup extends Canin {

    public Loup(){

    }

    public Loup(String couleur, int poids){

        this.couleur = couleur;

        this.poids = poids;

    }

    void crier() {
```



```
        System.out.println("Je hurle à la Lune en faisant ouhouh !");  
  
    }  
  
}
```

Lion.java

```
public class Lion extends Felin {  
  
    public Lion(){  
  
    }  
  
    public Lion(String couleur, int poids){  
  
        this.couleur = couleur;  
  
        this.poids = poids;  
  
    }  
  
    void crier() {  
  
        System.out.println("Je rugis dans la savane !");  
  
    }  
  
}
```

Tigre.java

```
public class Tigre extends Felin {  
  
    public Tigre(){  
  
    }  
  
}
```

```

public Tigre(String couleur, int poids){

    this.couleur = couleur;

    this.poids = poids;

}

void crier() {

    System.out.println("Je grogne très fort !");

}

}

```

Chat.java

```

public class Chat extends Felin {

    public Chat(){

    }

    public Chat(String couleur, int poids){

        this.couleur = couleur;

        this.poids = poids;

    }

    void crier() {

        System.out.println("Je miaule sur les toits !");

    }

}

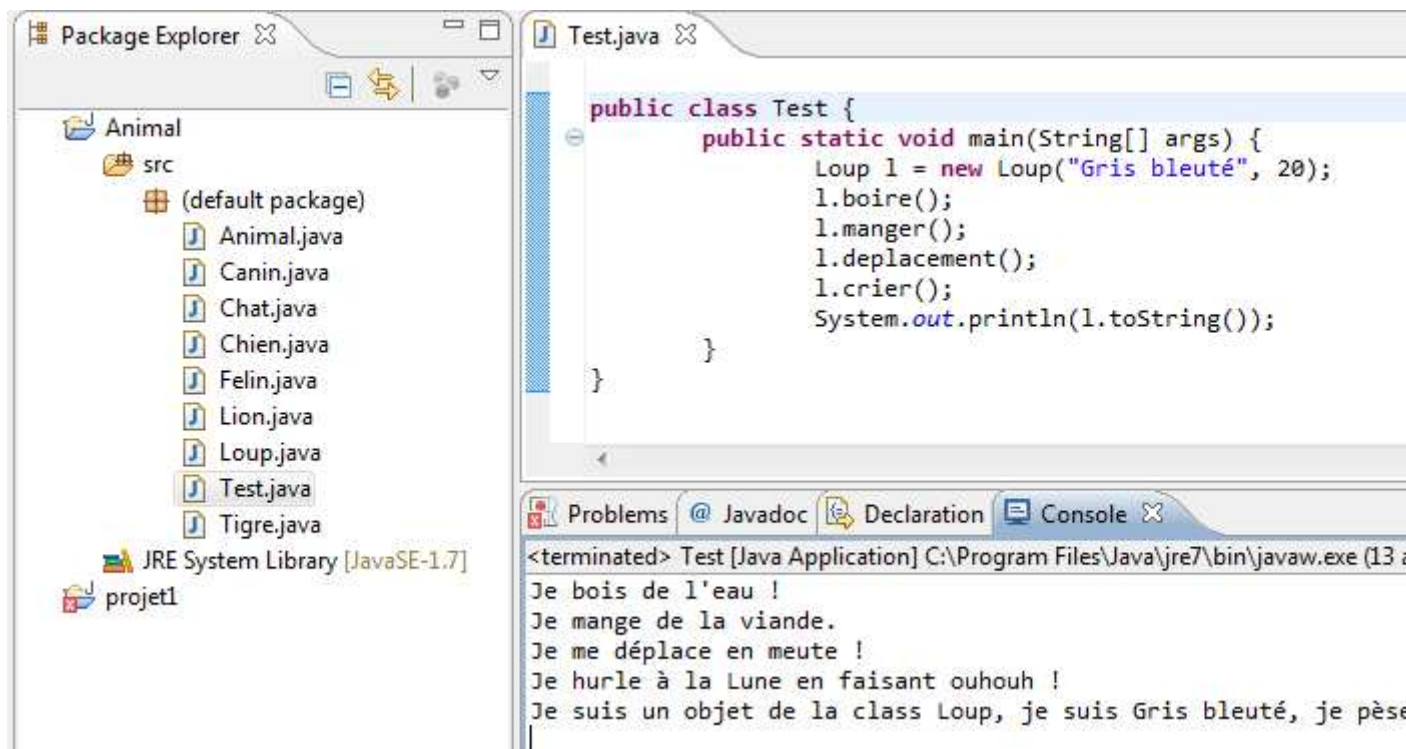
```

Dis donc ! Une classe abstraite ne doit-elle pas comporter une méthode abstraite ?

Je n'ai jamais dit ça ! Une classe déclarée abstraite n'est pas « instanciable », mais rien ne l'oblige à comprendre des méthodes abstraites. En revanche, une classe contenant une méthode abstraite doit être déclarée abstraite ! Je vous invite maintenant à faire des tests :

```
public class Test {  
  
    public static void main(String[] args) {  
  
        Loup l = new Loup("Gris bleuté", 20);  
  
        l.boire();  
  
        l.manger();  
  
        l.deplacement();  
  
        l.crier();  
  
        System.out.println(l.toString());  
  
    }  
}
```

Le jeu d'essai de ce code correspond à la figure suivante.



Test d'une classe abstraite

Dans la méthode `toString()` de la classe `Animal`, j'ai utilisé la méthode `getClass()` qui — je vous le donne en mille — se trouve dans la classe `Object`. Celle-ci retourne «class <nom de la classe>».

Dans cet exemple, nous pouvons constater que nous avons un objet `Loup`:

- À l'appel de la méthode `boire()`: l'objet appelle la méthode de la classe `Animal`.
- À l'appel de la méthode `manger()`: idem.
- À l'appel de la méthode `toString()`: idem.
- À l'appel de la méthode `deplacement()`: c'est la méthode de la classe `Canin` qui est invoquée ici.
- À l'appel de la méthode `crier()`: c'est la méthode de la classe `Loup` qui est appelée.

Remplacez le type de référence (ici, `Loup`) par `Animal`, essayez avec des objets `Chien`, etc. Vous verrez que tout fonctionne.

Les interfaces

L'un des atouts majeurs — pour ne pas dire l'atout majeur — de la programmation orientée objet est la *réutilisabilité* de vos objets. Il est très commode d'utiliser un objet (voire une architecture) que nous avons déjà créé pour une nouvelle application.

Admettons que l'architecture que nous avons développée dans les chapitres précédents forme une bonne base. Que se passerait-il si un autre développeur vous demandait d'utiliser vos objets dans un autre type d'application ? Ici, nous ne nous sommes occupés que de l'aspect générique des animaux que nous avons créés. Cependant, la personne qui vous a contacté, elle, développe une application pour un chenil.

La contrainte principale, c'est que vos chiens devront apprendre à faire de nouvelles choses telles que :

- faire le beau ;
- faire des câlins ;
- faire une « léchouille ».

Je ne vois pas le problème... Tu n'as qu'à ajouter ces méthodes dans la classe `Animal` !

Ouh là ! Vous vous rendez compte que vous obtiendrez des lions qui auront la possibilité de faire le beau ? Dans ce cas, on n'a qu'à mettre ces méthodes dans la classe `Chien`, mais j'y vois deux problèmes :

1. vous allez devoir mettre en place une convention de nommage entre le programmeur qui va utiliser vos objets et vous. Vous ne pourrez pas utiliser la méthode `faireCalin()`, alors que le programmeur oui ;
2. si vous faites cela, adieu au polymorphisme ! Vous ne pourrez pas appeler vos objets par le biais d'un supertype. Pour pouvoir accéder à ces méthodes, vous devrez obligatoirement passer par une référence à un objet `Chien`. Pas terrible, tout ça !

Tu nous as dit que pour utiliser au mieux le polymorphisme, nous devons définir les méthodes au plus haut niveau de la hiérarchie. Alors du coup, il faut redéfinir un supertype pour pouvoir utiliser le polymorphisme !

Oui, et je vous rappelle que l'héritage multiple est interdit en Java. Et quand je dis *interdit*, je veux dire que Java ne le gère pas ! Il faudrait pouvoir développer un nouveau supertype et s'en servir dans nos classes `Chien`. Eh bien nous pouvons faire cela avec des **interfaces**.

En fait, les interfaces permettent de créer un nouveau supertype ; on peut même en ajouter autant que l'on le veut dans une seule classe ! Quant à l'utilisation de nos objets, la convention est toute

trouvée. Pourquoi ? Parce qu'une interface n'est rien d'autre qu'une classe 100 % abstraite ! Allez : venons-en aux faits !

Votre première interface

Pour définir une interface, au lieu d'écrire :

```
public class A{ }
```

... il vous suffit de faire :

```
public interface I{ }
```

Voilà : vous venez d'apprendre à déclarer une interface. Vu qu'une interface est une classe 100 % abstraite, il ne vous reste qu'à y ajouter des méthodes abstraites, mais sans le mot clé `abstract`.

Voici des exemples d'interfaces :

```
public interface I{

    public void A();

    public String B();

}

public interface I2{

    public void C();

    public String D();

}
```

Et pour faire en sorte qu'une classe utilise une interface, il suffit d'utiliser le mot clé `implements`. Ce qui nous donnerait :

```
public class X implements I{

    public void A(){

        //...

    }

    public String B(){

        //...

    }

}
```

C'est tout. On dit que *la classe X implémente l'interface I*. Comme je vous le disais, vous pouvez implémenter plusieurs interfaces, et voilà comment ça se passe :

```
public class X implements I, I2{

    public void A(){

        //...

    }

    public String B(){

        //...

    }

    public void C(){

        //...

    }

    public String D(){

        //...

    }

}
```

Par contre, lorsque vous implémentez une interface, vous devez *obligatoirement* redéfinir les méthodes de l'interface ! Ainsi, le polymorphisme vous permet de faire ceci :

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

    //Avec cette référence, vous pouvez utiliser les méthodes de l'interface I

    I var = new X();

    //Avec cette référence, vous pouvez utiliser les méthodes de l'interface I2

    I2 var2 = new X();

    var.A();

    var2.C();

}
```

Implémentation de l'interface `Rintintin`

Voilà où nous en sommes :

- nous voulons que nos chiens puissent être amicaux ;
- nous voulons définir un supertype pour utiliser le polymorphisme ;
- nous voulons pouvoir continuer à utiliser nos objets comme avant.

Comme le titre de cette sous-section le stipule, nous allons créer l'interface `Rintintin` pour ensuite l'implémenter dans notre objet `Chien`.

Sous Eclipse, vous pouvez faire `File > New > Interface`, ou simplement cliquer sur la flèche noire à côté du «C» pour la création de classe, et choisir `interface`, comme à la figure suivante. Voici son code :

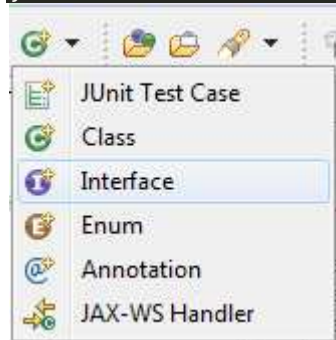
```
public interface Rintintin{

    public void faireCalin();

    public void faireLechouille();

    public void faireLeBeau();

}
```



Création d'une nouvelle interface

À présent, il ne nous reste plus qu'à implémenter l'interface dans notre classe `Chien`:

```
public class Chien extends Canin implements Rintintin {

    public Chien(){

    }

    public Chien(String couleur, int poids){

        this.couleur = couleur;

    }

}
```

```

    this.poids = poids;

}

void crier() {

    System.out.println("J'aboie sans raison !");

}

public void faireCalin() {

    System.out.println("Je te fais un GROS CÂLIN");

}

public void faireLeBeau() {

    System.out.println("Je fais le beau !");

}

public void faireLechouille() {

    System.out.println("Je fais de grosses léchouilles...");

}

}

```

L'ordre des déclarations est *primordial*. Vous devez mettre l'expression d'héritage avant l'expression d'implémentation, sinon votre code ne compilera pas.

Voici un code que vous pouvez utiliser pour tester le polymorphisme de notre implémentation :

```

public class Test {

    public static void main(String[] args) {

```



```

//Les méthodes d'un chien

Chien c = new Chien("Gris bleuté", 20);

c.boire();

c.manger();

c.deplacement();

c.crier();

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

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

//Les méthodes de l'interface

c.faireCalin();

c.faireLeBeau();

c.faireLechouille();

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

//Utilisons le polymorphisme de notre interface

Rintintin r = new Chien();

r.faireLeBeau();

r.faireCalin();

r.faireLechouille();

}
}

```

Objectif atteint ! Nous sommes parvenus à définir deux superclasses afin de les utiliser comme supertypes et de jouir pleinement du polymorphisme.

Dans la suite de ce chapitre, nous verrons qu'il existe une façon très intéressante d'utiliser les interfaces grâce à une technique de programmation appelée « *pattern strategy* ». Sa lecture n'est

pas indispensable, mais cela vous permettra de découvrir à travers un cas concret comment on peut faire évoluer au mieux un programme Java.

Le pattern strategy

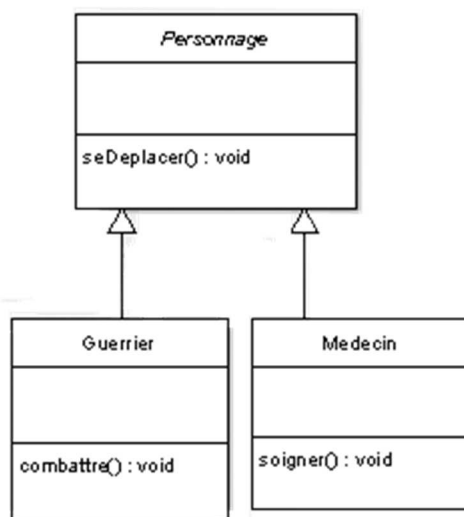
Nous allons partir du principe que vous avez un code qui fonctionne, c'est-à-dire un ensemble de classes liées par l'héritage, par exemple. Nous allons voir ici que, en dépit de la puissance de l'héritage, celui-ci atteint ses limites lorsque vous êtes amenés à modifier la hiérarchie de vos classes afin de répondre à une demande (de votre chef, d'un client etc.).

Le fait de toucher à votre hiérarchie peut amener des erreurs indésirables, voire des absurdités : tout cela parce que vous allez changer une structure qui fonctionne à cause de contraintes que l'on vous impose. Pour remédier à ce problème, il existe un concept simple (il s'agit même d'un des fondements de la programmation orientée objet) : **l'encapsulation !**

Nous allons parler de cette solution en utilisant un **design pattern** (ou « modèle de conception » en français). Un design pattern est un patron de conception, une façon de construire une hiérarchie des classes permettant de répondre à un problème. Nous aborderons le **pattern strategy**, qui va nous permettre de remédier à la limite de l'héritage. En effet, même si l'héritage offre beaucoup de possibilités, il a ses limites.

Posons le problème

Mettez-vous dans la peau de développeurs jeunes et ambitieux d'une toute nouvelle société qui crée des jeux vidéo. Le dernier titre en date, « Z-Army », un jeu de guerre très réaliste, a été un succès international ! Votre patron est content et vous aussi. Vous vous êtes basés sur une architecture vraiment simple afin de créer et utiliser des personnages, comme le montre la figure suivante.

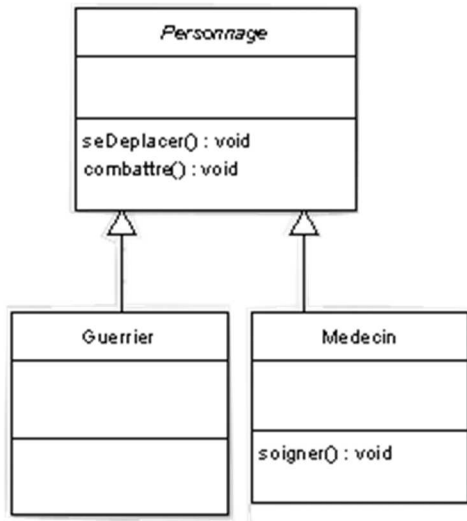


Hiérarchie des classes

Les guerriers savent se battre tandis que les médecins soignent les blessés sur le champ de bataille. Et c'est maintenant que commencent les ennuis !

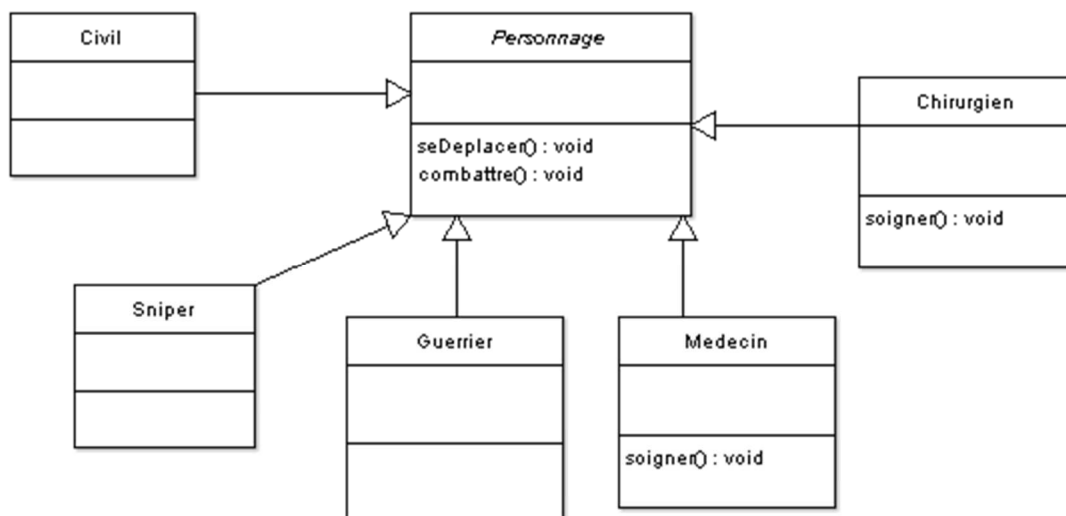
Votre patron vous a confié le projet « Z-Army 2 : The return of the revenge », et vous vous dites : « Yes ! Mon architecture fonctionne à merveille, je la garde. » Un mois plus tard, votre patron vous convoque dans son bureau et vous dit : « Nous avons fait une étude de marché, et il semblerait que les joueurs aimeraient se battre aussi avec les médecins ! » Vous trouvez l'idée séduisante et avez déjà pensé à une solution : déplacer la méthode `combattre()` dans la

superclasse `Personnage`, afin de la redéfinir dans la classe `Medecin` et jouer du polymorphisme ! La figure suivante schématise le tout.



Déplacement de la méthode `combattre()`

À la seconde étude de marché, votre patron vous annonce que vous allez devoir créer des civils, des snipers, des chirurgiens etc. Toute une panoplie de personnages spécialisés dans leur domaine, comme le montre la figure suivante.



personnages spécialisés
Le code source de ces classes

Personnage.java

```
public abstract class Personnage {

    //Méthode de déplacement de personnage

    public abstract void seDeplacer();
```

Nouveaux

```
//Méthode que les combattants utilisent

public abstract void combattre();

}
```

Guerrier.java

```
public class Guerrier extends Personnage {

    public void combattre() {

        System.out.println("Fusil, pistolet, couteau ! Tout ce que tu veux !");

    }

    public void seDeplacer() {

        System.out.println("Je me déplace à pied.");

    }

}
```

Medecin.java

```
public class Medecin extends Personnage{

    public void combattre() {

        System.out.println("Vive le scalpel !");

    }

    public void seDeplacer() {

        System.out.println("Je me déplace à pied.");

    }

    public void soigner(){
```

```
        System.out.println("Je soigne les blessures.");
    }
}
```

Civil.java

```
public class Civil extends Personnage{

    public void combattre() {

        System.out.println("Je ne combats PAS !");

    }

    public void seDeplacer() {

        System.out.println("Je me déplace à pied.");

    }

}
```

Chirurgien.java

```
public class Chirurgien extends Personnage{

    public void combattre() {

        System.out.println("Je ne combats PAS !");

    }

    public void seDeplacer() {

        System.out.println("Je me déplace à pied.");

    }

    public void soigner(){

        System.out.println("Je fais des opérations.");

    }

}
```

```
}  
}
```

Sniper.java

```
public class Sniper extends Personnage{  
  
    public void combattre() {  
  
        System.out.println("Je me sers de mon fusil à lunette !");  
  
    }  
  
  
    public void seDeplacer() {  
  
        System.out.println("Je me déplace à pied.");  
  
    }  
  
}
```

À ce stade, vous devriez remarquer que :

- le code contenu dans la méthode `seDeplacer()` est dupliqué dans toutes les classes ; il est identique dans toutes celles citées ci-dessus ;
- le code de la méthode `combattre()` des classes `Chirurgien` et `Civil` est lui aussi dupliqué !

La duplication de code est une chose qui peut générer des problèmes dans le futur. Je m'explique.

Pour le moment, votre chef ne vous a demandé que de créer quelques classes supplémentaires. Qu'en serait-il si beaucoup de classes avaient ce même code dupliqué ? Il ne manquerait plus que votre chef vous demande de modifier à nouveau la façon de se déplacer de ces objets, et vous courrez le risque d'oublier d'en modifier un. Et voilà les incohérences qui pointeront le bout de leur nez !

No problemo ! Tu vas voir ! Il suffit de mettre un comportement par défaut pour le déplacement et pour le combat dans la superclasse `Personnage`.

Effectivement, votre idée se tient. Donc, cela nous donne ce qui suit :

Personnage.java

```
public abstract class Personnage {  
  
    public void seDeplacer(){  
  
        System.out.println("Je me déplace à pied.");  
  
    }  
  
}
```

```

public void combattre(){

    System.out.println("Je ne combats PAS !");

}

}

```

Guerrier.java

```

public class Guerrier extends Personnage {

    public void combattre() {

        System.out.println("Fusil, pistolet, couteau ! Tout ce que tu veux !");

    }

}

```

Medecin.java

```

public class Medecin extends Personnage{

    public void combattre() {

        System.out.println("Vive le scalpel !");

    }


    public void soigner(){

        System.out.println("Je soigne les blessures.");

    }

}

```

Civil.java

```

public class Civil extends Personnage{    }

```

Chirurgien.java

```

public class Chirurgien extends Personnage{

    public void soigner(){

```

```

        System.out.println("Je fais des opérations.");
    }
}

```

Sniper.java

```

public class Sniper extends Personnage{

    public void combattre() {

        System.out.println("Je me sers de mon fusil à lunette !");

    }

}

```

Voici une classe contenant un petit programme afin de tester nos classes :

```

public static void main(String[] args) {

    Personnage[] tPers = {new Guerrier(), new Chirurgien(), new Civil(), new Sniper(), new
    Medecin()};

    for(Personnage p : tPers){

        System.out.println("\nInstance de " + p.getClass().getName());

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

        p.combattre();

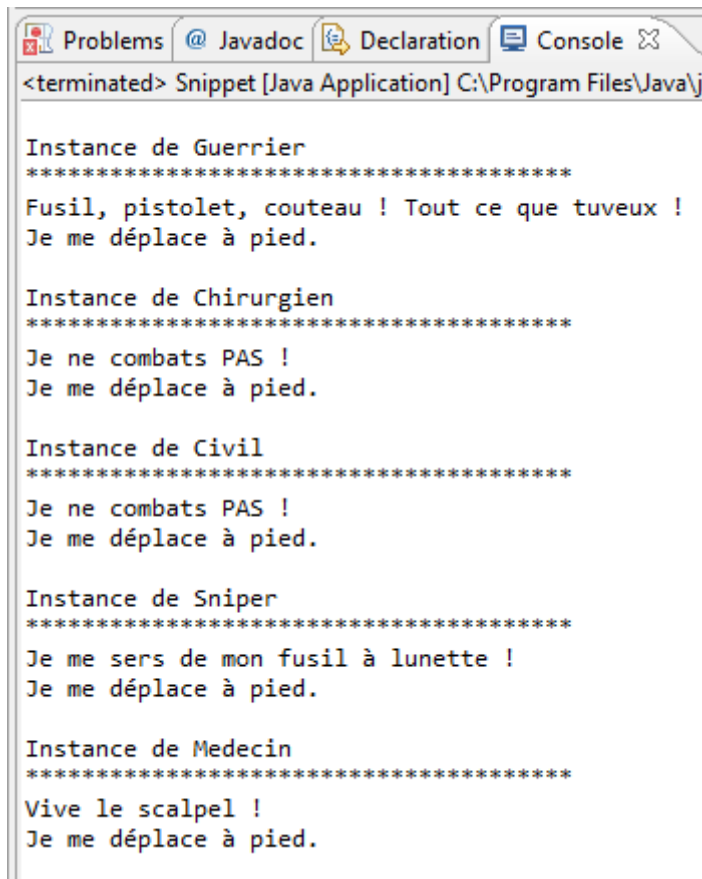
        p.seDeplacer();

    }

}

```

Le résultat correspond à la figure suivante.



```
<terminated> Snippet [Java Application] C:\Program Files\Java\j...

Instance de Guerrier
*****
Fusil, pistolet, couteau ! Tout ce que tu veux !
Je me déplace à pied.

Instance de Chirurgien
*****
Je ne combats PAS !
Je me déplace à pied.

Instance de Civil
*****
Je ne combats PAS !
Je me déplace à pied.

Instance de Sniper
*****
Je me sers de mon fusil à lunette !
Je me déplace à pied.

Instance de Medecin
*****
Vive le scalpel !
Je me déplace à pied.
```

Résultat du code

Apparemment, ce code vous donne ce que vous voulez ! Mais une chose me chiffonne : vous ne pouvez pas utiliser les classes `Medecin` et `Chirurgien` de façon polymorphe, vu que la méthode `soigner()` leur est propre ! On pourrait définir un comportement par défaut (ne pas soigner) dans la superclasse `Personnage` et le tour serait joué.

```
public abstract class Personnage {

    public void seDeplacer(){

        System.out.println("Je me déplace à pied.");

    }

    public void combattre(){

        System.out.println("Je ne combats PAS !");

    }

    public void soigner(){

        System.out.println("Je ne soigne pas.");

    }

}
```

}

Au même moment, votre chef rentre dans votre bureau et vous dit : « Nous avons bien réfléchi, et il serait de bon ton que nos guerriers puissent administrer les premiers soins. » À ce moment précis, vous vous délectez de votre capacité d'anticipation ! Vous savez que, maintenant, il vous suffit de redéfinir la méthode `soigner()` dans la classe concernée, et tout le monde sera content !

Seulement voilà ! Votre chef n'avait pas fini son speech : « Au fait, il faudrait affecter un comportement à nos personnages en fonction de leurs armes, leurs habits, leurs trousse de soin... Enfin, vous voyez ! Les comportements figés pour des personnages de jeux, de nos jours, c'est un peu ringard ! »

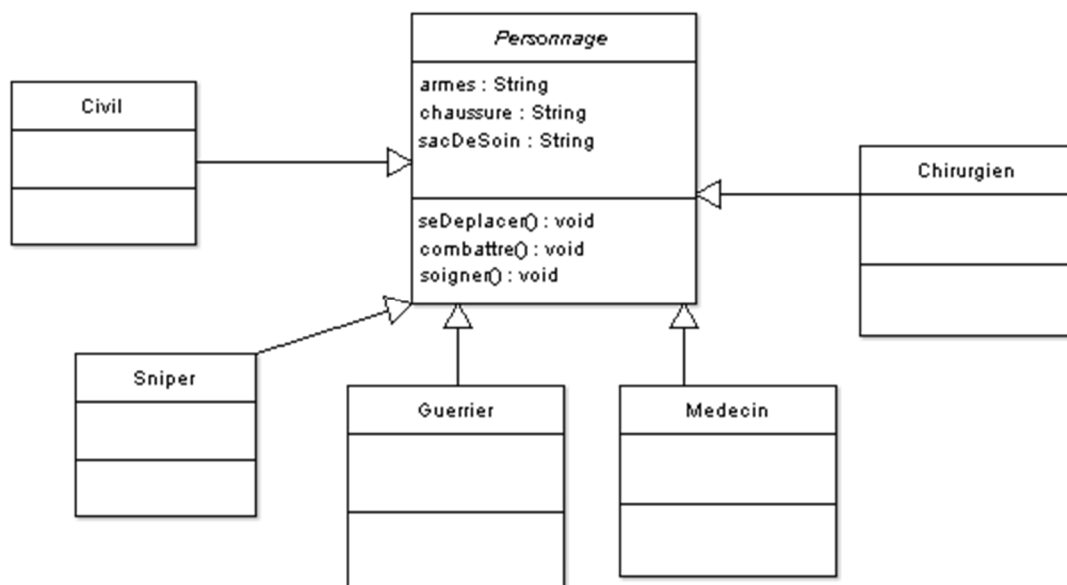
Vous commencez à voir ce dont il retourne : vous devrez apporter des modifications à votre code, encore et encore. Bon : pour des programmeurs, cela est le train-train quotidien, j'en conviens. Cependant, si nous suivons les consignes de notre chef et que nous continuons sur notre lancée, les choses vont se compliquer.

Un problème supplémentaire

Attelons-nous à appliquer les modifications dans notre programme. Selon les directives de notre chef, nous devons gérer des comportements différents selon les accessoires de nos personnages : il faut utiliser des variables d'instance pour appliquer l'un ou l'autre comportement.

Afin de simplifier l'exemple, nous n'allons utiliser que des objets `String`.

La figure suivante correspond au diagramme des classes de notre programme.



Modification de nos classes

Vous avez remarqué que nos personnages posséderont des accessoires. Selon ceux-ci, nos personnages feront des choses différentes. Voici les recommandations de notre chef bien-aimé :

- le guerrier peut utiliser un couteau, un pistolet ou un fusil de sniper ;
- le sniper peut utiliser son fusil de sniper ainsi qu'un fusil à pompe ;
- le médecin a une trousse simple pour soigner, mais peut utiliser un pistolet ;
- le chirurgien a une grosse trousse médicale, mais ne peut pas utiliser d'arme ;
- le civil, quant à lui, peut utiliser un couteau seulement quand il en a un ;
- tous les personnages hormis le chirurgien peuvent avoir des baskets pour courir ;

Il va nous falloir des mutateurs (inutile de mettre les méthodes de renvoi (getxxx), nous ne nous servirons que des mutateurs !) pour ces variables, insérons-les dans la superclasse ! Bon ! Les modifications sont faites, les caprices de notre cher et tendre chef sont satisfaits ? Voyons cela tout de suite.

Personnage.java

```
public abstract class Personnage {

    protected String armes = "", chaussure = "", sacDeSoin = "";

    public void seDeplacer(){

        System.out.println("Je me déplace à pied.");

    }

    public void combattre(){

        System.out.println("Je ne combats PAS !");

    }

    public void soigner(){

        System.out.println("Je ne soigne pas.");

    }

    protected void setArmes(String armes) {

        this.armes = armes;

    }

    protected void setChaussure(String chaussure) {
```

```

        this.chaussure = chaussure;

    }

    protected void setSacDeSoin(String sacDeSoin) {

        this.sacDeSoin = sacDeSoin;

    }

}

```

Guerrier.java

```

public class Guerrier extends Personnage {

    public void combattre() {

        if(this.armes.equals("pistolet"))

            System.out.println("Attaque au pistolet !");

        else if(this.armes.equals("fusil de sniper"))

            System.out.println("Attaque au fusil de sniper !");

        else

            System.out.println("Attaque au couteau !");

    }

}

```

Sniper.java

```

public class Sniper extends Personnage{

    public void combattre() {

        if(this.armes.equals("fusil à pompe"))

            System.out.println("Attaque au fusil à pompe !");

        else

```

```

        System.out.println("Je me sers de mon fusil à lunette !");

    }

}

```

Civil.java

```

public class Civil extends Personnage{

    public void combattre(){

        if(this.armes.equals("couteau"))

            System.out.println("Attaque au couteau !");

        else

            System.out.println("Je ne combats PAS !");

    }

}

```

Medecin.java

```

public class Medecin extends Personnage{

    public void combattre() {

        if(this.armes.equals("pistolet"))

            System.out.println("Attaque au pistolet !");

        else

            System.out.println("Vive le scalpel !");

    }


    public void soigner(){

        if(this.sacDeSoin.equals("petit sac"))

            System.out.println("Je peux recoudre des blessures.");

        else

```

```

        System.out.println("Je soigne les blessures.");
    }
}

```

Chirurgien.java

```

public class Chirurgien extends Personnage{

    public void soigner(){

        if(this.sacDeSoin.equals("gros sac"))

            System.out.println("Je fais des merveilles.");

        else

            System.out.println("Je fais des opérations.");

    }

}

```

Voici un programme de test :

```

public static void main(String[] args) {

    Personnage[] tPers = {new Guerrier(), new Chirurgien(), new Civil(), new Sniper(), new
    Medecin()};

    String[] tArmes = {"pistolet", "pistolet", "couteau", "fusil à pompe", "couteau"};

    for(int i = 0; i < tPers.length; i++){

        System.out.println("\nInstance de " + tPers[i].getClass().getName());

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

        tPers[i].combattre();

        tPers[i].setArmes(tArmes[i]);

        tPers[i].combattre();

        tPers[i].seDeplacer();

        tPers[i].soigner();
    }
}

```

```
}  
}
```

Le résultat de ce test se trouve à la figure suivante.

```
Instance de Guerrier  
*****  
Attaque au couteau !  
Attaque au pistolet !  
Je me déplace à pied.  
Je ne soigne pas.  
  
Instance de Chirurgien  
*****  
Je ne combats PAS !  
Je ne combats PAS !  
Je me déplace à pied.  
Je fais des opérations.  
  
Instance de Civil  
*****  
Je ne combats PAS !  
Attaque au couteau !  
Je me déplace à pied.  
Je ne soigne pas.  
  
Instance de Sniper  
*****  
Je me sers de mon fusil à lunette !  
Attaque au fusil à pompe !  
Je me déplace à pied.  
Je ne soigne pas.  
  
Instance de Medecin  
*****  
Vive le scalpel !  
Vive le scalpel !  
Je me déplace à pied.  
Je soigne les blessures.
```

Résultat du test d'accessoires

Vous constatez avec émerveillement que votre code fonctionne très bien. Les actions par défaut sont respectées, les affectations d'actions aussi. Tout est parfait !

Vraiment ? Vous êtes sûrs de cela ? Pourtant, je vois du code dupliqué dans certaines classes ! En plus, nous n'arrêtons pas de modifier nos classes. Dans le premier opus de « Z-Army », celles-ci fonctionnaient pourtant très bien ! Qu'est-ce qui ne va pas ?

Là-dessus, votre patron rentre dans votre bureau pour vous dire : « Les actions de vos personnages doivent être utilisables à la volée et, en fait, les personnages peuvent très bien apprendre au fil du jeu. » Les changements s'accumulent, votre code devient de moins en moins lisible et réutilisable, bref c'est l'enfer sur Terre.

Faisons un point de la situation :

- du code dupliqué s'insinue dans votre code ;
- à chaque modification du comportement de vos personnages, vous êtes obligés de retoucher le code source de la (ou des) classe(s) concernée(s) ;
- votre code perd en « réutilisabilité » et du coup, il n'est pas extensible du tout !

Une solution simple et robuste : le « pattern strategy »

Après toutes ces émotions, vous allez enfin disposer d'une solution à ce problème de modification du code source ! Si vous vous souvenez de ce que j'ai dit, un des fondements de la programmation orientée objet est l'encapsulation.

Le pattern strategy est basé sur ce principe simple. Bon, vous avez compris que le pattern strategy consiste à créer des objets avec des données, des méthodes (voire les deux) : c'est justement ce qui change dans votre programme !

Le principe de base de ce pattern est le suivant : « isolez ce qui varie dans votre programme et encapsulez-le ! »

Déjà, quels sont les éléments qui ne cessent de varier dans notre programme ?

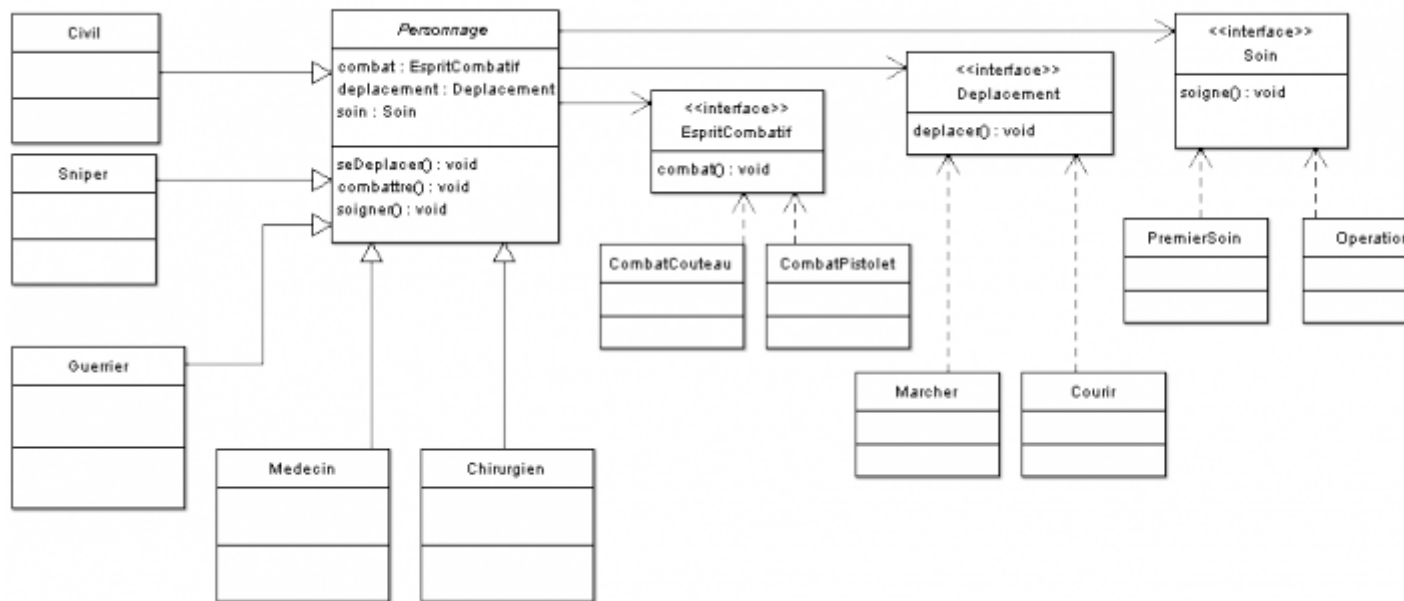
- La méthode `combattre()`.
- La méthode `seDeplacer()`.
- La méthode `soigner()`.

Ce qui serait vraiment grandiose, ce serait d'avoir la possibilité de ne modifier que les comportements et non les objets qui ont ces comportements ! Non ?

Là, je vous arrête un moment : vous venez de fournir la solution. Vous avez dit : « ce qui serait vraiment grandiose, ce serait d'avoir la possibilité de ne modifier que les comportements et non les objets qui ont ces comportements ».

Lorsque je vous ai présenté les diagrammes UML, je vous ai fourni une astuce pour bien différencier les liens entre les objets. Dans notre cas, nos classes héritant de `Personnage` héritent aussi de ses comportements et, par conséquent, on peut dire que nos classes filles sont des `Personnage`.

Les comportements de la classe mère semblent ne pas être au bon endroit dans la hiérarchie. Vous ne savez plus quoi en faire et vous vous demandez s'ils ont vraiment leur place dans cette classe ? Il vous suffit de sortir ces comportements de la classe mère, de créer une classe abstraite ou une interface symbolisant ce comportement et d'ordonner à votre classe `Personnage` d'avoir ces comportements. Le nouveau diagramme des classes se trouve sur la figure suivante.



Nouveau diagramme des classes

Vous apercevez une nouvelle entité sur ce diagramme, l'interface, facilement reconnaissable, ainsi qu'une nouvelle flèche symbolisant l'implémentation d'interface entre une classe concrète et une interface. N'oubliez pas que votre code doit être souple et robuste et que — même si ce chapitre vous montre les limites de l'héritage — le polymorphisme est inhérent à l'héritage (ainsi qu'aux implémentations d'interfaces).

Il faut vous rendre compte qu'utiliser une interface de cette manière revient à créer un supertype de variable ; du coup, nous pourrons utiliser les classes héritant de ces interfaces de façon polymorphe sans nous soucier de savoir la classe dont sont issus nos objets ! Dans notre cas, notre classe `Personnage` comprendra des objets de type `EspritCombatif`, `Soins` et `Deplacement` !

Avant de nous lancer dans le codage de nos nouvelles classes, vous devez observer que leur nombre a considérablement augmenté depuis le début. Afin de pouvoir gagner en clarté, nous allons gérer nos différentes classes avec différents packages.

Comme nous l'avons remarqué tout au long de ce chapitre, les comportements de nos personnages sont trop éparés pour être définis dans notre superclasse `Personnage`. Vous l'avez dit vous-mêmes : il faudrait que l'on ne puisse modifier que les comportements et non les classes héritant de notre superclasse !

Les interfaces nous servent à créer un supertype d'objet ; grâce à elles, nous utiliserons des objets de type :

- `EspritCombatif` qui présentent une méthode `combat()` ;
- `Soins` qui présentent une méthode `soigner()` ;
- `Deplacement` qui présentent une méthode `deplace()` .

Dans notre classe `Personnage`, nous avons ajouté une instance de chaque type de comportement, vous avez dû les remarquer : il y a ces attributs dans notre schéma ! Nous allons développer un comportement par défaut pour chacun d'entre eux et affecter cet objet dans notre superclasse. Les classes filles, elles, comprendront des instances différentes correspondant à leurs besoins.

Du coup, que fait-on des méthodes de la superclasse `Personnage` ?

Nous les gardons, mais plutôt que de redéfinir ces dernières, la superclasse va invoquer la méthode de comportement de chaque objet. Ainsi, nous n'avons plus à redéfinir ou à modifier nos

classes ! La seule chose qu'il nous reste à faire, c'est d'affecter une instance de comportement à nos objets. Vous comprendrez mieux avec un exemple. Voici quelques implémentations de comportements.

Implémentations de l'interface `EspritCombatif`

```
package com.sdz.comportement;

public class Pacifiste implements EspritCombatif {

    public void combat() {

        System.out.println("Je ne combats pas !");

    }

}

package com.sdz.comportement;

public class CombatPistolet implements EspritCombatif{

    public void combat() {

        System.out.println("Je combats au pitolet !");

    }

}

package com.sdz.comportement;

public class CombatCouteau implements EspritCombatif {

    public void combat() {

        System.out.println("Je me bats au couteau !");

    }

}
```

Implémentations de l'interface `Deplacement`

```
package com.sdz.comportement;

public class Marcher implements Deplacement {

    public void deplacer() {

        System.out.println("Je me déplace en marchant.");

    }

}
```

```
package com.sdz.comportement;

public class Courir implements Deplacement {

    public void deplacer() {

        System.out.println("Je me déplace en courant.");

    }

}
```

Implémentations de l'interface Soin

```
package com.sdz.comportement;

public class PremierSoin implements Soin {

    public void soigner() {

        System.out.println("Je donne les premiers soins.");

    }

}
```

```
package com.sdz.comportement;

public class Operation implements Soin {
```

```

    public void soigner() {

        System.out.println("Je pratique des opérations !");

    }

}

package com.sdz.comportement;

public class AucunSoin implements Soin {

    public void soigner() {

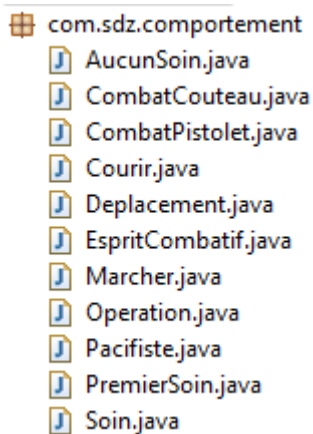
        System.out.println("Je ne donne AUCUN soin !");

    }

}

```

La figure suivante représente ce que vous devriez avoir dans votre nouveau package.



Package des comportements

Maintenant que nous avons défini des objets de comportements, nous allons pouvoir remanier notre classe `Personnage`. Ajoutons les variables d'instance, les mutateurs et les constructeurs permettant d'initialiser nos objets :

```

import com.sdz.comportement.*;

public abstract class Personnage {

    //Nos instances de comportement

```

```
protected EspritCombatif espritCombatif = new Pacifiste();

protected Soin soin = new AucunSoin();

protected Deplacement deplacement = new Marcher();


//Constructeur par défaut

public Personnage(){}


//Constructeur avec paramètres

public Personnage(EspritCombatif espritCombatif, Soin soin, Deplacement deplacement) {

    this.espritCombatif = espritCombatif;

    this.soin = soin;

    this.deplacement = deplacement;

}


//Méthode de déplacement de personnage

public void seDeplacer(){

    //On utilise les objets de déplacement de façon polymorphe

    deplacement.deplacer();

}


// Méthode que les combattants utilisent

public void combattre(){

    //On utilise les objets de déplacement de façon polymorphe

    espritCombatif.combat();

}
```

```

}

//Méthode de soin

public void soigner(){

    //On utilise les objets de déplacement de façon polymorphe

    soin.soigner();

}

//Redéfinit le comportement au combat

public void setEspritCombatif(EspritCombatif espritCombatif) {

    this.espritCombatif = espritCombatif;

}

//Redéfinit le comportement de Soin

public void setSoin(Soin soin) {

    this.soin = soin;

}

//Redéfinit le comportement de déplacement

public void setDeplacement(Deplacement deplacement) {

    this.deplacement = deplacement;

}

}

```

Que de changements depuis le début ! Maintenant, nous n'utilisons plus de méthodes définies dans notre hiérarchie de classes, mais des implémentations concrètes d'interfaces ! Les méthodes que nos objets appellent utilisent chacune un objet de comportement. Nous pouvons donc définir des

guerriers, des civils, des médecins... tous personnalisables, puisqu'il suffit de modifier l'instance de leur comportement pour que ceux-ci changent instantanément. La preuve par l'exemple.

Je ne vais pas vous donner les codes de toutes les classes. En voici seulement quelques-unes.

Guerrier.java

```
import com.sdz.comportement.*;

public class Guerrier extends Personnage {

    public Guerrier(){

        this.espritCombatif = new CombatPistolet();

    }

    public Guerrier(EspritCombatif esprit, Soins soin, Deplacement dep) {

        super(esprit, soin, dep);

    }

}
```

Civil.java

```
import com.sdz.comportement.*;

public class Civil extends Personnage{

    public Civil() {}

    public Civil(EspritCombatif esprit, Soins soin, Deplacement dep) {

        super(esprit, soin, dep);

    }

}
```

Medecin.java

```
import com.sdz.comportement.*;
```

```

public class Medecin extends Personnage{

    public Medecin() {

        this.soin = new PremierSoin();

    }

    public Medecin(EspritCombatif esprit, Soin soin, Deplacement dep) {

        super(esprit, soin, dep);

    }

}

```

Maintenant, voici un exemple d'utilisation :

```

class Test{

    public static void main(String[] args) {

        Personnage[] tPers = {new Guerrier(), new Civil(), new Medecin()};

        for(int i = 0; i < tPers.length; i++){

            System.out.println("\nInstance de " + tPers[i].getClass().getName());

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

            tPers[i].combattre();

            tPers[i].seDeplacer();

            tPers[i].soigner();

        }

    }

}

```

Le résultat de ce code nous donne la figure suivante.


```
Problems @ Javadoc Declaration Console
<terminated> Test (1) [Java Application] C:\Program Files\Java

Instance de Guerrier
*****
Je combats au pitolet !
Je me déplace en marchant.
Je ne donne AUCUN soin !

Instance de Civil
*****
Je ne combats pas !
Je me déplace en marchant.
Je ne donne AUCUN soin !

Instance de Medecin
*****
Je ne combats pas !
Je me déplace en marchant.
Je donne les premiers soins.
```

Test du pattern strategy

Vous pouvez voir que nos personnages ont tous un comportement par défaut qui leur convient bien ! Nous avons spécifié, dans le cas où cela s'avère nécessaire, le comportement par défaut d'un personnage dans son constructeur par défaut :

- le guerrier se bat avec un pistolet ;
- le médecin soigne.

Voyons maintenant comment indiquer à nos personnages de faire autre chose. Eh oui, la façon dont nous avons arrangé tout cela va nous permettre de changer dynamiquement le comportement de chaque Personnage. Que diriez-vous de faire faire une petite opération chirurgicale à notre objet Guerrier ?

Pour ce faire, vous pouvez redéfinir son comportement de soin avec son mutateur présent dans la superclasse en lui passant une implémentation correspondante !

```
import com.sdz.comportement.*;

class Test{

    public static void main(String[] args) {

        Personnage pers = new Guerrier();

        pers.soigner();

        pers.setSoin(new Operation());

        pers.soigner();

    }
}
```

}

En testant ce code, vous constaterez que le comportement de soin de notre objet a changé dynamiquement sans que nous ayons besoin de changer la moindre ligne de son code source ! Le plus beau dans le fait de travailler comme cela, c'est qu'il est tout à fait possible d'instancier des objets `Guerrier` avec des comportements différents.

- Une classe est définie comme abstraite avec le mot clé `abstract`.
- Les classes abstraites sont à utiliser lorsqu'une classe mère ne doit pas être instanciée.
- Une classe abstraite **ne peut donc pas être instanciée**.
- Une classe abstraite n'est pas obligée de contenir de méthode abstraite.
- Si une classe contient une méthode abstraite, cette classe doit alors être déclarée abstraite.
- Une méthode abstraite n'a pas de corps.
- Une interface est une classe 100 % abstraite.
- Aucune méthode d'une interface n'a de corps.
- Une interface sert à définir un supertype et à utiliser le polymorphisme.
- Une interface s'implémente dans une classe en utilisant le mot clé `implements`.
- Vous pouvez implémenter autant d'interfaces que vous voulez dans vos classes.
- Vous devez redéfinir toutes les méthodes de l'interface (ou des interfaces) dans votre classe.
- Le pattern strategy vous permet de rendre une hiérarchie de classes plus souple.
- Préférez encapsuler des comportements plutôt que de les mettre d'office dans l'objet concerné.