Chapitre 5

Héritage

L'héritage permet à un objet d'acquérir les propriétés d'un autre objet. Si on considère notre exemple de personnage, un romain est un type de personnage, un gaulois un autre. Pour coder une classe `Gaulois`, il semble inutile de coder de nouveau tout ce qui a été codé dans la classe `Personnage`. De plus, si on veut plus tard modifier la classe `Personnage`, on ne voudrait pas qu'on ait à changer quelque chose dans la classe `Gaulois`. La solution est donc de dire que la classe `Gaulois` hérite des propriétés de la classe `Personnage`. On peut ainsi factoriser les connaissances : la classe mère (ou classe de base) est plus générale et contient les propriétés commune à toutes les classes filles (ou classe dérivée ou héritée). Les classes filles ont des propriétés plus spécifiques. On peut ainsi avoir une hierarchie de classes. De fait, tous les objets en JAVA dérivent par défaut de la classe `java.lang.Object`. Cela implique que tous les objets possèdent déjà à leur naissance un certains nombre d'attributs et de méthodes qui dérivent d'`Object`. Pour exprimer qu'une classe est une classe fille d'une autre classe, on utilise le mot-clé `extends` comme suit :

```
1 class <nom classe fille> extends <nom classe mère>
```

JAVA supporte seulement *un seul* héritage : il n'est pas possible d'hériter de plusieurs parents (un langage comme C++ permet un tel héritage multiple).

1 Polymorphisme

Par définition, un objet est une instance d'une classe. Dans l'exemple qui suit, on définit quatre classes. Tout d'abord, la classe `Personnage` est la classe la plus générale. Ensuite, on définit les classes `Gaulois` et `Romain` qui sont deux classes qui héritent de `Personnage` : un romain ou un gaulois sera un personnage particulier. Finalement, on définit une troisième classe qui hérite de `Gaulois` : il s'agit de la classe `IrreductibleGaulois`. Un irréductible gaulois est par définition un gaulois, mais c'est aussi un personnage. On voit donc bien que désormais, un objet peut avoir plusieurs types. Quand un objet peut appartenir à plusieurs types, on parle de *polymorphisme.*

```
1 public class Personnage { ... }
```

```
2 public class Gaulois extends Personnage { ... }
```

```
3 public class IrreductibleGaulois extends Gaulois { ... }
```

```
4 public class Romain extends Personnage { ... }
```

Le polymorphisme et le transtypage implicite nous permettent de manipuler des objets qui sont issus de classes différentes, mais qui partagent un même type parent : il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance).

```
1 Personnage asterix = new Gaulois("Astérix") ;
```

On utilise alors le transtypage implicite de la classe dérivée vers la classe mère. Dans l'exemple précédent, la variable `asterix` est déclarée comme un `Personnage`. Mais cette variable fait référence à un `Gaulois`, cela est logique car un `Gaulois` est bien un `Personnage`.

A quoi cela peut bien servir ? Lorsqu'on veut utiliser une structure qui rassemble plusieurs objets, cela peut s'avérer très utile. Pour le moment, vous ne connaissez que les tableaux (on verra plus tard d'autres structures comme les listes, tas, etc). Dans l'exemple qui suit, on veut avoir un tableau contenant tous les personnages connus. On veut donc un tableau de `Personnages`, mais on ne veut pas s'interdire d'avoir des instances de sous classes de `Personnage` comme des `Gaulois`. Ici, on pourra donc utiliser une boucle pour écrire le nom de chaque personnage.

```
1 Gaulois obelix = new Gaulois("Obélix") ;
2 Gaulois asterix = new Gaulois("Astérix") ;
3 Personnage cleopatre = new Personnage("Cléopâtre") ;
3 Personnage[] distribution= new Personnage[3] ;
4 distribution[0]= asterix ;
5 distribution[1]= obelix ;
6 distribution[2]= cleopatre ;
7 for (int i=0 ;i<3 ;i++)
8    System.out.println(distribution[i].getName()) ;
```

Evidemment, si on utilise un type de base comme référence pour un type dérivé, on ne peut pas utiliser les méthodes spécialisées du type dérivé. En d'autres termes, si on a déclaré qu'une variable est du type `Parent`, on n'a accès qu'aux méthodes et attributs de la classe `Parent`. Même si le programmeur peut savoir qu'en fait, cet objet est de type `Enfant`, il ne peut appeler une méthode spécifique de la classe `Enfant`.

Dans notre exemple, on va supposer que la classe `Gaulois` possède une méthode `boisPotionMagique()` qui n'existe pas pour la classe `Personnage`. On ne pourra donc pas faire l'appel

`distribution[0]. boisPotionMagique().`

En effet, même si `distribution[0]` contient l'objet `asterix` qui est un `Gaulois`, `distribution[0]` est déclaré comme un `Personnage`, donc seulement les méthodes de la classe `Personnage` peuvent être utilisées.

2 `instance of`

On peut utiliser le mot clé `instance of` pour vérifier si un objet est bien une instance d'une classe ou s'il est une instance d'une sous classe. Avec le polymorphisme, un objet peut appartenir à plusieurs classes, et ainsi, il pourra retourner `True` à des appels différents de instance of. Par exemple, après l'exécution du code ci-dessous, on obtiendra `true`, `true` et `false` car Astérix est un `Personnage`, plus précisemment, c'est bien un `Gaulois`, mais ce n'est pas un `Romain`.

```
5 ...
6 IrreductibleGaulois asterix = new IrreductibleGaulois() ;
7 System.out.println( asterix instance of Personnage) ;
8 System.out.println( asterix instance of Gaulois) ;
9 System.out.println( asterix instance of Romain) ;
```

3 Les membres protégés – `protected`

On a vu deux portées pour les variables et les méthodes : `public` et `private`. Une classe fille peut accéder aux éléments public (évidemment, comme toutes les autres classes), mais malheureusement, elle ne peut pas accéder directement aux éléments privés et les manipuler tels quels. L'idée ici est que si la classe mère possède des attributs, c'est à cette classe d'offrir tous les outils pour travailler avec ces attributs. Si le développeur a choisi de cacher des détails d'implémentation (de façon à pouvoir les changer au besoin), ces details resteront cachés pour les classes filles (et donc un changement de détails d'implémentation n'affectera pas les classes filles).

Cependant, dans certains cas, on voudrait bien avoir une portée qui permette aux classes dérivées d'avoir accès à un élément, mais pas à des classes étrangères. Il existe donc une portée appelée `protected` qui permet précisemment cela : seules les classes dérivées peuvent accéder à des variables et méthodes protégées. Il vaut veiller à n'utiliser protected que lorsqu'on en a vraiment besoin.

4 Redéfinition des méthodes héritées

L'héritage permet d'ajouter des fonctionalités particulières qui ne se trouve pas dans la classe mère. Mais quant est-il pour les méthodes de la classe mère ? La classe dérivée hérite des méthodes `public` ou `protected` de sa classe mère. Ainsi, si le comportement est exactement le même que celui de la classe mère, on peut/doit omettre la ré-écriture de cette méthode. Si le comportement est différent, on peut ré-écrire la méthode avec la même signature et changer le corps de la méthode : on appelle cela la *redéfinition*.

Pour aider à faire la distinction entre une classe et sa classe mère, il existe deux références pour parcourir la hierarchie :

— *this* : est une référence sur l'instance de la classe.
— *super* : est une référence sur l'instance mère.

Dans l'exemple plus bas, la méthode `presentation()` de la classe `Gaulois` ajoute à la présentation de la classe `Personnage` le fait que le personnage est Gaulois.

Lorsqu'on redéfinit une méthode, la liste des arguments doit forcément rester la même que dans la classe mère (sinon, on effectue la création d'une nouvelle méthode et non la rédifinition d'une méthode de la classe mère). Pour cela, on peut/doit ajouter une annotation `@Override` qui permettra au compilateur de vérifier si c'est bien une rédéfinition. Par contre, il est permis de modifier le type de retour de la méthode. Par exemple, dans la classe `Romain`, on pourrait ajouter une méthode `public Romain getSuperior()`. Maintenant, si on veut faire une classe `Centurion` qui hérite de `Romain`, on peut redéfinir la méthode `getSuperior` de façon à indiquer qu'un centurion n'a pas pour supérieur un simple soldat romain, il ne peut rendre des comptes qu'à un autre centurion. On peut donc utiliser la signature `@Override public Centurion getSuperior()`.

4.1 Constructeur

Le constructeur est un cas particulier. On doit faire appel au constructeur de la classe mère à l'aide de `super`. L'appel au constructeur de la classe de base est la première instruction du constructeur de la classe dérivée. Ici, `super` indique l'appel au constructeur de la classe mère comme dans l'exemple ci-dessous.

On peut éventuellement se passer de l'appel au constructeur de la classe mère, mais seulement dans le cas où la classe mère possède un constructeur par défaut (i.e. sans arguments). Donc, que ce soit d'une manière explicite ou non, tout constructeur d'une classe fille fait *obligatoirement* appel au constructeur de la classe mère.

```
1  public class Personnage {
2
3     private String nom ;
4
5     public Personnage(String name){
6       this.nom = name ;
7     }
8
9     public String presentation(){
10      return "Je m'appelle" + name ;
11    }
12 }
```

```
1  public class Gaulois extends Personnage {
2
3     public Gaulois(String name){
4       super(name) ;
5     }
6
7     public String presentation(){
8       return super.presentation() + " je suis un gaulois";
9     }
10 }
11
12 public static void main(String[] args){
13    Gaulois asterix = new Gaulois("Astérix") ;
14    System.out.println( asterix.presentation()) ;
15 }
```

4.2 Recherche dynamique d'un membre

Si une méthode est redéfinie dans une classe fille, on peut se demander quelle méthode est effectivement appeler. Commençons par l'exemple suivant :

```java
public class Personnage {
    ...
  public String presentation(){
    return "je m'appelle "+nom ;
  }
}
```

```java
public class Gaulois extends Personnage {
  public Gaulois(String name){ super(name) ;}
  @Override[a]
  public String presentation(){
    return super.presentation() + "je suis un gaulois" ;
  }
  public void frappeRomains(){
    System.out.println("Qu'est-ce qu'on s'amuse") ;
  }
}
```

a. @Override est une annotation. Elle est une indication destinée au compilateur pour lui signifier qu'il s'agit d'une redéfinition de méthode. Le compilateur vigilant vérifiera que c'est bien le cas.

```java
public class Romain extends Personnage {

  public Romain(String name){ super(name) ;}
  @Override
  public String presentation(){
    return super().presentation() + " et je ne suis pas fou" ;
  }
```

```java
...
public static void main(String[] args){
  Random generator = new Random() ;
  Personnage mystere ;
  if (generator.nextBoolean())
    mystere = new Gaulois("Astérix") ;
  else
    mystere = new Romain("Jules") ;
  System.out.println( mystere.presentation()) ;
}
```

On trouve dans cet exemple une implémentation de la méthode `presentation()` dans chacune des classes. Dans la méthode `main`, on appelle la méthode `presentation()` d'un objet qui est déclaré comme un `Personnage`, mais qui est en fait soit un `Gaulois` soit un `Romain`[1]. Quelle méthode va être appelée ? Puisqu'on utilise un évènement aléatoire, on ne peut pas déduire au moment de la compilation le véritable type de l'objet `mystere`. JAVA détermine la méthode à utiliser au moment de l'exécution du programme, i.e., c'est une liasion *dynamique*. Lors de l'appel d'une méthode, JAVA remonte la hiérarchie de classes

1. Ici, on utilise la classe `Random` qui se trouve dans l'API de JAVA . Cette classe gère la création de séquence d'évènements pseudo aléatoire. Dans cet exemple, la méthode `nextBoolean` choisit de manière uniforme entre `true` et `false`.

jusqu'à trouver la méthode dont la signature correspond à l'appel. Donc dans l'exemple, JAVA va utiliser la méthode `presentation()` de la classe `Gaulois` si `mystere` est un `Gaulois`, et celle de la classe `Romain` si `mystere` est un `Romain`.

Attention ! Au moment de la compilation, on va vérifier si la méthode appliquée à un `Personnage` est bien une méthode de la classe `Personnage` ou de ses parentes. Dans l'exemple, il est vrai que la variable `mystere` fait référence à une instance de `Gaulois` ou `Romain`, néanmoins, on ne pourrait pas appeler `mystere.frappeRomains()` car la méthode `frappeRomains()` n'est définie que pour la classe `Gaulois` et non pour la classe `Personnage`.

5 le mot-clé `final`

Il est aussi possible d'indiquer qu'une méthode ne sera pas redéfinie dans une sous classe. Cela évitera à JAVA de faire la recherche de la bonne méthode à utiliser. Pour se faire, on peut utiliser le mot-clé `final`.

Ce mot-clé peut aussi être utilisé pour la définition d'une classe. Il indique alors que cette classe ne peut avoir de classes filles. Certaines classes de JAVA sont d'ailleurs déclarées comme `final`, comme par exemple la classe `String`. Le but est ici d'ordre de la sécurité : on veut être sûr que quand on manipule un `String`, on manipule bien un `String` fournie par JAVA et non une instance d'une sous classe qui pourrait faire des opérations non voulues.

Enfin, le mot-clé `final` peut aussi être utilisé pour déclarer un paramètre d'une méthode ou une variable locale d'une méthode. Cette qualification impose que le paramètre ou la variable ne sera pas modifiée (évidemment pour la variable, il faudra qu'elle soit initialisée avant la fin de l'exécution du constructeur).

6 La classe `Object`

En JAVA , que ce soit d'une façon directe ou non, toute classe hérite de la classe `Object`. Cette classe se trouve donc en haut de la hiérarchie des classes. La conséquence est qu'on pourra appeler n'importe quelle méthode définie dans la classe `Object` sur n'importe quel objet. Dans la suite, nous allons regarder plusieurs méthode de la classe `Object`. Ci-dessous, voici une partie du résumé des méthodes que l'on trouve dans la documentation JAVA .

Modifier and Type	Method Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this Object.
`int`	`hashCode()` Returns a hash code value for the object.
`String`	`toString()` Returns a string representation of the object.

6.1 toString

La méthode `toString` permet de retourner une réprésentation de l'objet avec une chaîne de caractères.

Comme c'est une méthode de la classe `Object`, on pourra donc toujours avoir une représentation d'un objet. L'implémentation dans la classe `Object` retourne le nom de la classe et le code de hachage (c'est un nombre qui identifie de maniere unique un objet, voir la section qui suit sur `hashCode()`).

Une manière un peu standard d'imprimer un objet est d'indiquer le nom de la classe suivie entre crochets de la liste des variable d'instance.

Les tableaux héritent aussi de la classe `Object`. Lorsque l'on utilise la méthode `toString`, on a une représentation quelque peu archaique, quelque chose comme `''[I@7852e922]''`. Ici `I[` désigne un tableau de `int`. Pour avoir une représentation plus standard, vous pouvez utiliser la méthode `toString` de la classe Arrays du package `java.util`.

```
1  import java.util.Arrays;
2  public class TableauInt {
3
4    public static void main(String[] args){
5      int[] chiffres = 0,1,2,3,4,5,6,7,8,9;
6      System.out.println(chiffres.toString());
7      System.out.println(Arrays.toString(chiffres));
8      }
9
10   }
```

L'exécution du code ci-dessus aura pour résultat :

```
[I@7852e922
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.2 hashCode

Un code de hachage est une empreinte servant à identifier rapidement, bien qu'incomplètement, un objet. Idéalement, on voudrait un code unique, mais cela pourrait prendre un peu de temps de calculer un code unique pour une instance d'une classe. Comme on veut une méthode rapide, on va sacrifier l'unicité : si deux objets ont un code de hachage différent, la probabilité que les objets soient différents est très haute.

JAVA utilise un entier. `Object.hasCode()` dérive un code de hachage qui peut utiliser la position en mémoire, un nombre associé à l'objet, ou une combinaison.

La chose qui est importante est que deux objets égaux doivent avoir le même code de hachage. La méthode `hashCode()` doit donc être cohérente avec la méthode `equals()`.

6.3 equals

L'égalité entre deux objets fait apparaître une subtilité. La méthode `equals` implémentée dans la classe `Object` determine seulement si les références sont identiques. Pour beaucoup de cas, c'est exactement ce que l'on désire.

Par contre, parfois, on peut vouloir vérifier que deux objets ont le même contenu. Par exemple, on peut avoir deux instances de la classe `String` (qui auront donc des références différentes), mais qui seront

composés de la même chaîne de caractères. Dans ce cas, on voudra redéfinir la méthode `equals` pour comparer le contenu des objets. Attention alors dans ce cas que la méthode `hashCode()` reste cohérente avec la méthode `equals` ainsi redéfinie.

6.4 `clone`

La méthode clone est aussi subtile, et pas forcément souvent nécessaire. Ne redéfinissez pas cette méthode si vous n'avez pas une bonne raison de le faire.

Le but du clonage est de faire une nouvelle instance qui ait le même état que l'objet original. Si un des objets est modifié, l'autre devrait rester inchangé.

La méthode est déclarée comme `protected`, vous devez donc si vous voulez que les utilisateurs de votre classe puisse cloner des instances.

L'implémentation de la classe `Object` fait une copie superficielle : elle copie toutes les variables d'instance de l'objet original. Pour les variables primitives, cela est parfait. Cependant, pour les variables qui sont des objets, JAVA va simplement copier la référence.

Prenons un exemple. On a une classe `Episode` qui contient un tableau de `Personnage`. Supposons qu'on a un premier épisode avec deux personnages Astérix et Obélix. On clone ce premier episode pour en faire un second. On ajoute au second épisode le personnage d'Idéfix. Avec un clone superficiel, le tableau n'est pas dupliqué : chaque épisode réfere au même tableau en mémoire. Ainsi, les deux épisodes auront trois personnages : Astérix, Obélix et Idéfix !

6.5 `finalize`

Cette méthode sert lorsque « garbage collector » détruit l'objet (voir Section 4). Si on veut ajouter un traitement particulier lors de la destruction de l'objet, on peut ajouter le code dans cette méthode. Par exemple si l'objet est utilisé pour de la communication, l'appel de cette méthode peut servir à terminer proprement les communications.

La méthode `finalize()` de la classe `Object` ne fait en fait rien, les sous classes peuvent rédéfinir un comportement plus particulier.

6.6 `getClass`

Cette méthode permet de récupérer un objet de type `Class` qui est automatiquement généré par la machine virtuelle de JAVA . A chaque création d'objet, la machine virtuelle crée un objet de type `Class` associé qui peut permettre d'apprendre beaucoup sur l'objet initial.

7 Classes et méthodes abstraites

Prenons l'exemple de notre classe `Personnage`. Lorsque l'on va utiliser notre programme, on n'utilisera jamais d'objet de la classe `Personnage` : à chaque fois, on utilisera une classe plus précise (`Romain`, `Gaulois`, `Egyptien`, `Chien`, etc.). La classe `Personnage` est néanmois utile comme racine de la hiérarchie et pour les méthodes que toutes ses classes filles possèderont. Pour certaines méthodes, l'implémentation dans la classe `Personnage` sera donc inutile si toutes les classes filles redéfinissent cette méthode. On voudrait déclarer ces méthodes dans `Personnage` mais sans fournir leur implémentation, et forcer à ce que l'implémentation soit fournie dans les classes filles. C'est exactement ce que propose les classes ou les méthodes dites *abstraites*.

Une méthode abstraite est une méthode qui ne possède pas de corps, seulement la signature de la méthode est donnée. Ainsi, une telle méthode doit être obligatoirement implémentée dans toutes les classes filles. L'intérêt est que l'on pourra appeler la méthode de la même façon pour tous les objets dérivés.

Une classe abstraite est une classe qui contient au moins une méthode abstraite. Une classe abstraite ne peut être instanciée directement (i.e., on ne peut pas écrire `new`), elle n'est utilisée qu'à travers sa descendance. Une classe dérivée d'une classe abstraite et qui ne redéfinit pas toutes les méthodes abstraites est elle-même abstraite.

Pour la syntaxe, les classes et les méthodes abstraites doivent être précédées du mot-clé `abstract`. Pour notre exemple, on peut faire de la classe `Personnage` une classe abstraite. Lorsqu'on veut instancier un vrai personnage, ce personnage aura un type plus précis (i.e., ce sera un gaulois, un romain, un goth, etc). Mais tous les sous-types devront implémenter un même ensemble de méthode. Ainsi quand on voudra manipuler les personnages en général, on pourra utiliser le type `Personnage`.

```
1 public abstract class Personnage {
2    public Personnage(String name) ;
3    public abstract presentation() ;
4 }
```

8 Héritage et types énumérés

On a vu dans le chapitre 7 une description simple des types énumérés grâce au mot clé `enum`. Ce qu'on vous a caché alors, c'est qu'un type énuméré héritent de la classe `java.lang.Enum`. C'est cet héritage qui permet entre autre l'utilisation des méthodes `ordinal()` et `values()`.

On peut donc redéfinir dans un type énuméré des constructeurs, des méthodes et des variables.

Attention, le constructeur d'un type énuméré est toujours `private` (ceci pour préserver les valeurs définies dans l'`enum`). Vous pouvez omettre le mot clé (mais mettre `public` ou `protected` provoquera une erreur de syntaxe). Dans l'exemple qui suit, les valeurs possibles du type énuméré sont `SMALL`, `MEDIUM`, `LARGE` et `EXTRA_LARGE`. A l'aide du constructeur, on va pouvoir ajouter de l'information : ici on ajoute une abbréviation pour la taille et une chaîne qui représente la taille française équivalente. Chaque valeur du type énuméré appelera donc une fois le constructeur. Dans la méthode `main`, on utilise donc la valeur `MEDIUM` et on demande son abbréviation et sa taille française équivalente.

```java
public enum Size {
   SMALL("S", "36/38"),
   MEDIUM("M", "40/42"),
   LARGE("L","44/46"),
   EXTRA_LARGE("XL","48/50") ;

   private String abbreviation ;
   private String eqFrance ;

   Size(String abbreviation, String eqFrance) {
     this.abbreviation = abbreviation ;
     this.eqFrance = eqFrance ;
   }

   public String getAbbreviation() { return abbreviation ; }
   public String getEqFrance(){ return eqFrance ; }

   public static void main(String[] args){
     Size mySize = Size.MEDIUM ;
     System.out.println("my size " + mySize.getAbbreviation()
               + " is equivalent in France to " + mySize.getEqFrance()) ;
    }
}
```

9 Interfaces

Commencons par un exemple. Parmi les personnages, certains sont des combattants, d'autres non. Par exemples certains gaulois vont aller combattre, mais pas tous (le druide par exemple ne combat pas), de même chez les romains, certains romains sont dans l'administration et ne combattent pas.

Une premiere solution serait de faire des sous classes de `Romain` et `Gaulois` et d'ajouter des méthodes pour combattre. Mais on sent bien qu'on va sûrement dupliquer du code. On serait tenté de faire un classe `Combattant` et on déserirerais que certains romains héritent à la fois de Combattant et de Romain. Malheureusement, JAVA permet seulement un héritage simple : on ne peut hériter que d'une seule classe. Les interfaces vont nous permettre de traiter ce probleme. Les interfaces offrent un mécanisme pour réaliser une sorte d'héritage multiple.

Une interface est un ensemble de déclarations et de constantes qui ressemble à une classe abstraite. C'est une sorte de standard qu'une classe peut suivre : dans notre exemple, l'interface combattant va définir toutes les méthodes qu'un combattant doit posséder. Pour suivre le standard, une classe doit posséder les méthodes déclarées dans l'interface. Dans ce cas, on dit que la classe *implémente* l'interface. Une classe peut suivre à la fois plusieurs standard, i.e. une classe peut implémenter plusieurs interfaces.

Pour déclarer une interface, on peut utiliser le modèle ci-dessous :

```
[public] interface <nom interface> [extends <nom interface 1> <nom interface 2> ... ] {
   // méthodes ou des attributs static
}
```

Tout attribut est implicitement déclaré comme `public`, `static` et `final`.

Dans les anciennes versions de JAVA , les méthodes d'une interface était toujours abstraites : aucune n'avait un corps et chaque classe qui implémentait l'interface devait avoir son implémentation de chacune des méthodes de l'interface. JAVA offre désormais la possibilité d'avoir des implémentations : on pourra ainsi trouver des méthodes `static` et des méthodes par défaut. Les méthodes `static` seront par exemples des outils. Les méthodes par défaut devront avoir le mot clé `default`. Dqns l'exemple ci-dessous, deux méthodes devront être implémentés par les classes qui implémenteront l'interface (`attack` et `defense`) et la méthode `runAway` possède une implémentation par défaut.

```
1 public interface Fighter {
2    public void attack(Personnage p) ;
3    public void defense(Combattant c) ;
4    default void runAway(){goHome() ;}
5 }
```

```
1  public class IrreductibleGaulois implements Combattant {
2      ...
3     public void attaque(Personnage p){
4       gourdePotionMagique.bois() ;
5       while(p.isDebout())
6         coupsDePoing(p) ;
7     }
8
9     public void defend(Combattant c){
10      esquive() ;
11      attaque(c) ;
12    }
13 }
```

L'utilisation d'interfaces s'inscrit dans la notion de polymorphisme. Un objet qui implémente une interface possède aussi le « type »de l'interface. Si la classe `Romain` implémente aussi l'interface `Combattant`, on pourra donc avoir un tableau de `Combattants` qui contiendra donc des objets qui implémentent l'interface `Combattant`, qu'ils soient des `Romains` ou des `IrreductiblesGaulois`. De plus, ceci permet d'avoir des méthodes spécialisées pour chaque type de combattants : les `Romains` n'auront évidemment pas accès à la potion magique pour combattre !

Notez qu'une classe pouvant implémenter plusieurs interfaces, il peut y avoir un conflit entre méthodes. Le compilateur indiquera une erreur et il faudra lever l'ambiguité. Dqns l'exemple ci-dessous, on appelle l'implémentation par défaut de l'interface `Identified`.

```
1 public interface Person {
2    String getName() ;
3    default int getId() { return 0 ; }
4    }
```

```
1 public interface Identified {
2    default int getId() { return Math.abs(hashCode()) ; }
3    }
```

```
1 public class Employee implements Person, Identified {
2   public int getId() {
3     return Identified.super.getId() ;
4     }
5   ...
6   }
```

10 Information durant l'exécution

Nous avons vu l'instruction `instance of` qui permet de savoir si un objet est une instance d'une classe (voir Section 2). Prenons un nouvel example inspiré de la section 4.2.

```
1  ...
2  public static void main(String[] args){
3     Random generator = new Random() ;
4     Personnage mystere ;
5     if (generator.nextBoolean())
6       mystere = new Gaulois("Astérix") ;
7     else
8       mystere = new Romain("Jules") ;
9     System.out.println( mystere instance of Gaulois) ;
10 }
```

Notez bien que dans l'example, on ne pouvait pas connaître le type de l'objet au moment de la compilation, mais seulement au moment de l'exécution du code. Nous allons voir maintenant d'autres moyen d'avoir des informations et de les utiliser pendant l'exécution.

10.1 La classe `Class`

Nous avons vu une méthode de la classe `Object` qui s'appelait `getClass` et qui retournait un objet de type `Class` (pour le moment, on va ignore les symboles `<?>`) qui est associé à la classe de l'objet en question. Grâce à cet objet `Class`, on va pouvoir accéder à beaucoup d'information sur la classe de l'objet, par exemple :

— on peut connaître le nom de sa classe avec la méthode `getName()`
— on peut connaître les interfaces implémentées par la classe
— on peut connaître toutes les méthodes, les variables d'instances, les constructeurs
— on peut même générer une nouvelle instance

L'exemple ci-dessous donnera toutes les méthodes de la classe `String`. On ne va pas ici donner tous les détails. Dans la boucle `for` ligne 8 on itère sur chaque méthode déclarée (on remarque le type `Method`), et pour chaque méthode `m`, on écrit le visibilité (« modifier ». i.e. `public`, `private`, ou `protected`), le type de retour, le nom de la méthode, et enfin la liste des arguments.

```java
import java.lang.reflect.*;
import java.util.Arrays;
public class Exemple{
   public static void main(String[] args){
     String word = "stephane";
     Class<?> cl = word.getClass();
     while (cl != null) {
       for (Method m : cl.getDeclaredMethods()) {
         System.out.println(
           Modifier.toString(m.getModifiers()) + " " +
           m.getReturnType().getCanonicalName() + " " +
           m.getName() +
           Arrays.toString(m.getParameters()));
       }
       cl = cl.getSuperclass();
     }
   }
}
```

Les premieres lignes de l'exécution sont les suivantes

```
public boolean equals[java.lang.Object arg0]
public java.lang.String toString[]
public int hashCode[]
...
```