Attributs & méthodes

Dans ce chapitre, nous allons revenir sur la déclaration d'une classe en Java et détailler les notions d'attributs et de méthodes.

Les attributs

Les attributs représentent l'état interne d'un objet. Nous avons vu précédemment qu'un attribut a une portée, un type et un identifiant. Il est déclaré de la façon suivante dans le corps de la classe :

```
[portée] [type] [identifiant];
```

```
public class Voiture {

  public String marque;
  public float vitesse;

}
```

La classe ci-dessus ne contient que des attributs, elle s'apparente à une simple structure de données. Il est possible de créer une instance de cette classe avec l'opérateur **new** et d'accéder aux attributs de l'objet créé avec l'opérateur **.** :

```
Voiture v = new Voiture();
v.marque = "DeLorean";
v.vitesse = 88.0f;
```

La portée

Jusqu'à présent, nous avons vu qu'il existe deux portées différentes : **public** et **private**. Java est un langage qui supporte l'encapsulation de données. Cela signifie que lorsque nous créons une classe nous avons le choix de laisser accessible ou non les attributs et les méthodes au reste du programme.

Pour l'instant nous distinguerons les portées :

public
: Signale que l'attribut est visible de n'importe quelle partie de l'application.

private
: Signale que l'attribut n'est accessible que par l'objet lui-même ou par un objet du même type. Donc seules les méthodes de la classe déclarant cet attribut peuvent accéder à cet attribut.

Lorsque nous parlerons de l'encapsulation et du principe du *ouvert/fermé*, nous verrons qu'il est très souvent préférable qu'un attribut ait une portée *private*.

L'initialisation

En Java, on peut indiquer la valeur d'initialisation d'un attribut pour chaque nouvel objet.

```
public class Voiture {

  public String marque = "DeLorean";
  public float vitesse = 88.0f;

}
```

En fait, un attribut possède nécessairement une valeur par défaut qui dépend de son type :

Initialisation par défaut des attributs

Type	Valeur par défaut
boolean	false
char	`'\0'`
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
référence d'objet	null

Donc, écrire ceci :

```
public class Voiture {

  public String marque;
  public float vitesse;
```

```
}
```

ou ceci

```
public class Voiture {

  public String marque = null;
  public float vitesse = 0.0f;

}
```

est strictement identique en Java.

attributs finaux

Un attribut peut être déclaré comme **final**. Cela signifie qu'il n'est plus possible d'affecter une valeur à cet attribut une fois qu'il a été initialisé. Dans cas, le compilateur exige que l'attribut soit initialisé *explicitement*.

```
public class Voiture {

  public String marque;
  public float vitesse;
  public final int nombreDeRoues = 4;

}
```

L'attribut *Voiture.nombreDeRoues* sera initialisé avec la valeur 4 pour chaque instance et ne pourra plus être modifié.

```
Voiture v = new Voiture();
v.nombreDeRoues = 5; // ERREUR DE COMPILATION
```

> **final** porte sur l'attribut et empêche sa modification. Par contre si l'attribut est du type d'un objet, il est possible de modifier l'état de cet objet.

Pour une application d'un concessionnaire automobile, nous pouvons créer un objet *Facture* qui contient un attribut de type *Voiture* et le déclarer **final**.

```
public class Facture {

  public final Voiture voiture = new Voiture();

}
```

Sur une instance de *Facture*, on ne pourra plus modifier la référence de l'attribut *voiture* par contre, on pourra toujours modifier les attributs de l'objet référencé

```
Facture facture = new Facture();
facture.voiture.marque = "DeLorean"; // OK
facture.voiture = new Voiture() // ERREUR DE COMPILATION
```

Attributs de classe

Jusqu'à présent, nous avons vu comment déclarer des attributs d'objet. C'est-à-dire que chaque instance d'une classe aura ses propres attributs avec ses propres valeurs représentant l'état interne de l'objet et qui peuvent évoluer au fur et à mesure de l'exécution de l'application.

Mais il est également possible de créer des *attributs de classe*. La valeur de ces attributs est partagée par l'ensemble des instances de cette classe. Cela signifie que si on modifie la valeur d'un attribut de classe dans un objet, la modification sera visible dans les autres objets. Cela signifie également que cet attribut existe au niveau de la classe et est donc accessible même si on ne crée aucune instance de cette classe.

Pour déclarer un attribut de classe, on utilise le mot-clé **static**.

```
public class Voiture {

  public static int nombreDeRoues = 4;
  public String marque;
  public float vitesse;

}
```

Dans l'exemple ci-dessus, l'attribut *nombreDeRoues* est maintenant un attribut de classe. C'est une façon de suggérer que toutes les voitures de notre application ont le même nombre de roues. Cette caractéristique appartient donc à la classe plutôt qu'à chacune de ses instances. Il est donc possible d'accéder directement à cet attribut depuis la classe :

```
System.out.println(Voiture.nombreDeRoues);
```

Notez que dans l'exemple précédent, out est également un attribut de la classe System. Si vous vous rendez sur la documentation de cette classe, vous constaterez que out est déclaré comme **static** dans cette classe. Il s'agit d'une autre utilisation des attributs de classe : lorsqu'il n'existe qu'une seule instance d'un objet pour toute une application, cette instance est généralement accessible grâce à un attribut **static**. C'est une des façons d'implémenter le design pattern singleton en Java. Dans notre exemple, out est l'objet qui représente la sortie standard de notre application. Cet objet est unique pour toute l'application et nous n'avons pas à le créer car il existe dès le lancement.

Si le programme modifie un attribut de classe, alors la modification est visible depuis toutes les instances :

```
Voiture v1 = new Voiture();
Voiture v2 = new Voiture();

System.out.println(v1.nombreDeRoues); // 4
System.out.println(v2.nombreDeRoues); // 4

// modification d'un attribut de classe
v1.nombreDeRoues = 5;

Voiture v3 = new Voiture();

System.out.println(v1.nombreDeRoues); // 5
System.out.println(v2.nombreDeRoues); // 5
System.out.println(v3.nombreDeRoues); // 5
```

Le code ci-dessus, même s'il est parfaitement correct, peut engendrer des difficultés de compréhension. Si on ne sait pas que *nombreDeRoues* est un attribut de classe, on peut le modifier en pensant que cela n'aura pas d'impact sur les autres instances. C'est notamment pour cela que Eclipse émet un avertissement si on accède ou si on modifie un attribut de classe à travers un objet. Même si l'effet est identique, il est plus lisible d'accéder à un tel attribut à travers le nom de la classe uniquement :

```
System.out.println(Voiture.nombreDeRoues); // 4

Voiture.nombreDeRoues = 5;

System.out.println(Voiture.nombreDeRoues); // 5
```

Attributs de classe finaux

Il n'existe pas de mot-clé pour déclarer une constante en Java. Même si **const** est un mot-clé, il n'a aucune signification dans le langage. On utilise donc la combinaison des mots-clés **static** et **final** pour déclarer une constante. Par convention, pour les distinguer des autres attributs, on écrit leur nom en majuscules et les mots sont séparés par _.

```
public class Voiture {

  public static final int NOMBRE_DE_ROUES = 4;
  public String marque;
  public float vitesse;

}
```

> Rappelez-vous que si l'attribut référence un objet, **final** n'empêche pas d'appeler des méthodes qui vont modifier l'état interne de l'objet. On ne peut vraiment parler de constantes que pour les attributs de type primitif.

Les méthodes

Les méthodes permettent de définir le comportement des objets. nous avons vu précédemment qu'une méthode est définie pas sa **signature** qui spécifie sa portée, son type de retour, son nom et ses paramètres entre parenthèses. La signature est suivie d'un bloc de code que l'on appelle le **corps** de méthode.

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

Dans ce corps de méthode, il est possible d'avoir accès au attribut de l'objet. Si la méthode modifie la valeur des attributs de l'objet, elle a un *effet de bord* qui change l'état interne de l'objet. C'est le cas dans l'exemple ci-dessous pour la méthode *accelerer* :

```
public class Voiture {

  private float vitesse;

  /**
   * @return La vitesse en km/h de la voiture
   */
  public float getVitesse() {
```

```
    return vitesse;
  }

  /**
   * Pour accélérer la voiture
   * @param deltaVitesse Le vitesse supplémentaire
   */
  public void accelerer(float deltaVitesse) {
    vitesse = vitesse + deltaVitesse;
  }
}
```

Il est possible de créer une instance de la classe ci-dessus avec l'opérateur **new** et d'exécuter les méthodes de l'objet créé avec l'opérateur **.** :

```
Voiture v = new Voiture();
v.accelerer(88.0f);
```

La portée

Comme pour les attributs, les méthodes ont une portée, c'est-à-dire que le développeur de la classe peut décider si une méthode est accessible ou non au reste du programme. Pour l'instant, nous distinguons les portées :

public
: Signale que la méthode est appelable de n'importe quelle partie de l'application. Les méthodes publiques définissent le contrat de la classe, c'est-à-dire les opérations qui peuvent être demandées par son environnement.

private
: Signale que la méthode n'est appelable que par l'objet lui-même ou par un objet du même type. Les méthode privées sont des méthodes utilitaires pour un objet. Elles sont créées pour mutualiser du code ou pour simplifier un algorithme en le fractionnant en un ou plusieurs appels de méthodes.

La valeur de retour

Une méthode peut avoir au plus un type de retour. Le compilateur signalera une erreur s'il existe un chemin d'exécution dans la méthode qui ne renvoie pas le bon type de valeur en retour. Pour retourner une valeur, on utilise le mot-clé **return**. Si le type de retour est un objet, la méthode peut toujours retourner la valeur spéciale **null**, c'est-à-dire l'absence d'objet. Une méthode qui ne

retourne aucune valeur, le signale avec le mot-clé **void**.

```
public class Voiture {

  private String marque;
  private float vitesse;

  public float getVitesse() {
    return vitesse;
  }

  public void setMarque(String nouvelleMarque) {
    if (nouvelleMarque == null) {
      return;
    }
    marque = nouvelleMarque;
  }

}
```

Les paramètres

Un méthode peut éventuellement avoir des paramètres (ou arguments). Chaque paramètre est défini par son type et par son nom.

```
public class Voiture {

  public float getVitesse() {
    // implémentation ici
  }

  public void setVitesse(float deltaVitesse) {
    // implémentation ici
  }

  public void remplirReservoir(float quantite, TypeEssence typeEssence) {
    // implémentation ici
  }

}
```

Il est également possible de créer une méthode avec un nombre variable de paramètres (*varargs parameter*). On le signale avec trois points après le type du paramètre.

```
public class Calculatrice {

  public int additionner(int... valeurs) {
    int resultat = 0;
    for (int valeur : valeurs) {
      resultat += valeur;
    }
    return resultat;
  }
}
```

Le paramètre variable est vu comme un tableau dans le corps de la méthode. Par contre, il s'agit bien d'une liste de paramètre au moment de l'appel :

```
Calculatrice calculatrice = new Calculatrice();

System.out.println(calculatrice.additionner(1)); // 1
System.out.println(calculatrice.additionner(1, 2, 3)); // 6
System.out.println(calculatrice.additionner(1, 2, 3, 4)); // 10
```

L'utilisation d'un paramètre variable obéit à certaines règles :

1. Le paramètre variable doit être le dernier paramètre
2. Il n'est pas possible de déclarer un paramètre variable acceptant plusieurs types

Au moment de l'appel, le paramètre variable peut être omis. Dans ce cas le tableau passé au corps de la méthode est un tableau vide. Un paramètre variable est donc également optionnel.

```
Calculatrice calculatrice = new Calculatrice();

System.out.println(calculatrice.additionner()); // 0
```

Il est possible d'utiliser un tableau pour passer des valeurs à un paramètre variable. Cela permet notamment d'utiliser un paramètre variable dans le corps d'une méthode comme paramètre variable à l'appel d'une autre méthode.

```
Calculatrice calculatrice = new Calculatrice();

int[] valeurs = {1, 2, 3};
```

```
System.out.println(calculatrice.additionner(valeurs)); // 6
```

Pour l'exemple de la calculatrice, il peut sembler *naturel* d'obliger à passer au moins deux paramètres à la méthode *additionner*. Dans ce cas, il faut créer une méthode à trois paramètres :

```
public class Calculatrice {

  public int additionner(int valeur1, int valeur2, int... valeurs) {
    int resultat = valeur1 + valeur2;
    for (int valeur : valeurs) {
      resultat += valeur;
    }
    return resultat;
  }
}
```

Paramètre final

Un paramètre peut être déclaré **final**. Cela signifie qu'il n'est pas possible d'assigner une nouvelle valeur à ce paramètre.

```
public class Voiture {

  public void accelerer(final float deltaVitesse) {
    deltaVitesse = 0.0f; // ERREUR DE COMPILATION

    // ...
  }
}
```

Rappelez-vous que **final** ne signifie pas réellement constant. En effet si le type d'un paramètre **final** est un objet, la méthode pourra tout de même appeler des méthodes sur cet objet qui modifient son état interne.

> Java n'autorise que le passage de paramètre par copie. Assigner une nouvelle valeur à un paramètre n'a donc un impact que dans les limites de la méthode. Cette pratique est généralement considérée comme mauvaise car cela peut rendre la compréhension du code de la méthode plus difficile. **final** est donc un moyen de nous aider à vérifier au moment de la compilation que nous n'assignons pas par erreur une nouvelle valeur à un paramètre. Cet usage reste tout de même très limité. Nous reviendrons plus tard sur l'intérêt principal de

> déclarer un paramètre **final** : la déclaration de classes anonymes.

Les variables

Il est possible de déclarer des variables où l'on souhaite dans une méthode. Par contre, contrairement aux attributs, les variables de méthode n'ont pas de valeur par défaut. Cela signifie qu'il est obligatoire d'initialiser les variables. Il n'est pas nécessaire de les initialiser dès la déclaration, par contre, elles doivent être initialisées avant d'être lues.

Méthodes de classe

Les méthodes définissent un comportement d'un objet et peuvent accéder aux attributs de l'instance. À l'instar des attributs, il est également possible de déclarer des *méthodes de classe*. Une méthode de classe ne peut pas accéder aux attributs d'un objet mais elle peut toujours accéder aux éventuels attributs de classe.

Pour déclarer une méthode de classe, on utilise le mot clé **static**.

```
public class Calculatrice {


  public static int additionner(int... valeurs) {
    int resultat = 0;
    for (int valeur : valeurs) {
      resultat += valeur;
    }
    return resultat;
  }
}
```

Comme pour l'exemple précédent, les méthodes de classe sont souvent des méthodes utilitaires qui peuvent s'exécuter sans nécessiter le contexte d'un objet. Dans un autre langage de programmation, il s'agirait de simples fonctions.

Les méthodes de classe peuvent être invoquées directement à partir de la classe. Donc il n'est pas nécessaire de créer une instance.

```
int resultat = Calculatrice.additionner(1, 2, 3, 4);
```

> Certaines classes de l'API Java ne contiennent que des méthodes de classe. On parle de classes utilitaires ou de classes outils puisqu'elles s'apparentent à une collection de fonctions. Parmi les plus utilisées, on trouve les classes java.lang.Math, java.lang.System,

> java.util.Arrays et java.util.Collections.

Il est tout à fait possible d'invoquer une méthode de classe à travers une variable pointant sur une instance de cette classe :

```
Calculatrice c = new Calculatrice();
int resultat = c.additionner(1, 2, 3, 4);
```

Cependant, cela peut engendrer des difficultés de compréhension puisque l'on peut penser, à tord, que la méthode *additionner* peut avoir un effet sur l'objet. C'est notamment pour cela que Eclipse émet un avertissement si on invoque une méthode de classe à travers un objet. Même si l'effet est identique, il est plus lisible d'invoquer une méthode de classe à partir de la classe elle-même.

La méthode de classe la plus célèbre en Java est sans doute **main**. Elle permet de définir le point d'entrée d'une application dans une classe :

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

Les paramètres *args* correspondent aux paramètres passés en ligne de commande au programme **java** après le nom de la classe :

```
$ java MaClasse arg1 arg2 arg3
```

Surcharge de méthode : overloading

Il est possible de déclarer dans une classe plusieurs méthodes ayant le même nom. Ces méthodes doivent obligatoirement avoir des paramètres différents (le type et/ou le nombre). Il est également possible de déclarer des types de retour différents pour ces méthodes. On parle de surcharge de méthode (**method overloading**). La surcharge de méthode n'a réellement de sens que si les méthodes portant le même nom ont un comportement que l'utilisateur de la classe jugera proche. Java permet également la surcharge de méthode de classe.

```
public class Calculatrice {

  public static int additionner(int... valeurs) {
    int resultat = 0;
    for (int valeur : valeurs) {
      resultat += valeur;
    }
```

```
    return resultat;
  }

  public static float additionner(float... valeurs) {
    float resultat = 0;
    for (float valeur : valeurs) {
      resultat += valeur;
    }
    return resultat;
  }
}
```

Dans l'exemple ci-dessus, la surcharge de méthode permet supporter l'addition pour le type entier et pour le type à virgule flottante. Selon le type de paramètre passé à l'appel, le compilateur déterminera laquelle des deux méthodes doit être appelée.

```
int resultatEntier = Calculatrice.additionner(1,2,3);
float resultat = Calculatrice.additionner(1f,2.3f);
```

> N'utilisez pas la surcharge de méthode pour implémenter des méthodes qui ont des comportements trop différents. Cela rendra vos objets difficiles à comprendre et donc à utiliser.

Si on surcharge une méthode avec un paramètre variable, cela peut créer une ambiguïté de choix. Par exemple :

```
public class Calculatrice {

  public static int additionner(int v1, int v2) {
    return v1 + v2;
  }

  public static int additionner(int... valeurs) {
    int resultat = 0;
    for (int valeur : valeurs) {
      resultat += valeur;
    }
    return resultat;
  }
```

```
}
```

Si on fait appel à la méthode *additionner* de cette façon :

```
Calculatrice.additionner(2, 2);
```

Alors les deux méthodes *additionner* peuvent satisfaire cet appel. La règle appliquée par le compilateur est de chercher d'abord une correspondance parmi les méthodes qui n'ont pas de paramètre variable. Donc pour notre exemple ci-dessus, la méthode *additionner(int, int)* sera forcément choisie par le compilateur.

Portée des noms et this

Lorsqu'on déclare un identifiant, qu'il s'agisse du nom d'une classe, d'un attribut, d'un paramètre, d'une variable…, il se pose toujours la question de sa portée : dans quel contexte ce nom sera-t-il compris par le compilateur ?

Pour les paramètres et les variables, la portée de leur nom est limitée à la méthode qui les déclare. Cela signifie que vous pouvez réutiliser les mêmes noms de paramètres et de variables dans deux méthodes différentes pour désigner des choses différentes.

Plus précisément, le nom d'une variable est limité au bloc de code (délimité par des accolades) dans lequel il a été déclaré. En dehors de ce bloc, le nom est inaccessible.

```
public int doSomething(int valeurMax) {
  int resultat = 0;

  // la variable i n'est accessible que dans la boucle for
  for (int i = 0; i < 10; ++i) {

    // la variable k n'est accessible que dans la boucle for
    for (int k = 0; k < 10; ++k) {
      // la variable m n'est accessible que dans ce bloc
      int m = resultat + i * k;
      if (m > valeurMax) {
        return valeurMax;
      }
      resultat = m;
    }
  }
```

```
  return resultat;
}
```

En Java, le masquage de nom de variable ou de nom de paramètre est interdit. Cela signifie qu'il est impossible de déclarer une variable ayant le même nom qu'un paramètre ou qu'une autre variable accessible dans le bloc de code courant.

```
public int doSomething(int valeurMax) {
  int valeurMax = 2; // ERREUR DE COMPILATION
}
```

```
public int doSomething(int valeurMax) {
  int resultat = 0;
  for (int i = 0; i < 10; ++i) {
    resultat += i;
    if (resultat > 10) {
      int resultat = -1; // ERREUR DE COMPILATION
      return resultat;
    }
  }
  return resultat;
}
```

Par contre, il est tout à fait possible de réutiliser un nom de variable dans deux blocs de code successifs. Cette pratique n'est vraiment utile que pour les variables temporaires (comme pour une boucle **for** contrôlée par un index). Sinon, cela gène généralement la lecture.

```
public void doSomething(int valeurMin, int valeurMax) {
  for (int i = 0; i < valeurMax; ++i) {
    // implémentation
  }

  // on peut réutiliser le nom de variable i car il est déclaré
  // dans deux blocs for différents
  for (int i = 0; i < valeurMin; --i) {
    // implémentation
  }
}
```

En Java, le masquage du nom d'un attribut par un paramètre ou une variable est autorisé car les attributs sont toujours accessibles à travers le mot-clé **this**.

```
public class Voiture {
  private String marque;

  public void setMarque(String marque) {
    this.marque = marque;
  }
}
```

this désigne l'instance courante de l'objet dans une méthode. On peut l'envisager comme une variable implicite accessible à un objet pour le désigner lui-même. Avec **this**, on peut accéder aux attributs et aux méthodes de l'objet. Il est même possible de retourner la valeur **this** ou la passer en paramètre pour indiquer une référence de l'objet courant :

```
public class Voiture {
  private float vitesse;

  public Voiture getPlusRapide(Voiture voiture) {
    return this.vitesse >= voiture.vitesse ? this : voiture;
  }
}
```

S'il n'y a pas d'ambiguïté de nom, l'utilisation du mot-clé **this** est inutile. Cependant, certains développeurs préfèrent l'utiliser systématiquement pour indiquer explicitement l'accès à un attribut.

> **this** désignant l'objet courant, ce mot-clé n'est pas disponible dans une méthode de classe (méthode **static**). Pour résoudre le problème du masquage des attributs de classe dans ces méthodes, il suffit d'accéder au nom à travers le nom de la classe.

Principe d'encapsulation

Un objet est constitué d'un état interne (l'ensemble de ses attributs) et d'une liste d'opérations disponibles pour ses clients (l'ensemble de ses méthodes publiques). En programmation objet, il est important que les clients d'un objet en connaissent le moins possible sur son état interne. Nous verrons plus tard avec les mécanismes d'héritage et d'interface qu'un client demande des services à un objet sans même parfois connaître le type exact de l'objet. La programmation objet introduit un niveau d'abstraction important et cette abstraction devient un atout pour la réutilisation et l'évolutivité.

Prenons l'exemple d'une classe permettant d'effectuer une connexion FTP et de récupérer un fichier distant. Les clients d'une telle classe n'ont sans doute aucun intérêt à comprendre les

mécanismes compliqués du protocole FTP. Ils veulent simplement qu'on leur rende un service. Notre classe FTP pourrait très grossièrement ressembler à ceci :

```
public class ClientFtp {


  /**
  * @param uri l'adresse FTP du fichier
  *            par exemple ftp://monserveur/monfichier.txt
  * @return le fichier sous la forme d'un tableau d'octets
  */
  public byte[] getFile(String uri) {
    // implémentation
  }


}
```

Cette classe a peut-être des attributs pour connaître l'état du réseau et maintenir des connexions ouvertes vers des serveurs pour améliorer les performances. Mais tout ceci n'est pas de la responsabilité du client de cette classe qui veut simplement récupérer un ficher. Il est donc intéressant de cacher aux clients l'état interne de l'objet pour assurer un *couplage faible de l'implémentation*. Ainsi, si les développeurs de la classe *ClientFtp* veulent modifier son implémentation, ils doivent juste s'assurer que les méthodes publiques fonctionneront toujours comme attendues par les clients.

En programmation objet, le principe d'encapsulation nous incite à contrôler et limiter l'accès au contenu de nos classes au strict nécessaire afin de permettre le couplage le plus faible possible. L'encapsulation en Java est permise grâce à la portée **private**.

On considère que tous les attributs d'une classe **doivent** être déclarés **private** afin de satisfaire le principe d'encapsulation.

Cependant, il est parfois utile pour le client d'une classe d'avoir accès à une information qui correspond à un attribut de l'état interne de l'objet. Plutôt que de déclarer cet attribut **public**, il existe en Java des méthodes dont la signature est facilement identifiable et que l'on nomme **getters** et **setters** (les accesseurs). Ces méthodes permettent d'accéder aux **propriétés** d'un objet ou d'une classe.

getter

> Permet l'accès en lecture à une propriété. La signature de la méthode se présente sous la forme :

```
public type getNomPropriete() {
  // ...
}
```

Pour un type booléen, on peut aussi écrire :

```
public boolean isNomPropriete() {
  // ...
}
```

setter

Permet l'accès en écriture à une propriété. La signature de la méthode se présente sous la forme :

```
public void setNomPropriete(type nouvelleValeur) {
  // ...
}
```

Ce qui donnera pour notre classe *Voiture* :

```
public class Voiture {

  // La vitesse en km/h
  private float vitesse;

  /**
  * @return La vitesse en km/h
  */
  public float getVitesse() {
    return vitesse;
  }

  /**
  * @param vitesse La vitesse en km/h
  */
  public void setVitesse(float vitesse) {
    this.vitesse = vitesse;
  }

}
```

Les *getters/setters* introduisent une abstraction supplémentaire : la **propriété**. Une propriété peut correspondre à un attribut ou à une expression. Du point de vue du client de la classe, cela n'a pas d'importance. Dans l'exemple ci-dessus, les développeurs de la classe *Voiture* peuvent très bien décider que l'état interne de la vitesse sera exprimé en mètres par seconde. Il devient possible de conserver la cohérence de notre classe en effectuant les conversions nécessaires pour passer de la propriété en km/s à l'attribut en m/s et inversement.

```
public class Voiture {

  // vitesse en m/s
  private float vitesse;

  private static float convertirEnMetresSeconde(float valeur) {
    return valeur * 1000f / 3600f
  }

  private static float convertirEnKilometresHeure(float valeur) {
    return valeur / 1000f * 3600f
  }

  /**
  * @return La vitesse en km/h
  */
  public float getVitesse() {
    return convertirEnKilometresHeure(vitesse);
  }

  /**
  * @param vitesse La vitesse en km/h
  */
  public void setVitesse(float vitesse) {
    this.vitesse = convertirEnMetresSeconde(vitesse);
  }

}
```

Avec les *getters/setters*, il est également possible de contrôler si une propriété est consultable et/ou modifiable. Si une propriété n'est pas consultable, il ne faut pas déclarer de *getter* pour cette propriété. Si une propriété n'est pas modifiable, il ne faut pas déclarer de *setter* pour cette propriété.

> Les*getters/setters* sont très utilisés en Java mais leur écriture peut être fastidieuse. Les IDE comme Eclipse introduisent un système de génération automatique. Dans Eclipse, faites un clic droit dans votre fichier de classe et choisissez *Source > Generate Getters and Setters...*

Revision #1
Created 10 February 2025 12:03:29 by Nicolas
Updated 10 February 2025 12:13:14 by Nicolas