Chapitre 5 : Programmation Orientée Objet

▼ Introduction à la POO

Dans les chapitres précédents, nous nous sommes concentrés sur le **paradigme** de la **programmation procédurale**. Le principe est d'écrire des procédures qui sont une liste d'instructions à exécuter.

Dans ce chapitre, un nouveau **paradigme** est détaillé : la **POO** (`Programmation Orientée Objet`). Le principe est d'utiliser des `objets` comme modèles afin de représenter le monde réel ou des concepts.

Les `objets` sont des **entités** qui stockent des **variables** (appelés **attributs**) et des **fonctions** (appelées **méthodes**). Les **attributs** représentent alors les **données de l'objet** et les **méthodes** représentent les **comportements possibles de l'objet**.

Pourquoi la POO ?

En utilisant uniquement la **programmation procédurale**, il n'est pas aisé de représenter des données complexes car les **types natifs** de données représentent des données simples comme des entiers, des caractères, des booléens, etc.

Les **objets** permettent notamment de **rassembler** un **ensemble de variables simple**s pour représenter des **entités complexes**. Or, ces entités complexes sont **assignables** au sein d'une **variable**.

Au final, le but de la `POO` est d'obtenir un **code** plus **clair**, plus **facile** à maintenir et à déboguer.

Les classes

Les `classes` sont en quelque sorte les *plans de fabrication* des `objets`.

Lors de l'écriture des classes, nous écrivons les différents **attributs** et **méthodes** de l'entité que nous souhaitons représenter dans notre code.

Pour l'illustrer, nous pouvons écrire une **classe Fruit** qui correspond au plan de fabrication de différents objets représentant des fruits. Cette classe contient par exemple les attributs suivants:

`Le nom`

`La couleur`

`L'origine`

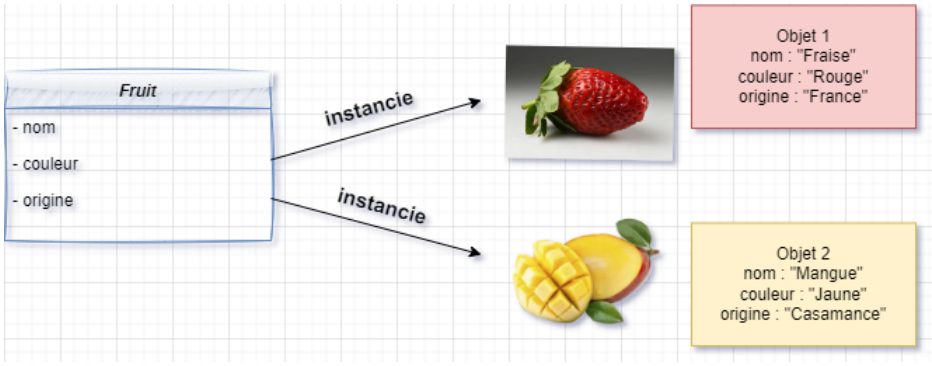
Les objets

Les **objets** sont les **instances** des **classes**, qui sont **assignables** à des **variables**.

Il est ensuite possible d'appeler les **méthodes** définies dans la **classe** d'un **objet** pour **effectuer** des **opérations** sur les **attributs** également définis dans la même classe.

Les `class` suivent la convention de nommage du `PascalCase`. **Tous les mots** commencent par une **majuscule** (y compris le premier). Le reste est en **minuscule**.

Pour l'illustrer, reprenons l'exemple des objets représentant des fruits. Il est alors possible de créer les objets suivants :



On voit bien ici que la même classe **Fruit** sert de **plan de fabrication à plusieurs objets représentant des fruits.**

Les concepts de la POO en Java

Dans ce chapitre, nous verrons ces différents concepts de la `POO` en Java :

- `Les classes et objets` : Comment écrire le code d'une classe et créer des objets à partir de celle-ci.
- `Les attributs (ou propriétés) et constructeurs` : Le fonctionnement des attributs dans une classe.
- `Les méthodes` : Comment ajouter des fonctions utilisables sur les **objets**
- `Les accesseurs` : Comment fonctionne le principe d'encapsulation des attributs
- `L'héritage` : Comment des classes peuvent hériter d'autres classes afin de définir des comportements communs.
- `Les méthodes de comparaison d'objets et d'affichage` : Le fonctionnement des méthodes **equals**, **hashCode** et **toString**
- `Les records` : Des **classes** plus restrictives qui génèrent les **accesseurs**, **equals** et **hashCode**
- `Introduction des interfaces` : Comment définir des contrats pour la définition de classes
- `Les classes internes` : Le fonctionnement des classes imbriquées et leur utilité
- `Les énumérations` : Comment définir des listes finies de valeurs en tant que type

▼ Les classes et objets

Les classes

Nommage des classes

En Java, il est **interdit** de définir **plusieurs** classes différentes dans le **même fichier source**. Il faut donc créer autant de fichier `.java` que de classe dans le code (à l'exception des classes internes que nous verrons plus tard dans le chapitre).

Le nom d'une classe doit respecter la convention de nommage du **PascalCase** (toutes les premières lettres de chaque mot doivent être en majuscule).

Le **nom d'une classe** et le **nom du fichier source** `.java` doivent être **identiques** (la classe Fruit doit se trouver dans le fichier source Fruit.java).

Syntaxe des classes

Exemple de création d'une **classe** `Fruit` dans un fichier `Fruit.java` :

```
public class Fruit {}
```

Par défaut, une classe est **forcément publique**, il n'est donc pas obligatoire de préciser le mot-clé **public** avant le **mot-clé class**. Cependant, c'est une **bonne pratique** de le préciser comme dans l'exemple ci-dessus.

Il est impossible de définir la classe avec le mot-clé **private** dans l'exemple ci-dessus. La raison est qu'une **classe privée de premier niveau** ne pourra **pas être utilisée** dans une autre classe, il n'est donc pas utile de laisser la possibilité de définir une classe de premier niveau en tant que classe privée.

Par conséquence, il n'est possible de définir des classes privées que dans les classes internes :

```
public class Fruit {
    private class InternalClass {}
}
```

Nous verrons l'utilité des classes internes dans une prochaine leçon.

Les objets

Création d'un objet

Pour rappel, un **objet** représente une **instance** d'une **classe**.

Il est possible de créer un objet issu de la classe Fruit ci-dessus de la manière suivante :

```
new Fruit();
```

Le mot-clé new est utilisé en Java pour créer des objets. Il est bien sûr possible d'assigner des objets dans des variables :

```
Fruit fruit = new Fruit();
var fruitParInference = new Fruit();
```

On voit également qu'on peut inférer le type d'un objet avec var de la même manière qu'avec un type natif.

Durée de vie d'un objet

Un **objet** est créé dans la **heap** lorsque le mot-clé `new` est utilisé. Une **référence** vers l'objet est alors créé dans la **stack**.

La suppression de l'objet est **automatique** en Java grâce à la JVM. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction **delete** comme en `C++`.

Le littéral null

A chaque fois qu'une référence vers un **objet** peut être créé, on peut utiliser le littéral `null` à la place :

```
Fruit fruit = null;
```

Il n'appartient pas à une **classe** mais il peut être utilisé à la place d'un **objet** de n'importe quelle type ou comme attribut. **null** ne peut pas être utilisé comme un **objet normal**. Nous verrons ce que ça implique dans la leçon sur les méthodes.

> **Petit conseil** : de manière générale, il est conseillé d'instancier des objets à chaque fois que c'est possible plutôt que de passer la valeur **null**. Appliquer cette règle évitera des potentiels bugs à l'exécution (appeler des méthodes sur une variable alimentée avec **null** résulte en une exception).

▼ Les attributs et constructeurs

Pour rappel, il est possible de définir des attributs dans les classes afin de représenter des concepts plus complexes que les types natifs. Dans cette leçon, nous allons apprendre à représenter des **fruits** avec une classe **Fruit** qui possède les données suivantes :

- Le nom du fruit
- La couleur du fruit
- L'origine du fruit

Les attributs (ou propriétés)

Le code suivante montre comment définir la **classe** Fruit mentionnée ci-dessus avec ses attributs :

```
public class Fruit {
    public String nom;
    public String couleur;
    public String origine;
}
```

Comme pour les **classes** et les **fonctions**, il est nécessaire de définir la **visibilité** des attributs. S'ils sont `public`, ils seront utilisables **en dehors de la classe Fruit**. S'ils sont `private` ils ne seront **pas utilisables en dehors de la classe Fruit**.

On commence à avoir l'habitude : Java est fortement typé. Il est donc nécessaire de donner un type à nos attributs.

Accessibilité aux membres d'une classe

Dans java, il existe **4 niveaux de protection:**

- `private (-)` : Un membre privé d'une classe n'est accessible qu'à l'intérieur de cette classe.
- `protected (#)` **: U**n membre protégé d'une classe est accessible à :
  - L'intérieur de cette classe
  - Aux classes dérivées de cette classe.
  - Aux classes du même package
- `public (+)` **:** accès à partir de toute entité interne ou externe à la classe
- `Autorisation par défaut` : Dans java, en l'absence des trois autorisations précédentes, l'autorisation par défaut est package. Cette autorisation indique que
  uniquement les classes du même package ont
  l'autorisation d'accès

Les constructeurs

Les `constructeurs` sont des **fonctions particulières** qui permettent de **créer** des `objets` grâce au **mot-clé** `new`.

N'importe quelle classe possède un **constructeur par défaut**. En effet, la classe Fruit définit dans l'exemple ci-dessus peut être instanciée de la manière suivante :

```
var fruit = new Fruit();
```

Cependant, ce **constructeur** construit un objet de type Fruit qui **n'a aucune valeur renseignée pour** ses attributs **nom**, **couleur** et **origine**.

Quand on ne définit aucun constructeur pour une classe, le **compilateur crée le constructeur par défaut.**

Le constructeur par défaut n'a aucun paramètre et ne fait aucune initialisation.

Il est possible de créer des constructeurs de la manière suivante :

```
public class Fruit {
    public String nom;
    public String couleur;
    public String origine;

    public Fruit(String nom, String couleur, String origine) {
        this.nom = nom;
        this.couleur = couleur;
        this.origine = origine;
    }
}
```

La **visibilité** des constructeurs doit également être **renseignée**. Un **constructeur** `public` pourra être utilisé dans d'autres classes alors qu'un **constructeur** `private` ne pourra être utilisé qu'au sein de la classe.

> Pour info : il est possible de générer les constructeurs en sélectionnant les attributs voulus grâce au raccourcis `Atl + insert` dans IntelliJ.

Le mot-clé this

Dans le **bloc d'instructions** du constructeur précédent, on voit l'utilisation du **mot-clé** `this`. Il s'agit en fait de **l'objet** qui sera instancié à l'utilisation du **new**.

Celui-ci est important ici car il permet de préciser que l'on utilise spécifiquement les **attributs** de l'**objet** : `this.nom` utilise la valeur présent dans l'attribut nom de l'objet de type Fruit instancié.

Dans l'exemple précédent, `this` permet d'éviter la **confusion** entre les **paramètres d'entrée** de la fonction du constructeur et les **attributs** de la classe.

En résumé `this` permet de référencé l'objet créé par le constructeur

Surcharge des constructeurs

Il est possible de définir plusieurs constructeurs comme dans l'exemple ci-dessous :

```
public class Fruit {
    public String nom;
    public String couleur;
    public String origine;

    public Fruit() {}
```

```
    public Fruit(String nom) {
        this.nom = nom;
    }

    public Fruit(String nom, String couleur, String origine) {
        this.nom = nom;
        this.couleur = couleur;
        this.origine = origine;
    }
}
```

Il n'y a pas de confusion lors de la définition des constructeurs car les signatures sont toutes différentes (les paramètres d'entrées sont différents en nombre à chaque fois).

Utilisation des constructeurs

Avec la définition des différents constructeurs de l'exemple précédent, il est alors possible de créer des objets fruits de différentes manières :

```
Fruit fruit = new Fruit();
Fruit fraise = new Fruit("Fraise");
Fruit mangue = new Fruit("Mangue", "Jaune", "Casamance");
```

Dans notre exemple, les attributs de la classe sont public. Il est donc possible de récupérer la valeur de nos attributs directement dans le code du main :

```
Fruit fraise = new Fruit("Mangue", "Jaune", "Casamance");
System.out.println(fraise.nom); //Affiche "Mangue"
System.out.println(fraise.couleur); //Affiche "Jaune"
System.out.println(fraise.origine); //Affiche "Casamance"
```

Attention aux null

```
Fruit banane = new Fruit("Banane");
```

Seul l'attribut **nom** de l'**objet** est renseigné. Les attributs **couleur** et **origine** sont donc alimentés à **null par défaut**. Donc si on essaie d'afficher tous les attributs de l'objet :

```
Fruit banane = new Fruit("Banane");
System.out.println(banane.nom); //Affiche "Banane"
System.out.println(banane.couleur); //Affiche null
System.out.println(banane.origine); //Affiche null
```

Définir une constante globale

Il est possible de définir des constantes globales grâce aux mot-clé `static` et `final` :

```
public class Constant {
    public final static double pi = 3.14;
}
```

Ici, ce n'est plus de la `POO`. À l'instar des **fonctions** qui sont en fait des **méthodes statiques**, il est possible de définir des **attributs statiques** que l'on peut ensuite utiliser n'importe sans créer de nouvel **objet** :

```
System.out.println(Constant.pi); //Affiche 3.14
```

▼ Les méthodes

Les **méthodes** sont des **fonctions particulières** applicables **directement** aux **objets**.

Pour la suite de la leçon, nous allons partir du code de la classe Fruit que nous avons défini dans la leçon précédente :

```
public class Fruit {
    public String nom;
    public String couleur;
    public String origine;

    public Fruit(String nom, String couleur, String origine) {
        this.nom = nom;
        this.couleur = couleur;
        this.origine = origine;
    }
}
```

Syntaxe d'une méthode

Une méthode se définit presque de la même manière qu'une fonction :

```
public class Fruit {

    ...

    public boolean estCasamancais() {
        if (origine.equals("Casamance")) return true;
        return false;
    }
}
```

La seule différence est que l'on définit la **méthode** sans utiliser le mot-clé `static`.

Pour rappel, `static` permet à l'inverse de définir des **fonctions** qui **ne s'appliquent à aucun objet**.

Dans la **méthode**, on utilise l'**attribut** `origine` de **l'objet** directement sans utiliser le **mot-clé** `this`. La raison est que `this` est **implicite**, nous n'avons pas besoin de le **préciser**. C'est possible car il n'y a pas de confusion avec un **paramètre d'entrée** qui aurait le **même nom**.

Utilisation des méthodes

L'intérêt des **méthodes**, c'est qu'elles permettent de définir des fonctions qui ont accès aux attributs des objets que l'on a instancié. En effet, il est possible d'utiliser la **méthode** sur un **objet** de type **Fruit** de la manière suivante dans le code du main :

```
var fraise = new Fruit("Fraise", "Rouge", "France");
var banane = new Fruit("Banane", "Jaune", "Casamance");
```

```
System.out.println(fraise.estCasamancais()); //Affiche true
System.out.println(banane.estCasamancais()); //Affiche false
```

On voit bien ici qu'on appelle directement la méthode sur les objets créés et que la vérification de l'origine du fruit se fait sur l'attribut de l'objet.

Surcharge des méthodes

À l'instar des constructeurs, il est possible de surcharger les méthodes. En effet, le code suivant est tout à fait valide :

```
public class Fruit {

    ...

    public boolean estCasamancais() {
        if (origine.equals("Casamance")) return true;
        return false;
    }

    public boolean estCasamancais(int entier) {
        if (origine.equals("Casamance")) return true;
        return false;
    }
}
```

La raison est la même : la signature des deux méthodes est différente, même si le nom est exactement le même. Le langage autorise ces surcharges car il est capable de différencier quel méthode appeler en fonction des paramètres d'entrée :

```
var banane = new Fruit("Banane", "Jaune", "Casamance");
banane.estCasamancais();
banane.estCasamancais(26); //Appelle la seconde méthode
```

En résumé les méthodes sont équivalentes à des fonctions mais que l'on applique à des objets. Cela nous permet de travailler directement sur les attributs de l'objet.

▼ Les accesseurs

Dans cette leçon, nous utilisons la classe **Fruit** suivante qui contient des `attributs private` :

```
public class Fruit {
    private String nom;
    private String couleur;
    private String origine;

    public Fruit(String nom, String couleur, String origine) {
        this.nom = nom;
        this.couleur = couleur;
        this.origine = origine;
```

```
    }
}
```

Le principe d'encapsulation

En programmation, **l'encapsulation** désigne le **regroupement de données** avec un ensemble de **routines** qui en permettent la **lecture** et la **manipulation**. En pratique, on va gérer l'encapsulation des données en utilisant les **accesseurs** dans du code Java.

Les `accesseurs` sont des **méthodes** tout à fait **normales**. Leur rôle **se limite à la lecture** ou à la **modification** des **attributs** d'un objet.

Cela implique également que l'on travaillera uniquement avec des **attributs private** par la suite pour s'assurer que l'accès aux attributs des objets se fera toujours de la manière prévue par le développeur.

Les getters

Les **getters** sont des **accesseurs de lecture**. Il est possible de définir le getter de l'attribut nom de la manière suivante :

```
public class Fruit {

    ...

    public String getNom() {
        return nom;
    }
}
```

> Rappel : `this` est optionnel car implicite ici. l'attribut nom est bien retourné par le getter.

Les setters

Les **setters** sont des **accesseurs d'écriture**. Il est possible de définir le setter de l'attribut nom de la manière suivante :

```
public class Fruit {
    ...

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

    public void setNom2(String nom) {
        this.nom = nom;
    }
}
```

La particularité de notre implémentation `setNom` à la différence de `setNom2` est qu'elle retourne l'objet lui-même (`this`) après avoir modifié l'attribut. Cette technique est souvent utilisée pour permettre l'appel en chaîne des méthodes, ce qui peut rendre le code plus fluide et lisible. Par exemple :

```
Fruit fruit = new Fruit();
fruit.setNom("Pomme").setCouleur("Rouge").setOrigine("Maroc");
System.out.println(String.format("Fruit: %s, %s", fruit.getNom(), fruit.getCouleur()));

//Fruit : Pomme, Rouge
```

Attention : Définir des setters dans une classe rend les objets créés par la suite **mutables**. Autant que possible, il est conseillé de ne pas rendre les objets mutables pour éviter des bugs liés aux modifications (un objet peut être référencé à différents endroits du code et donc il peut être modifié à différents endroits également).

Il est donc recommandé, autant que possible, de favoriser **l'immutabilité** des **objets**, c'est-à-dire de les rendre **non modifiables une fois qu'ils ont été créés**. Cela peut contribuer à rendre le code plus robuste et à éviter les erreurs de programmation liées à la modification inattendue des objets.

Utilisation des accesseurs

Il est possible d'utiliser les accesseurs dans le main de la manière suivante :

```
var banane = new Fruit("Banane", "Jaune", "Casamance");
System.out.println(banane.getNom()); //Affiche "Banane"
banane.setNom("Banane modifiée"); //Modifie la valeur de nom dans fraise
System.out.println(banane.getNom()); //Affiche "Banane modifiée"
```

On utilise les **accesseurs** pour respecter l'**encapsulation des données** vu que les attributs sont **private**.

Génération des accesseurs

Il est possible de générer les accesseurs dans les classes pour les paramètres voulu en utilisant le raccourcis `Alt + Insert` ou `Command + N` dans `IntelliJ`.

▼ L'héritage

Dans la **programmation orientée objet**, l'**héritage** offre un moyen très efficace qui permet la **réutilisation du code**.

L'**héritage** permet de créer des **classes** "**enfants**" d'autres classes afin d'hériter **automatiquement** de leurs **comportements**.

Grâce à l'**héritage**, les objets d'une classe **fille** ont accès aux données et aux méthodes de la classe **parente** et peuvent les **étendre**.

De plus quand il peut être exploité, l'**héritage** fait gagner beaucoup de
temps en termes de développement et en terme de maintenance des applications

> Note : l'héritage multiple n'existe pas en Java, ce qui signifie qu'une classe ne peut pas hériter de plusieurs classes parentes, mais d'une seule.

Définir une classe fille

La classe parente

Nous partons de la classe parente Fruit suivante :

```
public class Fruit {
    private String nom;
    private String couleur;
```

```
    private String origine;

    public Fruit(String nom, String couleur, String origine) {
        this.nom = nom;
        this.couleur = couleur;
        this.origine = origine;
    }

    public String descriptionFruit(){
        return "Vient de la classe parente. Nom : " + nom;
    }
}
```

La classe fille

```
public class Banane extends Fruit {
    public Banane() {
        super("Banane", "Jaune", "Guinée");
    }

    @Override
    public String descriptionFruit() {
        return "surcharge de la réponse " + super.descriptionFruit();
    }
}
```

Le **mot-clé** `extends` permet d'hériter de la classe Fruit.

La classe **Banane** hérite de la classe **Fruit** tous ses membres sauf le constructeur.

Dans java une classe hérite toujours d'**une seule classe**. Si une classe n'hérite pas explicitement d'une autre classe, elle hérite implicitement de la classe **Object**. La classe **Fruit** hérite de la
classe
Object.
La classe
Banane hérite directement de la classe **Fruit** et indirectement de la classe **Object**.

Le **constructeur parent** est appelé en utilisant l'invocation `super(...)`.

Il est possible de **surcharger** le **comportement** d'une **méthode** de la **classe parente** en utilisant l'annotation `@Override`. Les **annotations** sont une notation particulière en Java qui peuvent être appliquées sur des **classes** ou des **attributs** ou des **méthodes**. Nous verrons en détails leur fonctionnement dans un prochain chapitre.

N'importe quelle **méthode** de la **classe parente** peut être appelée en utilisant l'invocation `super.nomDeLaMethode(...)`.

Un **constructeur** peut appeler un autre **constructeur** de la **même classe** en utilisant le mot `this()` avec des paramètres du constructeur.

```
public class Fruit {
    private String nom;
    private String couleur;
    private String origine;

    public Fruit() {
```

```
    }

    public Fruit(String nom) {
        this.nom = nom;
    }
    public Fruit(String nom, String couleur, String origine) {
        this.nom = nom;
        this.couleur = couleur;
        this.origine = origine;
    }

    public String descriptionFruit(){
        return "Vient de la classe parente. Nom : " + nom;
    }
}

public class Banane extends Fruit {
    public Banane() {
        this("Banane");
    }
    public Banane(String nom) {
        super(nom);
    }
    @Override
    public String descriptionFruit() {
        return "surcharge de la réponse " + super.descriptionFruit();
    }
}
```

> Limitation : Il n'est pas possible d'utiliser un attribut private ou une méthode private de la classe parente dans la classe fille

Utilisation de la classe fille

La **classe fille Banane** s'utilise alors exactement de la même manière que n'importe quelle classe :

```
//Main
Banane banane = new Banane();
System.out.println(banane.descriptionFruit());

//surcharge de la réponse Vient de la classe parente. Nom : Banane
```

Classes final

Une **classe** de **type** `final` est une classe qui ne peut **pas être dérivée**. Autrement dit, on ne peut **pas hériter** d'une classe final.

Ainsi, le code suivant ne compilera pas :

```
public class StringCustom extends String { }
```

En effet, `String` est une **classe final**. Il n'est donc pas possible de l'étendre.

En résumé:

- Une classe final est une classe qui ne peut pas être dérivée.
- Une méthode final est une méthode qui ne peut pas être redéfinie dans les classes dérivées.
- Une variable final est une variable don la valeur ne peut pas changer
- On utilise final pour deux raisons: une raison de sécurité et une raison d'optimisation

Classes abstract

Les classes **abstract** permettent de définir des classes qui **ne peuvent pas créer d'objets**. Le but est de créer une classe avec des **données** et des **comportements par défaut** qui pourront ensuite être héritées par des **classes filles**.

Une classe **abstraite** est généralement créée pour en faire dériver de nouvelle classe par héritage.

On peut rendre la classe Fruit abstract ainsi :

```
public abstract class Fruit {
    ...
}
```

Méthodes abstract

Dans les classes `abstract`, il est possible de définir des **méthodes abstract**. Ce sont des méthodes qui n'ont **pas de bloc d'instructions** mais qui doivent **obligatoirement** être **surchargées** par les **classes filles**.

On peut définit une méthode **abstract** ainsi :

```
public abstract class Fruit {
    ...
    public abstract String afficher();
}
```

`afficher` devra obligatoirement être surchargée par ses classes filles.

▼ Les méthodes de comparaison d'objets et d'affichage

La classe **Fruit** suivante sera utilisée dans cette leçon :

```
public class Fruit {
    private String nom;
    private String couleur;
    private String origine;
}
```

La classe Object

Par défaut, **toutes les classes** Java **héritent** de la classe **Object** de manière **automatique**.

La classe **Object** possèdent **certaines méthodes** qui peuvent être **surchargées**. C'est le cas des méthodes `equals`, `hashCode` et `toString`. Comme vu dans une leçon précédente, on surcharge ces méthodes en utilisant l'annotation `@Override`.

La méthode equals

Cette méthode est utilisée pour **comparer si deux objets sont égaux**. **Par défaut**, elle compare simplement les **adresses mémoire des objets**, mais elle peut être **surchargée** dans les **classes dérivées** pour effectuer une comparaison plus significative basée sur le contenu des objets.

> **A retenir: Il ne suffit pas d'utiliser l'opérateur == pour comparer deux objets**
> **Pour comparer deux références différentes, il est nécessaire de s'appuyer sur la méthode equals des classes.**

La méthode `equals` de la classe Fruit se définit de la manière suivante :

```
public class Fruit {
    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Fruit fruit = (Fruit) o;

        if (!nom.equals(fruit.nom)) return false;
        if (!couleur.equals(fruit.couleur)) return false;
        return origine.equals(fruit.origine);
    }
}
```

La méthode hashCode

Cette méthode retourne un code de hachage pour l'objet. Les codes de hachage sont utilisés dans diverses structures de données Java, telles que les tables de hachage, pour une recherche efficace. Lorsque la méthode `equals` est surchargée, il est recommandé de surcharger également `hashCode` pour garantir la cohérence avec le comportement d'égale comparaison.

La méthode `hashCode` de la classe Fruit se définit de la manière suivante :

```
public class Fruit {
    ...

    @Override
    public int hashCode() {
        int result = nom.hashCode();
        result = 31 * result + couleur.hashCode();
        result = 31 * result + origine.hashCode();
        return result;
    }
}
```

La méthode toString

Cette méthode retourne une représentation sous forme de chaîne de caractères de l'objet. Par défaut, elle retourne une chaîne de la forme `"NomDeLaClasse@adresseMémoire"`. Surcharger `toString` permet de fournir une représentation plus significative de l'objet, ce qui est utile pour le débogage et la lisibilité du code.

La méthode `toString` de la classe Fruit se définit de la manière suivante :

```
public class Fruit {
    ...

    @Override
    public String toString() {
        return "Fruit{" +
                "nom='" + nom + '\'' +
                ", couleur='" + couleur + '\'' +
                ", origine='" + origine + '\'' +
                '}';
    }
}
```

Générer les méthodes avec Intellij

Il est possible de générer automatiquement les méthodes `equals`, `hashCode` et `toString` en utilisant le raccourci `Alt + Insert` ou `Command + N` et de choisir les attributs sur lesquels les méthodes doivent travailler.

▼ Introduction aux interfaces

Les `interfaces` Java permettent de **définir** des **méthodes génériques** qui devront **être définies** par la suite par **différentes classes**. C'est assez similaire aux **classes abstract**, mais nous allons voir dans cette leçon qu'il y a quelques différences qui justifient l'existence des interfaces.

Syntaxe d'une interface

Exemple de définition d'une **interface** :

```
public interface Motorise {
    void pleinEssence();
    void vidangeHuileMoteur();
}
```

Les méthodes des interfaces **ne peuvent pas être private**. C'est normal car le but est de forcer l'utilisation de ces méthodes pour les classes filles. Il n'est donc pas nécessaire de préciser le **mot-clé public** dans le cas des méthodes d'interfaces.

Utilisation d'une interface

L'interface peut ensuite être implémentée par une classe fille à l'aide du **mot-clé** `implements` :

```
public class Voiture implements Motorise {
    @Override
    public void pleinEssence() {}

    @Override
```

```
    public void vidangeHuileMoteur() {}
}
```

On voit ici que les **méthodes héritées** de l'interface doivent être **surchargées**.

Le mot-clé default

Pour les méthodes des interfaces, il est possible de définir une **implémentation par défaut**.

Pour cela, il est possible d'utiliser le mot-clé `default` :

```
public interface Motorise {
    default void pleinEssence(){
        System.out.println("Implémentation par défaut");
    };
}
```

Ainsi, toutes les classes filles n'ont plus l'obligation de surcharger les méthodes des interfaces (mais elles peuvent le faire quand même !). Il y a juste une implémentation par défaut qui sera appelée en cas d'absence de surcharge.

"Héritage" multiple

En Java, nous avons vu que **l'héritage multiple n'est pas autorisé**. Or, une classe **peut implémenter plusieurs interfaces** sans problèmes :

```
public class Voiture implements Motorise, Pneumatiques {}
```

C'est une manière de **contourner** le **problème de l'absence** d'héritage multiple. Evidemment, toutes les méthodes des différentes interfaces implémentées doivent être surchargées s'il n'y a pas d'implémentation par défaut.

Cette **possibilité** de **multiples implémentations** est la principale différence avec les classes **abstraites** qui ne permettent pas l'héritage multiple.

En résumé les interfaces Java permettent d'apporter un **substitut** à l'héritage multiple

▼ Les énumérations

Créer une énumération

Les `énumérations` sont un t**ype de données** que l'on peut utiliser en Java pour représenter une liste de données, comme le montre l'exemple ci-dessous :

```
public enum Jour {
  LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

L'énumération Jour permet ici de lister l'ensemble des jours de la semaine. Sa visibilité doit être précisée et le mot-clé enum permet de la définir.

Comme c'est un type, cette énumération peut être définie comme n'importe quel type de variable :

```
Jour jour;
```

Et comme attribut de classe puisque ce sont aussi des variables

```
public class Exemple(public Jour jour) {}
```

Utiliser une énumération

Enfin, une énumération peut être utilisée de la manière suivante :

```
Exemple exemple= new Exemple();
exemple.jour = Jour.SAMEDI;
```

Il est en fait possible de lister toutes les valeurs de l'énumération avec la notation Jour. suivit d'une des valeurs de l'énumération.

▼ Polymorphisme

Le **polymorphisme** est un concept essentiel en **programmation orientée objet**. Il permet de modifier le comportement d'une **classe fille** par rapport à sa **classe mère**. Voici comment il fonctionne en **Java** :

Principe du Polymorphisme

Prenons l'exemple d'une classe `Animal`. Cette classe offre une méthode `crier()`. Pour simplifier, supposons que cette méthode affiche le cri de l'animal sur la sortie standard.

Nous avons également deux classes spécialisées : `Chat` et `Chien`, qui héritent de la classe `Animal`.

Chaque classe fille peut **redéfinir** (`override`) la méthode `crier()` pour changer son comportement.

Utilisation du Polymorphisme

```
package entity;

public class Animal {
    String couleur;
    public Animal() {
        System.out.println("création d'un animal");
    }
    public void crier() {
        System.out.println("un cri d'animal");
    }

}

public class Chien extends Animal {
    public Chien(String c){
        couleur = c;
    }
    public void crier() {
        System.out.println("Whouaf whouaf !");
    }
    public void afficherCouleur() {
        System.out.println("La couleur du chien est: " + couleur);
    }
```

```
}

public class Chat extends Animal {
    public Chat(String c){
        couleur = c;
    }
    public void crier() {
        System.out.println("Miaou !");
    }
    public void afficherCouleur() {
        System.out.println("La couleur du chat est: " + couleur);
    }
}
```

```
public class Main {
    public static void main(String[] args){
        Animal animal = new Animal();
        animal.crier();

        Chat chat = new Chat("Marron");
        chat.crier();

        Chien chien = new Chien("Noir");
        chien.crier();

        System.out.println("----------------------------");
        // Polymorphisme : la méthode crier dépend du type réel de l'objet
        animal = chat;//Sur casting
        animal.crier(); // Affiche "Miaou !"
        ((Chat) animal).afficherCouleur();//Sous-casting explicite

        animal = chien;
        animal.crier(); // Affiche "Whouaf whouaf !
    }
}
```

Le résultat affiché par le programme est:

```
création d'un animal
un cri d'animal
création d'un animal
Miaou !
création d'un animal
Whouaf whouaf !
```

Nous constatons qu'avant de créer une Chat, le programme crée un Animal, comme le montre l'exécution du constructeur de cette classe. La même chose se passe lorsque nous créons une Chien.

Un objet de type Chat peut être affecté à une variable de type animal sans aucun problème : animal = chat;

Dans ce cas l'objet Chat est converti automatiquement en Animal. On dit que l'objet Chat est **sur-casté** en Animal.

Dans java, le sur-casting peut se faire implicitement

Signature de Méthode

- Le polymorphisme repose sur la **redéfinition de méthodes**
- Pour que la redéfinition fonctionne, la méthode qui la redéfinit doit avoir une **signature correspondante** à celle de la méthode originale.
- Les méthodes doivent avoir la même portée, le même type de retour, le même nom et les mêmes paramètres
- Exception : les méthodes privées ne supportent pas le polymorphisme, car elles ne sont accessibles que par la classe qui les déclare.

Le polymorphisme permet d'utiliser l'héritage comme un mécanisme d'extension, en adaptant le comportement des objets. C'est un concept puissant pour créer des hiérarchies de classes flexibles et évolutives en Java