

# Cycle de vie d'un objet

Ce chapitre détaille la création d'un objet et son cycle de vie. Nous aborderons notamment les constructeurs et les mécanismes de gestion de la mémoire de la JVM.

## Le constructeur

Il est possible de déclarer des méthodes particulières dans une classe que l'on nomme **constructeurs**. Un constructeur a pour objectif d'initialiser un objet nouvellement créé afin de garantir qu'il est dans un état cohérent avant d'être utilisé.

Un constructeur a la signature suivante :

```
[portée] [nom de la classe]([paramètres]) {  
  
}
```

Un constructeur se distingue d'une méthode car il n'a jamais de type de retour (pas même **void**). De plus un constructeur a obligatoirement **le même nom que la classe**.

```
public class Voiture {  
  
    public Voiture() {  
        // Le constructeur  
    }  
  
}
```

Lorsqu'une voiture est créée par l'application avec l'opérateur **new** comme avec l'instruction suivante :

```
Voiture voiture = new Voiture();
```

Alors, la JVM crée l'espace mémoire nécessaire pour le nouvel objet de type *Voiture*, puis elle appelle le constructeur et enfin elle assigne la référence de l'objet à la variable *voiture*. Donc le constructeur permet de réaliser une initialisation complète de l'objet selon les besoins des développeurs.

# Paramètres de constructeur

Comme les méthodes, les constructeurs peuvent accepter des paramètres et comme les méthodes, les constructeurs supportent la surcharge (*overloading*). Une classe peut ainsi déclarer plusieurs constructeurs à condition que la liste des paramètres diffère par le nombre et/ou le type.

```
public class Voiture {  
  
    private String marque;  
    private float vitesse;  
  
    public Voiture(String marque) {  
        this.marque = marque;  
    }  
  
    public Voiture(String marque, float vitesseInitiale) {  
        this.marque = marque;  
        this.vitesse = vitesseInitiale;  
    }  
  
}
```

Dans l'exemple ci-dessus, la classe *Voiture* déclare deux constructeurs avec des paramètres différents. Il est maintenant nécessaire de passer des paramètres au moment de la création d'une instance de cette classe. Pour cet exemple, on voit que le constructeur permet de forcer la création d'une instance de *Voiture* en fournissant au moins sa marque.

```
Voiture voiture = new Voiture("DeLorean");  
Voiture voiture2 = new Voiture("DeLorean", 88.0f);
```

## Valeur par défaut des attributs

Nous avons vu précédemment que les attributs d'une classe peuvent être initialisés explicitement à la déclaration. Dans le cas contraire, ils sont initialisés avec une valeur par défaut. Java garantit que cette initialisation a lieu avant l'appel au constructeur.

```
public class Vehicule {  
  
    private String marque;  
    private int nbRoues = 4;  
    private float vitesse;
```

```

public Vehicule(String marque) {
    this.marque = marque;
    // la vitesse vaudra 0 et nbRoues vaudra 4
}

public Vehicule(String marque, int nbRoues) {
    this.marque = marque;
    // On ne peut créer que des véhicules avec au plus 4 roues
    if (nbRoues < this.nbRoues) {
        this.nbRoues = nbRoues;
    }
    // la vitesse vaudra 0
}
}

```

Les attributs déclarés **final** sont traités un peu différemment. Ces attributs doivent être obligatoirement et explicitement initialisés avant la fin de la création de l'objet. Donc, il est possible de les initialiser dans le constructeur. Par contre, le compilateur générera une erreur si :

- un constructeur tente d'accéder à un attribut **final** qui n'a pas encore été initialisé
- un constructeur se termine sans avoir initialisé explicitement tous les attributs **final**
- un constructeur tente d'affecter une valeur à un attribut **final** qui a déjà été initialisé au moment de sa déclaration.

```

public class Vehicule {

    private static final int DEFAULT_NBRROUES = 4;

    private final String marque;
    private final int nbRoues;
    private float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
        this.nbRoues = DEFAULT_NBRROUES;
        // la vitesse vaudra 0
    }
}

```

```
public Vehicule(String marque, int nbRoues) {  
    this.marque = marque;  
    // On ne peut créer que des véhicules avec au plus 4 roues  
    this.nbRoues = nbRoues < DEFAULT_NBRouES ? nbRoues : DEFAULT_NBRouES;  
    // la vitesse vaudra 0  
}  
  
}
```

## Constructeur par défaut

Le compilateur Java garantit que toutes les classes ont au moins un constructeur. Si vous créez la classe suivante :

```
public class Voiture {  
  
}
```

Alors, le compilateur ajoutera le code nécessaire qui correspondrait à :

```
public class Voiture {  
  
    public Voiture() {  
    }  
  
}
```

Ce constructeur est appelé le **constructeur par défaut**. Par contre si votre classe contient au moins un constructeur, quelle que soit sa signature, alors le compilateur n'ajoutera pas le **constructeur par défaut**.

```
public class Voiture {  
  
    private final String marque;  
  
    /* Le compilateur ne générera pas de constructeur par défaut.  
     * Pour créer une voiture, je suis obligé de fournir sa marque en paramètre  
     * de création.  
     */  
    public Voiture(String marque) {  
        this.marque = marque;  
    }  
}
```

```
}
```

```
}
```

Si votre classe ne contient qu'un seul constructeur sans paramètre dont le corps est vide, alors vous pouvez supprimer cette déclaration car le compilateur le générera automatiquement.

## Constructeur privé

Il est tout à fait possible d'interdire l'instantiation d'une classe en Java. Pour cela, il suffit de déclarer tous ses constructeurs avec une portée **private**.

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

Comme montré dans l'exemple ci-dessus, un cas d'usage courant est la création d'une classe util. Une classe util ne contient que des méthodes de classe. Il n'y a donc aucun intérêt à instancier une telle classe. Donc, on déclare un constructeur privé pour éviter une utilisation incorrecte.

On peut aussi considérer que la classe *Calculatrice* est simplement un espace de nom contenant un ensemble de fonctions. Même si les fonctions n'existent pas en Java, les classes outils sont un moyen de les simuler.

## Appel d'un constructeur dans un constructeur

Certaines classes peuvent offrir différents constructeurs à ses utilisateurs. Souvent ces constructeurs vont partiellement exécuter le même code. Pour simplifier la lecture et éviter la duplication de code, un constructeur peut appeler un autre constructeur en utilisant le mot-clé **this**

comme nom du constructeur. Cependant, un constructeur ne peut appeler qu'un **seul** constructeur et, s'il le fait, cela doit être sa première instruction.

```
public class Vehicule {

    private static final int DEFAULT_NBRoues = 4;

    private final String marque;
    private final int nbRoues;
    private float vitesse;

    public Vehicule(String marque) {
        this(marque, DEFAULT_NBRoues, 0f);
    }

    public Vehicule(String marque, int nbRoues) {
        this(marque, nbRoues, 0f);
    }

    public Vehicule(String marque, int nbRoues, float vitesseInitiale) {
        this.marque = marque;
        this.nbRoues = nbRoues < DEFAULT_NBRoues ? nbRoues : DEFAULT_NBRoues;
        this.vitesse = vitesseInitiale;
    }

}
```

La classe *Vehicule* ci-dessus offre plusieurs possibilités d'initialisation, mais les développeurs de cette classe ont évité la duplication en plaçant le code d'initialisation dans le troisième constructeur.

## Appel d'une méthode dans un constructeur

Il est tout à fait possible d'appeler une méthode de l'objet dans un constructeur. Cela est même très utile pour éviter la duplication de code et favoriser la réutilisation. Attention cependant au statut particulier des constructeurs. Tant qu'un constructeur n'a pas achevé son exécution, l'objet n'est pas totalement initialisé. Il peut donc y avoir des cas où l'appel à une méthode peut avoir des comportements inattendus.

Prenons l'exemple suivant :

```

public class Vehicule {
    private static final int DEFAULT_NBROUES = 4;

    private final String marque;
    private final int nbRoues;
    private float vitesse;

    public Vehicule(String marque, int nbRoues, float vitesseInitiale) {
        faireQuelqueChoseDInattendue();
        this.marque = marque;
        this.nbRoues = nbRoues < DEFAULT_NBROUES ? nbRoues : DEFAULT_NBROUES;
        this.vitesse = vitesseInitiale;
    }

    private void faireQuelqueChoseDInattendue() {
        System.out.println(this.nbRoues); // 0
    }
}

```

Le constructeur appelle la méthode *faireQuelqueChoseDInattendue* qui affiche la valeur de l'attribut *nbRoues*. Cet attribut est déclaré **final** donc il n'est pas modifiable durant la vie de l'objet et la tâche du constructeur va être, entre autres, de lui assigner une valeur. Mais comme la méthode *faireQuelqueChoseDInattendue* est appelée avant l'initialisation, elle affichera 0. Il s'agit d'un comportement aberrant du point de vue de la définition de **final** mais qui compile et s'exécute sans erreur.

Plus généralement, si vous souhaitez appeler des méthodes de l'objet dans un constructeur, il faut prendre soin de s'assurer que l'état de l'objet nécessaire à l'exécution de ces méthodes est correctement initialisé avant par le constructeur.

## Injection de dépendances par le constructeur

L'état interne d'un objet (ses attributs) inclut souvent des références vers d'autres objets. Parfois, ces objets peuvent eux-même avoir une représentation interne complexe qui nécessite des références vers d'autres objets... Par exemple, une classe *Voiture* peut nécessiter une instance d'une classe *Moteur* :

```
package com.cgi.udev;

public class Moteur {

    private int nbCylindres;
    private int nbSoupapesParCylindre;
    private float vitesseMax;

    public Moteur(int nbCylindres, int nbSoupapesParCylindre, float vitesseMax) {
        this.nbCylindres = nbCylindres;
        this.nbSoupapesParCylindre = nbSoupapesParCylindre;
        this.vitesseMax = vitesseMax;
    }

    // ...
}
```

À partir de la classe *Moteur* ci-dessus, nous pouvons fournir l'implémentation suivante de la classe *Voiture* :

```
package com.cgi.udev;

public class Voiture {

    private String marque;
    private Moteur moteur;

    public Voiture(String marque, int nbCylindres, int nbSoupapesParCylindre, float vitesseMax)
    {
        this.marque = marque;
        this.moteur = new Moteur(nbCylindres, nbSoupapesParCylindre, vitesseMax);
    }

    // ...
}
```

Et créer une instance de la classe *Voiture* :

```
Voiture clio = new Voiture("Clio Williams", 4, 4, 216);
```



Cependant, si nous considérons le type de relation qui unit la classe *Voiture* à la classe *Moteur*, nous constatons que non seulement la classe *Voiture* est dépendante de la classe *Moteur* mais qu'en plus la classe *Voiture* crée l'instance de la classe *Moteur* dont elle a besoin. Donc la classe *Voiture* a un couplage très fort avec la classe *Moteur*. Par exemple, si le constructeur de la classe *Moteur* évolue alors le constructeur de la classe *Voiture* doit également évoluer.

En programmation objet, créer un objet n'hésite souvent de disposer des informations nécessaires pour invoquer le constructeur de sa classe. La plupart du temps, les classes qui sont dépendantes d'autres classes n'ont pas vocation à les créer car il n'y a pas vraiment de raison à ce qu'elles connaissent les informations nécessaires à leur création. Dans le cas de notre classe *Voiture* nous pouvons proposer simplement l'implémentation :

```
package com.cgi.udev;

public class Voiture {

    private String marque;
    private Moteur moteur;

    public Voiture(String marque, Moteur moteur) {
        this.marque = marque;
        this.moteur = moteur;
    }

    // ...
}
```

La création d'une instance de *Voiture* se fait maintenant en deux étapes :

```
Moteur moteur = new Moteur(4, 4, 216);
Voiture clio = new Voiture("Clio Williams", moteur);
```

On dit qu'une instance de la classe *Moteur* est **injectée** par le constructeur dans une instance de *Voiture*. En programmation objet, cela signifie que nous avons découplé l'utilisation de l'instance de la classe *Moteur* de sa création.

L'injection de dépendances est une technique de programmation qui permet à une classe de disposer des instances d'objet dont elle a besoin sans avoir à les créer directement.

L'injection de dépendance est la technique qui est à la base de [l'inversion de dépendances](#) (appelée aussi parfois inversion de contrôle) qui est un des principes [SOLID](#) en

programmation objet. Beaucoup de frameworks Java (comme le [Spring framework](#)) sont basés sur ce principe.

# Le bloc d'initialisation

Il est possible d'écrire un traitement d'initialisation s'effectuant avant l'appel au constructeur. Il suffit de déclarer un bloc anonyme dans la classe.

```
public class Voiture {  
  
    private final int nbRoues;  
  
    {  
        Configuration cfg = getConfiguration();  
        nbRoues = cfg.nbRouesParVoiture;  
    }  
  
    private Configuration getConfiguration() {  
        // le code ici pour consulter la configuration  
    }  
  
}
```

Dans l'exemple précédent, on suppose qu'il existe une classe *Configuration* et qu'il est possible de consulter la configuration de l'application pour connaître le nombre de roues par voiture. Le bloc d'initialisation accède à la configuration et affecte la bonne valeur à l'attribut **final** *nbRoues*.

Le bloc d'initialisation est très rarement employé en Java. On peut systématiquement obtenir le même comportement en déclarant un constructeur.

# Le bloc d'initialisation de classe

Il est possible d'écrire un traitement d'initialisation d'une classe. Ce traitement ne sera effectué qu'une seule fois : au moment du chargement de la définition de la classe dans la mémoire de la JVM. Une initialisation de classe se fait à l'aide d'un bloc d'instructions **static**.

```
public class Voiture {
```

```

private static final int NB_ROUES;

static {
    Configuration cfg = getConfiguration();
    NB_ROUES = cfg.nbRouesParVoiture;
}

private static Configuration getConfiguration() {
    // le code ici pour consulter la configuration
}

}

```

Dans l'exemple précédent, on suppose qu'il existe une classe *Configuration* et qu'il est possible de consulter la configuration de l'application pour connaître le nombre de roues par voiture. Le bloc **static** donne la possibilité d'initialiser une constante à partir d'un traitement plus complexe.

On peut obtenir un résultat similaire en initialisant la constante *NB\_ROUES* à partir d'un appel à une méthode de classe :

```

public class Voiture {

    private static final int NB_ROUES = getConfiguration().nbRouesParVoiture;

    private static Configuration getConfiguration() {
        // le code ici pour consulter la configuration
    }

}

```

## Mémoire heap et stack

Comme pour la plupart des langages de programmation, Java utilise deux espaces mémoires : la **stack** (ou *call stack*, la pile d'appel) et le **heap** (le tas).

La **stack** correspond à l'espace alloué pour gérer la mémoire nécessaire à l'exécution des méthodes (d'un thread). C'est dans cet espace que les variables déclarées dans la méthode sont stockées. Cet espace a la structure d'une pile car lorsqu'une méthode appelle une autre méthode, l'espace mémoire nécessaire à cet appel s'empile au dessus de l'espace mémoire précédent. Lorsqu'une méthode se termine l'espace mémoire qui lui est alloué dans la stack est libéré. Cela signifie que lorsqu'une méthode se termine, il n'est plus possible d'accéder aux variables qu'elle a

déclarées.

Le **heap** permet de stocker de l'information en allouant dynamiquement de l'espace mémoire lorsque cela est nécessaire et de le libérer lorsqu'il n'est plus utile. Le heap a une structuration plus complexe qui tient compte de la durée de vie présumée des éléments qui le composent. Dans le heap se trouve, la description des classes chargées par la JVM mais surtout tous les objets créés. En effet, le mot-clé **new** a pour fonction de créer un nouvel objet en stockant ses informations dans le heap.

Tous les objets Java étant créés dans le heap, leur durée de vie peut être plus longue que le temps d'exécution d'une méthode. Il n'est pas possible pour un développeur de demander explicitement la destruction d'un objet. Par contre il existe un procédé appelé le *ramasse-miettes* (**garbage collector**) qui se charge de libérer la mémoire lorsqu'il détecte qu'elle n'est plus utilisée.

La machine virtuelle Java gère elle-même l'espace mémoire allouable à la stack et au heap (alors qu'il s'agit normalement d'une activité prise en charge par le système d'exploitation lui-même). Du coup, il est possible de paramétrer au lancement de la JVM la taille mémoire allouable si on souhaite introduire des quotas par processus avec les paramètres :

```
-Xms<taille>  
Taille initiale du heap  
-Xmx<taille>  
Taille maximale du heap  
-Xss<taille>  
Taille de la stack (par thread)
```

Par exemple :

```
$ java -Xms512M -Xmx512M MonApplication
```

Dans l'exemple précédent, l'application est lancée avec un heap d'une taille fixe de 512 Mo.

## Le ramasse-miettes

Le ramasse-miettes (*garbage collector*) est un processus léger (*thread*) qui est créé par la JVM et qui s'exécute régulièrement pour contrôler l'état de la mémoire. S'il détecte que des portions de mémoire allouées ne sont plus utilisées, il les libère afin que l'application ne manque pas de ressource mémoire.

La présence du ramasse-miettes évite aux développeurs de devoir demander explicitement la libération de la mémoire. D'ailleurs il n'est pas possible en Java de demander explicitement la

libération de la mémoire. Cependant, il est important que les développeurs comprennent le fonctionnement du ramasse-miettes.

Le ramasse-miettes vérifie périodiquement si les objets sont référencés. Un objet est référencé si :

- la méthode en cours d'exécution ou une méthode de la pile d'appel possède une variable référençant cet objet
- il existe un objet référencé qui possède un attribut référençant cet objet

Le ramasse-miettes gère également le problème de la référence circulaire. Un objet qui contiendrait un attribut qui le référence directement ou indirectement lui-même n'est pas réellement considéré comme une référence par le ramasse-miettes.

Donc, si un développeur souhaite qu'un objet soit détruit et son espace mémoire récupéré, il doit s'assurer que plus aucune référence n'existe vers cet objet. Par exemple, il peut affecter la valeur **null** aux variables et aux attributs qui référencent cet objet.

Il est également possible de forcer l'appel au ramasse-miettes grâce à la méthode [java.lang.System.gc\(\)](#). Cependant, cette méthode ne donne aucune garantie quant au résultat. Vous ne pouvez pas vous baser sur son appel pour garantir la suppression d'un objet non référencé. Le ramasse-miettes utilise un algorithme complexe qui rend son comportement difficilement prédictible.

Le ramasse-miettes est parfois la préoccupation des ingénieurs système. En effet, les serveurs implémentés en Java dépendent du ramasse-miettes pour gérer la désallocation de la mémoire. Même si l'exécution du ramasse-miettes est rapide, elle peut avoir des effets sur des serveurs très sollicités en entraînant des micro-interruptions du service. Java propose non pas un mais des algorithmes de ramasses-miettes configurables. Il est donc possible de choisir au lancement de la JVM le type de ramasse-miettes à utiliser.

Le ramasse-miettes fait l'objet de modification et d'évolution à toutes les versions de Java. Pour Java 9, vous pouvez vous reporter au [guide de tuning](#) du ramasse-miettes.

Java propose un mécanisme de ramasse miettes mais ce dernier ne peut libérer l'espace mémoire que des objets non référencés. Si vous développez une application qui crée beaucoup d'objets sans donner la possibilité au ramasse-miettes de les collecter, votre application peut se retrouver à cours d'espace mémoire. Lors de la création d'un nouvel objet, vous obtiendrez alors une erreur du type [java.lang.OutOfMemoryError](#).

La mémoire n'est pas la seule ressource système avec laquelle les développeurs doivent composer. Si Java propose un mécanisme pour la gestion de la mémoire, il ne propose pas de mécanisme automatique pour réclamer les autres types de ressources, notamment les descripteurs de fichier et de socket.

# La méthode finalize

Si un objet souhaite effectuer un traitement avant sa destruction, il peut implémenter la méthode *finalize*. Cette méthode a la signature suivante :

```
protected void finalize() {  
}
```

Dans la pratique cette méthode n'est utilisée que pour des cas d'implémentation très avancés. En effet, la JVM ne donne strictement **aucune** garantie sur le moment où la méthode **finalize** est appelée. Elle peut même ne jamais être appelée si l'application se termine avant que le ramasse-miettes ne réclame l'espace mémoire de l'objet. Elle n'a donc pas le même statut ni la même importance qu'un destructeur dans le langage C++.

```
public class ObjetCurieux {  
  
    protected void finalize() {  
        System.out.print("je vais disparaître !");  
    }  
  
    public static void main(String[] args) {  
        ObjetCurieux objetCurieux = new ObjetCurieux();  
        objetCurieux = null;  
  
        System.gc();  
  
        for(int i = 0; i < 1000 ; ++i) {  
            System.out.print('.');  
        }  
    }  
}
```

Dans l'exemple ci-dessus, Un objet qui n'implémente que la méthode *finalize* est créé puis la variable qui le référence est mise à **null**. Ensuite, le programme appelle explicitement le ramasse-miettes avec la méthode [java.lang.System.gc\(\)](#). Enfin, une boucle se contente d'afficher mille points sur la sortie standard. Cette boucle **for** est utile car généralement le programme s'arrête trop vite et le ramasse-miettes n'a pas le temps d'appeler *finalize*. Si vous exécutez ce programme plusieurs fois, vous constaterez que le message « je vais disparaître ! » ne s'affiche pas au même moment. Cela traduit bien le fait que le comportement du ramasse-miettes varie d'une exécution à l'autre.

---

Revision #1

Created 10 February 2025 12:07:31 by Nicolas

Updated 10 February 2025 12:13:14 by Nicolas