La gestion de mêmoire :

Garbage collector Le ramasse-miettes :

Le ramasse-miettes est une fonctionnalité de la JVM qui a pour rôle de gérer la mémoire notamment en libérant celle des objets qui ne sont plus utilisés.

La règle principale pour déterminer qu'un objet n'est plus utilisé est de vérifier qu'il n'existe plus aucun autre objet qui lui fait référence. Ainsi un objet est considéré comme libérable par le ramasse-miettes lorsqu'il n'existe plus aucune référence dans la JVM pointant vers cet objet.

Lorsque le ramasse-miettes va libérer la mémoire d'un objet, il a l'obligation d'exécuter un éventuel finalizer définit dans la classe de l'objet. Attention, l'exécution complète de ce finalizer n'est pas garantie : si une exception survient durant son exécution, les traitements sont interrompus et la mémoire de l'objet est libérée sans que le finalizer soit entièrement exécuté.

La mise en oeuvre d'un ramasse-miettes possède plusieurs avantages :

- elle améliore la productivité du développeur qui est déchargé de la libération explicite de la mémoire
- elle participe activement à la bonne intégrité de la machine virtuelle : une instruction ne peut jamais utiliser un objet qui n'existe plus en mémoire

Mais elle possède aussi plusieurs inconvénients :

- le ramasse-miettes consomme des ressources en terme de CPU et de mémoire
- il peut être à l'origine de la dégradation plus ou moins importante des performances de la machine virtuelle
- le mode de fonctionnement du ramasse miettes n'interdit pas les fuites de mémoires si le développeur ne prend pas certaines

précautions. Généralement issues d'erreurs de programmation subtiles, ces fuites sont assez difficiles à corriger.

Le rôle du ramasse-miettes

Le garbage collector a plusieurs rôles :

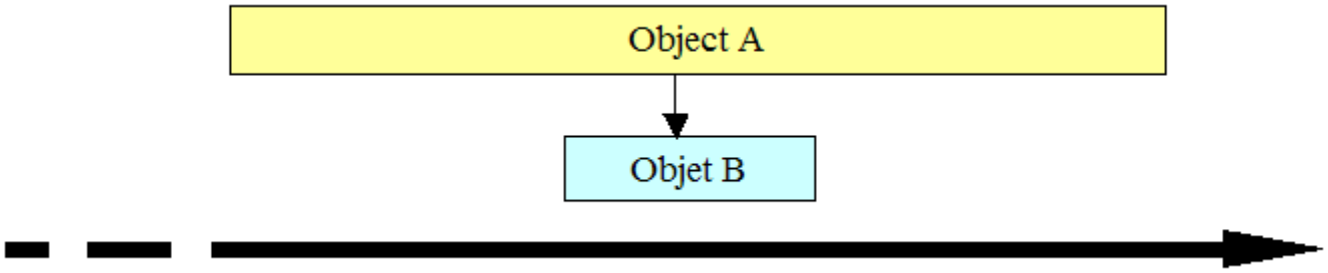
- s'assurer que tout objet dont il existe encore une référence n'est pas supprimé
- récupérer la mémoire des objets inutilisés (dont il n'existe plus aucune référence)
- éventuellement défragmenter (compacter) la mémoire de la JVM selon l'algorithme utilisé
- intervenir dans l'allocation de la mémoire pour les nouveaux objets à cause du point précédent

Le ramasse-miettes s'exécute dans un ou plusieurs threads de la JVM.

Les objets en cours d'utilisation (dont il existe encore une référence) sont considérés comme "vivants". Les objets inutilisés (ceux dont plus aucun autre objet ne possède une référence) sont considérés comme pouvant être libérés. Les traitements pour identifier ces objets et libérer la mémoire qu'ils occupent se nomment garbage collection. Ces traitements sont effectués par le garbage collector ou ramasse-miettes en français.

Le rôle primaire d'un ramasse-miettes est de trouver les objets de la mémoire qui ne sont plus utilisés par l'application et de libérer l'espace qu'ils occupent. Le principe général d'exécution du ramasse-miettes est de parcourir l'espace mémoire, marquer les objets dont il existe au moins une référence de la part d'un autre objet. Tous les objets qui ne sont pas marqués sont éligibles pour récupérer leur mémoire. Leur espace mémoire sera libéré par le ramasse-miettes ce qui augmentera l'espace mémoire libre de la JVM.

Il est important de comprendre comment le ramasse-miettes détermine si un objet est encore utilisé ou pas : un objet est considéré comme inutilisé s'il n'existe plus aucune référence sur cet objet dans la mémoire.



Dans l'exemple ci-dessus, un objet A est créé. Au cours de sa vie, un objet B est instancié et l'objet A possède une référence sur l'objet B. Tant que cette référence existe, l'objet B ne sera pas supprimé par le ramasse-miettes même si l'objet B n'est plus considéré comme utile d'un point de vue fonctionnel. Ce cas de figure est fréquent notamment avec les objets des interfaces graphiques, les listeners ou avec les collections.

L'algorithme le plus basique pour un ramasse-miettes, parcourt tous les objets, marque ceux dont il existe au moins une référence. A la fin de l'opération, tous les objets non marqués peuvent être supprimés de la mémoire. Le gros inconvénient de cet algorithme est que son temps d'exécution est proportionnel au nombre d'objets contenus dans la mémoire. De plus, les traitements de l'application sont arrêtés durant l'exécution du ramasse-miettes.

Plusieurs autres algorithmes ont été développés pour améliorer les performances et diminuer les temps de pauses liés à l'exécution du ramasse-miettes.

Les différents concepts des algorithmes du ramasse-miettes

Plusieurs considérations doivent être prises en compte dans le choix de l'algorithme à utiliser lors d'une collection par le ramasse-miettes :

serial ou parallel : avec une collection de type serial, une seule tâche peut être exécutée à un instant donné même si plusieurs processeurs sont disponibles sur la machine.

Avec une collection de type parallel, les traitements du garbage collector sont exécutés en concomitance par plusieurs processeurs. Le temps global de traitement est ainsi plus court mais l'opération est plus complexe et augmente généralement la fragmentation de la mémoire

stop the world ou concurrent : avec une collection de type stop the world, l'exécution de l'application est totalement suspendue durant les traitements d'une collection. Stop the world utilise un algorithme assez simple puisque durant ses traitements, les objets ne sont pas modifiés. Son inconvénient majeur est la mise en pause de l'application durant l'exécution de la collection.

Avec une collection de type concurrent, une ou plusieurs collections peuvent être exécutées simultanément avec l'application. Cependant, une collection de type concurrent ne peut pas réaliser tous ses traitements de façon concurrente et doit parfois en réaliser certains sous la forme stop the world.

De plus, l'algorithme d'une collection de type concurrent est beaucoup plus complexe puisque les objets peuvent être modifiés par l'application durant la collection : il nécessite généralement plus de ressources CPU et de mémoire pour s'exécuter.

compacting, non compacting ou copying : une fois la libération de la mémoire effectuée par le garbage collector, il peut être intéressant pour

ce dernier d'effectuer un compactage de la mémoire en regroupant les objets alloués d'une part et la mémoire libre de l'autre.

Ce compactage nécessite un certain temps de traitement mais il accélère ensuite l'allocation de mémoire car il n'est plus utile de déterminer quel espace libre utiliser : s'il y a eu compactage, cette espace correspond obligatoirement à la première zone de mémoire libre rendant l'allocation très rapide.

Si la mémoire n'est pas compactée, le temps nécessaire à la collection est réduit mais il est nécessaire de parcourir la mémoire pour rechercher le premier espace de mémoire qui permettra d'allouer la mémoire requise, ce qui augmente les temps d'allocation de mémoire aux nouveaux objets et la fragmentation de cette dernière.

Il existe aussi une troisième forme qui consiste à copier les objets survivants à différentes collections dans des zones de mémoires différentes (copying). Ainsi la zone de création des objets se vide au fur et à mesure, ce qui rend l'allocation rapide. Le copying nécessite plus de mémoire.

LA CLASSE OBJECT

Voyons tout d'abord les méthodes de cette classe.

Exemple 3. Les méthodes de la classe Object

```
public  class Object {
    public Object() {...}  // contructeur

    public String toString() {...}

    protected native Object clone() throws
CloneNotSupportedException {...}

    public equals(java.lang.Object) {...}
    public  native  int hashCode() {...}

    protected  void finalize()  throws Throwable {...}

    public  final  native Class getClass() {...}

    // méthodes utilisées dans la gestion des threads
    public  final  native  void notify() {...}
    public  final  native  void notifyAll() {...}

    public  final  void wait(long)  throws InterruptedException {...}
    public  final  void wait(long,  int)  throws InterruptedException {...}
}
```

Nous reviendrons en détails plus tard sur la signification des clauses throws ... Exception. De même, les méthodes notify(), notifyAll(), wait(long) et wait(long, int) seront également revues en détails dans le chapitre sur les unités d'exécution (threads).

Une des possibilités offertes par l'héritage, est que tout objet instance d'une classe qui hérite d'une autre classe, peut utiliser les méthodes de cette autre classe. Donc, tout objet Java peut utiliser, sous certaines conditions, l'ensemble des méthodes de la classe Object. Certaines de ces méthodes sont très spécialisées, mais d'autres sont utilisées très fréquemment. D'autres enfin, sont utilisées de manière spéciale par la machine Java, et ont donc un statut particulier.

La méthode toString()

La première de ces méthodes est la méthode toString(). Cette méthode est utilisée par la machine Java toutes les fois où elle a besoin de représenter un objet sous forme d'une chaîne de caractères. Par exemple, il est légal d'écrire la ligne suivante :

```
Marin marin =  new Marin ("Surcouf",  "Robert",  25000) ;
System.out.println("Marin : " + marin) ;
```

La méthode println de l'objet System.out prend en paramètre un objet de type String. La machine Java a donc besoin de convertir ("Marin : " + marin) en objet de type String. Cela passe par la conversion de l'objet marin en objet String. Cette conversion s'effectue par appel à la méthode toString() de la classe Object.

Il est donc presque équivalent d'écrire le code précédent et celui-ci :

```
System.out.println("Marin : " + marin.toString()) ;
```

En apparence, les deux codes sont équivalents. En fait, en toute rigueur, le second échouera avec une ignoble NPE si l'objet marin est nul, alors que le premier affichera null.

Dans la pratique, on peut se demander ce que ce code va nous afficher. La réponse est immédiate si l'on tente de l'exécuter : une chaîne de caractères un peu cabalistique, qui va ressembler à Marin@b82e3f203. Les chaînes de caractères retournées par la méthode toString() de la

classe Object ont toutes cette forme : le nom de la classe, suvie du caractère @, et une adresse en hexadécimal, qui est l'adresse mémoire où l'objet considéré est enregistré. Cela garantit l'unicité de ce code en fonction de l'objet, ce qui a son importance. En tout cas, il apparaît clairement que ce résultat c'est pas très sympathique, et en tout cas peu explicite pour un utilisateur humain normalement constitué.

Heureusement pour nous, il est possible de changer ce comportement, et d'afficher une chaîne de caractères plus amicale. On utilise pour cela un mécanisme qui s'appelle la "surcharge". Sans entrer dans les détails de ce mécanisme, disons pour le moment qu'il est possible de réécrire cette méthode toString() dans la classe Marin. Il suffit pour cela d'ajouter les lignes de code suivantes à la classe que nous avons déjà écrite.

Exemple 4. Surcharge de toString() dans la classe Marin

```
public String toString() {
   String resultat =  super.toString() ;
   resultat +=  "\nNom : " + nom ;
   resultat +=  "\nPrénom : " + prenom ;
   resultat +=  "\nSalaire : " + salaire ;
    return resultat ;
}
```

Cette méthode commence par un appel à la méthode super.toString(). Cette syntaxe signifie qu'elle appelle une méthode toString() qui doit être définie parmi les super-classes de Marin. Sur notre exemple simple, cette classe n'en étend aucune autre par déclaration, elle étend donc Object. Il se trouve par chance que la classe Object possède une méthode toString(), c'est donc celle-là qui va être appelée tout d'abord. Si cela n'avait pas été le cas, on aurait eu une erreur de compilation.

Les lignes suivantes ajoutent des éléments supplémentaires. Le résultat de l'appel à cette méthode toString() devra ressembler à ceci :

```
Marin@b82e3f203
Nom : Surcouf
Prénom : Robert
Salaire :  25000
```

La méthode clone()

La méthode clone() est une méthode déclarée native. Une méthode native est une méthode qui n'est pas écrite en Java, mais dans un autre langage, qui peut être le C, le C++, ou tout autre langage. Java utilise un mécanisme spécial pour éventuellement passer des paramètres aux méthodes natives, les invoquer, et récupérer ce qu'elles retournent. Une méthode native n'est en général pas portable d'une machine à l'autre, on perd donc un des intérêts majeurs de Java en écrivant des méthodes natives. Ici, cette méthode fait partie de l'API standard, qui de toute façon existe pour toutes les machines / OS existantes.

Le rôle de la méthode clone() est de dupliquer un objet rapidement, en dupliquant la zone mémoire dans laquelle il se trouve à l'aide d'un processus rapide. Pour cloner un objet, il suffit donc d'appeler cette méthode, qui nous renverra une copie conforme de cet objet.

Attention toutefois le clonage des objets est interdit par défaut. Afin de l'utiliser, il faut surcharger la méthode clone() de la classe Object, qui est protected, par une méthode public de la classe de l'objet que l'on veut cloner. En plus, il faut que la classe dont on veut cloner les instances, implémente l'interface Cloneable. Cette interface ne comporte pas de méthode, elle est juste là pour autoriser le clonage. Tenter de cloner un objet qui n'implémente pas cette interface génèrera une exception de type CloneNotSupportedException (moins ignoble que l'ignoble NPE, mais tout de même...).

La surcharge de cette méthode n'a pour objet que d'exposer publiquement la méthode clone() de la classe Object. Rendre une classe Marin clonable, peut donc se faire de la façon suivante.

Exemple 5. Surcharge de clone() dans la classe Marin

```
public  class Marin
 implements Cloneable {  // déclaration indispensable

    // ici on propage l'exception, on aurait pu aussi
    // l'attraper localement
    public Object clone()  throws CloneNotSupportedException {
       return  super.clone() ;
   }
}
```

Notons que l'on peut surcharger la méthode clone() par une méthode qui ne jette aucune exception, dans la mesure où la clause throws ne fait pas partie de la signature d'une méthode. Dans ce cas, l'exception que peut jeter l'appel à super.clone() doit être attrapée localement.

La méthode equals()

Cette méthode permet, comme son nom peut le laisser supposer, de comparer deux objets, et notamment de savoir s'ils sont égaux. Un peu plus haut nous avons créé deux objets de la classe Marin, qui possédaient même nom, même prénom et même salaire, et nous les avons comparés avec ==. Nous avions alors vu que == comparait les adresses mémoire des objets, et dans ce cas renvoyait false. Ce comportement est logique, mais il serait utile d'avoir à disposition un moyen de comparer des objets qui puisse nous dire que si leurs champs sont égaux, alors ces objets sont égaux. En d'autres termes, remplacer une égalité technique en égalité sémantique. C'est l'objet de la méthode equals().

Écrivons une méthode equals() pour notre classe Marin, qui retourne true si les champs des deux objets comparés sont égaux.

Exemple 6. Surcharge de equals() dans la classe Marin

```
public  boolean equals(Object o) {
    if (!(o  instanceof Marin))
       return false ;

   Marin marin = (Marin)o ;

    return nom.equals(marin.nom) &&
         prenom.equals(marin.prenom) &&
         salaire == marin.salaire ;
}
```

Analysons ce code.

Remarquons tout d'abord que l'opérateur instanceof retourne systématiquement false si l'objet testé est nul. Cela garantit que l'objet marin est non nul.

Tout d'abord, remarquons que la méthode equals() prend en paramètre un objet de type Object. Une erreur commune consisterait à déclarer l'objet passé en paramètre comme devant être de type Marin. Cette erreur est un peu subtile, et nous la détaillerons dans la suite. Disons ici qu'il est absolument nécessaire que cette méthode equals() prenne un Object en paramètre.

La première chose à faire est de tester si l'objet passé en paramètre est non nul, et s'il est bien de la classe Marin. La comparaison d'un objet de type Marin avec un objet nul ou d'un autre type est légale, et elle retournera false systématiquement, ce qui est normal.

L'opérateur instanceof, utilisé ligne 5, permet de tester la classe d'un objet. En l'occurrence, il retourne true pour tous les objets de la classe Marin, et de toute classe qui hériterait de Marin.

Une fois que nous sommes sûr d'avoir un objet Marin en paramètre, alors il nous faut comparer ses champs un par un. Pour pouvoir accéder à ses champs, il faut le convertir en objet de la bonne classe, c'est ce

que fait la ligne 8. Cette opération s'appelle un *cast* , elle consiste à déclarer un objet (ici marin), et à lui affecter la valeur de l'objet à convertir, en mettant devant et entre parenthèses le type dans lequel on veut faire cette conversion. Il faut toujours vérifier le type de l'objet que l'on caste à l'aide d'un instanceof, avant de faire le *cast* (il existe également une autre méthode, que nous verrons dans la suite). Un *cast* réalisé sur un objet qui ne serait pas de la bonne classe jetterait une exception.

La comparaison des champs de marin est faite entre les lignes 10 et 12. On remarquera que la comparaison des chaînes de caractères se fait en utilisant aussi la méthode equals(), de la classe String. La classe String possède sa propre méthode equals(), qui retourne true si les deux chaînes de caractères contiennent les mêmes caractères.

Nous reverrons en détails la méthode equals() de la classe String, à titre d'exemple supplémentaire.

En toute rigueur, avant de tester l'égalité de nom et prenom, il nous faudrait tester si ces deux chaînes de caractères sont nulles ou pas. Si nom est nul (par exemple), alors notre méthode equals() échouera, en jetant l'ignoble NPE. Comme on peut le voir, l'écriture d'une méthode equals() correcte et complète est un processus qui mérite de l'attention.

La méthode hashCode()

Le rôle de la méthode hashCode() est de calculer un code numérique pour l'objet dans lequel on se trouve. Ce code numérique est censé être représentatif de l'objet, nous allons expliciter ce point immédiatement.

Techniquement, la méthode hashCode() est une méthode native qui permet de calculer un nombre (int) unique associé à une instance de n’importe quelle classe. Par défaut, la méthode de la classe Object retourne l'adresse à laquelle est rangé cet objet, nombre effectivement unique, puisqu'on ne peut pas ranger deux objets au

même endroit en mémoire. En toute rigueur, ce point dépend de la JVM que l'on utilise, mais c'est le cas dans la JVM de Sun.

Le point délicat est le contrat qui lie les méthodes equals() et hashCode() dans les spécifications de Java.

- Deux objets égaux au sens de equals() doivent retourner le même hashCode() ;
- Il n'est pas nécessaire que deux objets différents au sens de equals() retournent deux hashCode() différents...

Donc, surcharger la méthode equals() d'une classe entraîne systématiquement la surcharge de la méthode hashCode(). Ne pas respecter cette règle revient à s'exposer à des bugs obscurs et très difficiles à corriger, nous verrons des exemples précis dans la suite.

Surcharger une méthode hashCode() se fait en respectant un algorithme précis. Il existe plusieurs variantes de cet algorithme, nous en donnons une ici. Supposons que nous ayons surchargé une méthode equals(), en écrivant l'égalité entre plusieurs champs de notre classe.

La première chose à faire est de choisir deux nombres entiers, pas trop petits, 17 et 31 peuvent faire l'affaire.

On initialise l'algorithme en prenant hashCode = 17.

Pour chacun des autres champs c pris en compte par la méthode equals(), on construit l'entier hash suivant :

- si c est un booléen, hash vaut 1 si c est true, 0 s'il est false ;
- si c est de type byte, short, int ou char, alors hash vaut (int)c ;
- si c est de type long, alors hash vaut (int)(c^(c >>> 32)) ;
- si c est de type float, alors hash vaut Float.floatToIntBits(f) ;
- si c est de type double, alors hash vaut Double.doubleToLongBits(f), et l'on prend le code de hachage du long que l'on récupère ;
- si c est nul alors hash vaut 0 ;

- si c est un objet non nul, alors hash vaut c.hashCode() ;
- si c est un tableau, alors chacun des éléments du tableau est traité comme un champ à part entière.

Et pour chacun de ces champs, on met à jour hashCode :

```
hashCode =  31 * hashCode + hash
```

Pour notre classe Marin, la méthode hashCode() est la suivante.

Exemple 7. Surcharge de hashCode() dans la classe Marin

```
public  int hashCode() {
    int hashCode =  17 ;
   hashCode =  31 * hashCode + ((nom == null) ?  0 : nom.hashCode())
;
   hashCode =  31 * hashCode + ((prenom == null) ?  0 :
prenom.hashCode()) ;
   hashCode =  31 * hashCode + salaire ;
    return hashCode ;
}
```

Notons que cette implémentation de la méthode hashCode() est une implémentation parmi d'autres. En particulier, les nombres entiers choisis peuvent varier.

La méthode finalize()

La présentation de la méthode finalize() est une bonne occasion de parler de la destruction des objets. Effectivement, nous avons vu comment construire des objets, mais rien n'a été dit sur leur destruction. Et pour cause : en Java, il n'y a rien à faire pour détruire un objet, la notion de destructeur n'existe tout simplement pas. La machine Java fonctionne avec un ramasse-miettes (*garbage collector* en anglais), qui est censé détecter les objets qui ne sont plus référencés par aucune

variable, et les effacer lui-même. Idéalement, lorsque la dernière référence vers un objet est coupée, le ramasse-miettes enregistre cet objet, et de temps en temps, il se met en marche et libère la mémoire.

Dans la réalité, les choses sont en fait un peu plus complexes. La notion de *garbage collection* est en fait très complexe. Une machine Java a à sa disposition plusieurs *garbage collectors* , qui fonctionnent sur des algorithmes différents. La méthode finalize() a été conçue aux tous débuts de Java, les dernières versions de machines virtuelles ont des *garbage collectors* conçus très récemment.

La sémantique de la méthode finalize() est donc la suivante. Lorsque la machine Java sait qu'elle va supprimer un objet de la mémoire, elle invoque la méthode finalize(). Cette méthode est donc un callback, appelé par la machine Java avant l'effacement d'un objet. Notons tout de suite que l'appel de cette méthode ne se fait pas au moment où un objet n'est plus référencé, mais au moment où la machine Java décide de l'effacer. On n'a donc aucune maîtrise sur le temps au bout duquel l'objet est effacé, ni sur le temps au bout duquel la méthode finalize() est appelée.

Quel thread appelle-t-il la méthode finalize() ? Cette question est épineuse, car tout dépend du type de *garbage collector* utilisé. Ce qui est sûr, c'est que cette méthode n'est pas appelée dans le thread applicatif, et que toute modification de l'objet dans lequel on se trouve risque fort de poser un problème de *race condition* .

D'un façon générale, si l'on choisit d'écrire du code dans cette méthode finalize(), en aucun cas ce code doit avoir une quelconque importance pour le bon déroulement de l'application. Entre autres, il est inutile, et même nuisible, de fermer des fichiers, connexions à des bases de données ou toute autre ressource de ce type. Il est également inutile de tenter de vider le contenu des collections ou des tables de hachage.

Poser du code dans une méthode finalize() a toute les chances d'apporter plus de bugs dans notre application, que de solutions à un quelconque problème.

Notons qu'à partir de Java 9 la méthode finalize()est dépréciée. Même si l'on travaille sur une application de version antérieure, surcharger cette méthode dans une classe est à présent une mauvaise pratique.

Les fuites de mémoire peuvent exister en Java, même si le mécanisme d'allocation et de libération de mémoire mis au point dans les dernières versions de JVM se révèlent très performant, en fait plus performant même que la classique allocation / libération manuelle du C ou du C++. Tenter de les résoudre en posant du code dans la méthode finalize() est inutile et risque d'introduire plus de problèmes que de solutions.

La méthode getClass()

Cette méthode retourne un objet, instance d'une classe particulière appelée Class. Tout est objet en Java, y compris les classes elles-mêmes ! Il existe donc une classe Class, qui modélise les classes Java. Nous verrons par la suite que cette classe est à la base des mécanismes d'introspection, et ouvre la porte à des méthodes de programmation très puissantes.

Notons que la méthode toString() de la classe Class est surchargée, mais ne retourne pas le nom de la classe, comme on pourrait s'y attendre.

```
Marin m =  new Marin("Surcouf",  "Robert") ;
System.out.println("Classe de marin : " + m.getClass()) ;

> Classe de marin :  class Marin
```

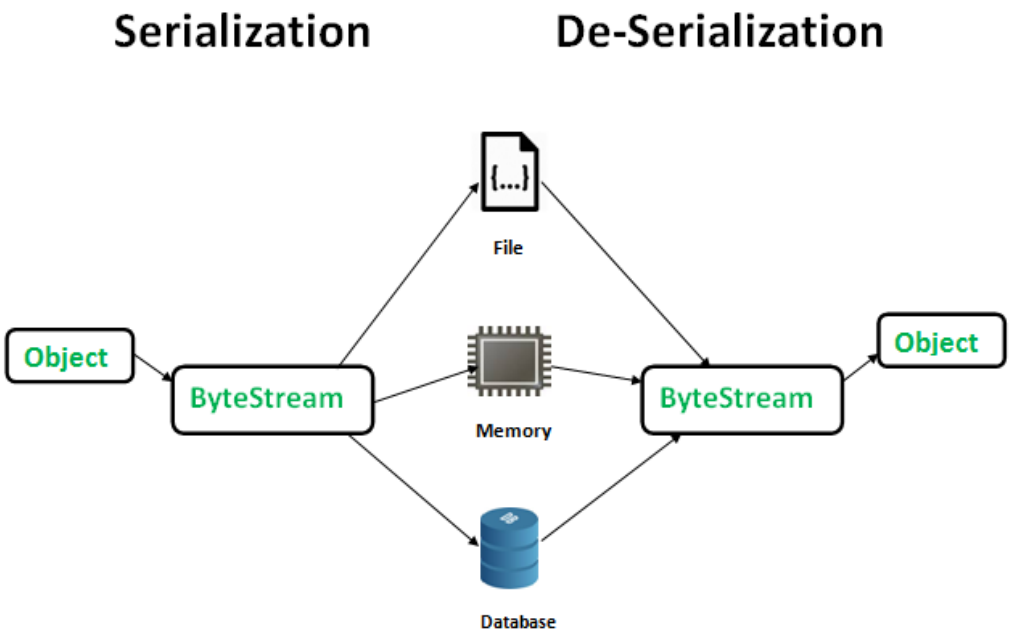
Si l'on veut juste le nom de la classe, il faut invoquer sa méthode getName().

```
Marin m =  new Marin("Surcouf",  "Robert") ;
System.out.println("Classe de marin : " + m.getName()) ;

> Classe de marin : Marin
```

Sérialisation et désérialisation

La sérialisation est un mécanisme de conversion de l'état d'un objet en un flux d'octets. La désérialisation est le processus inverse dans lequel le flux d'octets est utilisé pour recréer l'objet Java réel en mémoire. Ce mécanisme est utilisé pour conserver l'objet.



Le flux d'octets créé est indépendant de la plate-forme. Ainsi, l'objet sérialisé sur une plateforme peut être désérialisé sur une plateforme différente.

Pour rendre un objet Java sérialisable, nous implémentons l' interface **java.io.Serializable** .
La classe ObjectOutputStream contient la méthode **writeObject ()** pour sérialiser un Object.

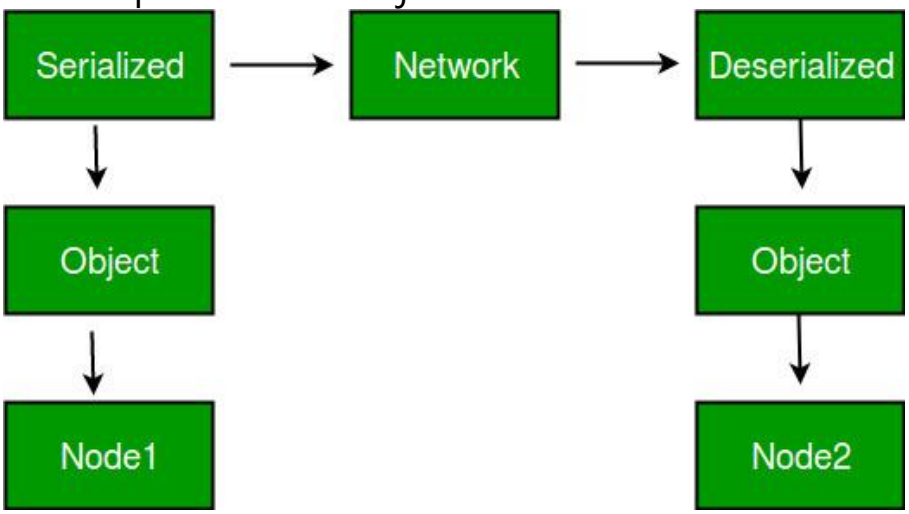
```
public final void writeObject (Object obj)
                    jette IOException
```

La classe ObjectInputStream contient la méthode **readObject ()** pour désérialiser un objet.

```
Objet public final readObject ()
                lance IOException,
              ClassNotFoundException
```

Avantages de la sérialisation

1. Pour enregistrer / conserver l'état d'un objet.
2. Pour parcourir un objet sur un réseau.



Seuls les objets de ces classes peuvent être sérialisés qui implémentent l' interface **java.io.Serializable** .

Serializable est une **interface de marqueur** (n'a pas de membre de données ni de méthode). Il est utilisé pour «marquer» les classes Java afin que les objets de ces classes puissent obtenir certaines capacités. D'autres exemples d'interfaces de marqueurs sont: - Clonable et Remote.

Points à retenir

1. Si une classe parent a implémenté une interface Serializable, alors la classe enfant n'a pas besoin de l'implémenter mais l'inverse n'est pas vrai.
2. Seuls les membres de données non statiques sont enregistrés via le processus de sérialisation.
3. Les membres de données statiques et les membres de données transitoires ne sont pas enregistrés via le processus de sérialisation. Ainsi, si vous ne souhaitez pas enregistrer la valeur d'un membre de

données non statique, rendez-le transitoire.
4. Le constructeur d'objet n'est jamais appelé lorsqu'un objet est désérialisé.
5. Les objets associés doivent implémenter une interface sérialisable.
Exemple :

```
la classe A implémente Serializable {


// B implémente également Serializable
// interface.
B ob = nouveau B ();
}
```

SerialVersionUID

Le runtime de sérialisation associe un numéro de version à chaque classe Serializable appelée SerialVersionUID, qui est utilisé pendant la désérialisation pour vérifier que l'expéditeur et le destinataire d'un objet sérialisé ont chargé des classes pour cet objet qui sont compatibles avec la sérialisation. Si le destinataire a chargé une classe pour l'objet qui a un UID différent de celui de la classe de l'expéditeur correspondant, la désérialisation entraînera une **InvalidClassException** . Une classe Serializable peut déclarer son propre UID explicitement en déclarant un nom de champ.
Il doit être statique, définitif et de type long.
ie- ANY-ACCESS-MODIFIER statique final long serialVersionUID = 42L;
Si une classe sérialisable ne déclare pas explicitement un serialVersionUID, le runtime de sérialisation en calculera un par défaut pour cette classe en fonction de divers aspects de la classe, comme décrit dans Spécification de sérialisation d'objets Java. Cependant, il est fortement recommandé que toutes les classes sérialisables déclarent explicitement la valeur serialVersionUID, car son calcul est très sensible aux détails de classe qui peuvent varier en fonction des implémentations du compilateur, tout changement de classe ou l'utilisation d'un identifiant différent peut affecter les données sérialisées.

Il est également recommandé d'utiliser un modificateur privé pour l'UID car il n'est pas utile en tant que membre hérité.

serialver

Le serialver est un outil fourni avec JDK. Il est utilisé pour obtenir le numéro serialVersionUID des classes Java.
Vous pouvez exécuter la commande suivante pour obtenir serialVersionUID