

La méthode finalize()

Philippe PRADOS

pp@philippe.prados.name



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIERES

1.	Îlot.....	3
2.	Résurrection d'une instance	3
3.	L'appel de finalize().....	3
4.	clause finally	7
5.	Conclusion	8

Avant-propos

Quand est appelée la méthode `finalize()` ? Comment la rédiger ?

La méthode `finalize()` est appelée par le ramasse-miettes avant qu'une instance ne soit détruite de la mémoire.

Le ramasse miette parcourt périodiquement toutes les instances reliées à touche les tâches actives, et détecte ainsi les instances perdues. Elles seront alors détruites de la mémoire. Auparavant, la méthode `finalize()` est éventuellement appelée. Cela permet à l'instance de libérer les ressources qu'elle utilise. Par exemple, une classe utilisant un fichier temporaire, peut supprimer ce fichier du disque lors de l'appel à la méthode `finalize()`.

```
class TempFile
{
  private File name_=new File("file.tmp");
  private FileOutputStream out_;
  ...
  public void finalize() throws IOException
  {
    out_.close(); // Ferme le fichier
    name_.delete(); // et l'efface.
  }
}
```

Pour que la méthode `TempFile.finalize()` soit appelée, il faut qu'il n'existe plus de pointeur sur l'instance et que le ramasse-miettes entre en action.

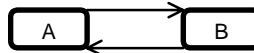
```
TempFile tmp=new TempFile();
tmp=null; // Va détruire le fichier
```

Le ramasse-miettes intervient en tâche de fond. Il n'est pas possible de prédire quand la méthode `finalize()` sera appelée. Cela veut dire que la ressource `file.tmp` ne sera pas disponible tant que le ramasse-miettes n'est pas intervenu. Pour demander explicitement le nettoyage de la mémoire, il faut appeler la méthode `System.gc()`. Cette méthode lance en tâche de fond le ramasse-miettes. La méthode `System.runFinalization()` lance le ramasse-miettes et attend que toutes les méthodes `finalize()` soient effectuées avant de rendre la main.

Lorsqu'un programme se termine, les méthodes `finalize()` ne sont pas forcément appelées. Pour garantir l'appel aux méthodes `finalize()` avant la fin du programme, il faut appeler `System.runFinalizationOnExit()` avec la valeur `true`. Ce paramétrage permet d'imposer l'appel de toutes les méthodes `finalize()` de toutes les instances, avant de sortir du programme. Cela permet de garantir que le fichier temporaire précédent sera détruit.

1. ÎLOT

Une instance peut être référencée par une autre instance, tout en étant candidate pour le ramasse-miettes. Si deux instances se réfèrent mutuellement, mais qu'aucune autre instance les références, il y a un îlot non rattaché aux différentes tâches.



Le ramasse-miettes est capable d'identifier les cycles pour décider d'en supprimer toutes les instances.

2. RESURRECTION D'UNE INSTANCE

Une instance peut être sélectionnée afin d'être supprimée de la mémoire, mais l'appel de la méthode `finalize()` peut la ressusciter. Par exemple,

```
public class MaClass
{
  private static MaClass ref;
  public void finalize()
  {
    ref=this;
  }
}
```

Lorsque la méthode `finalize()` d'une instance `MaClass` est appelée, le pointeur `ref` décide, en extrémiste, de référencer l'objet. L'instance ne doit plus être candidate au ramasse-miettes. Il ne faut plus détruire l'instance, sinon le pointeur `ref` serait dans le vide.

Pour résoudre cela, l'algorithme du ramasse-miettes suit un diagramme d'état transition complexe.

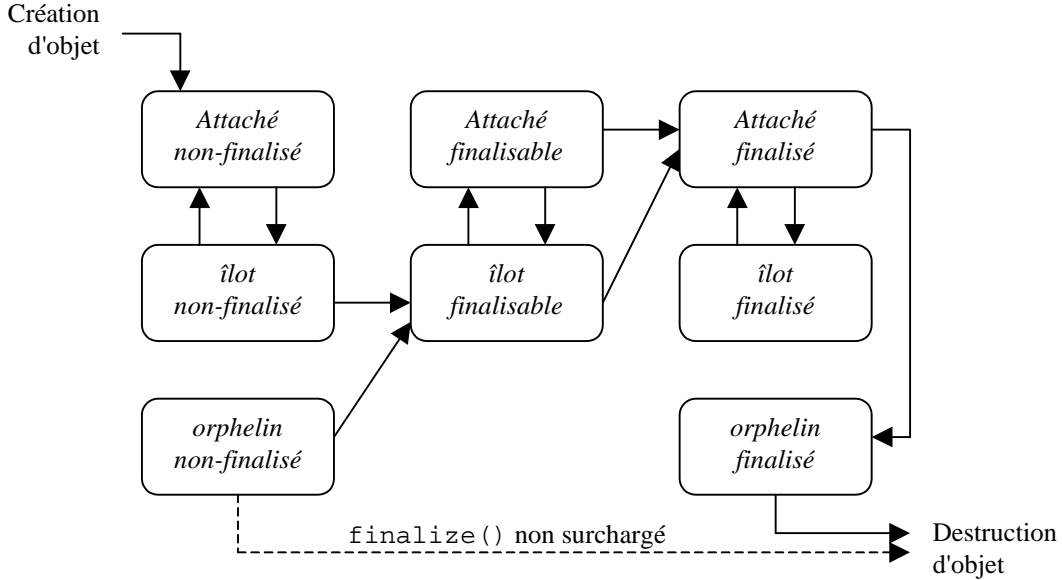
3. L'APPEL DE FINALIZE()

Chaque instance possède deux états. Le premier permet d'indiquer si l'instance est *attachée* à une tâche, dans un *îlot* ou *orpheline*. Le deuxième état permet d'indiquer le comportement à avoir vis-à-vis de la méthode `finalize()`. Il comporte trois états : *non-finalisé*, *finalisable* et *finalisé*.

Une instance *attachée* est une instance reliée directement ou indirectement à une des tâches du système. L'état *îlot* indique que l'instance est attachée à une autre instance, mais pas à une tâche du système. L'instance fait partie d'un îlot d'instances se référant mutuellement. L'état *orphelin* indique que l'instance n'est référencée par aucune instance. Elle ne fait pas partie d'un îlot.

L'état *non-finalisé* indique que l'instance n'a pas encore été identifiée comme devant être détruite. L'état *finalisable* indique que l'instance est candidate à l'invocation de la méthode `finalize()`. L'état *finalisé* indique que la méthode `finalize()` a été invoquée par le ramasse-miettes.

L'automate suivant, décrit les différentes transitions possibles des états d'une instance.



Il n'est pas nécessaire de maîtriser exactement ce schéma. Il faut savoir que : le suivi de l'appel à `finalize()` s'effectue par une transition de la gauche vers la droite. Il n'est pas possible de revenir vers la gauche. Il n'existe pas de flèches dans ce sens ; la machine virtuelle identifie différemment les états *orphelins* et *îlot*.

Ce schéma entraîne plusieurs remarques.

- La méthode `finalize()` peut être invoquée comme n'importe quelle méthode.

La spécification de la machine virtuelle garantit cela. Un appel explicite ne modifie pas l'état de l'instance. Elle ne passe pas à l'état *finalisé*. Il faut alors rédiger la méthode avec une approche défensive vis-à-vis d'un appel multiple.

```
public class MaClass
{
    FileInputStream in_;
    public void finalize() throws Throwable
    { // Vérification du premier appel
      if (in_!=null)
      { // Traitement
        in_.close();
        // Indique que l'appel est effectué
        in_=null;
      }
    }
}
```

- Les exceptions générées par la méthode `finalize()` sont ignorées.

Il faut faire attention aux exceptions générées par l'invocation de la méthode `super.finalize()`. Pour éviter tous problèmes, il faut déclarer l'appel à `super.finalize()` dans une section `finally`.

```
public class MaClass
{
    public void finalize() throws Throwable
    {
        try
        { ...
        } finally
        {
            super.finalize();
        }
    }
}
```

- La méthode `finalize()` n'est jamais appelée deux fois par le ramasse-miettes.

Elle peut être appelée explicitement par le programme, donc invoquée plusieurs fois, mais le ramasse-miettes ne l'appellera qu'une seule fois. Par exemple, si la méthode `finalize()` ressuscite `this`, lors d'une deuxième perte de l'instance, la méthode ne sera pas rappelée.

```
public class MaClass
{
    private static MaClass ref;
    public void finalize()
    {
        ref=this; // Ressuscite une seule fois
    }
}
```

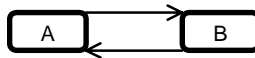
Il est préférable de dupliquer l'instance avant de la ressusciter, afin de garder un état *non-finalisé*.

```
public class MaClass
{
    public Object clone()
    { ...
    }
    private static MaClass ref;
    public void finalize()
    {
        ref=clone(); // Ressuscite une copie
    }
}
```

Ainsi, la méthode `finalize()` de la copie pourra être appelée par le ramasse-miettes.

- L'ordre de destruction d'un îlot n'est pas défini.

Si plusieurs instances se réfèrent mutuellement, le ramasse-miettes peut sélectionner n'importe laquelle pour lui appeler la méthode `finalize()`. Il peut même appeler simultanément dans plusieurs tâches la méthode `finalize()` de plusieurs instances appartenant au même îlot. Il est donc difficile de ressusciter une instance lors de la perte d'un îlot. En effet, pour la situation suivante :



Imaginons que la méthode `finalize()` de `A` ressuscite l'instance `this`. Deux situations peuvent se produire. Si le ramasse miette décide d'appeler la méthode `finalize()` de `A` avant la méthode `finalize()` de `B`, l'instance `A` sera ressuscitée, donc l'instance `B` également (via `A`). Si `B.finalize()` est appelé avant `A.finalize()`, l'instance `B` passe à l'état *finalisé*, avant d'être ressuscitée par l'intermédiaire de `A`. Au final, `A` et `B` sont ressuscités, mais suivant les cas, l'instance `B` est déclaré *finalisable* ou *finalisé*.

- Les méthodes `finalize()` peuvent être appelées dans des tâches différentes.

Cela impose une rédaction rigoureuse de la méthode. En effet, la méthode `finalize()` ne peut utiliser que des méthodes propres vis-à-vis du multitâche.

```
public class AutreClass
{
    ...
    void addInstance(Object x)
    { ...
    }
    static AutreClass dico=new AutreClass();
}
public class MaClass
{
    public void finalize()
    {
        AutreClass.dico.addInstance(this);
    }
}
```

Le code précédent est erroné si deux méthodes `finalize()` sont appelées simultanément par le ramasse-miettes. En effet, la méthode `addInstance()` n'est pas `synchronized`. Cette erreur est très difficile à détecter car elle est extrêmement difficile à reproduire.

Il faut faire attention au parcourt des relations dans la méthode `finalize()`. Les agrégations ne posent pas de problèmes car la méthode `finalize()` est la seule qui peut les parcourir.

La méthode `finalize()` peut être invoquée par n'importe quelle tâche. Par exemple, une tâche ayant pris le moniteur d'un objet par une méthode `synchronized` peut appeler une méthode `finalize()` sur l'invocation d'un `new`. Dans ce cas, la méthode `finalize()` peut modifier l'instance bloquée car elle est dans la même tâche ! La version 1.2 de Java garantie que la tâche qui appellera la méthode `finalize()` n'aura pas de moniteur en cours.

Par contre, si la méthode `finalize()` appelle une méthode sur une instance bloquée, il y a un risque d'étreinte mortelle.

- La méthode `finalize()` peut être appelée par la machine virtuelle, même si l'instance est référencée.

C'est le cas lorsque la méthode `System.exit()` est appelée alors que la méthode `System.runFinalizersOnExit()` a été appelée avec la valeur `true`. Avant de sortir du programme, la machine virtuelle appelle la méthode `finalize()` de toutes les instances n'étant pas à l'état *finalisé*. Ces instances sont à ce moment précis, reliées à d'autres instances.

Attention, la méthode `runFinalizersOnExit()` est déclarée `deprecated` dans la version 2.0 de Java. Cela entraîne que les méthodes `finalize()` ne sont pas forcément appelées !

- Ne pas concevoir de programme qui dépende du moment où une méthode `finalize()` sera appelée.

La destruction d'une méthode n'est pas prédictive. Les ressources utilisées par un objet ne seront pas libérées tant que le ramasse-miettes n'aura pas décidé de détruire l'instance.

- Ne pas relier la libération d'une ressource non-mémoire à une méthode `finalize()`.

Par exemple, si un objet ouvre un fichier, et le ferme dans la méthode `finalize()`, il est possible que le nombre de fichiers ouverts simultanément soit dépassé, avant que le ramasse-miettes intervienne. La machine Java possède généralement un nombre limité de fichiers pouvant être ouverts simultanément. Il faut offrir une méthode explicite pour libérer la ressource, et, éventuellement, la libérer in extrémiste dans la méthode `finalize()`.

```
public class MaClass
{
    FileInputStream in_;
    ...
    public void close()
    {
        in_.close();
        in_=null;
    }
    public void finalize() throws Throwable
    {
        // Vérification du premier appel
        if (in_!=null)
        {
            // Traitement
            in_.close();
            // Indique que l'appel est effectué
            in_=null;
        }
    }
}
```

Le traitement présent dans la méthode `finalize()` permet de corriger une erreur, un oubli de l'appel à la méthode `close()`. Il peut être préférable d'afficher une erreur dans ce cas.

```
public class MaClass
{
    FileInputStream in_;
    ...
    public void close()
    {
        in_.close();
        in_=null;
    }
    public void finalize() throws Throwable
    {
        // Vérification du close
        if (in_!=null)
        {
            System.out.println("Pre-condition: appelez la méthode close()");
        }
    }
}
```

- Une classe peut également être finalisée.

Pour cela, il faut déclarer la méthode statique `classFinalize()`.

```
public class MaClass
{
    static void classFinalize() throws Throwable
    {
        ...
    }
}
```

La classe est supprimée de la mémoire lorsque le `ClassLoader` l'ayant instancié est supprimé.

Lorsqu'une instance doit être supprimée de la mémoire, et qu'elle déclare une méthode `finalize()`, elle est placée dans une liste d'instance à finaliser. Les méthodes `finalize()` seront appelées lorsque le premier niveau du ramasse-miettes échoue. Le premier niveau peut échouer seulement lorsqu'il n'existe plus assez de mémoire pour répondre à une allocation, ou lorsque l'espace disponible est inférieur à 25%.

Il est préférable d'appeler explicitement la méthode `finalize()` sur les instances agrégées car cela permet de libérer immédiatement toutes les ressources. La méthode devant être résistante à un appel multiple, le traitement ultérieur sur l'instance agrégée, effectué par le ramasse-miettes, n'aura aucun effet. Pour cela, il faut que la méthode `finalize()` soit déclarée public.

```
public class MaClass
{ private Agregation agr_;
  ...
  public void finalize() throws Throwable
  {
    ...
    agr_.finalize();
    agr_=null;
  }
}
```

En règle générale, il est préférable d'éviter les méthodes `finalize()` complexe. Pour gérer les ressources autres que la mémoire, il est préférable d'indiquer uniquement une post-condition dans la méthode `finalize()` pour signaler au développeur qu'il a oublié de libérer une ressource.

```
public class MaClass
{ FileInputStream in_;
  ...
  public void close()
  { in_.close();
    in_=null;
  }
  public void finalize() throws Throwable
  { // Vérification du premier appel
    if (in_!=null)
    { System.err.println("Vous n'avez pas appelé la méthode close() !");
    }
  }
}
```

CLAUSE

Lors de l'appel d'une méthode, il est possible de libérer les ressources utilisées, quelles que soient les causes de sortie.

```
try
{ ...
} finally
{ // When exit
  ...
}
```

Il y peut y avoir des surprises lors de la rédaction de la clause `finally`.

```
void f()
{
  int i=1;
  try
  { return i;
  } finally
  { i=2;
  }
}
```

Que retourne la méthode `f()` ? Un ou deux ?

En fait, la clause `finally` est appelée comme une sous-routine avant chaque sortie du programme. La méthode `f()` précédente est compilé comme ceci.

```
void f()
{
  int i=1;
  int _tmp;
  try
  { tmp=i;
    call_finally();
    return tmp;
  } finally
  { i=2;
  }
}
```

La méthode `f()` retourne la valeur un, celle calculé juste avant l'appel de la clause.

La clause `finally` peut également interrompre un traitement est ainsi, empêcher le retour de la méthode.

```
void f()
{ for (;;)
  { try
    { return;
    } finally
    { continue;
    }
  }
}
```

L'instruction `return` entraîne l'appel de la clause `finally`. Celle-ci `continue` la boucle. Cette méthode est une boucle sans fin. La méthode ne retournera jamais à l'appelant. Une instruction `break` peut également être placée dans la clause.

```
boolean f()
{ for (;;)
  { try
    { return false;
    } finally
    { break;
    }
  }
  return true;
}
```

Cette version de la méthode `f()` retourne la valeur `true`, suite à l'instruction `return false` !

5. CONCLUSION

Les différentes règles à retenir pour la méthode `finalize()` sont les suivantes.

- La méthode `finalize()` peut être invoquée comme n'importe quelle méthode.
- Elle n'est jamais appelée deux fois par le ramasse-miettes.
- L'ordre de destruction d'un filot n'est pas défini.
- Les méthodes `finalize()` peuvent être appelées dans des tâches différentes.
- La méthode `finalize()` peut être appelée par la machine virtuelle, même si l'instance est référencée.
- Les exceptions générées sont ignorées.
- Le corps de la méthode doit être rédigée avec une approche défensive vis-à-vis d'un deuxième appel.
- Ne pas concevoir de programme qui dépende du moment ou une méthode `finalize()` sera appelé.
- Ne pas relier la libération d'une ressource non-mémoire à une méthode `finalize()`.
- La méthode `finalize()` n'est pas toujours appelée (JDK 2.0)
- Une classe peut également être finalisée.

Le modèle pour la méthode `finalize()` est le suivant :

```
public void finalize() throws Throwable
{
  try
  { // Vérification du premier appel
    if (...)
    { // Traitement
      ...
      // Indique que l'appel est effectué
      ...
    }
  } finally
  {
    super.finalize();
  }
}
```