final, finally et finaliser en Java

Il s'agit d'une question importante concernant le point de vue de l'entretien.

mot-clé final

final (minuscule) est un mot-clé réservé en java. Nous ne pouvons pas l'utiliser comme identifiant car il est réservé. Nous pouvons utiliser ce mot-clé avec des variables, des méthodes et aussi avec des classes. Le mot clé final en java a une signification différente selon qu'il est appliqué à une variable, une classe ou une méthode.

1. **final avec Variables:** La valeur de variable ne peut pas être modifiée une fois initialisée.


```
class A {
    public static void main(String[] args)
    {
        // Non final variable
        int a = 5;

        // final variable
        final int b = 6;

        // modifying the non final variable : Allowed
        a++;

        // modifying the final variable :
        // Immediately gives Compile Time error.
        b++;
    }
}
```

Si nous déclarons une variable final, nous ne pouvons pas modifier son contenu car elle est final, et si nous la modifions alors nous obtenons une erreur de compilation.

2. **final avec classe:** la classe ne peut pas être sous-classée. Chaque fois que nous déclarons une classe final, cela signifie que nous ne pouvons pas étendre cette classe ou que cette classe **ne peut pas être étendue** ou que nous ne pouvons pas créer de sous-classe de cette classe.

filter_none

luminosité_4

```
final class RR {
    public static void main(String[] args)
    {
        int a = 10;
    }
}
// here gets Compile time error that
// we can't extend RR as it is final.
class KK extends RR {
    // more code here with main method
}
```

3. **final avec Méthode:** La méthode ne peut pas être remplacée par une sous-classe. Chaque fois que nous déclarons une méthode final, cela signifie que nous ne pouvons pas remplacer cette méthode.

filter_none

luminosité_4

```
class QQ {
    final void rr() {}
    public static void main(String[] args)
    {
    }
}


class MM extends QQ {
```

```
    // Here we get compile time error
    // since can't extend rr since it is final.
    void rr() {}
}
```

Remarque: Si une classe est déclarée final, **par défaut,** toutes les méthodes présentes dans cette classe sont automatiquement finals, mais pas les **variables** .

filter_none

Éditer

play_arrow

luminosité_4

```
// Java program to illustrate final keyword
final class G {

    // by default it is final.
    void h() {}

    // by default it is not final.
    static int j = 30;

public static void main(String[] args)
    {
        // See modified contents of variable j.
        j = 36;
        System.out.println(j);
    }
}
```

Sortie :

```
36
```

finally mot-clé

Tout comme final est un mot-clé réservé, de la même manière, finalment, est également un mot-clé réservé en java, c'est-à-dire que nous ne pouvons pas l'utiliser

comme identifiant. Le mot clé finally est utilisé en association avec un bloc try / catch et garantit qu'une section de code sera exécutée, même si une exception est levée. Le bloc finally sera exécuté après les blocs try et catch, mais avant que le contrôle ne revienne à son origine.

filter_none

Éditer

play_arrow

luminosité_4

```
// A Java program to demonstrate finally.
class Geek {
    // A method that throws an exception and has finally.
    // This method will be called inside try-catch.
    static void A()
    {
        try {
            System.out.println("inside A");
            throw new RuntimeException("demo");
        }
        finally
        {
            System.out.println("A's finally");
        }
    }

    // This method also calls finally. This method
    // will be called outside try-catch.
    static void B()
    {
        try {
            System.out.println("inside B");
            return;
        }
        finally
        {
            System.out.println("B's finally");
```

```
        }
    }

    public static void main(String args[])
    {
        try {
            A();
        }
        catch (Exception e) {
            System.out.println("Exception caught");
        }
        B();
    }
}
```

Production:

```
à l'intérieur de A

A est finally

Exception interceptée

à l'intérieur de B

B's finally
```

Il existe différents cas où finalment peut être utilisé. Il en est question ci-dessous:

1. **Cas 1: aucune exception ne se produit dans le programme**

filter_none

Éditer

play_arrow

luminosité_4

```
// Java program to illustrate finally in
// Case where exceptions do not
// occur in the program
class B {
    public static void main(String[] args)
    {
```

```
            int k = 55;
            try {
                System.out.println("In try block");
                int z = k / 55;
            }

            catch (ArithmeticException e) {
                System.out.println("In catch block");
                System.out.println("Dividing by zero but caught");
            }

            finally
            {
                System.out.println("Executes whether exception occurs or not");
            }
        }
}
```

Sortie :

```
Dans le bloc try

Exécute si une exception se produit ou non
```

Ici, l'exception ne se produit pas, mais finalment, le bloc s'exécute, car finalment est destiné à exécuter si une exception se produit ou non.

Flux du programme ci-dessus : il commence d'abord par la méthode principale, puis va essayer bloc et dans essayer, car aucune exception ne se produit, donc le flux ne va pas au bloc de capture, donc le flux va directement de la tentative au bloc final.

2. **Cas 2: une exception se produit et des correspondances de bloc de capture correspondantes**

filter_none

Éditer

play_arrow

luminosité_4

```
// Java program to illustrate finally in
```

```
// Case where exceptions occur
// and match in the program
class C {
    public static void main(String[] args)
    {
        int k = 66;
        try {
            System.out.println("In try block");
            int z = k / 0;
            // Carefully see flow dosen't come here
            System.out.println("Flow dosen't came here");
        }

        catch (ArithmeticException e) {
            System.out.println("In catch block");
            System.out.println("Dividing by zero but caught");
        }

        finally
        {
            System.out.println("Executes whether an exception occurs or not");
        }
    }
}
```

Sortie :

```
Dans le bloc try

Dans le bloc de capture

Diviser par zéro mais attraper

Exécute si une exception se produit ou non
```

Ici, l'exception ci-dessus se produit et le bloc catch correspondant a été trouvé mais toujours finalment le bloc s'exécute puisque finalment est destiné à exécuter si une exception se produit ou non ou si le bloc catch correspondant a été trouvé ou non.

Flux du programme ci-dessus : commence d'abord par la méthode principale, puis essaie le bloc et, dans l'essai, une exception arithmétique se produit et le bloc de capture correspondant est également disponible, donc le flux va au bloc

de capture. Après que l' écoulement ne va pas essayer de nouveau bloc car une fois une exception se produit dans le bloc d'essai puis couler **doesnt** revenir à nouveau pour essayer bloc. Après cela, finalment, execute puisque finally est destiné à exécuter si une exception se produit ou non ou si le bloc catch correspondant a été trouvé ou non.

3. **Cas 3: Une exception se produit et le bloc de capture correspondant est introuvable / correspond**

filter_none

Éditer

play_arrow

luminosité_4

```
// Java program to illustrate finally in
// Case where exceptions occur
// and do not match any case in the program
class D {
    public static void main(String[] args)
    {
        int k = 15;
        try {
            System.out.println("In try block");
            int z = k / 0;
        }

        catch (NullPointerException e) {
            System.out.println("In catch block");
            System.out.println("Dividing by zero but caught");
        }

        finally
        {
            System.out.println("Executes whether an exception occurs or not");
        }
    }
}
```

Sortie :

```
Dans le bloc try

Exécute si une exception se produit ou non

Exception dans le thread "principal":
java.lang.ArithmeticException:

/ par zéro suivi d'une trace de pile.
```

Ici ci-dessus, une exception se produit et le bloc catch correspondant n'est pas trouvé / correspond, mais finalment le bloc s'exécute, car finalment est destiné à exécuter si une exception se produit ou non ou si le bloc catch correspondant est trouvé / correspond ou non.

Flux du programme ci - dessus : Tout d' abord commence à partir de la méthode principale puis va essayer bloc et essayer une exception arithmétique se produit et bloc catch correspondant est **pas** disponible si l' écoulement **doesnt** va à bloc catch. Après que l' écoulement ne va pas essayer de nouveau bloc car une fois une exception se produit dans le bloc d'essai puis couler **doesnt** revenir à nouveau pour essayer bloc. Après cela, finalment, execute puisque finally est censé exécuter si une exception se produit ou non ou si le bloc catch correspondant a été trouvé / correspond ou non.

Application du bloc finally : Donc, fondamentalement, l'utilisation du bloc finalment est **la désallocation des ressources** . Signifie que toutes les ressources telles que les connexions réseau, les connexions à la base de données, que nous avons ouvertes dans le bloc try, doivent être fermées afin que nous ne perdions pas nos ressources telles qu'elles sont ouvertes. Ces ressources doivent donc être fermées dans le bloc finally.

```
// Java program to illustrate
// use of finally block
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

class K {
private static final int SIZE = 10;
    public static void main(String[] args)
    {

        PrintWriter out = null;
        try {
            System.out.println("Entered try statement");

            // PrintWriter, FileWriter
            // are classes in io package
            out = new PrintWriter(new FileWriter("OutFile.txt"));
        }
        catch (IOException e) {
            // Since the FileWriter in
            // try block can throw IOException
        }

        // Following finally block cleans up
        // and then closes the PrintWriter.

        finally
        {
            if (out != null) {
                System.out.println("Closing PrintWriter");
                out.close();
            } else {
                System.out.println("PrintWriter not open");
```

```
            }
        }
    }
}
```

Sortie :

```
Instruction try entrée

PrintWriter pas ouvert
```

Remarque : Le bloc finally est un outil clé pour éviter les fuites de ressources. Lorsque vous fermez un fichier ou récupérez d'autres ressources, placez le code dans un bloc finally pour vous assurer que la ressource est toujours récupérée.

Comment jdk 1.7 utilise-t-il finalment le bloc comme facultatif?

Jusqu'à ce que jdk 1.6 finalment, le bloc soit comme un héros, c'est-à-dire qu'il est recommandé de l'utiliser pour la désallocation des ressources mais à partir de jdk 1.7 finally, le bloc est maintenant optionnel (mais vous pouvez l'utiliser). Étant donné que les ressources que nous avons ouvertes dans le bloc try seront automatiquement libérées / fermées lorsque le flux de programme atteint la fin du bloc try.
Ce concept de désallocation automatique des ressources sans utiliser finalment le bloc est appelé **instruction try-with-resources** .

finaliser la méthode

C'est une **méthode** que le **Garbage Collector** appelle toujours juste **avant** la suppression / destruction de l'objet éligible au Garbage Collection, afin d'effectuer **une activité de nettoyage** . L'activité de nettoyage signifie la fermeture des ressources associées à cet objet comme la connexion à la base de données, la connexion réseau ou nous pouvons dire la désallocation des ressources. N'oubliez pas qu'il **ne** s'agit **pas d'** un mot clé réservé.
Une fois la méthode de finalisation terminée, Garbage Collector détruit immédiatement cet objet. La méthode finalize est présente dans la classe Object et sa syntaxe est:

```
void protégé finaliser les lancers Throwable {}
```

Étant donné que la classe Object contient la méthode finalize, la méthode finalize est disponible pour chaque classe java, car Object est la superclasse de toutes les classes java. Comme il est disponible pour chaque classe java, Garbage Collector

peut appeler la méthode finalize sur **n'importe quel objet java.**
Maintenant, la méthode finalize qui est présente dans la classe Object a une implémentation vide, dans notre classe, les activités de nettoyage sont là, alors nous avons pour **remplacer** cette méthode afin de définir nos propres activités de nettoyage.
Cas liés à la méthode de finalisation:

1. **Cas 1:** l'objet éligible à la récupération de place, la méthode de finalisation de classe correspondante de cet objet va être exécutée

```
class Hello {
    public static void main(String[] args)
    {
        String s = new String("RR");
        s = null;

        // Requesting JVM to call Garbage Collector method
        System.gc();
        System.out.println("Main Completes");
    }

    // Here overriding finalize method
    public void finalize()
    {
        System.out.println("finalize method overriden");
    }
}
```

Sortie :

```
Achèvement principal
```

Remarque : Ici, la sortie ci-dessus n'est venue que pour les **achèvements principaux** et **non pour** «la finalisation de la méthode» car le garbage collector appelle la méthode de finalisation sur cet objet de classe qui est éligible pour le garbage collection. Ci-dessus, nous avons fait->
s = null et 's' est l'objet de la classe String, donc la méthode de finalisation de la classe String va être appelée et non notre classe (c'est-à-dire la classe Hello). Nous modifions donc notre code comme->

```
Bonjour s = nouveau Bonjour () ;
s = null ;
```

Maintenant, notre classe c'est-à-dire la méthode de finalisation de la classe Hello est appelée. **Sortie** :

```
méthode de finalisation remplacée

Achèvement principal
```

Donc, fondamentalement, Garbage Collector appelle la méthode finalize sur cet objet de classe qui est éligible pour le garbage collection. Donc, si l'objet String est éligible pour le Garbage Collection, la méthode **String** class finalize va être appelée et **non la** méthode **Hello class** finalize.

2. **Cas 2:** Nous pouvons appeler la méthode finalize de manière explicite, puis elle sera exécutée comme un appel de méthode normal, mais l'objet ne sera pas supprimé / détruit

```
class Bye {
    public static void main(String[] args)
    {
        Bye m = new Bye();

        // Calling finalize method Explicitly.
        m.finalize();
        m.finalize();
        m = null;

        // Requesting JVM to call Garbage Collector method
        System.gc();
        System.out.println("Main Completes");
    }

    // Here overriding finalize method
    public void finalize()
    {
        System.out.println("finalize method overriden");
    }
```

```
}
```

Sortie :

```
méthode de finalisation remplacée

// appeler par le programmeur mais l'objet ne sera pas détruit.

méthode de finalisation remplacée

// appeler par le programmeur mais l'objet ne sera pas détruit.

Achèvement principal

méthode de finalisation remplacée

// appel par Garbage Collector juste avant de détruire l'objet.
```

Remarque : Comme finalize est une méthode et non un mot clé réservé, nous pouvons donc appeler **explicitement la** méthode **finalize** , alors elle sera exécutée comme un appel de méthode normal mais l'objet ne sera pas supprimé / détruit.

3. **Cas 3:**
   - **Partie a)** Si le programmeur appelle la méthode finalize, lors de l'exécution de la méthode finalize, une exception non vérifiée se lève.

```
class Hi {
    public static void main(String[] args)
    {
        Hi j = new Hi();

        // Calling finalize method Explicitly.
        j.finalize();

        j = null;

        // Requesting JVM to call Garbage Collector method
        System.gc();
```

```
            System.out.println("Main Completes");
        }

        // Here overriding finalize method
        public void finalize()
        {
            System.out.println("finalize method overriden");
            System.out.println(10 / 0);
        }
}
```

Sortie :

```
exception dans le thread "main" java.lang.ArithmeticException:

/ par zéro suivi d'une trace de pile.
```

Donc, le **point clé** est le suivant: si le programmeur appelle la méthode finalize, alors que l'exécution de la méthode finalize, une exception non vérifiée se lève, la JVM termine le programme anormalement en augmentant l'exception. Donc, dans ce cas, la fin du programme est **anormale** .

- **partie b)** Si garbage Collector appelle la méthode finalize, lors de l'exécution de la méthode finalize, une exception non vérifiée se lève.

```
class RR {
    public static void main(String[] args)
    {
        RR q = new RR();
        q = null;

        // Requesting JVM to call Garbage Collector method
        System.gc();
        System.out.println("Main Completes");
    }

    // Here overriding finalize method
    public void finalize()
    {
        System.out.println("finalize method overriden");
        System.out.println(10 / 0);
```

```
    }
}
```

Sortie :

```
méthode de finalisation remplacée

Achèvement principal
```

Le **point clé** est donc le suivant: si Garbage Collector appelle la méthode finalize, alors que l'exécution de la méthode finalize génère une exception non vérifiée, la JVM **ignore** cette exception et le reste du programme se poursuit normalement. Dans ce cas, la fin du programme est donc **normale** et non anormale

Les points importants:

- Il n'y a aucune garantie quant au moment où la finalisation est appelée. Il peut être appelé à tout moment après que l'objet ne soit référé nulle part (la cabine doit être récupérée).
- JVM n'ignore pas toutes les exceptions lors de l'exécution de la méthode de finalisation, mais elle ignore **uniquement les exceptions non vérifiées** . Si le bloc catch correspondant est là, la JVM ne l'ignorera pas et le bloc catch correspondant sera exécuté.
- System.gc () n'est qu'une demande à JVM d'exécuter le Garbage Collector. C'est à JVM d'appeler ou non Garbage Collector. Habituellement, VM appelle Garbage Collector lorsqu'il n'y a pas assez d'espace disponible dans la zone Heap ou lorsque la mémoire est faible.

Variables statiques en Java avec des exemples

Lorsqu'une variable est déclarée comme statique , une seule copie de la variable est créée et partagée entre tous les objets au niveau de la classe. Les variables statiques sont essentiellement des variables globales. Toutes les instances de la classe partagent la même variable statique.

Points importants pour les variables statiques:

- Nous pouvons créer des variables statiques au niveau de la classe uniquement. Voir ici
- le bloc statique et les variables statiques sont exécutés dans l'ordre où ils sont présents dans un programme.

Ci-dessous, le programme java pour démontrer que le bloc statique et les variables statiques sont exécutés dans l'ordre dans lequel ils sont présents dans un programme.

```
// Java program to demonstrate execution
// of static blocks and variables

class Test {

    // static variable
    static int a = m1();

    // static block
    static
    {
```

```
        System.out.println("Inside static block");
    }

    // static method
    static int m1()
    {
        System.out.println("from m1");
        return 20;
    }

    // static method(main !!)
    public static void main(String[] args)
    {
        System.out.println("Value of a : " + a);
        System.out.println("from main");
    }
}
```

Production:

```
de m1
À l'intérieur du bloc statique
Valeur de: 20
de la main
```