final, finally and finalize in Java

Difficulty Level : Easy Last Updated : 13 Jan, 2022

This is an important question concerning the interview point of view.

final keyword

final(lowercase) is a reserved keyword in java. We can't use it as an identifier as it is reserved. We can use this keyword with variables, methods and also with classes. The final keyword in java has different meaning depending upon it is applied to variable, class or method.

1. **final with Variables :** The value of variable cannot be changed once initialized.

```java
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++;
    }
}
```

   If we declare any variable as final, we can't modify its contents since it is final, and if we modify it then we get Compile Time Error.

2. **final with Class :** The class cannot be subclassed. Whenever we declare any class as final, it means that we can't extend that class or that class **can't be extended** or we can't make subclass of that class.

```
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 with Method :** The method cannot be overridden by a subclass. Whenever we declare any method as final, then it means that we can’t override that method.

```
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() {}
}
```

Note : If a class is declared as final then **by default** all of the methods present in that class are automatically final but **variables are not**.

```
// 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);
    }
}
```

Output:

```
36
```

finally keyword

Just as final is a reserved keyword, so in same way finally is also a reserved keyword in java i.e, we can't use it as an identifier. The finally keyword is used in association with a try/catch block and guarantees that a section of code will be executed, even if an exception is thrown. The finally block will be executed after the try and catch blocks, but before control transfers back to its origin.

```
// 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();
    }
}
```

Output:

```
inside A
A's finally
Exception caught
inside B
B's finally
```

There are various cases when finally can be used. There are discussed below:

1. **Case 1 : Exceptions do not occur in the program**

```
// Java program to illustrate finally in
// Case where exceptions do not
// occur in the program
```

```java
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");
        }
    }
}
```

Output:

```
In try block
Executes whether exception occurs or not
```

Here above exception not occurs but still finally block executes since finally is meant to execute whether an exception occurs or not.

Flow of Above Program: First it starts from the main method and then goes to try block and in try since no exception occurs so flow doesn’t goes to catch block hence flow goes directly from try to finally block.

2. **Case 2 : Exception occurs and corresponding catch block matches**

```java
// 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 doesn't come here
            System.out.println("Flow doesn'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'
        }
    }
}
```

Output:

```
In try block
In catch block
Dividing by zero but caught
Executes whether an exception occurs or not
```

Here, the above exception occurs and corresponding catch block found but still finally block executes since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found or not.
Flow of Above Program: First, starts from the main method and then goes to try block and in try an Arithmetic exception occurs and the corresponding catch block is also available so flow goes to catch block. After that flow doesn't go to try block again since once an exception occurs in try block then flow **doesn't** come back again to try block. After that finally, execute since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found or not.

3. **Case 3 : Exception occurs and corresponding catch block not found/match**

```
// 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'
        }
    }
}
```

Output:

```
In try block
Executes whether an exception occurs or not
Exception in thread "main":java.lang.ArithmeticException:
/ by zero followed by stack trace.
```

Here above exception occurs and corresponding catch block not found/match but still finally block executes since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found/match or not.

Flow of Above Program: First starts from main method and then goes to try block and in try an Arithmetic exception occurs and corresponding catch block is **not** available so flow **doesn't** goes to catch block. After that flow doesn't go to try block again since once an exception occurs in try block then flow **doesn't** come back again to try block. After that finally, execute since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found/match or not.

Application of finally block: So basically the use of finally block is **resource deallocation**. Means all the resources such as Network Connections, Database Connections, which we opened in try block are needed to be closed so that we won't lose our resources as opened. So those resources are needed to be closed in finally block.

```java
// 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");
            }
        }
    }
}
```

Output:

```
Entered try statement
PrintWriter not open
```

Note: The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered.

How jdk 1.7 makes usage of finally block as optional?

Until jdk 1.6 finally block is like a hero i.e, it is recommended to use it for resource deallocation but from jdk 1.7 onwards finally, block is now optional(however you can use it). Since the resources which we opened in the try block will automatically gets deallocated/closed when the flow of program reaches to the end of the try block.
This concept of automatic resource deallocation without using finally block is known as **try-with-resources Statement**.

finalize method

It is a **method** that the Garbage Collector always calls just **before** the deletion/destroying the object which is eligible for Garbage Collection, so as to perform **clean-up activity**. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation. Remember it is **not** a reserved keyword.
Once the finalize method completes immediately Garbage Collector destroy that object. finalize method is present in Object class and its syntax is:

```
protected void finalize throws Throwable{}
```

Since Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class hence Garbage Collector can call finalize method on **any java object**
Now, the finalize method which is present in the Object class, has an empty implementation, in our class clean-up activities are there, then we have to **override** this method to define our own clean-up activities.

Cases related to finalize method:

1. **Case 1 :** The object which is eligible for Garbage Collection, that object's corresponding class finalize method is going to be executed

```
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");
    }
}
```

Output:

```
Main Completes
```

Note : Here above output came only **Main Completes** and **not** “finalize method overriden” because Garbage Collector call finalize method on that class object which is eligible for Garbage collection. Here above we have done->
s = null and ‘s’ is the object of String class, so String class finalize method is going to be called and not our class(i.e, Hello class). So we modify our code like->

```
Hello s = new Hello();
s = null;
```

Now our class i.e, Hello class finalize method is called. **Output**:

```
finalize method overriden
Main Completes
```

So basically, Garbage Collector calls finalize method on that class object which is eligible for Garbage collection.So if String object is eligible for Garbage Collection then **String** class finalize method is going to be called and **not the Hello class** finalize method.

2. **Case 2 :** We can call finalize method Explicitly then it will be executed just like normal method call but object won’t be deleted/destroyed

```java
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");
    }
}
```

Output:

```
finalize method overriden
//call by programmer but object won't gets destroyed.
finalize method overriden
//call by programmer but object won't gets destroyed.
Main Completes
finalize method overriden
//call by Garbage Collector just before destroying the object.
```

Note : As finalize is a method and not a reserved keyword, so we can call finalize method **Explicitly**, then it will be executed just like normal method call but object won't be deleted/destroyed.

3. **Case 3 :**
   - **Part a)** If programmer calls finalize method, while executing finalize method some unchecked exception rises.

Java

```
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);
    }
}
```

Output:

```
exception in thread "main" java.lang.ArithmeticException:
/ by zero followed by stack trace.
```

So **key point** is : If programmer calls finalize method, while executing finalize method some unchecked exception rises, then JVM terminates the program abnormally by rising exception. So in this case the program termination is **Abnormal**.

- **part b)** If garbage Collector calls finalize method, while executing finalize method some unchecked exception rises.

Java

```
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);
        }
    }
```

Output:

```
finalize method overriden
Main Completes
```

So **key point** is : If Garbage Collector calls finalize method, while executing finalize method some unchecked exception rises then JVM **ignores** that exception and rest of program will be continued normally. So in this case the program termination is **Normal** and not abnormal.

Important points:

- There is no guarantee about the time when finalize is called. It may be called any time after the object is not being referred anywhere (cab be garbage collected).
- JVM does not ignore all exceptions while executing finalize method, but it ignores **only** Unchecked exceptions. If the corresponding catch block is there then JVM won't ignore and corresponding catch block will be executed.
- System.gc() is just a request to JVM to execute the Garbage Collector. It's up-to JVM to call Garbage Collector or not.Usually JVM calls Garbage Collector when there is not enough space available in the Heap area or when the memory is low.
  This article is contributed by **Rajat Rawat and Manav**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

  Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.