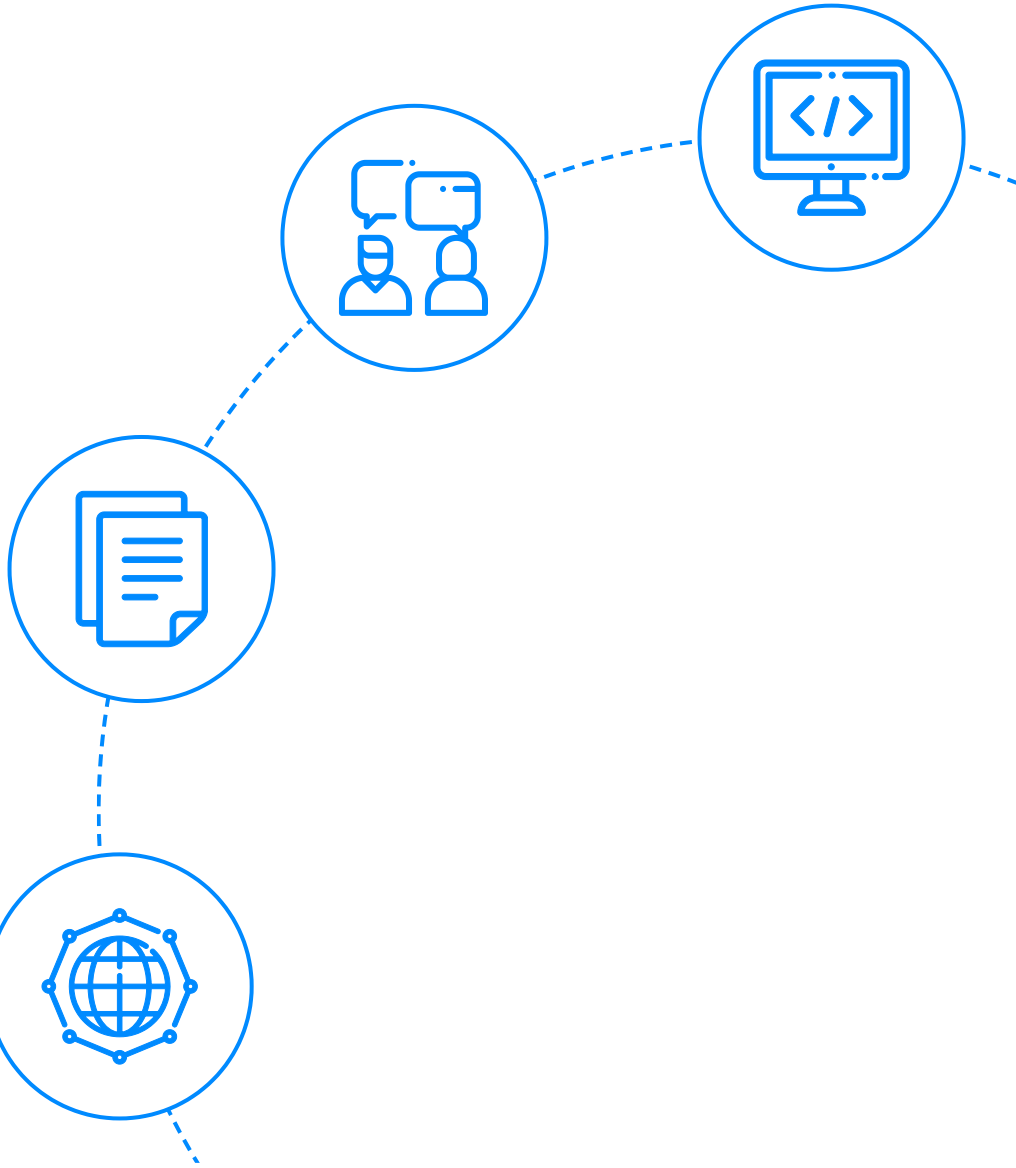
InterviewBit

Exception Handling Interview Questions



To view the live version of the page, click here.

© Copyright by Interviewbit

Contents

Exception Handling Interview Questions for Freshers

Exception Handling Interview Questions for Experienced

InterviewBit

© Copyright by Interviewbit

Exception Handling Interview Questions for Experienced

(.....Continued)

19. What does JVM do when an exception occurs in a program?
20. What happens when an exception is thrown by the main method?
21. Is it possible to throw checked exceptions from a static block?
22. What happens to the exception object after exception handling is complete?
23. What are different scenarios where “Exception in thread main” types of error could occur?
24. Under what circumstances should we subclass an Exception?
25. What happens when you run the below program?
26. Does the finally block always get executed in the Java program?
27. Why it is always recommended to keep the clean-up activities like closing the I/O resources or DB connections inside a finally block?
28. Are we allowed to use only try blocks without a catch and finally blocks?
29. What happens when the below program is run?
30. Is it possible to throw an Exception inside a Lambda Expression’s body?
31. What are the rules we should follow when overriding a method throwing an Exception?
32. What are some of the best practices to be followed while dealing with Java Exception Handling?

© Copyright by Interviewbit

Let's get Started

Introduction

An Exception refers to abnormal behaviour of an application that occurs at the time of execution that could lead to the termination of that application if not handled. Exceptions could include sudden network errors, database connection errors, errors due to non-existent or corrupt files, logical errors that were not handled by the developers and many more. **Java** provides a great way of handling Exceptions that results in a robust application that does not fail in case of abnormalities. Due to its importance, exception handling has become a very important and favourite topic amongst interviewers. Every software developer should know how to handle unexpected errors while developing an application.

In this article, we will go through the commonly asked Exception-Handling Interview Questions for both freshers and experienced software developers:

- Exception Handling Interview Questions for Freshers
- Exception Handling Interview Questions for Experienced
- MCQ Questions on Exception Handling in Java

Exception Handling Interview Questions for Freshers

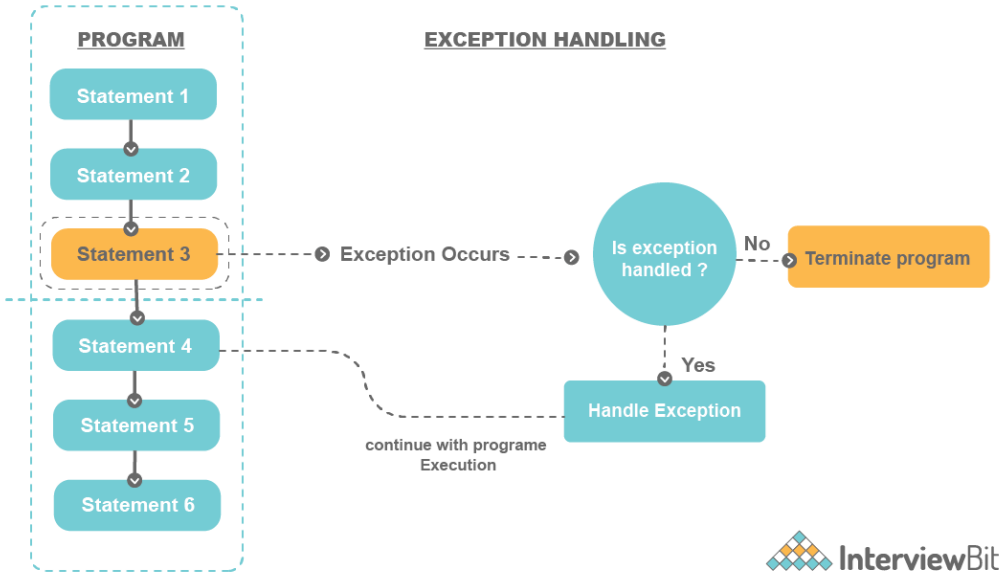
1. What is an exception in Java?

© Copyright by Interviewbit

An exception is an abnormal event that disrupts the flow of normal program execution that unless handled could lead to the termination of that program. In real-world scenarios, a program could run into an exception if it tries to access a corrupted/non-existent file or if a network error happens or if the JVM runs out of memory, or if the code is trying to access an index of an array which does not exist and so on.

2. What is exception handling in Java and what are the advantages of exception handling?

Exception Handling is the technique of handling unexpected failures that could occur in a program so that the program does not terminate and normal execution flow is maintained. Consider an example program where we have 6 statement blocks as shown in the below image. Statements 1 and 2 execute successfully. While executing the statement 3 block, an unexpected error occurred. Now the Java program checks if there is any exception-handling mechanism present. If an exception handling block is present, then the exception is handled gracefully and the program continues with the execution of the statement 4 block. If not, the program gets terminated.



The following are some of the A**dvantages of using Exception Handling in Java**:

© Copyright by Interviewbit

1. The most important advantage of having an exception-handling technique is that it avoids abnormal program termination and the normal flow of the program is maintained.
2. Provides flexibility to the programmers to define/handle what should occur in cases of failures/errors thereby making the applications more robust and immune to unexpected scenarios.
3. Provides stable user experience in cases of failures by providing means to let the user know what made the program fail.

3. How are exceptions handled in Java?

In Java, exceptions could be handled in the following ways:

1. **try-catch block:** The try section holds the code that needs to be normally executed and it monitors for any possible exception that could occur. The catch block "catches" the exception thrown by the try block. It could consist of logic to handle failure scenario or the catch block could simply rethrow the exception by using the "throw" keyword.
2. **finally block:** Regardless of whether an exception has occurred or not, if we need to execute any logic, then we place it in the final block that is usually associated with the try-catch block or just with the try block. The final block is not executed when `System.exit(0)` is present in either the try or catch block.

Consider an example where we need to get the value of the score from a file called "resultFile". When we access the file, the file could exist and we get the score and everything happens normally. But there could be cases where the file was accidentally deleted. In this case, when the program tries to access that file, then it throws `FileNotFoundException`. There are 2 ways to handle this exception (Depending on the requirements, we need to choose what way is the most appropriate for us):-

- **Case 1:** Throw `IllegalArgumentException` stating the file does not exist as shown in the logic below.

© Copyright by Interviewbit

```
public int getResultScore(String resultFile) {
    try {
        Scanner fileContents = new Scanner(new File(resultFile));
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException fileNotFound) {
        // handle exception by throwing a new exception
        throw new IllegalArgumentException("Result file does not exist");
    }
}
```

- **Case 2:** Return the score as 0 and log the error that file doesn't exist as shown in the logic below.

```
public int getResultScore(String resultFile) {
    try {
        Scanner fileContents = new Scanner(new File(resultFile));
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException fileNotFound) {
        // handle exception by returning 0 and logging error
        logger.error("Result file does not exist");
        return 0;
    }
}
```

Finally, irrespective of whether the code running normally or not, we would want to close the resources. This could run in the finally block as shown below:

```
public int getResultScore(String resultFile) {
    Scanner fileContents;
    try {
        fileContents = new Scanner(new File(resultFile));
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException fileNotFound) {
        // handle exception by returning 0 and logging error
        logger.error("Result file does not exist");
        return 0;
    } finally {
        if (fileContents != null) {
            fileContents.close();
        }
    }
}
```


© Copyright by Interviewbit

> **Note:**
>
> As of Java 7, the new feature “try-with-resources” helps to autoclose the resources that extend the “AutoCloseable” interface. The close() method will be called when it exits the try-with-resources block. For example, the above code which has a close() method call could be simplified as shown below:

```
public int getResultScore(String resultFile) {
    try (Scanner fileContents = new Scanner(new File(resultFile))) {
      return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException e ) {
        // handle exception by returning 0 and logging error
        logger.error("Result file does not exist");
        return 0;
    }
}
```

- Sometimes, the program could throw more than 1 exception. In this case, Java supports the usage of multiple catch blocks. Our example could also encounter a `NumberFormatException` exception while parsing integer numbers and this could be handled as shown below:

```
public int getPlayerScore(String playerFile) {
    try (Scanner fileContents = new Scanner(new File(resultFile))) {
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException e ) {
        // If an exception occurs while opening the file
        logger.error("Result file does not exist");
        return 0;
    } catch (NumberFormatException e) {
        // If an exception occurs during parsing the contents to integer format
        logger.error("Couldn't format number", e);
        return 0;
    }
}
```


© Copyright by Interviewbit

- Multiple catch blocks are useful if we have different exception-handling logic for different types of exceptions. If our logic of handling the exception is the same (as in the above scenario), then Java7 introduced the feature to union the catch blocks - handle multiple exceptions in the same catch block. The above could now be simplified to:

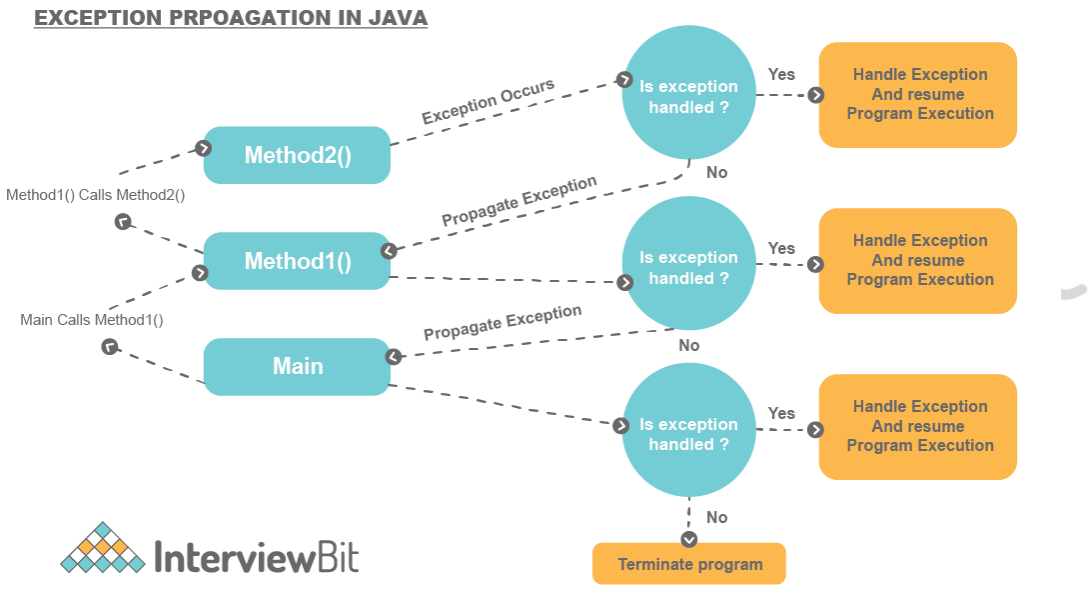
```
public int getPlayerScore(String playerFile) {
  try (Scanner fileContents = new Scanner(new File(resultFile))) {
      return Integer.parseInt(contents.nextLine());
  } catch (FileNotFoundException | NumberFormatException e ) {
      //e.getMessage() prints the detailed message
      logger.error("Score couldn't be loaded - " + e.getMessage());
      return 0;
  }
}
```

4. What is exception propagation in Java?

Exception propagation is a process where the compiler makes sure that the exception is handled if it is not handled where it occurs. If an exception is not caught where it occurred, then the exception goes down the call stack of the preceding method and if it is still not caught there, the exception propagates further down to the previous method. If the exception is not handled anywhere in between, the process continues until the exception reaches the bottom of the call stack. If the exception is still not handled in the last method, i.e, the main method, then the program gets terminated. Consider an example -

We have a Java program that has a main() method that calls method1() and this method, in turn, calls another method2(). If an exception occurs in method2() and is not handled, then it is propagated to method1(). If method1() has an exception handling mechanism, then the propagation of exception stops and the exception gets handled. The same has been represented in the image below:

© Copyright by Interviewbit



5. What are the important methods defined in Java's Exception Class?

The throwable class is the base class of Exception. Exception and its subclasses do not provide specific methods. All the methods that could be used by Exception are defined in the Throwable class. Some of the most important methods are as follows:

© Copyright by Interviewbit

1. **String getMessage():** This method returns the Throwable message of type String. The message could be given at the time of creating an exception by using the constructor.
2. **String getLocalizedMessage():** This method can be overridden by the Exception and its subclasses to give locale-specific messages to the calling program. The implementation of this method in the Throwable class by default uses the getMessage() that returns the exception message. Hence, to use this method, it should be overridden to avoid this default behaviour.
3. **synchronized Throwable getCause():** This returns the exception cause if it is known. If the cause is not known, it returns null. The cause is returned if it was provided via the constructors requiring `Throwable` or if it was set after the creation of exception by using the `initCause(Throwable)` method. Exception and its subclasses may override this method to return a cause set by different means.
4. **String toString():** This returns the Throwable information in String format consisting of the Throwable class name and localized message.
5. **void printStackTrace():** As the name indicates, this prints the stack trace data to the standard error stream. By default, it prints on the console. But, there are overloaded versions of this method where we can pass either `PrintWriter` or `PrintStream` as an argument to write stack trace data to a specific file or stream.

6. What are runtime exceptions in Java?

Runtime exceptions are those exceptions that occur at the run time of the program execution. These exceptions are not noticed by the compiler at the compile time and hence the program successfully gets compiled. Therefore, they are also called unchecked exceptions. All subclasses of the `java.lang.RunTimeException` class and `java.lang.Error` class belongs to runtime exceptions. Examples of runtime exceptions include NullPointerException, NumberFormatException, ArrayIndexOutOfBoundException, StackOverflowError, ClassCastException, ArithmeticException, ConcurrentModificationException, etc.

© Copyright by Interviewbit

7. What is the difference between the throw and throws keywords in Java?

The `throw` keyword allows a programmer to throw an exception object to interrupt normal program flow. The exception object is handed over to the runtime to handle it. For example, if we want to signify the status of a task is outdated, we can create an `OutdatedTaskException` that extends the `Exception` class and we can throw this exception object as shown below:

```
if (task.getStatus().equals("outdated")) {
    throw new OutdatedTaskException("Task is outdated");
}
```

The `throws` keyword in Java is used along with the method signature to specify exceptions that the method could throw during execution. For example, a method could throw `NullPointerException` or `FileNotFoundException` and we can specify that in the method signature as shown below:

```
public void someMethod() throws NullPointerException, FileNotFoundException {
    // do something
}
```

8. How do you handle checked exceptions?

Checked Exceptions can be handled by either using a try-catch block or by using the throws clause in the method declaration. If these exceptions are not handled properly, then the program would fail to compile.

9. Differentiate between Checked Exception and Unchecked Exceptions in Java.

The followings programs are the differences between Checked and Unchecked Exceptions:

© Copyright by Interviewbit

Checked Exceptions	Unchecked Exceptions
These exceptions are checked and handled at the time of compilation.	These Exceptions are not checked and handled at compilation time. They occur during the runtime of the program and can not be anticipated by the compiler.
These exceptions are a direct subclass of the `Exception` class.	These exceptions are a direct subclass of the `RuntimeException` class.
Program would give compilation error if checked exceptions are not handled.	Program compiles successfully, but the exception could occur at runtime due to logical errors in the program.
Checked Exceptions require handling using a try-catch block or at least the method should use the `throws` keyword to let the calling method know that a checked exception could be thrown from this method.	Unchecked Exceptions do not require try-catch or `throws` handling. These exceptions could occur due to mistakes in programming logic.
Examples: IOException, FileNotFoundException, DataAccessException, InterruptedException, etc.	Examples: ArithmeticException, ArrayIndexOutOfBoundException, NullPointerException, InvalidClassException, etc.

© Copyright by Interviewbit

10. Can you catch and handle Multiple Exceptions in Java?

There are three ways to handle multiple exceptions in Java:

- Since Exception is the base class for all exception types, make use of a catch block that catches the Exception class-

```
try {
    // do something
} catch (Exception exception) {
    // handle exception
}
```

However, it is recommended to use accurate Exception handlers instead of generic ones. This is because having broad exception handlers could make the code error-prone by catching exceptions that were not anticipated in the software design and could result in unexpected behavior.

- From Java 7 onwards, it is now possible to implement multiple catch blocks as shown below -

```
    try {
       // do something
    } catch (FileNotFoundException fileNotFoundException) {
        // handle FileNotFoundException
    } catch (EOFException eofException) {
        // handle EOFException
    }
```

When we are using multiple catch blocks, we need to ensure that in a case where the exceptions have an inheritance relationship, the child exception type should be the first and the parent type later to avoid a compilation error.

- Java 7 also began to provide the usage of multi-catch blocks for reducing duplication of code if the exception handling logic was similar for different exception types. The syntax of the multi-catch block is as shown below-

© Copyright by Interviewbit

```
try {
    // ...
} catch (FileNotFoundException | EOFException exception) {
    // ...
}
```

11. What is a stack trace and how is it related to an Exception?

A stack trace is information consisting of names of classes and methods that were invoked right from the start of program execution to the point where an exception occurred. This is useful to debug where exactly the exception occurred and due to what reasons. Consider an example stack trace in Java,

```
Exception in thread "main" java.lang.NullPointerException
        at com.example.demoProject.Book.getBookTitle(Book.java:26)
        at com.example.demoProject.Author.getBookTitlesOfAuthor(Author.java:15)
        at com.example.demoProject.DemoClass.main(DemoClass.java:14)
```

To determine where an exception has occurred, we need to check for the beginning of the trace that has a list of “at …”. In the given example, the exception has occurred at Line Number 26 of the `Book.java` class in the `getBookTitle()` method. We can look at this stack trace and go to the method and the line number mentioned in the trace and debug for what could have caused the NullPointerException. Furthermore, we can get to know that the `getBookTitle()` method in the `Book.java` class was called by the `getBookTitlesOfAuthor()` method in the `Author.java` class in Line Number 15 of the file and this in turn was called by the `main()` method of the `DemoClass.java` file in Line Number 14.

12. What is Exception Chaining?

© Copyright by Interviewbit

Exception Chaining happens when one exception is thrown due to another exception. This helps developers to identify under what situation an Exception was thrown that in turn caused another Exception in the program. For example, we have a method that reads two numbers and then divides them. The method throws `ArithmeticException` when we divide a number by zero. While retrieving the denominator number from the array, there might have been an `IOException` that prompted to return of the number as 0 that resulted in `ArithmeticException`. The original root cause in this scenario was the `IOException`. The method caller would not know this case and they assume the exception was due to dividing a number by 0. Chained Exceptions are very useful in such cases. This was introduced in JDK 1.4.

13. Can we have statements between try, catch and finally blocks?

No. This is because they form a single unit.

14. How are the keywords final, finally and finalize different from each other?

- **final keyword:** By using this keyword,
  - we can declare a variable as final (meaning, variable value cannot be changed).
  - we can declare a method as final (meaning, that method cannot be overridden).
  - we can declare a class as final (meaning, that class cannot be extended).
- **finally keyword:** This is used in conjunction with the try-catch block or the try block where we want to run some logic whether or not an exception has occurred.
- **finalize keyword:** This is a method called by the Garbage Collector just before destroying the objects no longer needed in the program.

15. What is the output of this below program?

© Copyright by Interviewbit

```
public class TestClass
{
    public static void main(String[] args)
    {
        int a = 30;
        int b = 40;
        int c = 10;
        int expression = (a * b)/(a - b + c);
        System.out.println("Result: " +expression);
      }
}
```

When the expression is evaluated, the denominator becomes zero. Hence Java would throw ArithmeticException when it's executed. The exception logs would be:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestClass.main(TestClass.java:8)
```

16. What is the difference between ClassNotFoundException and NoClassDefFoundError?

Both these exceptions occur when ClassLoader or JVM is not able to find classes while loading during run-time. However, the difference between these 2 are:

- **ClassNotFoundException:** This exception occurs when we try to load a class that is not found in the classpath at runtime by making use of the `loadClass()` or `Class.forName()` methods.
- **NoClassDefFoundError:** This exception occurs when a class was present at compile-time but was not found at runtime.

17. What do you understand by an unreachable catch block error?

This error is thrown by the compiler when we have multiple catch blocks and keep parent classes first and subclasses later. The catch blocks should follow the order of the most specific ones at the top to the most general ones at the bottom. If this is not followed, an unreachable catch block error is thrown during compile time.

© Copyright by Interviewbit

Exception Handling Interview Questions for Experienced

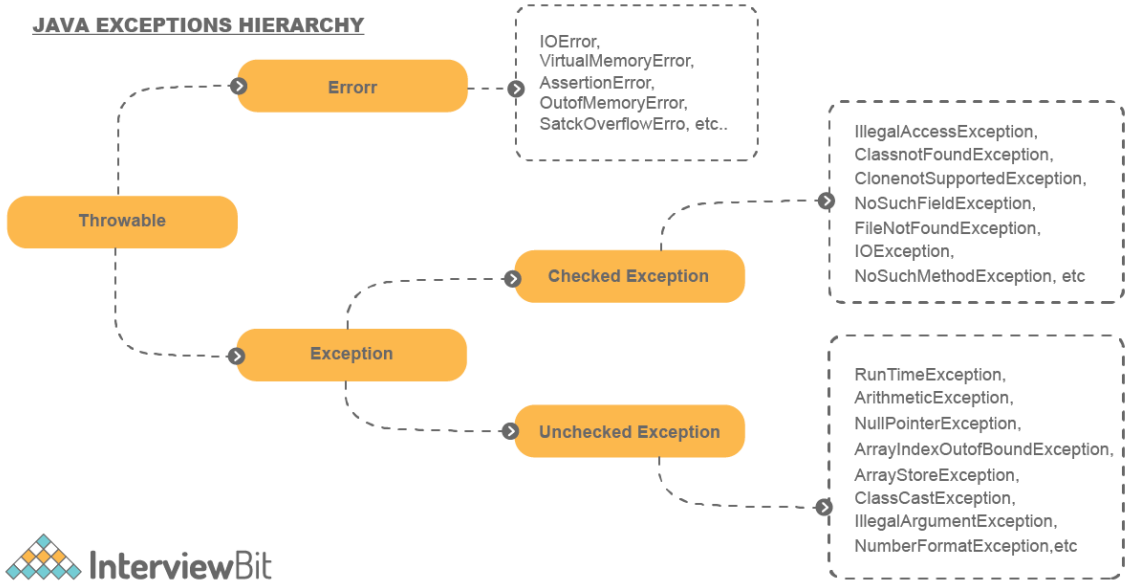
18. Explain Java Exception Hierarchy.

Exceptions in Java are hierarchical and follow inheritance to categorize different kinds of exceptions. The parent class of all exceptions and errors is Throwable. This class has 2 child classes - Error and Exception.

1. Errors are those abnormal failures that are not in the scope of recovery for the application. It is not possible to anticipate errors or recover from them. Errors could be due to failures in hardware, out-of-memory, JVM crash, etc.
2. Exception is divided into 2 categories - Checked Exception and Unchecked Exception.
    - Checked Exceptions are those that can be anticipated in the program at the time of compilation and we should try to handle this otherwise it leads to a compilation error. IllegalAccessException, ClassNotFoundException, and FileNotFoundException are some of the types of checked exceptions. Exception is the parent class of all checked exceptions.
    - Unchecked exceptions are those exceptions that occur during runtime and are not possible to identify at the time of compilation. These exceptions are caused due to mistakes in application programming - for example, we try to access an element from an array index that is out of bounds from the length of the array, while trying to run operations on null objects and so on. RuntimeException, ArrayIndexOutOfBoundException, NullPointerException, ClassCastException, NumberFormatException, IllegalArgumentException, etc are some of the types of unchecked exceptions. The parent class of all runtime/unchecked exceptions is RuntimeException.

The exception hierarchy in Java has been depicted in the image below:

© Copyright by Interviewbit



19. What does JVM do when an exception occurs in a program?

When there is an exception inside a method, the method creates and passes (throws) the Exception object to the JVM. This exception object has information regarding the type of exception and the program state when this error happens. JVM is then responsible for finding if there are any suitable exception handlers to handle this exception by going backwards in the call stack until it finds the right handler. If a matching exception handler is not found anywhere in the call stack, then the JVM terminates the program abruptly and displays the stack trace of the exception.

20. What happens when an exception is thrown by the main method?

When there is an exception thrown by the main() method if the exception is not handled, then the program is terminated by the Java Runtime, and the exception message along with the stack trace is printed in the system console.

21. Is it possible to throw checked exceptions from a static block?

© Copyright by Interviewbit

We cannot throw a check exception from a static block. However, we can have try-catch logic that handles the exception within the scope of that static block without rethrowing the exception using the throw keyword. The exceptions cannot be propagated from static blocks because static blocks are invoked at the compiled time only once and no method invokes these blocks.

22. What happens to the exception object after exception handling is complete?

The exception object will be garbage collected.

23. What are different scenarios where “Exception in thread main” types of error could occur?

The following are some of the scenarios causing an exception in the main thread:

1. **Exception in thread main java.lang.UnsupportedClassVersionError:** This occurs when the Java class is compiled from one JDK version and we run the class in another JDK version.
2. **Exception in thread main java.lang.NoSuchMethodError: main:** This occurs when we try to run a class that has no main method.
3. **Exception in thread main java.lang.NoClassDefFoundError:** This occurs if a ClassLoader is unable to find that class in the classpath at the time of loading it.
4. **Exception in thread “main” java.lang.ArithmeticException:** This is an example exception. Here, in the logic, ArithmeticException was thrown from the main method due to some erroneous logic in the main or due to the methods called from the main that threw this exception but was not handled there.

24. Under what circumstances should we subclass an Exception?

© Copyright by Interviewbit

When there are exceptions that are to be thrown by the application, but the type is not represented by any of the existing Exception types in Java, then we can provide more detailed information by creating a custom exception. The creation of custom exceptions depends on the business logic. While creating a custom exception, it is recommended to inherit from the most specific Exception class that relates to the exception we want to throw. If there are no such classes available, then we can choose the Exception class as the parent for this new custom exception. Consider an example -
We want to access a file and return the first line of the file content.

```
try (Scanner file = new Scanner(new File(fileName))) {
    if (file.hasNextLine()) return file.nextLine();
} catch(FileNotFoundException fnfeException) {
    // Handle exception if the file is not found
}
```

In the above case, when there is an exception while reading the file - we do not know if it was caused due to absence of the file or if the name of the file provided was wrong. To narrow this down, we will create a custom exception called `InvalidFileNameException` by extending the Exception class as shown below:

```
public class InvalidFileNameException extends Exception {
    public InvalidFileNameException(String exceptionMessage) {
        super(exceptionMessage);
    }
}
```

To create a custom exception, we should also provide a constructor that makes a call to the parent class constructor by using the super keyword. Our custom exception is ready. The code logic now becomes:

© Copyright by Interviewbit

```
try (Scanner file = new Scanner(new File(fileName))) {
    if (file.hasNextLine()) return file.nextLine();
} catch(FileNotFoundException fnfeException) {
    // check if the file name is valid or not
    if (!ifFileNameValid(fileName)) {
        throw new InvalidFileNameException("Filename : " + fileName + " is invalid");
    }
    // Handle exception if the filename was valid and if the file is not found
}
```

25. What happens when you run the below program?

```
import java.io.IOException;
import java.io.FileNotFoundException;

import javax.xml.bind.JAXBException;

public class TestExceptionDemo {

    public static void main(String[] args) {
            try {
                demoException();
            } catch (IOException e) {
                System.out.println(e.getMessage());
            } catch (FileNotFoundException e) {
                System.out.println(e.getMessage());
            } catch (JAXBException e) {
                System.out.println(e.getMessage());
            }
    }

    public static void demoException() throws IOException, FileNotFoundException, JAXBE

    }
}
```


© Copyright by Interviewbit

We know that the FileNotFoundException is the subclass of IOException - By the rule of multiple catch block, we need to have the most specific exception at the top and the most generic exception at the bottom. If there are no parent-child relationships between the exceptions, they can be placed anywhere. In this example, JAXBException is not related to IOException and FileNotFoundException exceptions. Hence, it can be placed anywhere in the multiple catch block structure. However, IOException being the parent class is mentioned at the beginning and that is followed by FileNotFoundException. This results in an error - Unreachable catch block for FileNotFoundException. It is already handled by the catch block for IOException. To fix this issue, depending on the business requirements, we can either remove the FileNotFoundException from the throws list and catch block OR we can rearrange the catch blocks as shown below:

```
public static void main(String[] args) {
    try {
        demoException();
    } catch (FileNotFoundException e) {
        System.out.println(e.getMessage());
    } catch (IOException e) {
        System.out.println(e.getMessage());
    } catch (JAXBException e) {
        System.out.println(e.getMessage());
    }
}
```

26. Does the finally block always get executed in the Java program?

The finally blocks are meant to be executed whenever there is an exception or not in the try-catch blocks. However, there is one scenario where this block does not get executed. It is when we use `System.exit(0)` in either try or catch block. This is because the `System.exit(0)` terminates the running JVM.

27. Why it is always recommended to keep the clean-up activities like closing the I/O resources or DB connections inside a finally block?

© Copyright by Interviewbit

When there is no explicit logic to terminate the system by using `System.exit(0)`, finally block is always executed irrespective of whether the exception has occurred or not. By keeping these cleanup operations in the finally block, it is always ensured that the operations are executed and the resources are closed accordingly. However, to avoid exceptions in the finally block, we need to add a proper validation check for the existence of the resource and then attempt to clean up the resources accordingly.

28. Are we allowed to use only try blocks without a catch and finally blocks?

Before Java 7, we were not allowed to use only try blocks. The try block should have been followed by a catch or a finally block. But from Java 7 onwards, we can simply have a try block with a catch or finally blocks in the form of `try-with-resources` that takes parameters implementing the `AutoCloseable` interface. Note that, if the parameters do not implement the `AutoCloseable` interface, then it leads to an error.

29. What happens when the below program is run?

```
public class TestExceptionDemo {

    public static void main(String[] args) {
        try{
            demoMethod();
        }catch(NullPointerException e){
            System.out.println(e.getMessage());
        }catch(Exception e){
            System.out.println(e.getMessage());
        }

        foobar();
    }

    public static void demoMethod(){

    }

    public static void foobar() throws NullPointerException{

    }
}
```


© Copyright by Interviewbit

The program compiles successfully and it runs and displays nothing as no logic in both methods could result in an exception.

If there are any exceptions in the `demoMethod()`, we can always catch the exception and handle it. If the method `foobar()` declares `NullPointerException` using throws clause, the method doesn't need to throw an exception. However, if there is a logic that results in performing operations on null objects inside the `foobar()` method, then `NullPointerException` will be thrown by the method to the `main()` method. Since there is no exception handling for this `foobar()` method, the program will terminate displaying the error message and stack trace.

30. Is it possible to throw an Exception inside a Lambda Expression's body?

Yes, it is possible. However, we need to note two points:

- If we are using a standard functional interface that is given by Java, then we can throw only unchecked exceptions. This is because the standard functional interfaces of Java do not have a "throws" clause defined in their signature. For example, we can throw `IllegalArgumentException` inside a function interface as shown below:

```
List<Integer> list = Arrays.asList(2,3,5,10,20);
list.forEach(i -> {
    if (i < 0) {
        throw new IllegalArgumentException("Negative numbers are not allowed.");
    }
    System.out.println(i);
});
```

- If we are using custom functional interfaces, then we can throw checked as well as unchecked exceptions. Custom functional interfaces can be defined by using the `@FunctionalInterface` keyword.

31. What are the rules we should follow when overriding a method throwing an Exception?

© Copyright by Interviewbit

The following are some of the rules that have to be followed while overriding method throwing Exception:

Rule 1: If the parent class method is not throwing any exceptions, then the overridden child class method should not throw checked exceptions. But it can throw an unchecked exception. Consider an example that demonstrates this- We have 2 classes - ParentDemo and ChildDemo where the latter is the subclass of the ParentDemo class. We have the `doThis()` the method that is overridden in the ChildDemo class. The overridden method may only throw an unchecked exception. The below example is valid since `IllegalArgumentException` is an unchecked exception.

```
class ParentDemo {
    void doThis() {
        // ...
    }
}

class ChildDemo extends ParentDemo {
    @Override
    void doThis() throws IllegalArgumentException {
        // ...
    }
}
```

Rule 2: If the parent class method is throwing one or more checked exceptions, then the overridden method in the child class can throw any unchecked exceptions or any exceptions that are the same as checked exceptions of the parent method or the subclasses of those checked exceptions. Consider the below example code:

© Copyright by Interviewbit

```
class ParentDemo {
    void doThis() throws IOException, ParseException {
        // ...
    }

    void doThat() throws IOException {
        // ...
    }
}

class ChildDemo extends ParentDemo {
    void doThis() throws IOException {
        // ...
    }

    void doThat() throws FileNotFoundException, EOFException {
        // ...
    }
}
```

In this example, the `doThis()` method throws few exceptions than the parent method and the `doThat()` method throws a greater number of exceptions, however, the scope of the exceptions is not greater than the parent exceptions. If we try to throw a checked exception that was not declared in the parent method or we throw an exception that has a broader scope, then it results in a compilation error. For example, the below code is not valid as the parent method throws a `FileNotFoundException` exception and the child method throws `IOException` which is broader in scope as it is the parent class of `FileNotFoundException` :

```
class ParentDemo {
    void doThis() throws FileNotFoundException {
        // ...
    }
}

class ChildDemo extends ParentDemo {
    void doThis() throws IOException {
        // Compilation error because IOException is of broader scope than FileNotFoundE
    }
}
```


© Copyright by Interviewbit

Rule 3: If the parent class method has a throws clause having unchecked exceptions, then the overriding child method can throw any number of unchecked exceptions even if they are not related to each other. Consider the below example. While the parent class doThis() method threw only 1 unchecked exception which is `IllegalArgumentException`, the child class could throw multiple unchecked exceptions as shown below:

```
class ParentDemo {
    void doThis() throws IllegalArgumentException {
        // ...
    }
}

class ChildDemo extends ParentDemo {
    void doThis() throws ArithmeticException, NumberFormatException, NullPointerExcepti
        // ...
    }
}
```

32. What are some of the best practices to be followed while dealing with Java Exception Handling?

The following are some of the best practices that have to be followed while developing Exception Handling in Java:

© Copyright by Interviewbit

- Use Specific Exception as much as possible for the sake of better messages and easier debugging.
- Throw Exceptions early in the program. This follows the Fail-Fast approach to development.
- Catch Exceptions late in the program, i.e, the caller has to handle the exceptions.
- If you are using Java 7 and above versions, make use of the feature of the try-with-resources block to close the resources properly. In this way, we do not have to remember to ensure that all resources have been closed in the finally block.
- Always log the exception messages with meaningful messages. This helps in easier debugging.
- Use multiple catches that follow the rule of most specific exceptions at the top and the most generic exceptions at the bottom.
- If the exception types provided by Java do not match your use case, make sure to use custom exceptions with meaningful error messages.
- While creating custom messages, the naming convention of ending the exception name with `Exception` has to be followed.
- Whenever your method throws any exception, document this using the Javadoc and describe that briefly using the @throws parameter.
- Since exceptions are costly, be responsible while throwing them. Only throw them if it makes sense.

Conclusion

Exception handling is a very important technique for handling abnormal errors or failures that could happen at run time. It makes the program more robust and immune to failure scenarios thereby providing a better user experience to the users. Hence, software developers need to know how to manage and handle exceptions. In this article, we have seen what are the most commonly asked interview questions for both freshers and experienced software developers.

© Copyright by Interviewbit

Useful Resources

- https://www.interviewbit.com/problems/exception-handling/
- https://www.interviewbit.com/problems/try-catch-block/
- https://www.scaler.com/topics/java/error-vs-exception-in-java/
- https://www.interviewbit.com/technical-interview-questions/

© Copyright by Interviewbit

Links to More Interview Questions

C Interview Questions	Php Interview Questions	C Sharp Interview Questions
Web Api Interview Questions	Hibernate Interview Questions	Node Js Interview Questions
Cpp Interview Questions	Oops Interview Questions	Devops Interview Questions
Machine Learning Interview Questions	Docker Interview Questions	Mysql Interview Questions
Css Interview Questions	Laravel Interview Questions	Asp Net Interview Questions
Django Interview Questions	Dot Net Interview Questions	Kubernetes Interview Questions
Operating System Interview Questions	React Native Interview Questions	Aws Interview Questions
Git Interview Questions	Java 8 Interview Questions	Mongodb Interview Questions
Dbms Interview Questions	Spring Boot Interview Questions	Power Bi Interview Questions
Pl Sql Interview Questions	Tableau Interview Questions	Linux Interview Questions
Ansible Interview Questions	Java Interview Questions	Jenkins Interview Questions

© Copyright by Interviewbit