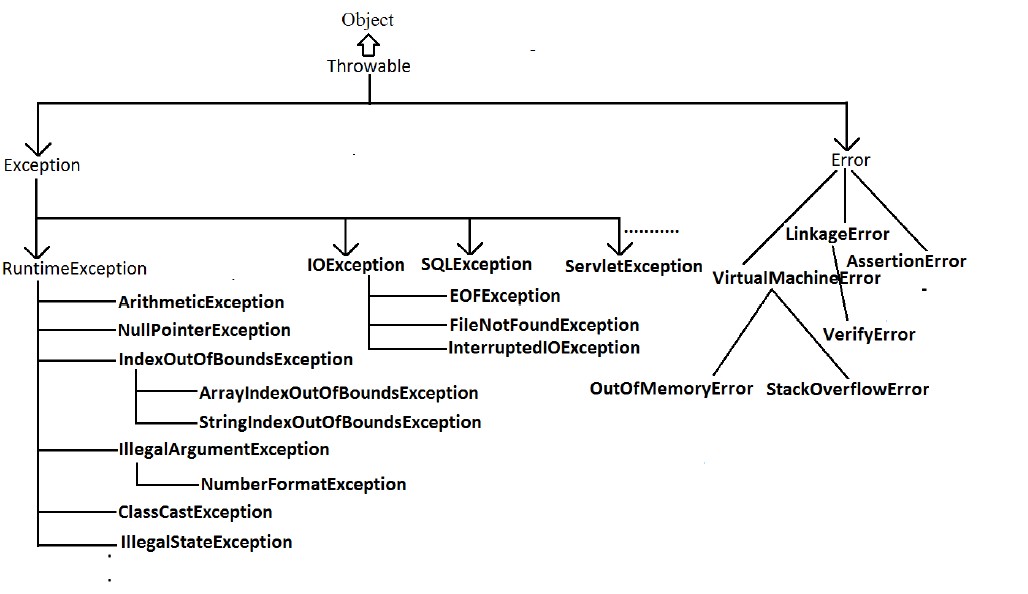
**Explain Java Exception Hierarchy.**

**What is Exception in Java?**

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn’t continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled. We will cover the handling part later in this same tutorial.

**When an exception can occur?**

Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

**Reasons for Exceptions**

There can be several reasons for an exception. For example, following situations can cause an exception – Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on.

**What are the Exception Handling Keywords in Java?**

**throw** – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. throw keyword is used to throw exception to the runtime to handle it.

**throws** – When we are throwing any exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it’s caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.

**try-catch** – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.

**finally** – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

**What are important methods of Java Exception Class?**

Some of the useful methods of Throwable class are;

**public String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it’s constructor.

**public String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.

**public synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.

**public String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.

**public void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

**What changes has been introduced in JDK7 related to Exception handling in Java ?**

If you are catching a lot of exceptions in a single try block, you will notice that catch block code looks very ugly and mostly consists of redundant code to log the error, keeping this in mind Java 7 one of the feature was multi-catch block where we can catch multiple exceptions in a single catch block. The catch block with this feature looks like below:

**catch(IOException | SQLException | Exception ex){**

**logger.error(ex);**

**throw new MyException(ex.getMessage());**

**}**

Most of the time, we use finally block just to close the resources and sometimes we forget to close them and get runtime exceptions when the resources are exhausted. These exceptions are hard to debug and we might need to look into each place where we are using that type of resource to make sure we are closing it. So java 7 one of the improvement was try-with-resources where we can create a resource in the try statement itself and use it inside the try-catch block. When the execution comes out of try-catch block, runtime environment automatically close these resources. Sample of try-catch block with this improvement is:

try (MyResource mr = new MyResource()) {

System.out.println("MyResource created in try-with-resources");

} catch (Exception e) {

e.printStackTrace();

}

**What is difference between Checked and Unchecked Exception in Java?**

1) Checked Exception is required to be handled by compile time while Unchecked Exception doesn't.

2) Checked Exception is direct sub-Class of Exception while Unchecked Exception are of RuntimeException.

3) CheckedException represent scenario with higher failure rate while UnCheckedException are mostly programming mistakes.

**Example of unchecked Exception in Java API**

Here are few examples of Unchecked Exception in Java library:

NullPointerException

ArrayIndexOutOfBound

IllegalArgumentException

IllegalStateException

**Summary:**

1. Both Checked and Unchecked Exception are handled using keyword try, catch and finally.

2. In terms of Functionality Checked and Unchecked Exception are same.

3. Checked Exception handling verified during compile time.

4. Unchecked Exception are mostly programming errors

5. JDK7 provides improved Exception handling code with catching multiple Exception in one catch block and reduce amount of boiler plate code required for exception handling in Java.

**What is difference between throw and throws keyword in Java?**

1) Java throw keyword is used to explicitly throw an exception.

Java throws keyword is used to declare an exception.

2) Checked exception cannot be propagated using throw only.

Checked exception can be propagated with throws.

3) Throw is followed by an instance.Throws is followed by class.

4) Throw is used within the method.Throws is used with the method signature.

5) You cannot throw multiple exceptions.You can declare multiple exceptions e.g.

public void method()throws IOException,SQLException.

**How to write custom exception in Java?**

We can extend Exception class or any of it’s subclasses to create our custom exception class. The custom exception class can have it’s own variables and methods that we can use to pass error codes or other exception related information to the exception handler.

A simple example of custom exception is shown below.

package com.rehan.exceptions;

import java.io.IOException;

public class MyException extends IOException {

private static final long serialVersionUID = 4664456874499611218L;

private String errorCode="Unknown\_Exception";

public MyException(String message, String errorCode){

super(message);

this.errorCode=errorCode;

}

public String getErrorCode(){

return this.errorCode;

}

}

**What is OutOfMemoryError in Java?**

OutOfMemoryError in Java is a subclass of java.lang.VirtualMachineError and it’s thrown by JVM when it ran out of heap memory. We can fix this error by providing more memory to run the java application through java options.

$>java MyProgram -Xms1024m -Xmx1024m -XX:PermSize=64M -XX:MaxPermSize=256m

**What are different scenarios causing “Exception in thread main”?**

Some of the common main thread exception scenarios are:

Exception in thread main java.lang.UnsupportedClassVersionError: This exception comes when your java class is compiled from another JDK version and you are trying to run it from another java version.

Exception in thread main java.lang.NoClassDefFoundError: There are two variants of this exception. The first one is where you provide the class full name with .class extension. The second scenario is when Class is not found.

Exception in thread main java.lang.NoSuchMethodError: main: This exception comes when you are trying to run a class that doesn’t have main method.

Exception in thread “main” java.lang.ArithmeticException: Whenever any exception is thrown from main method, it prints the exception is console. The first part explains that exception is thrown from main method, second part prints the exception class name and then after a colon, it prints the exception message.

**What happens when exception is thrown by main method?**

When exception is thrown by main() method, Java Runtime terminates the program and print the exception message and stack trace in system console.

### What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?

### The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

### If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

**Can we have an empty catch block?**

We can have an empty catch block but it’s the example of worst programming. We should never have empty catch block because if the exception is caught by that block, we will have no information about the exception and it wil be a nightmare to debug it. There should be at least a logging statement to log the exception details in console or log files.

**What is the reason of NoClassDefFoundError in Java?**

NoClassDefFoundError in Java comes when Java Virtual Machine is not able to find a particular class at runtime which was available at compile time. For example, if we have a method call from a class or accessing any static member of a Class and that class is not available during run-time then JVM will throw NoClassDefFoundError. It’s important to understand that this is different than ClassNotFoundException which comes while trying to load a class at run-time only and the name was provided during runtime, not at compile-time. Many Java developer mingles this two Error and gets confused.

In short, NoClassDefFoundError will come if a class was present during compile time but not available in java classpath during runtime. Normally you will see below line in log when you get NoClassDefFoundError:

**The difference between java.lang.NoClassDefFoundError and ClassNotFoundException in Java.**

Many times we confused ourselves with java.lang.ClassNotFoundException and java.lang.NoClassDefFoundError, though both of them related to Java Classpath they are completely different to each other. ClassNotFoundException comes when JVM tries to the load a class at runtime dynamically means you give the name of the class at runtime and then JVM tries to load it and if that class is not found in the classpath it throws java.lang.ClassNotFoundException. While in the case of NoClassDefFoundError the problematic class was present during Compile time and that's why the program successfully compiled but not available during runtime for any reason. NoClassDefFoundError is easier to solve than ClassNotFoundException in my opinion because here we know that Class was present at build time, but it totally depends upon environment if you are working in the J2EE environment than you can get NoClassDefFoundError even if the class is present because it may not be visible to the corresponding class loader.

**How to resolve java.lang.NoClassDefFoundError in Java.**

The obvious reason of NoClassDefFoundError is that a particular class is not available in Classpath, so we need to add that into Classpath or we need to check why it’s not available in Classpath if we are expecting it to be. There could be multiple reasons like:

1) The class is not available in Java Classpath.

2) You might be running your program using jar command and class was not defined in manifest file's ClassPath attribute.

3) Any start-up script is overriding Classpath environment variable.

4) Because NoClassDefFoundError is a subclass of java.lang.LinkageError it can also come if one of it dependency like native library may not available.

4) Check for java.lang.ExceptionInInitializerError in your log file. NoClassDefFoundError due to the failure of static initialization is quite common.

5) If you are working in J2EE environment than the visibility of Class among multiple Classloader can also cause java.lang.NoClassDefFoundError.

**What is Exception chaining in Java?**

Exception chaining is a popular exception handling concept in Java, where another exception is thrown in response of an exception and creating a chain of Exceptions. This technique mostly used to wrap a checked exception into an unchecked or RuntimeException. By the way if you are throwing new exception due to another exception then always include original exception so that handler code can access root cause by using methods like getCause() and initCause().

**Does code form finally executes if method returns before finally block or JVM exits ?**

This Java exception interview question can also be asked in code format, where given a code with System.exit() in try block and something in finally block. It’s worth knowing that, finally block in Java executes even when return keyword is used in try block. Only time they don’t execute is when you call JVM to exit by executing System.exit(0)from try block in Java.

**Can someone explain “ClassCastException” in Java?**

if you are trying to typecast an object of class A into an object of class B, and they aren't compatible, you get a class cast exception.

Let's think of a collection of classes.

class A {...}

class B extends A {...}

class C extends A {...}

You can cast any of these things to Object, because all Java classes inherit from Object.

You can cast either B or C to A, because they're both "kinds of" A

You can cast a reference to an A object to B only if the real object is a B.

You can't cast a B to a C even though they're both A's.

**Reason for UnSupportedClassVersionError in java**

This class was compiled with a JDK more recent than the one used for execution.

The easiest is to install a more recent JRE on the computer where you execute the program. If you think you installed a recent one, check the JAVA\_HOME and PATH environment variables.

Version 49 is java 1.5. That means the class was compiled with (or for) a JDK which is yet old. You probably tried to execute the class with JDK 1.4.

**Difference between final, finally, and finalize**:

**Final:**

 Final is the modifier applicable for class, methods and variables.

 If a class declared as the final then child class creation is not possible.

 If a method declared as the final then overriding of that method is not possible.

 If a variable declared as the final then reassignment is not possible.

**Finally:**

 It is the block always associated with try catch to maintain clean up code which

should be executed always irrespective of whether exception raised or not raised

and whether handled or not handled.

**Finalize:**

It is a method which should be called by garbage collector always just before

destroying an object to perform cleanup activities.

**Note:** To maintain clean up code faunally block is recommended over finalize() method because we can’t expert exact behavior of GC.

**ArrayIndexOutOfBoundsException:**

It is the child class of RuntimeException and hence it is unchecked. Raised

automatically by the JVM whenever we are trying to access array element with out

of range index.

**Example:**

class Test{

public static void main(String[] args){

int[] x=new int[10];

System.out.println(x[0]);//valid

System.out.println(x[100]);//AIOOBE

System.out.println(x[-100]);//AIOOBE

}

}

**NullPointerException:**

It is the child class of RuntimeException and hence it is unchecked. Raised

automatically by the JVM, whenever we are trying to call any method on null.

**Example:**

class Test{

public static void main(String[] args){

String s=null;

System.out.println(s.length());R.E: NullPointerException

}

}

**StackOverFlowError:**

It is the child class of Error and hence it is unchecked. Whenever we are trying to invoke recursive method call JVM will raise StackOverFloeError automatically.

**Example:**

class Test

{

public static void methodOne()

{

methodTwo();

}

public static void methodTwo()

{

methodOne();

}

public static void main(String[] args)

{

methodOne();

}

}

**Output:**

Run time error: StackOverFlowError

**NoClassDefFound:**

It is the child class of Error and hence it is unchecked. JVM will raise this error

automatically whenever it is unable to find required .class file.

**Example:** java Test

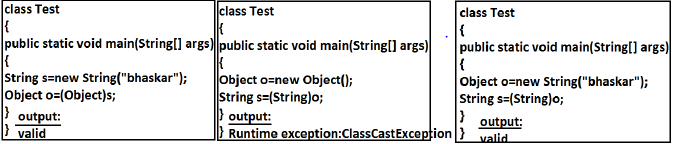
If Test.class is not available. Then we will get NoClassDefFound error.

**ClassCastException:**

It is the child class of RuntimeException and hence it is unchecked. Raised

automatically by the JVM whenever we are trying to typecast parent object to child type.

**Example:**



**ExceptionInInitializerError:**

 It is the child class of Error and it is unchecked. Raised automatically by the JVM, if any exception occurs while performing static variable initialization and static block execution.

**Example 1:**

class Test{

static int i=10/0;

}

**Output:**

Runtime exception:

Exception in thread "main" java.lang.ExceptionInInitializerError

**Example 2:**

class Test{

static {

String s=null;

System.out.println(s.length());

}}

**Output:**

Runtime exception: Exception in thread "main" java.lang.ExceptionInInitializerError

**IllegalArgumentException:**

It is the child class of RuntimeException and hence it is unchecked. Raised explicitly

by the programmer (or) by the API developer to indicate that a method has been

invoked with inappropriate argument.

**Example:**

class Test{

public static void main(String[] args){

Thread t=new Thread();

t.setPriority(10);valid

t.setPriority(100);invalid

}}

**Output:**

Runtime exception

 Exception in thread "main" java.lang.IllegalArgumentException.

**NumberFormatException:**

 It is the child class of IllegalArgumentException and hence is unchecked. Raised

explicitly by the programmer or by the API developer to indicate that we are

attempting to convert string to the number. But the string is not properly formatted.

**Example:**

class Test{

public static void main(String[] args){

int i=Integer.parseInt("10");

int j=Integer.parseInt("ten");

}}

**Output:**

Runtime Exception

Exception in thread "main" java.lang.NumberFormatException: For input string:

"ten"

**IllegalStateException:**

It is the child class of RuntimeException and hence it is unchecked. Raised explicitly

by the programmer or by the API developer to indicate that a method has been

invoked at inappropriate time.

**Example:**

Once session expires we can’t call any method on the session object otherwise we will get IllegalStateException

HttpSession session=req.getSession();

System.out.println(session.getId());

session.invalidate();

System.out.println(session.getId()); 🡪illgalstateException