### Exception

**Exception** is an **event which occurs during the execution of program**, that **disrupts normal flow of the program’s instruction**.

Exception results abnormal termination of program, and rest of the program won’t be executed.

To avoid the abnormal termination of program, all possible exceptions that could be thrown needs to be handled. This is known as Exceptional Handling. This **helps to maintain graceful termination of the program**.

**Throwable** is root class for Exception and Error.

**Java.lang.Throwable** extends from java.lang.Object class.

**Exception** is the **super class** for **all type of exception**.

**Exceptions** are **caused due to programmatic logic**.

It is **recoverable**. Exception **is an event that represents a condition from which possible to recover.**

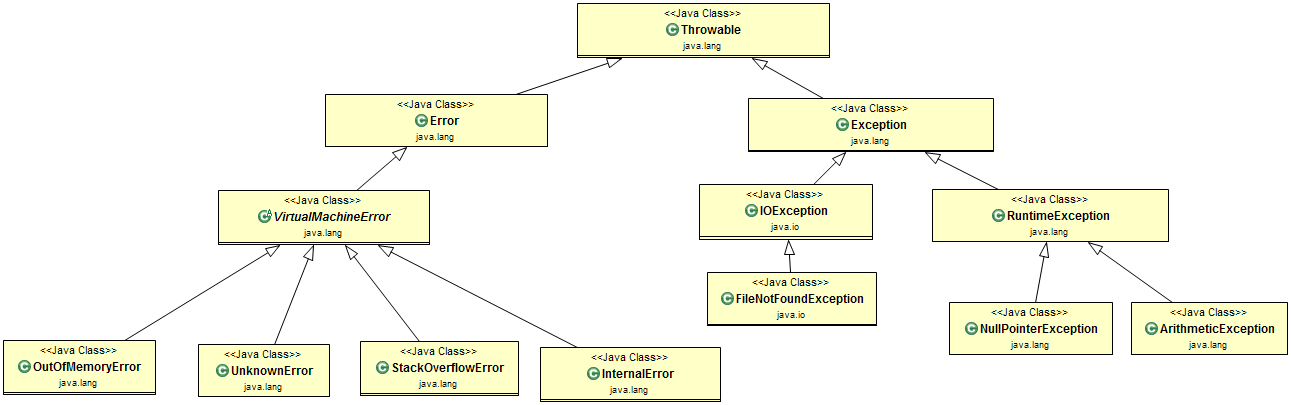
**Exceptions are categorized into checked exception and unchecked exception.**

Error is the **super class** for **all type of error**.

Errors are caused due **to lack of system resources**.

It is **non-recoverable**. Error **represents an external situation usually impossible to recover from.**

**All errors are fall into unchecked Exception category**, as it is raised due to lack of system resources at run time.

**[](https://2.bp.blogspot.com/--f8P24mVPK0/W3v8Qq2JzrI/AAAAAAAADSA/PR-7hjBxP1QPhzc5UsYyZBgFseJ7QM2XgCLcBGAs/s1600/exception-handling.png)**

**OutOfMemoryError** is subclass of **VirutalMachineError** and its **thrown by JVM when it** **ran out** **of heap memory**. We can **fix this error** by **providing more memory** to run the Java Application throw Java options.

Java myprogram -Xms1024m -Xmx1024m -XX:PermSize=64M -XX:MaxPermSize=256m

**StatckOverflowError** occurs when **the stack space for a thread has run out**, typically because an **application recurses too deeply**.

**Checked Exception**

**Exceptions which are checked at compile time** **during compilation** is known as Checked Exception.

**Checked Exceptions must be caught at compile time.**

Compilers forces programmers to deal with **either try/catch block** **or by declaring throws in a method.**

**Exception is super class for all the checked Exception**.

**Checked Exception** – IOException, FileNotFoundException, InterruptedException, CloneNotSupportedException, ClassNotFoundException

**UnChecked Exception**

**Exception which are not checked at compile time** is known as Unchecked Exception.

**UnChecked Exceptions are run time exception.**

Compiler does not force programmers either to catch the exception or declare it in throws clause.

**RuntimeException is the super class for all the unchecked exception.**

RuntimeException – NullPointerException, ArithmeticException, NoClassDefFoundError.

**Throws**

Throws keyword **is used with method signature** to declare exception that the method might throw.

Throws keyword **is used to specify that a method may raise an exception during its execution.**

Public void methodA () throws IOException {

}

**Throws clause is applicable to methods and constructors but strictly not applicable to classes.**

**Throws is used next to method signature.**

**Any number of exceptions can be thrown using throws keyword.**

**Throw**

Throw is used to **throw an exception manually**.

It is **used within method to throw an exception explicitly**.

It is **generally used for throwing user defined exception**.

**Maximum only one exception can be thrown using throw keyword.**

**Throw keyword followed by instance.**

Throw is used to **define user defined exception explicitly**.

Throw keyword is used to **throw an exception to the run time** to **handle** it.

**Sometimes we explicitly want to create an exception object for that we need to use throw keyword** **and then throw it to halt the normal processing of the program.**

If (task.isNotCompleted()) {

Throw new NotCompletedException (“This task is not completed”);

}

**Try-with-resources statement**

Using try-with-resources statement, **programmer doesn’t need to explicitly close opened resources**.

**Rather it will be automatically closed once control reaches end of try-catch block**.

This new feature introduced in Java 1.7 version is alternatively referred **as Automatic Resource Management** i.e., ARM

Try-with-resources statement is **a try statement that declares one or more resources**.

The resource is an object that must be closed after finishing the program.

The try-with-resources ensures that each resource is closed at the end of the statement execution.

public class BufferedReaderExample {

public static void main(String[] args) {

try (FileReader fr = new FileReader("C:/workspace/java-io-guide/sample.txt"); BufferedReader br = new BufferedReader(fr);) {

String sCurrentLine;

while ((sCurrentLine = br.readLine()) != null) {

System.out.println(sCurrentLine);

}

} catch (IOException e) {

e.printStackTrace();

}

}}

**Stack Trace**

A stack trace provides the name of the classes and method that were called, from the start of the application to the point an exception occurred.

It is very useful debugging tool since it enables us to determine exactly where the exception was thrown in the application and the original causes that led to it.

**Write a custom Exception**

* Create a new class whose name should end with Exception.
* Make the class **extends any one of the Exceptions w**hich are subtypes of the java.lang.Exception class.
* **Create a constructor with a string parameter which is detailed message of an exception**.

public class InvalidAgeForDrivingLicenseException extends Exception {

    // member variable for Exception Description

    String expDescription;

    // public constructor with String argument

    InvalidAgeForDrivingLicenseException(String expDescription) {

         super(expDescription);

    }

}

**Exception propagation:**

* Whenever exception is raised from method and if it isn’t handled in the same method, then it is propagated back to the caller-method
* This step is repeated until handler code is found in one of the caller-method in the runtime stack or else it reaches bottom of the runtime-stack
* This is known as Exception propagation

**Rules for Exception propagation:**

* By default, unchecked-exception is propagated back to the runtime-stack one-by-one until it finds handler code or it reaches bottom of the stack
* Checked-exception isn’t propagated, rather compiler forces the programmer to handle checked-exception in the same method by surrounding with try-catch block or declaring with throws keyword.

**Every thread**, **JVM creates a Stack. All the method calls performed by Thread to be stored in the corresponding stack.**

**finalize () method**

**finalize () method is protected and non-static method of java.lang.Object class.**

This method is available in all objects you create in Java.

This method is **used to perform some final operation or clean up operations on an object** before it removed from memory.

You can override the finalize () method to keep those operations you want to perform before an object is destroyed.

**finalize** () is a **method** is **associated with garbage collector.**

**This method is invoked before destroying an object** i.e. provide clean-up activities.

**After garbage collector invokes finalize () method, then immediately it destroys an object.**

|  |
| --- |
| public class FinalizeMethodExample {       protected void finalize() throws Throwable {           System.out.println("finalize() method invoked"                  + " to clean-up Object resources");      }        public static void main(String[] args) {           // create Object of type FinalizeMethodExample          FinalizeMethodExample fme = new FinalizeMethodExample();            // explicitly making null          fme = null;      }  } |

**Explanation:**

* When above program is executed, it doesn’t print any sysout statement from finalize() method
* Because, it is the garbage collector’s duty to invoke finalize() method just before destroying the Object
* And hence programmer can’t make sure that it is compulsorily invoked although we can make any Object as null explicitly.

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| ClassNotFoundException occurs when the required .class file is missing when program encounters class load statement such as  Class.forName(“class.name”)  Class.loadClass(“class.name”)  ClassLoader.findSystemClass(“class.name”) | NoClassDefFoundError occurs when the class is found during compile time but not at the run time. |
| ClassNotFoundException is checked exception. It extends from java.lang.Exception class. | NoClassDefFoundError is an error. It extends from java.lang.LinkageError class. |
| **It is a checked exception that needs to be handled.** | **All errors come under unchecked Exception. Therefore, NoClassDefFoundError is also unchecked Exception.** |

[**rules for Exception handling w.r.t Method Overriding in Java**](http://www.benchresources.net/rules-for-exception-handling-w-r-t-method-overriding-in-java/)

**Rule 1:** If parent-class method **doesn’t**declare any exception,

1. Then child-class overriding-method can declare **any type of unchecked-exception** (this is the only possibility)
2. If child-class overriding-method declares checked-exception, then compiler throws **compile-time error** stating “**Exception <**exception-type**> is not compatible with throws clause in ParentClass.testMethod()**”
3. Then child-class overriding-method can declare **no exception** (this is very much same as that of overridden-method of parent-class –> exactly same method signature)

**Rule 2:** If parent-class method declares**unchecked**exception,

1. Then child-class overriding-method can declare **any type of unchecked-exception** (not necessarily same exception as that of parent-class’ method)
2. If child-class overriding-method declares any checked-exception, then compiler throws **compile-time error** stating “**Exception <**exception-type**> is not compatible with throws clause in ParentClass.testMethod()**”
3. Then child-class overriding-method can declare **no exception**

**Rule 3:** If parent-class method declares**checked**exception,

1. Then child-class overriding-method can declare **any type of unchecked-exception**
2. Then child-class overriding-method can declare **same type of checked-exception**or**one of its sub-class**or**no exception**
3. Then child-class overriding-method can declare **no exception**

**Rule 4:** If parent-class method declares both**checked**&**unchecked**exceptions,

1. Then child-class overriding method can declare **any type of unchecked-exception**
2. Then child-class overriding-method can declare **same type of checked-exception**or **one of its sub-class**or**no exception**
3. Then child-class overriding-method can declare **no exception**