**EXCEPTION HANDLING**

***Agenda:***

* Introduction
* Runtime Stack mechanism
* Default Exception Handling in java
* Exception Hierarchy
* Customized Exception-Handling by try-catch
* Control Flow in try-catch
* Methods to print Exception information
* try with Multiple catch blocks
* finally
* Difference between final, finally and finalize
* Control flow in try-catch-finally
* Various possible combinations of try-catch-finally
* throw keyword
* throws keyword
* Exception handling keywords summary
* Various possible compile-time errors in exception handling
* Customized Exceptions
* Top 10 Exceptions

1. **INTRODUCTION:**

An unwanted un-excepted event that disturbs normal flow of the program called **Exception.**

**Eg:** SleepingException

TyrePuncheredException

FileNotFoundException…….etc.

* It’s highly recommended to handling Exception, the main objective of exception handling is graceful termination of the program.--, -9j-

Eg: Open DB Connection

Read DB Connection --------🡪**SQLException**

Close DB Connection

* Exception Handling doesn’t mean repairing an Exception. We have to define alternative way to continue rest of the program normally. This way of defining alternative is nothing but Exception Handling.

**Eg:** Suppose our programming requirement is to read data from UK file at runtime. If UK file is not available our program should be terminated normally. We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but Exception Handling.

try

{

Read datafrom UK file

}catch(FileNotFoundException e)

{

Use local file and continue the rest of the program normally

}………

1. **Runtime Stack Mechanism:**

For every thread JVM will create a separate stack, all method calls performed by thread will be stored in that stack. Each entry in the stack is called one **Activation record/Stack Frame**. After completing every method call JVM removes the corresponding entries from the stack.

After completing all method calls JVM destroys the empty stack and terminates the program normally.

**Eg:**

class Test {

public static void main(String[] a){

doStuff();

}

public static void doStuff(){

doMoreStuff();

}

public static void doMoreStuff(){

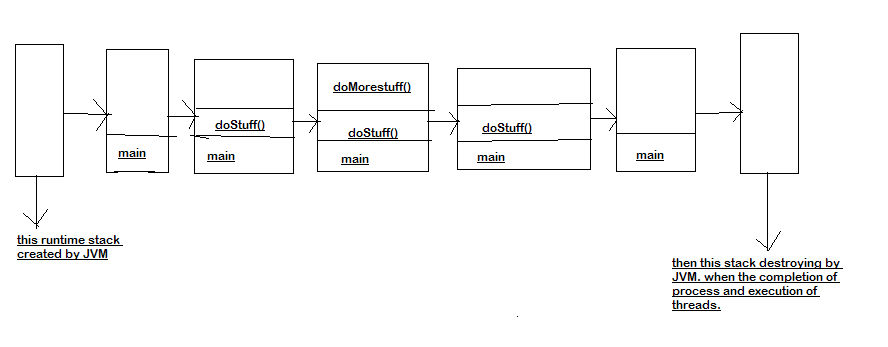
System.out.println(“hello”);

}

}

**Process:** For every java program internally one thread is created for java

(eg: i.e ‘main’) for that JVM created one Runtime stack.



1. **Default Exception handling in Java:**

If an exception raised inside any method then the method is responsible to create Exception object with the following information.

* Name of the Exception
* Description of the Exception
* Location of the Exception
* After creating that exception object the method handovers that object to the JVM.
* JVM checks whether the method contains any exception handling code or not. If the method won’t contain any handling code then JVM terminates the method abnormally and removes corresponding entry from the stack.
* JVM identifies the caller method and checks whether the caller method contain any handler code or not. If the caller method also doesn’t contain handle code then JVM terminates that caller method also abnormally and removes corresponding entry from the stack.
* This process will be continued until main() method and if the main() method doesn’t contain any exception handling code then JVM terminates main() method and removes corresponding entry from the stack.
* Then JVM handovers the responsibility of exception handling to the **‘Default Exception Handler’**
* **‘Default Exception Handler’**just print exception information to the console in the following format and terminates the program abnormally.

|  |
| --- |
| Name of the Exception: Description  Location of the Exception (Stack trace) |

**Eg:**

class Test {

public static void main(String[] a){

doStuff();

}

public static void doStuff(){

doMoreStuff();

}

public static void doMoreStuff(){

System.out.println(10/0);

}

}

**Output:**

RE:Exception in thread "main" java.lang.ArithmeticException: / by zero

at Test.doMoreStuff(Test.java:13)

at Test.doStuff(Test.java:9)

at Test.main(Test.java:5)

1. **Exception Hierarchy:**

* **‘Throwable’** acts as a root for the exception hierarchy.
* **‘Throwable’**class contains the following two child classes.
* Exception
* Error

**Exception:**

Most of the cases Exceptions are caused by our program and these are

**‘recoverable’**

**Error:**

Most of the cases Errors aren’t caused by our program these are due to

lack of system resources and these are **‘non-recoverable’.**

4.1)**Checked VS Unchecked Exception:**

**Checked Exception:**

The Exceptions which are checked by the compiler for smooth execution of the program at runtime are called **‘Checked Exception’.**

Eg: HallTicketMissingException

PenNotWorkingException

FilenotFoundException………etc.

**Un Checked Exception:**

The Exceptions which are not checked by the compiler for smooth execution of the program at runtime are called **‘UnChecked Exception’**

Eg: BombBlastException

AirthmeticException

NullPointerException………etc.

**Note:**Runtime Exception and it’s child classes, Error and it’s child classes are **“Un Checked Exceptions”** and all the remaining are considered as **“Checked Excepions”.**

Whether the exception is checked or un checked compulsory it should occur at runtime only. There is no chance of occurring any exception at compiletime.

4.2) **Partially Checked VS Fully Checked:**

If checked exception is said to be fully checked iff all it’s child classes also checked.

Eg: IOException--->Checked Exception

InturrptedException----> Checked Exception

If checked execption is said to be prtially checked iff some it’s child classes also un checked.

Eg: Exception------> Checked Exception

|

RuntimeException-------> Unchecked Exception

Eg: At supermarket-------> mother---- checked

Baby------ n’t checked

**Note:**The only **Partially Checked exceptions** in java are

**Trowable , Exception.**

**Q) Which of the following are checked?**

**1) RuntimeException--------> unchecked Exception**

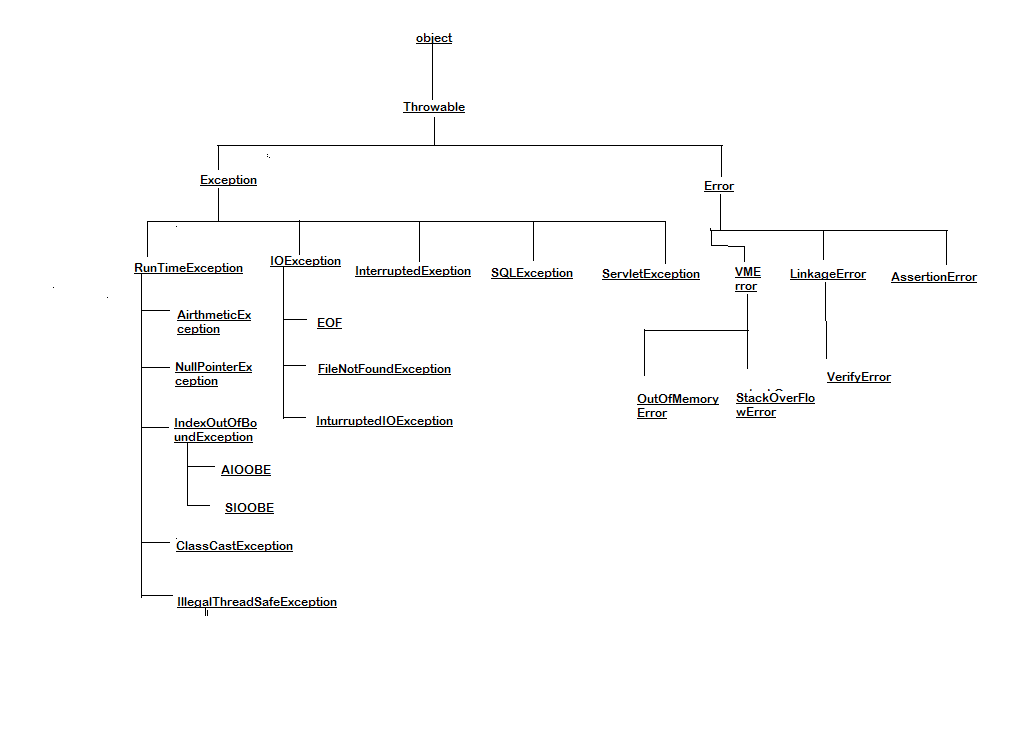
**2) Error --------> unchecked Exception**

**3) IOException --------->checked, fully checked**

**4) Exception -----------> partially checked**

**5) InterruptedException------> fully checked**

**6) Throwable ---------> partially checked**

****

1. **Customized Exception Handling by try-catch:**

It is highly recommended to handle exception. In our program the code which may cause an exception is called **risky code**. We have to place risky code inside try-block and the corresponding handling code inside catch block.

**Syntax:**

**try**

**{**

**Risky code**

**}**

**catch (Ex e)**

**{**

**Handling code**

**}**

**Without try-catch:**

**class Test1 {**

**public static void main(String[] args) {**

**System.out.println("hi");**

**System.out.println(10/0);**

**System.out.println("Hello”);**

**}**

**}**

**Output:**

**hi**

**Exception in thread "main" java.lang.ArithmeticException: / by zero**

**at Test1.main(Test1.java:6)**

**with try-catch:**

class Test1

{

public static void main(String[] args)

{

System.out.println("hi");

try{

System.out.println(10/0);

}catch(ArithmeticException e){

System.out.println(10/2);

}

System.out.println("Hello");

}

}

**Output:**

**hi**

**5**

**Hello**

1. **Control flow in try-catch:**

try

{

Stat 1;

Stat 2;

Stat 3:

}catch(x e)

{

Stat 4;

}

Stat 5;

**Case 1:**

If there is **‘no exception’**

Statement 1,2,3,5 executed i.e normal termination

**Case 2:**

If an exception raised at Statement 2 and corresponding catch

block matched will be executed.

Statement 1,4 ,5 executed normal termination

try

{

Stat 1;

Stat 2;---------------->// At statement 2 exception raised in try block then

Stat 3; Controller goes to corresponding catch block

}catch(x e) and executed that statement then controller

{ doesn’t go back to execute stat 2 and stat 3.

Stat 4;

}

Stat 5;

**Case 3:**

If an exception raised at statement 2 but the corresponding catch block not matched

**Output:**1 followed by abnormal termination.

**Case 4:**

If an exception raised at statement 4 or statement 5

Then it always abnormal termination of the program

**NOTE:**

* Within the try-block if any where an exception raised then rest of the try-block won’t be executed even though we handled exception.

Hence we have to place risky code inside try-block and length of the try-block should be as less as possible.

* If any statement which raises an exception and it isn’t part of the any try-block then it is always abnormal termination of the program.
* finally blocks also in condition to try-block.

1. **Various method to print Exception Information:**

**’Throwable’**class defines the following methods to print exception information to the console.

* + 1. printStackTrace():

This method prints exception information in the following format

|  |
| --- |
| Name of the Exception: description of the exception’  Stack trace |

ii) toString():

This method prints exception information in the following format

|  |
| --- |
| Name of the exception: description |

iii)getMessage():

This method prints exception information in the following format.

|  |
| --- |
| Description |

**NOTE:** Default exception handler internally uses printStacktrace(), to print exception information to the console.

1. **Try with multiple catch blocks:**

The way of handling an exception varied from exception to exception. Hence every exception type a separate catch block is required i.e. try with multiple catch blocks is possible and recommended to use.

Eg1:

try{

………

}catch(Exception e)

{

Default Handler

}

**Note:** This approach isn’t recommended because for any type of exception we are using the same catch block.

Eg:

**try**{

//statements.....

}**catch** (FileNotFoundException e) {

//use local file;

}**catch** (ArithmeticException e) {

// perform these airthmetic operations

}**catch** (SQLException e) {

// don't use oracle DB use my sql DB

}**catch** (Exception e) {

// for default handler

}

**Note:**This approach is highly recommended because for different exception types we are defining a separate catch blocks.

**Note:** If try with multiple catch blocks present then order of catch blocks

is very important it should be from child to parent. By mistake if we are taking from parent to child then we will get compile time error saying.

|  |
| --- |
| Exception XXXException has already been caught |

**EG: CE: e**xception java.lang.ArithmeticException has already been caught.

**Eg2:**

**try**{

//statements

}**catch** (ArithmeticExceptione) {

// **TODO**: handle exception

}**catch** (Exception e) {

// **TODO**: handle exception

}

This is highly recommended.

1. **finally:**

* It is never recommended to take **‘clean up’** code inside try block because there is no guarantee for the execution of every statement inside a try.

**Eg:**

**try**{

/\*statements

\* open Db connection

\* Read data connection

\* Db connection close----->clean-up code

\*/

}**catch** (Exception e) {

// **TODO**: handle exception

}

* It is never recommended to clean-up code inside a catch block. Because if there is no exception then catch block won’t be execute

**Eg:**

**try**{

//statements

}**catch** (Exception e) {

// clean-up code

}

* We required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not and whether handled or not handled. Such type of place is nothing but **‘finally’** block.
* Hence the main objective of **‘finally’** block is to maintain clean up code

**try**{

//Risky code

}**catch** (Exception e) {

// handling code

}**finally**{

//clean up code

}

* The specialty of ‘finally’ block is, It will be executed always irrespective of whether exception is raised or not raised and whether handled or not handled.

**Eg1:**

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

**try**{

System.*out*.println("try");

}**catch** (Exception e) {

System.*out*.println("catch");

}**finally**{

System.*out*.println("finally");

}

}

}

**Output:**

try

finally

**Eg2:**

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

**try**{

System.*out*.println("try");

System.*out*.println(10/0);

}**catch** (ArithmeticException e) {

System.*out*.println("catch");

}**finally**{

System.*out*.println("finally");

}

}

}

**Output:**

try

catch

finally

**Eg3:**

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

**try**{

System.*out*.println("try");

System.*out*.println(10/0);

}**finally**{

System.*out*.println("finally");

}

}

}

**Output:**

try

finally

Exception in thread "main" java.lang.ArithmeticException: / by zero

**return VS finally:**

Even though return statement present in try or catch blocks first finally will be executed and after that only return statement will be considered. i.e. **finally** block dominates the **return** statement.

**Eg:**

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

**try**{

System.*out*.println("try");

**return**;

//System.out.println("try1");

}**catch** (ArithmeticException e) {

System.*out*.println("catch");

}**finally**{

System.*out*.println("Finally");

}

}

}

**Output:**

try

Finally

**Note:**

* Generally when we write a return in any method then after returns, those are not executed.
* But in the case of finally, finally executed first and then return.

If return statement present inside try, catch, finally then finally block return statement will be considered.

* There is only situation , where the finally block won’t be executed is whenever we are using **System.exit(0)** method.

**Eg:**

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

**try**{

System.*out*.println("try");

System.*exit*(0);

//System.out.println("try1");

}**catch** (ArithmeticException e) {

System.*out*.println("catch");

}**finally**{

System.*out*.println("finally");

}

}

}

**Output:**

try

1. **Difference between final ,finalize, finally:**

**final:**

final modifiers is applicable for classes, methods and variables.

* If a **class** declared as final then the child class creation is not possible.
* If a **method** declared as final then overriding of that is not possible.
* If a **variable** declared as final then re-assignment 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. Just before destroying an object to perform clean-up activities.

**NOTE:**

To maintain clean-up code finally block is recommended over finalize() method because we cannot expect exact behavior of the **Garbage Collector.**

1. **Control flow in try-catch-finally:**

**Eg:**

**try**{

statement 1;

statement 2;

statement 3

}**catch** (Exception e) {

stsement 4;

}**finally**

{

statenent 5;

}

statement 6;

**case 1:**

If there is no exception statement 1,2,3,5,6 executed as be in

normal termination.

**Case 2:**

If an exception rose at statement 2 and corresponding catch block matched then 1,4,5,6, normal termination.

**Case 3:**

If an exception raised at statement 2 and corresponding catch block not matched then 1,5 abnormal termination.

**Case 4:**

If an exception raised at statement 4then it is always abnormal termination but before that finally block will be executed then 5, abnormal termination.

**Case 5:**

If an exception raised at statement 5 and 6 then it is always abnormal termination.

1. **Control flow in nested try-catch-finally:**

**Eg:**

**try**{

statement 1;

statement 2;

statement 3;

**try**{

statement 4;

statement 5;

statement 6;

}**catch** (x e) {

statement 7;

}**finally**{

statenent 8;

}

statement 9;

}**catch** (Y e) {

statement 10;

}**finally**{

statement 11;

}

statement 12;

**Case 1:**

If there is no exception

1--------- 12 executed ,normal termination.

**Case 2:**

If an exception raised at statement 2 and the corresponding catch block matched

1,10,11,12 executed and normal termination

**Case 3:**

If an exception raised at statement2 and the corresponding is not matched then 1,11 ,abnormal termination

**Case 4:**

If an exception raised at statement 5 and the corresponding inner catch is matched then 1,2,3,4,7,8,9,11,12 , normal termination

**Case 5:**

If an exception raised at statement 5 and inner catch has not matched but outer catch block has matched

1,2,3,4,8,10,11,12, normal termination

**Case 6:**

If an exception raised at statement 5 and both inner and outer catch blocks are not matched

1,2,3,4,8,11,abnormal termination

**Case 7:**

If an exception raised at statement 7 and the corresponding inner catch is matched

1,2,3,.,.,.8,10,11,12, normal termination

**Case 8:**

If an exception raised at statement 7 and the corresponding inner catch is not matched

1,2,3,.,.,.8,11, abnormal termination

**Case 9:**

If an exception raised at statement 8 and the corresponding inner catch is matched

1,2,3,.,.,.,.,10,11,12, normal termination

**Case 10:**

If an exception raised at statement 8 and the corresponding inner catch is not matched

1,2,3,.,.,., .,11, abnormal termination

**Case 11:**

If an exception raised at statement 9 and the corresponding inner catch is matched

1,2,3,.,.,., .,8,10,11,12, normal termination

**Case 12:**

If an exception raised at statement 9 and the corresponding inner catch is not matched

1,2,3,.,.,., .,8,11, abnormal termination

**Case 13:**

If an exception raised at statement 10 is always abnormal termination but before that finally block 11 will be executed

**Case 14:**

If an exception raised at statement 11 or 12 is always abnormal termination

**NOTE:**

If we are not entering into the try block then the finally block won’t be executed once we entered into the try block without executing finally block we cannot come out.

**Eg:**

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

**try**{

System.*out*.println(10/0);

}**catch** (ArithmeticException e) {

System.*out*.println(10/0);

}**finally**

{

String s=**null**;

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

}

}

}

**Output:**

RE: Exception in thread "main" java.lang.NullPointerException

**NOTE:**Default exception handler can handle only one exception at a time and which is mostly recently raised exception.

1. **Various possible combinations of try-catch-finally:**

|  |  |  |
| --- | --- | --- |
| 1)  **try**{  } **catch** (Exception e)  {  // **TODO**: handle exception  }    //correct | 2)  **try**{    }**catch** (x e) {  // **TODO**: handle exception  }**catch** (Y e) {  // **TODO**: handle exception  }  //correct | 3)  **try**{    }**catch** (x e) {  // **TODO**: handle exception  }**catch** (x e) {  // **TODO**: handle exception  }  //CE:exception x has already been caught |

|  |  |  |
| --- | --- | --- |
| 4)  **try**{    }  //CE: try with out catch or finally | 5)  **catch** (Exception e) {  // **TODO**: handle exception  }  //CE: catch without try | 6)  **try**{    }  System.out.println("hello");  **catch** (Exception e) {  // **TODO**: handle exception  }  //CE1: try without catch or finally  //CE2: catch without try |
| 7)  **try**{    }**catch** (Exception e) {  // **TODO**: handle exception  }  **finally**{    }//correct | 8)  **try**{    }**finally**{    }  //correct | 9)  **try**{    }**finally**{    }**finally**{    }  // finally with out try |

|  |  |  |
| --- | --- | --- |
| 10)  **try**{    }**catch** (Exception e) {  // **TODO**:  handle exception  }  System.out.println  ("hello");  **finally**{    }  //finally without try | 11)  **try**{    }**finally**{    }  **catch** (Exception e) {  // **TODO**: handle exception  }  //catch without try | 12)  **finally**{    }  // finally without try |
| 13)  **try**{  **try**{  }**catch** (Exception e) {  // **TODO**: handle  exception  }  }**catch** (Exception e) {  // **TODO**: handle  exception  }  //correct | 14)  **try**{    }**catch** (Exception e)  {  // **TODO**: handle exception  }**try**{    }  **finally**{    }  }  //correct | 15)  **try**{    }**catch** (Exception e)  {  // **TODO**: handle exception  ***try****{*  *}****catch*** *(Exception e) {*  *//* ***TODO****: handle*  *exception*  *}*  }**finally**{  }  //correct |

|  |  |
| --- | --- |
| 16)  **finally**{    }  **try**{    }**catch** (Exception e) {  // **TODO**: handle exception  }  //CE: finally without try | 17)  **try**{    }**catch** (Exception e) {  // **TODO**: handle exception  }**finally**{  **try**{    }**catch** (Exception e) {  // **TODO**: handle exception  }  }  //correct |

1. **Throw statement:**

Sometimes we can create an exception object explicitly and we can hand over to the JVM manually by using **throw** keyword.

|  |
| --- |
| **throw new ArithmeticException(“/by zero”);** |

**The result of the following two programs is exactly same**

|  |  |
| --- | --- |
| **class** Test{  **public static void** main(String[] args) {  System.*out*.println(10/0);  }  }  // in this case creation of AE object and handover to the jvm will be performed by automatically by the main() | **class** Test{  **public static void** main(String[] args) {  **throw new** ArithmeticException();  }  }  //in this case we are creating an exception object explicitly and handover to the jvm manually. |

**NOTE:** In general we can use ‘throw’ keyword for customized exception but not for predefined exception.

**Case 1:**

throw e;

if e refers null then we will get NullpointerException

|  |  |
| --- | --- |
| 1)  **class** Test{  **static** ArithmeticException *ae*=  **new** ArithmeticException();  **public static void** main(String[] args) {  **throw***ae*;  }  }  //RE: AE | 2)  **class** Test{  **static** ArithmeticException *ae*;  **publicstaticvoid** main(String[] args) {  **throw***ae*;  }  }  //RE:NPE |

**Case 2:**

**After throw statement we can’t take any statement directly otherwise we will get compile time error saying unreachable statement**.

|  |  |
| --- | --- |
| 1)  **class** Test{    **public static void** main(String[] args) {  System.*out*.println(10/0);  System.*out*.println("hello");  }  }//RE:AE:/ by zero | 2)  **class** Test{    **public static void** main(String[] args) {  **throw new** ArithmeticException("/by  zero");  System.*out*.println("hello");  }  }// unreachable statement |

**Case 3:**

**We can use throw keyword only for throwable types. Otherwise we will get compile time error saying incompatible types.**

|  |  |
| --- | --- |
| **1)**  **class** Test{    **public static void** main(String[] args) {  **throw new** Test();  }  }  //CE: incompatible types  Found: test  Required:int | **2)**  **Class T**est**extends** RuntimeException{    **public static void** main(String[] args) {  **throw new** Test();  }  }  //RE:Exception in thread ‘main’ test |

1. **Throws keyword:**

* Throws keyword just used for checkedException.
* In our program, if there is any chance of raising checked exception compulsory we should handle either by try/catch or by throws keyword otherwise the code won’t compile

**Eg:**

**class**Test {

**publicstaticvoid** main(String[] args) {

Thread.*sleep*(5000);

}

}

**Output:**

CE: unreported Exception: java.lang.InturupptedExcception

Must be called are declared to be thrown.

We can handle these compile time error by using the following ways

**By using try/catch:**

Eg:

**class** Test {

**publicstaticvoid** main(String[] args) {

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**By using throws keyword:**

We can use throws keyword to delegate the responsibility of Exception Handling to the caller method then the caller method is responsible to handle that exception.

**Eg:**

**class**Test {

**publicstaticvoid** main(String[] args) **throws** InterruptedException {

Thread.*sleep*(5000);

}

}

* Hence the main objective of throws keyword is to delegate the responsibility of exception handling to the caller method.
* ‘throws’ keyword required only for checked Exception usage of throws for unchecked Exception there is no use.
* Throws keyword required only to convince compiler usage of throws keyword doesn’t prevent abnormal termination of the program.

**Eg:**

**class** Test {

**publicstaticvoid** main(String[] args) **throws** InterruptedException {

dostuff();

}

**publicstaticvoid** doStuff()**throws** InterruptedException{

domorestuff();

}

**publicstaticvoid** domoreStuff()**throws** InterruptedException{

Thread.*sleep*(5000);

}

}

In the above program if we are removing at least one throws keyword then the program won’t compile.

**Case 1:**

**We can use throws keyword only for throwable types otherwise we will get compile time error saying incompatible types.**

|  |  |
| --- | --- |
| 1)  **class** Test {    **public static void** main(String[] args)  **throws T**est {    }  }  //CE: incompatible types  Found : test  Equired java.lang.throwable | 2)  **class** Test **extends** RuntimeException{    **publicstaticvoid** main(String[] args)  **throws T**est {    }  }  //correct |

**Case 2:**

In our program

|  |  |
| --- | --- |
| 1)  **class** Test {    **public static void** main(String[] args) {  **throw new** Exception();    }  }  // unreported Exception j.l Exception  Must be caught or declared thrown | 2)  **class** Test {    **public static void** main(String[] args) {  **throw new** Error();    }  }  //RE:exception in thread ‘main’ j.l.Error |

**Case 3:**

In our program if there is no chance of raising an exception then we can’t write catch block for that exception otherwise we will get compile error saying

**CE: Exception: xxxException is never thrown in body of corresponding try statement**

But this rule is applicable for only fully checked exceptions.

|  |  |  |
| --- | --- | --- |
| 1)  **try**{  System.*out*.println("hi");  }**catch** (Exception e) {  // **TODO**: handle exception  }  //correct  //partially checked | 2)  **try**{  System.*out*.println("hi");  }**catch** (ArithmeticException e) {  // **TODO**: handle exception  }  //correct  // un checked | 3)  **try**{  System.*out*.println("hello");  }**catch** (IOException e) {  // **TODO**: handle exception  }  //wrong  //fully Checked |

Some more……

|  |  |
| --- | --- |
| 1)  **try**{  System.*out*.println("hello");  }**catch** (InterruptedException e) {  // **TODO**: handle exception  }  //wrong  //fully checked | 2)  **try**{  System.*out*.println("hello");  }**catch** (Error e) {  // **TODO**: handle exception  }  //correct  //un checked |

1. **Exception Handling Keyword Summay:**

Try---------------> To maintain risky code

Catch-------------> to maintain handling code

finally------------>to maintain clean up code

throw-------------> to hand-over our created exception object to the

JVM manually.

Throws-----------> to delegate responsibilities of exception handling

to the caller method .

1. **Various possible compile time errors in exception Handling:**

* Exception xxx has already been caught
* Unreported exception xxx must be caught or declared to be thrown
* Exception xxx is never thrown in body of corresponding try statement
* Try without catch or finally
* Catch without try
* Finally without try
* Incompatible types

Found: Test

Required: java.lang.Throwable

* Unreachable statement

1. **Customized Exception:(user defined Exceptions):**

Sometimes we can create our own Exceptions to meet our programming requirements. Such type of exceptions is called custom Exceptions.

Eg: InSufficientException

TooYoungException

TooOldException

Eg:

**Class**TooYoungException**extends** RuntimeException

{

**public** TooYoungException(String s) {

**super**(s);

}

}

**Class**TooOldException**extends** RuntimeException{

TooOldException(String s){

**super**(s);

}

}

**class**CustException

{

**publicstaticvoid** main(String[] args) {

**int** age=Integer.*parseInt*(args[0]);

**if**(age>60){

**thrownew** TooOldException("ur age is already

crossed..........no chance to getting married");

}

**elseif**(age<18){

**thrownew** TooYoungException("plz wait some more time

.......u will get match"); }

**else**{

System.*out*.println("u will get match datails soon by e-

mail");

}

}

}

**Note:**

**It is highly recommended to maintain our customized Exception as unchecked by throwable type including errors also**

1. **Top 10 Exceptions:**

**JVM Exceptions:**

Raised automatically by the JVM whenever a particular event occurs.

Eg:

ArrayIndexOutOfBoundsException(AIOOBE)

NullPointerException(NPE)

**Programmatic Exceptions:**

The exceptions which are raised explicitly by the programmer (or) by the API developer are called programmatic Exception

Eg: IllegalArgumentException

**ArrayIndexOutOfBoundsException:**

It is the child class of **RuntimeException** and hence it is unchecked raised automatically by the jvm when ever we are trying to access Array element without of range index

Eg:

Int[] x=new int[10];

s.o.p(x[9]);----->valid

s.o.p(x[100]);-------->RE:AIOOBE

s.o.p(x[-100]);------->RE:AIOOBE

**NullPointerException:**

It is the child class of **RuntimeException** and hence it is unchecked raised automatically by the jvm when ever we are trying to call method null.

Eg: String s=null;

s.o.p(s.length());---------->NullPointerException

**StackOverFlowError:**

It is the child class of Error and hence it is unchecked whenever we are trying to invoke recursive method call to jvm will raise StackoverFlowError automatically.

Eg:

**class**Test {

**publicstaticvoid**m1{

*m2*();

}

**publicstaticvoid** m2(){

m1();

}

**publicstaticvoid** main(String[] args) {

m1();

}

}

**Output:**

**RuntimeException: StackOverFlowError**

**NoClassDefFoundError:**

It is the child class of Error and hence it is unchecked JVM will raised automatically whenever it is unable to find required .class file

**Eg:**

C:/>Java Test

If Text.class is not available then we will get NoClassDefFoundError.

**ClassCastException:**

It is the child class of **RuntimeException** and hence it is unchecked raised automatically by the jvm when ever we are trying to typecast parent object to the child object type.

Eg:

Stirng s=new String(“sekhar”);

Object o =(Object)s;----------->valid

Object o=new Object();

String s=(string)o;

**Output:**RE:CCE

**ExceptionInInitializerError:**

It is the child class of Error and hence it is unchecked raised automatically by the jvm. If an exception occurs while performing static variable initialization and static block execution.

Eg:

class Test{

Static int i=10/0;

}

**Output: RE:ExceptionInInitializerError:caused by AE:/by zero**

**Eg:**

class Test{

static{

String s=null;

s.o.p(s.length());

}

}

**Output:**

**RE:ExceptionInInitializerError:caused by NPE**

**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

Eg:

Thread t =new thread();

t.setpriority(10);------------->valid

t.setPriority(100);-------->RE: IAE

**NumberFormatException:**

It is the child class of **IllegalargumentException** and hence it 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 string is not properly formatted.

Eg:

int i=Integer.parseInt(“10”);--------------->valid

int i=Integer.parseInt(“ten”);--------------->RE: NumberFormatExcception

**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 with inappropriate time.

Eg: Once session expiries we can’t call any method on the session object otherwise we will get **IllegalStateException.**

**HttpSession session req.getSession();**

**Sop(session.getId());**

**Session.invalidate();**

**Sop(session.getId());----->RE:IllegalStateException**

**Assertion Error:** It is the child class of **Error** and hence it is unchecked raised explicitly by the programmer or by the AP developer to indicate that Assert statement fails.

**Eg: assert(false);**