

1. **Introduction :**

* An unexpected unwanted event that disturbs normal flow of the program is called exception

e.g

tyer puncture exception

sleeping exception

file not found exception

* It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of the program.
* Exception handling doesn’t mean repairing an exception we have to provide alternative way to continue rest of the program normally is the concept of exception handling.

e.g

- our program requirement is to read data from remote file locating at London at run time if London file is not available our program should not be terminated abnormally we have to provide some local file to continue rest of the program normally this way of defining alternative is nothing but exception handling

e.g

try{

read data from remote file locating at London

}

Catch(FileNotFoundException e){

Use local file and continue rest of the program.

}

1. **Runtime stack mechanism :**

* For every thread JVM will create a run time stack. Each and every method performed by that thread will be stored in the corresponding stack.
* Each entry in the stack is called stack frame or activation record after completing every method call the corresponding entry from the stack will be removed.
* After completing all method calls the stack will become empty and that empty stack will be destroyed by JVM just before terminating thread

**e.g**

|  |  |
| --- | --- |
| **class test{**  **public static void main(String[] args){**  **doStuff();**  **}**  **Public void doStuff(){**  **doMoreStuff();**  **}**  **Public void doMoreStuff(){**  **SOP(“Hello”);**  **}}** | Run time  stack for  main thread doStuff()  **main() main()**  doMoreStff()  doStuff() dostuff()  Main() main() main()  This empty stack will be destroyed by JVM |

1. **Default exception handler in java:**

* Inside a method if any exception occurs the method in which it is raised is responsible to create exception object by including the following information.

1. Name of exception
2. Description of exception
3. Location at which exception occurs (stack trace)

* After creating exception object method handovers that object to the JVM.
* JVM will check whether the method contains any exception handling code or not if the method doesn’t contain exception handling code then JVM terminates that method abnormally and removes corresponding entry from the stack.
* Then JVM identifies caller method and checks whether caller method contains any handling code or note.
* If the caller method doesn’t contain handling 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 the main method also doesn’t contain handling code then JVM terminate main method also abnormally and removes corresponding entry from the stack.
* Then JVN handovers responsibility of exception handling to default exception handler, which is the part of JVM.
* Default exception handler prints exception information in the following format and the terminates program abnormally.

**Exception in thread “XXX” name of the exception : description of the exception**

**Stack trace**

**e.g 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| class test{  public static void main(String[] args){  doStuff();  }  Public static doStuff(){  doMoreStuff();  }  Public static doMoreStuff(){  SOP(10/0);  }} | |  | | --- | |  | | DoMoreStuff() | | doStuff() | | Main() |   Exception  Exception in thread “main” j.l.arithmeticException division by zero  at Test.doMoreStuff()  at. Test.doStuff()  at. Test.main() |

**e.g 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| class test{  public static void main(String[] args){  doStuff();  SOP(10/0);  }  Public static doStuff(){  doMoreStuff();  SOP(“hi”);  }  Public static doMoreStuff(){  SOP(“hello”);  }} | |  | | --- | |  | | DoMoreStuff() | | doStuff() | | Main() |   Output:  Hello  hi  Exception  Exception in thread “main” j.l.arithmeticException division by zero  at. Test.main() |

**Note :**

* In a program at least one method terminates abnormally the program termination is abnormal termination. If all methods terminated normally then only program termination is normal termination.

1. **Exception hierarchy :**

* Throwable class acts as root for java exception hierarchy.
* Throwable class defines two child classes Exception and Error
* **Exception**
* Most of the time exceptions are caused by our program and these are recoverable.

e.g if our program requirement is to read data from remote file locating at London at runtime if remote file is not available then we will get run time exception saying FIleNotFountException. If FIleNotFountException occours we can provide local file and continue rest of the program normally.

try{

read data from remote file locating at London

}

Catch(FileNotFoundException e){

Use local file and continue rest of the program.

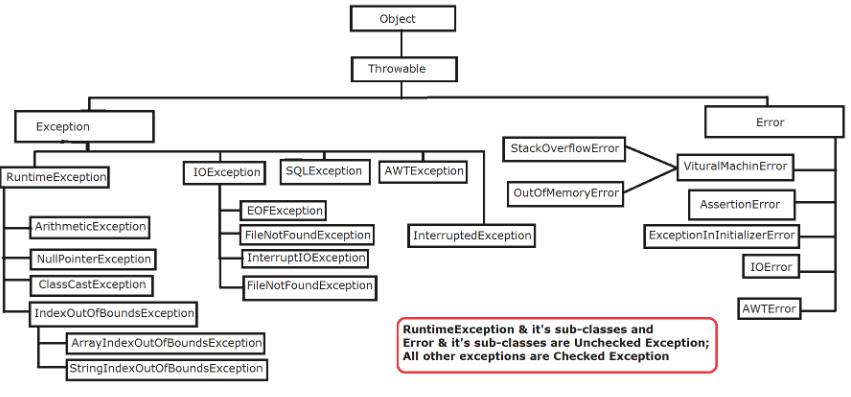
}

* **Error :**
* Most of the times errors are not caused by our program and these are due to lack of system resources. Errors are non-recoverable

E.g

If OutOfMemoryError occurs being a programmer we cant do anything and the program will be terminated abnormally.

* System admin or server admin is responsible to increase heap memory.



* **Checked Vs Unchecked**
* The exception which are checked by compiler for smooth execution of the program are called checked exception

E.g: HallTicketMissingException

PenNotWorkingException

FileNotFoundException

* In our program if there is chance of raising checked exception then compulsory we should handle that checked exception either by try-catch or by throws key word otherwise we will get compile time exception
* The exceptions which are not checked by compiler whether programmer handling or not such type of exceptions are called unchecked exceptions.

e.g : ArithmeticException

BomBlastException

Note:

* Whether it is checked or unchecked every exception occur at runtime only there is no chance of occurring any exception yet compile time.
* RuntimeException and its child classes, Error and its child classes are unchecked except these remaining all are checked.
* **Fully checked and partially checked exception**
* A checked exception is said to be fully checked if and only if all its child classes also checked.

e.g : IOException

InterruptedException

* A checked exception said to be partiality checked if and only if some of its child classes are unchecked.

e.g : Exception

Throwable

**Note:**

* The only possible partially checked exception in java are

1. Exception
2. Throwable

* **Describe the behavior of following exceptions**

1. IOException **fully checked**
2. RuntimeException **Unchecked**
3. InterruptedException **fully checked**
4. Error **Unchecked**
5. Throwable **Partially checked**
6. ArithmeticException **Unchecked**
7. NullPointerException **Unchecked**
8. Exception **Partially checked**
9. FileNotFoundException **Fully checked**
10. **Customized exception handling by using try catch**

* It is highly recommended to handle exceptions. The code which may raised an exception is called risky code and we have to define that code inside try block and corresponding handling code we have to define inside catch block.

**e.g**

|  |  |  |
| --- | --- | --- |
| **Try{**  **Risky code**  **}**  **Catch{**  **Handling code**  **}** | **Class test{**  **Public static void main(String[] args){**  **SOP(“state1”);**  **SOP(10/0);**  **SOP(“state2”);**  **}}**  **Output**  **State1**  **Runtime Exception: divide by zero** | **Class test{**  **Public static void main(String[] args){**  **SOP(“state1”);**  **Try{**  **SOP(10/0);**  **}**  **Catch(ArithmeticException e ){**  **SOP(10/2);**  **}**  **SOP(“state2”);**  **}}**  **Output**  **State1**  **5**  **State2** |

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

**e.g**

|  |  |
| --- | --- |
| **Try{**  **State1;**  **State2;**  **State3;**  **}Catch(Exceptione){**  **State4;**  **}**  **State5;** |  |

* If there is no exception the output will be 1235 and it will normal termination
* If exception raised at statement 2 and corresponding catch block matched the output will 145 and it will be normal termination.
* If exception raised at statement 2 and corresponding catch block not matched the output will 1 and it will be abnormal termination.
* If exception raised at statement 4 or statement 5 then it is always abnormal termination.

**Note**

* Within the try block if anywhere an exception raised then rest of the try block wont be executed even to we handled that exception hence within the try block we have to take only risky code and length of try block should be as less as possible.
* In addition to try block there may a chance of raising exception inside catch and finally blocks.
* If any statement which is not part of try block and raises an exception then it is always abnormal termination.

1. **Methods to print Exception information:**

* Throwable class defines the following methods to print exception information

|  |  |
| --- | --- |
| **Method** | **Printable format** |
| e.PrintStackTrace() | Name of Exception: description  Stack trace |
| SOP(e) or SOP(e.toString()) | Name of Exception: description |
| SOP(e.getMessage()) | description |

e.g :

Class test{

Try{

SOP(10/0);

}catch(ArithmeticException e){

e.printStackTrace();

SOP(e) or SOP(e.toString())

SOP(e.getMessage())

}}

**Note:**

* Internally default exception handler will use printStackTrace() to print exception information to the console.

1. **Try with multiple catch blocks**

* The way of handling an exception is varied from exception to exception hence for every exception type it is highly recommended to take separate catch block that is try with multiple catch block is always possible and recommended to use.

e.g :

|  |  |
| --- | --- |
| Try{  Risky code  }catch(Exception e){  }  //worst kind of programing | Try{  Risky code  } catch(ArithmeticException e){  Perform alternative opration  }  catch(FileNotFoundException e){  use local file instead of remote file  }  catch(SQLException e){  use mySql db instead of oracle  }  catch(Exception e){  default exception handling  }  //best programming |

* If try with multiple catch blocks present then the order of catch block is very importent, we have to take child first and then parent otherwise we will get compile time error saying “Exception XXX has already been caught”.

**e.g**

|  |  |  |
| --- | --- | --- |
| Try{  Risky code  }catch(Exception e){  }  catch(ArithmeticException e){  }  // Exception : Exception ArithmeticException has already been caught | Try{  Risky code  }catch(ArithmeticException e){  }  catch(Exception e){  }  // will work fine | Try{  Risky code  }catch(Exception e){  }  catch(ArithmeticException e){  }  catch(ArithmeticException e){  }  // Exception : Exception ArithmeticException has already been caught |

* We cant declare two catch block for the same exception otherwise we will get compile time error

1. **Final block:**

* Final is the modifier applicable for classes methods and variables.
* If a class declared as final then we cant extend that class that is we cant create child class for that class inheritance is not applicable for final classes.
* if a method is final then we cant override that method in the child class.
* If a variable declared as final then we can t perform reassignment for that variable.
* **Finally** is block always associated with try catch to maintain clean up code.

e.g:

try{

risky code

}catch{

Handling code

}finally{

Clean up code

}

* The spatiality of finally block is it will be executed always irrespective of whether exception is raised or not raised and whether handle or not handle.
* **Finalized :**
* Finalized is a method always invoked by garbage collector just before destroying an object to perform clean up activities.
* Once finalized method completes immediately garbage collector will destroy that object.

**Note:**

* Finally block is responsible to perform clean up activities related to try block that is whatever resources we open as part of try block will be closed inside finally block.
* Whereas finalized method is responsible to perform clean up activities related to object that is whatever resources associated with object will be deallocated before destroying an object by using finalized method.
* **Various combination of try-catch-finally.**
* In try-catch-finally order is important.
* Whenever we are writing try compulsory we should write either catch or finally otherwise we will get compile time error that is try without catch or finally is invalid.
* Whenever we are writing catch block compulsory try block must be required that is catch without try is invalid.
* Whenever we are writing finally block compulsory we should write try block that is finally without try is invalid.
* Inside try catch finally and finally block we can declare try-catch-finally blocks that is nesting of try-catch-finally is allowed.
* For try-catch and finally blocks curly braces are mandatory.

1. **Throw :**

* Sometimes we can create exception object explicitly and we can handover to JVM manually for this we have to use throw keyword.

**Throw new ArithmeticException(“/ by 0”) ;**

**Handover our created Creation of exception object**

**Exception object to JVM manually**

* Hence the main objective of throw keyword is to handover our created exception object to the JVM manually.
* Hence the result of following two program is exactly same.

e.g

|  |  |
| --- | --- |
| Class test{  Public static main(String[] rags){  SOP(10/0);  }}  Output  Exception in thread “main” j.l.ArithmaticException: / by zero  At test.main() | Class test{  Public static void main(String[] args){  Throw new ArithmeticException(“/ by zero”)  }}  Output  Exception in thread “main” j.l.ArithmaticException: / by zero  At test.main() |
| In this case main method is responsible to create exception object and handover to JVM. | In this case programmer creating exception object and handover to JVM manually. |

* Best use of throw keyword is for user define exception or customized exceptions
* **Case 1:**
* **Throw e if e refers null then we will get nullPointerException**

|  |  |
| --- | --- |
| Class test{  ArithmeticException e =new ArithmeticException()  Public static main(String[] rags){  Throw e;  }}  **Output**  Exception in thread “main” j.l.ArithmaticException: / by zero  At test.main() | Class test{  ArithmeticException e;  Public static void main(String[] args){  Throw e  }}  Output  Exception in thread “main” j.l.NullPointerException: |

* **Case 2:**
* After throw statement we are not allowed to write any statement directly otherwise we will get unreachable statement.

|  |  |
| --- | --- |
| Class test{  Public static main(String[] rags){  SOP(10/0);  SOP(“hello”)  }}  **Output**  **RE:** ArithmeticException : / by zero | Class test{  Public static void main(String[] args){  Throw new ArithmeticException(“/ by zero”);  SOP(“hello”);  }}  **Output**  **CE:** unreachable statement |

* **Case 3:**
* We can use throw key word only for throwable types if we are trying to use for normal java object we will get compile time error saying “incompatible types”

|  |  |
| --- | --- |
| Class test{  Public static main(String[] rags){  Throw test();  }}  **Output**  **RE: incompatible types** | Class test extends RuntimException{  Public static void main(String[] args){  Throw new Test();  }}  **Output**  **RE: Exception is in thread “main” test**  **At test.main()** |

1. **Throws :**

* In our program if there is a possibility of raising checked exception then compulsory, we should handle that checked exception otherwise we will get compile time error saying **“unreported exception XXX must be caught or declare to be thrown”**

**e.g**

|  |  |
| --- | --- |
| Class test{  Public static main(String[] rags){  PrintWriter pr = new PrintWriter(abc.txt);  Pr.write(“xyz”);  }}  **Output**  **CE: unreported exception FileNotFoundException; must be caught or declare to be thrown** | Class test{  Public static main(String[] rags){  Thread.sleep(2000);  }}  **Output**  **CE: unreported exception InterruptedException ; must be caught or declare to be thrown** |

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

|  |  |
| --- | --- |
| **By using Try-catch block** | **By using throws keyword** |
| Class test{  Public static main(String[] rags){  Try{  Thread.sleep(2000);  }catch(InterruptedException e){  }}} | Class test{  Public static main(String[] rags) throws InterruptedException {  Thread.sleep(2000);  }} |

* We can use throws keyword to delegate responsibility of exception handling to the caller (it may be another method or JVM) then caller method is responsible to handle that exception.
* Throws keyword required only for checked exception and usage of throws keywords for unchecked exceptions there no use or impact .
* Throws keyword required only to convince compiler and usage of throws keyword doesn’t prevent abnormal termination of the program.

e.g

|  |
| --- |
| Class test{  Public static main(String[] rags)throws InterruptedException{  doStuff();  }  Public static void doStuff()throws InterruptedException {  doMoreStuff();  }  Public static void doMoreStuff()throws InterruptedException {  Thread.sleep(1000);  }} |

* In the above program if we remove at least one throws statement then the code wont compile.

|  |  |
| --- | --- |
| Throws | 1. We can use to delegate the responsibility of exception handling to the caller method. 2. It is require only for checked exception and usage of throws keyword for unchecked exception there wont be any impact. 3. It is require only to convince to the compiler and usage of throws wont prevent abnormal termination of program. |

**Note :**

* It is recommended to use try-catch block over throws keyword.
* **Case 1 :**
* We can use throws keyword for methods and constructor but not for classes.

e.g

class test throws Exception{ // invalid

test() throws Exception{ // valid

}

Public void m1()throws Exception{ valid

}}

* **Case 2:**
* We can use throws keyword only throwable types if we are trying to use for normal java classes then we will get compile time error saying “incompatible types”

e.g

|  |  |
| --- | --- |
| class test{  Public void m1()throws test{  }}  //Exception: incompatobleException | class test extends RuntimeException{  Public void m1()throws test{  }}  //will work fine |

* **Case 3:**

|  |  |
| --- | --- |
| Class test{  Public static void main(String[] args){  Throw new Exception();  }}  **CE: unreported exception J.I.Exception;**  **Must be caught or declare to be thrown** | Class test{  Public static void main(String[] args){  Throw new Error();  }}  **RE: Exception in thread “main” J.I.Error**  **At test.main()** |

* **Case 4:**
* Within the try block if there is no chance of raising the exception then we can write catch block for that exception otherwise we will get compile time error saying “**Exception XXX is never thrown in body of corresponding try statement**”.
* Butt his rule is applicable only for fully checked exceptions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class test{  P s v main(String[] a){  Try{  SOP(“hello”);  }catch(AE e){  }}}  //will work fine | Class test{  P s v main(String[] a){  Try{  SOP(“hello”);  }catch(Exception e){  }}}  //will work fine | Class test{  P s v main(String[] a){  Try{  SOP(“hello”);  }catch(IOE e){  }}}  // **Exception IOexception is never thrown in body of corresponding try statement** | Class test{  P s v main(String[] a){  Try{  SOP(“hello”);  }catch(InterruptedException e){  }}}  **Exception InterruptedException** **is never thrown in body of corresponding try statement** | Class test{  P s v main(String[] a){  Try{  SOP(“hello”);  }catch(Error e){  }}}  //will work fine |

1. **Exception handling keywords summary:**

|  |  |
| --- | --- |
| **Keyword** | **Usage** |
| Try | To maintain risky code |
| Catch | To maintain exception handling code |
| Finally | To maintain clean up code |
| Throw | To create our own exception object and handover that object to the JVM manually |
| Throws | To delegate the responsibility of exception handling to the caller method |

* **Various possible compile time errors in exception handling :**

1. Unreported “XXX” must be caught or to be thrown.
2. Exception “XXX” has already been caught
3. Exception “XXX” is never thrown in body of corresponding try statement.
4. Unreachable statement
5. Incompatible type
6. Try without catch or finally
7. Catch without try
8. Finally without try
9. **Customized and user define exception:**

* Sometimes to meet programing requirements we can define our own exceptions such type of exceptions are called customized exception or user define exception.

e.g TooYoungException

TooOldException

InsufficientFundsException

e.g

|  |  |
| --- | --- |
| Class TooYoungException extend RuntimeException{  TooYoungException(String s){  Supper(s);  }}  // “s” is required to make information available to default exception handler | Class TooOldException extend RuntimeException{  TooOldException(String s){  Supper(s);  }}  // “s” is required to make information available to default exception handler |
| Class test{  Public static void main(Sting[] arg){  Int age = Integer.parseInt(arg[0]);  If(age > 60){  Throw New TooYoungException(“please wait for some more time. You will get best match soon”);  }  Else if(age<18){  Throw New TooOldException (“your age is already crossed for the marriage age.. no chance of getting mariage”);  }  Else{  System.out.println(“you will get match detail soon by email”);  }}} | |

* Throw keyword is best suitable for user define or customized exception, not for predefine exceptions.
* It is highly recommended to define customized exception as unchecked that is we have to extends RuntimeException but not Exception

1. Top 10 exception:

* Based on the person who is raising an exception all exceptions are divided into two categories

1. JVM Exceptions
2. Programmatic Exception

* JVM Exceptions :
* The exception which are raised automatically by JVM whenever a particular event occurs are called JVM exceptions.

E.g ArithmeticException

NullPointerException etc

* Programmatic exceptions :
* The exceptions which are raised explicitly either by programmer or by API developer to indicate the something goes wrong are called programmatic exceptions.

e.g TooOldException

IllegalArgumentException

1. **ArrayIndexOutOfBoundException** : // unchecked Exception

* It is the child class of runtime exception and hence it is unchecked.
* Raised automatically by JVM whenever we are trying to access array elemt with out of range index.

1. **NullPointerException :** // uncheck Exception

* It is the child class of runtime exception and hence it is unchecked.
* Raised automatically by JVM whenever we are trying to perform any operation on null.

1. **ClassCastException : //** uncheck Exception

* It is the child class of runtime exception and hence it is unchecked.
* Raised automatically by JVM whenever we are trying to type cast parent object to child type.

E.g

Object o = new Object();

String s =(String)o;

1. **StackOverFlowError :** // unchecked Error

* It is the child class of Error and hence it is unchecked
* Raised automatically by JVM whenever we are trying to perform recursive method call.

**e.g :**

Public void m1(){

m2();

}

Public void m2(){

m1();

}

1. **NoClassDefFoundError** : unchecked Error

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

e.g java test

if test class file not available the we will get runtime exception saying NoClassDefFoundError: test

1. **ExceptionInInitializationError: // unchecked Error**

* It is the child class of Error and hence it is unchecked
* Raised automatically by JVM whenever if any exception occurs while executing static variable assignments and static blocks

|  |  |
| --- | --- |
| **Static{**  **SOP(10/0);**  **}**  **ExceptionInInitializationError: caused by : J.l.ArethmaticException** | **Static{**  **String s=null;**  **SOP(s.lenght());**  **}**  **ExceptionInInitializationError: caused by : J.l.NullPointerException** |

1. **IllegalArgumentException : // unchecked Exception**

* It is the child class of runtime exception and hence it is unchecked.
* Raised explicitly either by programmer or by API developer to indicate that a method has been invoked with illegal argument.
* The valid range of thread priority is 1- 10 if we are trying to set the priority with any other value then we will get runtime exception saying ilegalArgumentException

**e.g**

thread t -new Thread();

t.setPriority(7);

t.setPriority(15); //runtimeException illegalArgumetException.

1. **NumberFormatException : //unchecked Exception**

* It is the child class of runtime exception and hence it is unchecked.
* Raised explicitly either by programmer or by API developer to indicate that we are trying to convert string to number and the String is not properly formatted

e.g

int I =Integer.parseInt(“str”); // numberFormateException

1. **IllegalStateException : // uncheckedException**

* It is the child class of runtime exception and hence it is unchecked.
* Raised explicitly either by programmer or by API developer to indicate that a method has been invoked at wrong time.

e.g : after staring of a thread we are not allowed to restart same thread once again otherwise we will get runtime exception saying illegalThreadStateException.

Thread t = new Thread();

t.start();

t.start(); /// illegalThreadStateException

1. **AssertionError** : // unchecked Error

* It is the child class of Error and hence it is unchecked.
* Raised explicitly either by programmer or by API developer to indicate that assert statement fails.

e.g assert(x >10 );

if x is not grater then we will get RuntimeException saying assertionError.

1. As the part of 1.7 version in exception handling the following to concepts introduced
2. Try with resources
3. Multi catch block

* **Try with resources:**
* **Until 1.6 vertion it is highly reccomonded to write finaly block to close resources which are open as the part of try block.**

|  |
| --- |
| BufferReader br =null;  Try{  Br= new BufferReader(new FileReader(“a.txt”))  //use br based on our requierment  }catch(IOException e){  //handling code  }finally{  If(br!=null){  Br.close()  }**}** |

* **The problem in this approach are as below.**

1. Compulsory programmer is require to close resources inside finally block it increases complexity of programing.
2. We have to write finally block compulsory and hence it increases length of the code reduces readability.
3. To overcome above problems sun people introduced try with resources in 1.7 version
4. The main advantages of try with resources is whatever resources we opened as the part of try block will be closed automatically once the control reaches end of try block either normally or abnormally and hence, we are not require to close resources explicitly hence complexity of programming will be reduced.
5. We are not require to write finally block so that length of the code will be reduced and readability will be improved.

e.g

|  |
| --- |
| Try(BufferReader Br= new BufferReader(new FileReader(“a.txt”))){  //use br based on our requirement  Br will be closed automatically once control reaches to the end of try block by normally of abnormally.  }catch(IOException e){  //handling code  }} |

* Conclusions 1:
* We can declare multiple resources but these resources should be separated with semi colon.

Try(r1,r2,r3,r4){

}

e.g

|  |
| --- |
| Try(FileWrite fw= new FileWriter( “a.txt”); FileReader fr= new FileReader( “b.txt”);){  //use fw and fr based on our requirement  Br will be closed automatically once controle reaches to the end of try block by normally of abnormally.  }catch(IOException e){  //handling code  }} |

* All resources should be autoClosable resources.
* A resource is said to be autocloseble if and only if corresponding class implements autoClosable interface
* All IO , database and network related resources are already implemented autoClosable interface.
* Being a programmer we are not require to do anything just we should aware the point.
* AutoClosable interface came in 1.7 version and it contain only one method “public void close()”
* Conclusions 2:
* All resource reference variable are implicitly final and hence within the try block we cannot perform reassignment otherwise we will get compile time error.

|  |
| --- |
| Try(BufferReader br= new BufferReader(new FileReader(“a.txt”))){  br = new BufferReader(new FileReader(“b.txt”))  }catch(IOException e){  //handling code  }}  // compile time error : autoClosable resource br may not be assigned |

* Conclusions 3:
* Until 1.6 version try should be associated with either catch or finally but from 1.7 vision onward we can take only try with resource without catch or finally.

e.g

try(r){

}

* The main advantage of try with resources is we are not require to write finally block explicitly because we are not require to close resources explicitly hence 1.6 version finally block is just like hero but 1.7 version onward it is dummy and becomes zero.

1. **Multi-catch block:**

* Until 1.6 version even though multiple different exception having same handling code for every exception type we have to write a separate catch block it increases length of the code and reduces readability.

e.g

|  |
| --- |
| Try{  SOP(“hello”);  }catch(Arithmetic e){  e.printStackTrace()  } catch(IOException e){  e.printStackTrace()  } catch(NullPointerException e){  SOP(e.getMassage());  } catch(InterruptedException e){  SOP(e.getMassage());  } |

* To over come this problem sun people introduced multicatch block in 1.7 version.
* According to this we can write a single catch block that can handle multiple different type of exceptions

|  |
| --- |
| Try{  SOP(“hello”);  } catch(Arithmetic| IOException e){  e.printStackTrace()  } catch(NullPointerException| InterruptedException e){  SOP(e.getMassage());} |

* The main advantage of this approach is length of the code will be reduced and readability will be improved.

e.g

|  |
| --- |
| Try{  SOP(10/0);  String s=null;  SOP(s.length());  } catch(Arithmetic| NullPointerException e){  e.printStackTrace();  } |

* In the above example whether raised exception is either ArithmeticException or NullPointerEception the same catch block can listen.
* In multi catch block there should not be any relation between exception types(either child to parent or parent to child or same type) otherwise compile time error

e.g

|  |
| --- |
| Try{  SOP(10/0);  } catch(ArithmeticException| Exception e){  e.printStackTrace();  }  // Compile time error : alternatives in a multicatch statement can not be related by subclassing. |

* **Exception propagation :**
* Inside a method if an exception raised and the if we are not handling that exception then exception object will be propogated to called then caller method is responsible handle that exception this process is called exception propogation.
* Read throwing exception :
* We can use this approach to convert one exception type to another exception type

|  |
| --- |
| Try{  SOP(10/0);  } catch(ArithmeticException e){  Throw new NullPointerException();  } |

* **Difference between throw and throws keyword**

| **Sr. No.** | **Key** | **throw** | **throws** |
| --- | --- | --- | --- |
| 1 | Definition | Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code. | Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code. |
| 2 | Internal implementation | Internally throw is implemented as it is allowed to throw only single exception at a time i.e we cannot throw multiple exception with throw keyword. | On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws keyword is used. |
| 3 | Type of exception | With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw. | On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name. |
| 4 | Syntax | Syntax wise throw keyword is followed by the instance variable. | On other hand syntax wise throws keyword is followed by exception class names. |
| 5 | Declaration | In order to use throw keyword we should know that throw keyword is used within the method. | On other hand throws keyword is used with the method signature. |