**Content:**

1. **Introduction.**
2. **Runtime Stack Mechanism.**
3. **Default exception handling in java.**
4. **Exception Hierarchy.**
5. **Checked Exception Vs Un checked Exception**
6. **Customized exception handling by using try catch.**
7. **Control flow in try catch.**
8. **Method to print exception information.**
9. **Try with multiple catch blocks.**
10. **Finally block.**
11. **Difference between final, finally finalize.**
12. **Control flow in try-catch-finally.**
13. **Control flow in nested try-catch-finally.**
14. **Various possible combinations of try-catch-finally.**
15. **throw keyword.**
16. **throws keyword.**
17. **Exception handling keywords summary.**
18. **Various possible compile time errors in exception handling.**
19. **Customized or user defined exception.**
20. **Top- 10 exceptions**
21. **1.7 version enhancement.**
    1. **Try with resources.**
    2. **Multi-catch block.**
22. **Introduction:**

An unwanted, un-expected event that disturbs the normal flow of the program is called exception.

Ex: Tyre Puncture exception, sleeping exception, file not found exception etc.

**Example 1:** You have to pick up one bus at 7 AM in the morning, so to get the bus at 7 AP the flow of action is.

1. You will have to get up by 5 AM
2. You will have to leave your house by 6 AM
3. You will have to get bus by 7 AM

Now suppose while coming from home to get the bus at bus stand your bike tyre got punctured in the middle of the way and now in the very early in the morning you will not get bike repair shop and you will have to go back to your home and thus you won’t get bus at 7 AM and thus it disturbed the normal flow of the planned action.

**Example 2:** Suppose you have to attend one class in the interview at 7 AM and your house is very far from the interview location so to attend the interview at 7 AP the flow of action is.

1. You will have to wake up by 4 AM in the morning.
2. You will have to leave the house by 5 AM.
3. And you reached by sharp 7 AM to attend the class.
4. Now class is started that will run for 3 hours.

Now suppose during attending the class after 15 min your start sleeping then in this case your sleeping is disturbing you to get the knowledge in the class, in other way it disturbed you to complete your flow of action as defined above.

Hence in both of the example an unwanted and un-excepted event disturbs to complete the flow of action, this is nothing but an exception.

**Example 3:** Suppose you have to access one file which is located remotely in the London and when you try to access the file, file is not available in the London server. Then in that case you will get fileNotFound exception.

**Q: is it recommended to handle exception or not?**

A: Yes highly recommended to handle exception so that if anything happens in middle of the something then we have to handle the exception in such a way that program could not be interrupted or terminated in middle and we could not miss anything.

**Example:**  Suppose we have to retrieve the data from any database then to retrieve the data we have to fallow the fallowing steps.

1. Open or create the database connection.
2. Retrieve or reading the data from the database using sql statement.
3. Close the database connection.

Now suppose database connected successfully and while retrieving the data using sql statement then we got one sql exception. Now the third activity (close the database) cannot be completed just because of this sql exception. Now suppose we have opened 10 database connection and just because of this sql exception we could not close all the 10 database connection and suppose if this database supporting only 10 connection then only then while creating a new connection 11th connection , the database will not support at all and database server will be down.

So if we have not handled the exception then such type of unwanted event will occur and which will terminate your program without completing the whole action. So to avoid such type of scenario

We need to handle the exception. If exception found then program should be terminated gracefully.

“It is highly recommended to handle exceptions and the main objective of EH is graceful termination of the program.”

**Objective or purpose of Exception Handling:**

It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of the program.

**Concept of handling exception (meaning of Exception Handling☺:**

**Example:**

Suppose while coming to attend the class, your bike got punctured then here you have tree option.

1. Return back to your home as you bike got punctured (In this option you are going to miss your class so here you are not handling the exception).
2. Stand your bike somewhere and continue to come to attend the class room (In this option after attending the class may be you could have lost your bike. So here also you are missing)
3. In the third you will put your bike in nearby stand or in your relative house and take an auto to come for attending the class (In this option neither you are missing your class not your bike so here you are handling exception or situation gracefully). So defining an alternative way to continue our program is nothing but the exception handling, here we are not repairing the exception we are defining an alternative way to continue our program.

So as we can see that the 3rd option is correct because here we are not missing anything (Bike or Class). So the meaning of exception handling is to defining an alternative way to continue our program without missing anything. EH does not mean repairing the exception, instead of repairing we have to provide an alternative way to continue or programnormally without missing anything.

For Ex: Our programming requirement is to read data from remote file locating at London. Then at run time if London file is not available then 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

**package** com.ibm.testspring;

**import** com.ibm.wps.spa.data.atom.exceptions.FileNotFoundException;

**public class** TestException {

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

**try** {

// Read data from file located at London

} **catch** (FileNotFoundException e) {

//Use Local files and continues rest of program normally

}

}

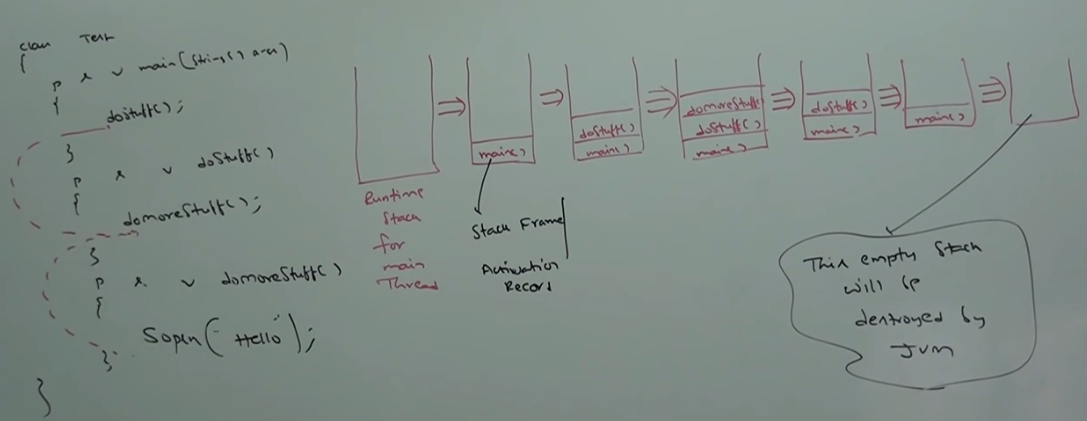
}

1. **Runtime Stack Mechanism:**

For every thread JVM will create a Runtime Stack. Each and every method called performed by that thread will be stored in the corresponding stack.

Each Entry in 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 the thread.

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** TestException {  **publicstaticvoid** main(String[]args)  { *doStuff*(); }  **privatestaticvoid** doStuff()  {*doMoreStuff*(); }  **privatestaticvoid** doMoreStuff() { System.*out*.println(“Hello”);}  }  doMoreStuff()  doStuff()  Main() | In this example here if we observe in stack trace the JVM is going to create one separate run time stack for main thread   1. The first method called by main thread is main () method so the first entry in the run time stack will be of main () method. 2. Main () internally calls doStuff () method so the second entry in the run time stack will be of doStuff () method. 3. doStuff () method internally calls doMoreStuff () method so the third entry in the runtime stack will of doMoreStuff (). 4. Within the doMoreStuff () we have one statement (Hello) which is will be printed on the console. 5. As soon as doMoreStuff () gets executed the controller move to doStuff () method from where doMoreStuff () is being called and the corresponding entry of doMoreStuff () will be removed by JVM from Runtime stack. 6. As soon as doStuff () gets executed the controller move to main () method from where doStuff () is being called and the corresponding entry of doStuff () will be removed by JVM. 7. Finally main () gets executed and finally the corresponding entry of main () will be removed from the Runtime stack by JVM and finally stack will be empty. 8. Finally the empty stack will be removed by JVM as shown in the below image |



1. **Default Exception Handling:**

In the last example we didn’t get any exception, everything goes fine but if something goes wrong then how will handle.

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** TestException {  **publicstaticvoid** main(String[]args)  {*doStuff*();}  **privatestaticvoid** doStuff()  {*doMoreStuff*();}  **privatestaticvoid** doMoreStuff() {System.*out*.println(10/0);}  }  doMoreStuff ()  doStuff ()  Main() | In this example here if we observe in stack trace then JVM is going to create one separate run time stack for main thread   1. The first method called by main thread is main () method so the first entry in the run time stack will be of main () method. 2. Main () internally calls doStuff () method so the second entry in the run time stack will be of doStuff () method. 3. doStuff () method internally calls doMoreStuff () method so the third entry in the runtime stack will of doMoreStuff (). 4. Within the doMoreStuff () we have one statement (10/0) which is an arithmetic exception, it means something goes wrong now in this case default exception comes into picture .Detail explanation is given below. |

**Example:**

* Let us assume two people on bike are travelling from HYD to VJW. Now after crossing place, he takes left turn and immediately a heavy truck hits the bike and people on bike fall on the road and they both have no any momentum in their body.
* Now here if we observer in the planned action something goes wrong i.e. unexpected, unwanted accident happened in other word an exception occurs.
* Now the immediate action will be to call 108. Now the question arises who will be going to call 108 and 100 so the answer is the place where this accident happed there nearby people will be responsible to call 108 and 100.
* Now suppose after 15 to 20 min 100 Police man came the first activity of these police man will be to identify their identity to belong to which location like contact number etc.
* In the same way in the above program main calls doStuff, which interns calls doMoreStuff and within the doMoreStuff one accident i.e. exception occurs so the method where this accidents / exception happens, will be responsible to create an exception object by including Name , Description and Location of exception where it occurs. In other word.....
* Inside a method if any exception occurs then the method in which it is raised is responsible to create exception object by including the fallowing information.

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

* After creating exception object method handovers the object to the JVM.
* JVM will check whether the method contains any exception handling code or not. If the method does not contain exception handling code, then JVM terminates that method abnormally and remove corresponding entry from the stack.
* After that JVM identifies caller method and checks whether caller method contains any handling code or not. If the caller method does not contain handling code then JVM terminates the caller method abnormally and removes the corresponding entry from the stack.
* This process will be continued until main () method and if the main () method also does contain handling code then JVM terminates main () method also abnormally and removes corresponding entry from the stack.
* Now JVM handovers responsibility of exception handling to default exception handler, which is the part of JVM only.
* Default exception handler prints exception information in the fallowing format and terminates program abnormally.

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

**Explanations with Example**

**package** com.ibm.testspring;

**publicclass** TestException {

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

d*oStuff* ();

}

**Privatestaticvoid**doStuff () {

*doMoreStuff* ();

}

**Privatestaticvoid**doMoreStuff () {

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

}

}

* In the above example JVM first will check the method where exception occurs is having the exception handling code or not if they don’t then immediately JVM terminate this program abnormally from that place without executing remaining code and remove the corresponding entry of this method doMoreStuff() from run time stack
* After that JVM will identify from where this doMoreStuff () is being called. As in the program it is being called in doStuff () method. Now immediately the control will go to doStuff () method. Here again JVM will check that whether doStuff () as a caller of doMoreStuff () is having the exception handling code or not and here again this method does not have any exception handling code so the JVM will terminate the program abnormally from that place without executing remaining code and remove the corresponding entry of this method doStuff () from run time stack.
* This process will be continue till main () method because doStuff () method is being called from main (). So the controller now goes to main (). Here again the JVM will check whether main() is having exception handling code or not and again this method does not contain any exception handling code , so the JVM will terminate the program abnormally from that place without executing remaining code and remove the corresponding entry of this method main() from run time stack.
* Now the question arises that who is the caller of main () method. Answer is, JVM only is caller of main (). So finally responsibility comes to JVM only to handle the exception. To handle this type of situation exception JVM is going to maintain one assistant “Default Exception Handler”. Now JVM will call the Default exception handler and will explain the situation that one exception occurs in the program but no one handle the exception so please handle the exception. Now immediately JVM will hand over the object having the information of Name, Description and Location of exception where it occurs.
* Now Default Exception Handler just prints the information in the console in the given below format and terminates the program abnormally.

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

at com.ibm.testspring.TestException.doMoreStuff(TestException.java:14)

atcom.ibm.testspring.TestException.doStuff(TestException.java:10)

atcom.ibm.testspring.TestException.main(TestException.java:6)

Note: Finally when all the entries will be removed from stack then empty stack will be removed by the JVM

|  |  |
| --- | --- |
|  |  |

(Terminate the program and remove entry from Stack) – Printing the exception information on console

So as we can see that a complete stack trace of exception will be printed by Default Exception Handler which makes the debugging easy.

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** TestException {  **publicstaticvoid** main(String[] args) {d*oStuff*();  System.*out*.println(10/0);  }  **Privatestaticvoid** doStuff() {  *doMoreStuff*();  System.*out*.println(“Hi”);  }  **Privatestaticvoid** doMoreStuff() {  System.*out*.println(“Hello”);  }  } | Out Put :  hello  Hi  Exception in thread "main" java.lang.ArithmeticException: divide by zero  at com.ibm.testspring.TestException.main(TestException.java:7)  **Note:**  In a program if any one method terminates abnormally then program termination is abnormal termination. If all methods terminated normally then only program termination is normal termination. |

So this all about default exception handling when we don’t use any try catch block to handle the exception.

But if we want that our program should not execute abnormally then in that case we use our customized exception handling using try-catch block.

1. **Exception Hierarchy.**

Throwable class access roots for java exception hierarchy. Throwable class defines two child classes

1. Exception
2. Error
3. **Exception:**

Most of the times exceptions are caused by our program and these are recoverable. For example if our programming requirements are to read data from remote file locating at London. At run time if remote file is not available then we will get run time exception saying FileNotFound exception. If FileNotFound exception occurs, we can provide local file and continue rest of the program normally.

**import** com.ibm.wps.spa.data.atom.exceptions.FileNotFoundException;

**publicclass** TestException {

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

**try** {

// Read data from file located at London

} **catch** (FileNotFoundException e) {

//Use Local files and continues rest of program normally

}

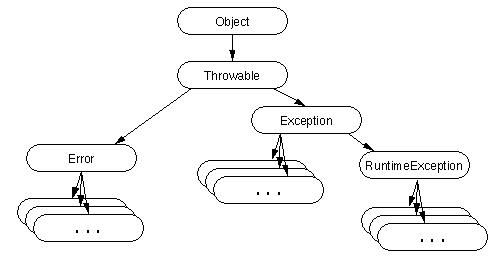
}

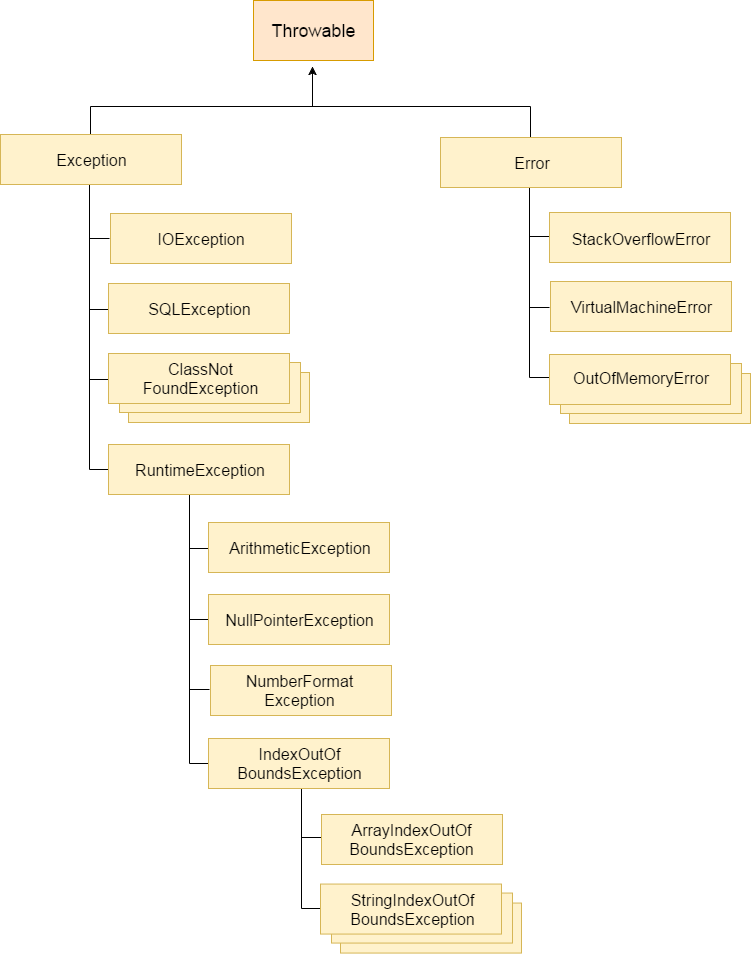
}

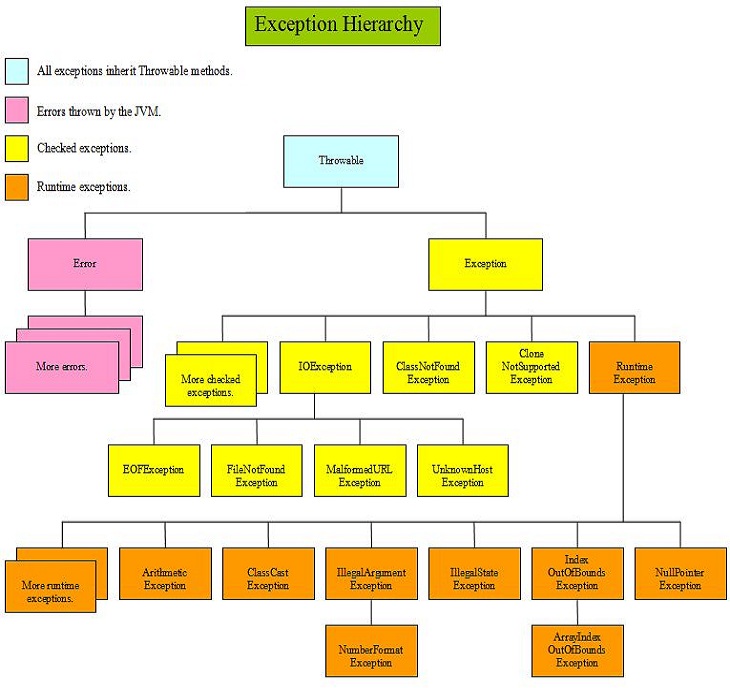
1. **Errors:**

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

For Example: if OutOfMemory occurs, being a program we can’t do anything and the program will be terminated abnormally. System or server admin responsible to increase heap memory







1. **Checked Exception Vs Un checked Exception:**

The exceptions which are checked by compiler for smooth execution of the program are called checked exceptions. Example **HaalTicketMissingException, PenNotWorkingException, FileNotFoundException** etc.

In our program if there is a chance of rising checked exception then compulsory we should handle the checked exception (Either by try-catch or by throws Keyword) otherwise we will get compile time error.

The exceptions which are not checked by compiler whether programmer handling or not, such type of exception are called unchecked exceptions.

Example: **ArithmatixException, BompBlastException** etc.

**Note:** Whether it is checked or unchecked every exception occurs at run time only, there is no chance of occurring any exception at compile time. Compile time exceptions are nothing but syntactical errors.

**Note:** Runtime Exception and its child classes, Error and its child classes are un-checked (Unexpected). Accept these, remaining are checked

1. Example: **HallTicketMissingException- Vs - BombBlasException**:

Let us assume you have to attain on exam so before leaving your house you definitely have to check the fallowing things so that you can completed your examination smoothly.

* Hall Ticket or Admit card.
* Pen, pencil, sharpener etc.

So here whatever you are checking before leaving your house will be treated as compile time check so that you can completed your examination (run time activity) very smoothly. These checks are called compile time exceptions.

Now suppose during examination one accident happened like Bomb Blast or some major accident which interrupted your examination will be considered as run time exceptions because run time exception

1. Example: **PetrolChckedException- Vs-TyreGoTPuncturedException**.

Let us assume you have to go for long drive so before leaving your house or before start your journey you will have to check the fallowing things so that your journey or long drive can be done smoothly.

* Petrol Check
* Stepney Check

So here whatever you are checking before leaving your house will be treated as compile time check so that you can completed your examination (run time activity) very smoothly.

Now suppose during the journey your tyre got punctured which interrupted your journey will be considered as run time exception

1. Example**: FileNotFoundException**

|  |  |
| --- | --- |
| **package** com;  **Import** java.io.\*;  **Publicclass** CompileTimeExcption {  **Publicstaticvoid** main(String[] args) {  **PrintWriter pw = newPrintWriter("abc.txt");**  pw.println("Hello");  }  } | When we try to run this program we will get compile time exception FileNotFoundException because here compiler don’t know that that abc.txt file is already available or not.  Exception in thread "main" java.lang.Error: Unresolved compilation problem:  Unhandled exception type FileNotFoundExceptionat com.RunTimeStack.main(RunTimeStack.java:7) |
| **package** com;  **Import** java.io.\*;  **Publicclass** CompileTimeExcption {  **Publicstaticvoid** main(String[] args)  {  PrintWriter pw;  **try** {pw = **new** PrintWriter("abc.txt");  pw.println("Hello");  } **catch** (FileNotFoundException e) {  System.*out*.println("exception caught");  e.printStackTrace();  }  }  } | **package** com;  **Import** java.io.\*;  **Publicclass** CompileTimeExcption {  **Publicstaticvoid** main(String[] args) **throws** FileNotFoundException {  PrintWriter pw = **new** PrintWriter("abc.txt");  pw.println("Hello");    }  }  **Note: To Resolve the compile time exception we need to use try-catch or need use throws key word to handle compile time exception.** |

1. **Fully Checked Vs partially checked**

A checked exception is said to be fully checked if and only if all its child classes are also checked. For Example IOException, InturrptedExceptions.

A checked is said to be partially checked if and only if some of its child classes are un-checked. For Example Exceptions, Throwable

Note: The only possible partially checked exception in java are.

1- Exception

2. Throwable

Note: Here the Throwable and Exception classes are the combination of fully checked and un-checked exception that is why these two classes are called partially checked.

**Example of partially checked and fully checked:**

Assume you are going for shopping in one Mall with your kids, then while entering inside the Mall security will check you properly but he will not check the kid because to check the kid is not mandatory so this case will considered as **partially checked**.

Now let assume you are going to airport with our kid then here security will check you as well as your kid i.e. here they are checked fully this case will be considered as **fully checked.**

Q: Describe the behaviour of fallowing exceptions.

1. IOException - Checked Exception (Fully)
2. RunTimeException. - Unchecked Exception
3. InterruptedException. - checked Exception (Fully)
4. Error. - Unchecked Exception
5. Throwable. - Checked(Partially)
6. ArithmeticException. - Unchecked Exception
7. NullPointerException - Unchecked Exception
8. Exception. - Checked(Partially)
9. FileNotFoundException - Checked Exception(Fully)

----------------------------------------------------------Theory End-----------------------------------------------------------------

1. **Customized Exception Handling By Using Try-Catch Block**

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

Syntax: **try {**

**// Risky Code has to be written inside try**

**} catch (Exception e) {**

**// Handling Code has to be written inside catch**

**}**

|  |  |
| --- | --- |
| **Publicclass** CompileTimeExcption {  **Publicstaticvoid** main(String[] args) **throws** NullPointerException{  System.*out*.println("Hi No exception");  System.*out*.println(10/0);  System.*out*.println("Handled the exceptions");  }  }  }  Without try-catch :  And with try-catch: 🡪 | **Publicclass** CompileTimeExcption {  **Publicstaticvoid** main(String[] args){  System.*out*.println("Hi No exception");  **try** {  **//Risky Code has to be written inside try**  System.*out*.println("before excetion");  System.*out*.println(10/0);  System.*out*.println("after excetion ");  } **catch** (Exception e) {  **// Handling Code has to be written inside catch**  System.*out*.println("Inside catch block");  System.*out*.println(10/2);  System.*out*.println**("Handled the exceptions"**);  }  System.*out*.println("Exit");  }  } |
| Output: Abnormal termination of the program. | Output: Graceful Termination of program , without missing any thing:  ----------------------------------------------------------  Hi No exception  before exception  Inside catch block  5  Handled the exceptions  Exit |

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

: **try {**

**// Risky Code has to be written inside try**

**Statement 1**

**Statement 2**

**Statement 3**

**} catch (Exception e) {**

**// Handling Code has to be written inside catch**

**Statement 4**

**}**

**Statement 5**

**Case1-If there is no exception then order of execution:**

Statement 1, 2, 3 and 5: Normal termination

**Case2-if exception raised at statement 2 and corresponding catch block match then order of execution**

**Statement 1, and 4, 5: Normal termination:**With the try block where the exception occurs from there exception code will not be execute even though we have handled the code in catch block. Only risky code should be in try and the length of the code inside try as much low as we can

**Case3-** If an exception raised at statement 2 and corresponding catch block not matched then order of execution.

**Statement 1, abnormal termination**.

**Case4-** If an exception raised at statement 4 - i.e. in catch block . Or statement 5 then it is always abnormal termination. Because for exception

**Note 1- :** Within the try block if anywhere an exception raised then from there rest of the try block won’t be executed even though 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.

**Note2-:**In addition to try block there may be a chance of raising an exception inside catch and finally blocks because inside catch and finally we are writing java code. If any statement with is not part of try block and raises exception then it is always abnormal termination.

1. **Method to print exception information.**

Throwable class defines the fallowing methods to print exception information

|  |  |
| --- | --- |
| Method | Printable Format |
| e.printStackTrace(); | Name of exception : Description of Exception: stack trace |
| e.toString() | Name of exception : Description of Exception: |
| e.getMessage() | Description of Exception Like “Divide by zero” |

|  |  |
| --- | --- |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (Exception e) {  e.printStackTrace();  }  }  } | Output Will be: (Name of exception : Description of Exception: stack trace)  java.lang.ArithmeticException:  divide by zero  at com.CompileTimeExcption.main  (CompileTimeExcption.java:6) |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (Exception e) {  System.*out*.println(e.toString());  }  }  } | Output will be :( Name of exception: Description of Exception :)  java.lang.ArithmeticException:  divide by zero |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (Exception e) {  System.*out*.println(e.getMessage());  }  }  } | Output will be (Description)  divide by zero |

1. **Try With Multiple Catch Blocks**

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

|  |  |
| --- | --- |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (Exception e) {  System.*out*.println(e.getMessage());  }  }  }  **Worst Programming practice**. Here for every different type of exception we will get same result so it not recommended. While other side we have included every possible catch block so that we can get the result or message according to caught exception. | **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  //Risky Code  System.*out*.println(10/0);  } **catch** (ArithmeticException e) {  System.*out*.println(“Perform alternative arithmetic operation”);  }  } **catch** (SQLException e) {  System.*out*.println(“Use MySQL DB instead of Oracle DB”);  }  } **catch** (FileNotFoundException e) {  System.*out*.println(“Use Local file instead of remote file”);  }  } **catch** (Exception e) {  System.*out*.println(e.getMessage());  }  }  }**// Best Programming practice** |

**Some Loop Holes**

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

|  |  |  |
| --- | --- | --- |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (Exception e) {  System.*out*.println(e.getMessage());  }  } **catch** (ArithmeticException e) {  System.*out*.println(“Perform alternative arithmetic operation”);  }  }  }  **Note:** Here we will get compile time error saying java.lang.ArithmeticException exception has already been caught.  Here child Exception Handling ArithmeticException catch block should come first. Here the child class job is played by its parent class. Hence child class is not required that is why throwing compile time error. | **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (ArithmeticException e) {  System.*out*.println(“Perform alternative arithmetic operation”);  }  } **catch** (Exception e) {  System.*out*.println(e.getMessage());  }  }  }  **Note:** Here we won’t get any exception because the order of exception is correct i.e. child class has been used first and then parent class is being used. | **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  System.*out*.println(10/0);  } **catch** (ArithmeticException e) {  System.*out*.println(e.getMessage());  }  } **catch** (ArithmeticException e) {  System.*out*.println(“Perform alternative arithmetic operation”);  }  }  }  **Note:** Here we will get compile time error saying ArithmeticException has already been caught.  Here we are using same type of exception handling code twice and this job has already been done so another same type of exception code is not required. We can’t declare two catch block for the same exception |

1. **Difference Between Final , Finally and Finalize**

|  |  |  |
| --- | --- | --- |
| **Final** | **Finally** | **Finalize** |
| 1. It is a modifier and is applicable for classes, methods and variables. 2. **If a class declared as final** then we can’t extend that class i.e. we can’t create child class for that class i.e. Inheritance is not possible for final classes. 3. **If a Method declared as final** then we can override that method in the child class. 4. **If a variable declared as final** then we can’t perform re-assignment for that variable. | 1. Finally is block always associated with try catch to maintain clean-up code 2. The speciality of finally block is it will executed always irrespective of whether exception is raised or not and whether handled or not handled   **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **try** {  //Risky Code  } **catch** (Exception e) {  //Handling Code  }  **finally**{  // Clean up code  }  }  } | 1. Finalize is a method always invoked by garbage collector just before destroying an object to perform clean-up activities. 2. Once finalize () method completes, immediately y garbage collector destroys that object. 3. finalize() method is responsible to perform clean-up activities related to object .i.e. whatever resources associated with the object will be de-allocated before destroying an object by using finalize() method |

**Note:**

finally block is responsible to perform clean-up activities related to try block i.e. whatever resources we opened at the part of try block will be closed inside finally block .

Whereas finalize() method is responsible to perform clean-up activities related to object .i.e. whatever resources associated with the object will be de-allocated before destroying an object by using finalize() method

1. **Various possible combination of try-catch-finally**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **}**  **Valid** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **} catch(y e){**  // Handling Code  **}**  **Valid**  **Two different catch block** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **} catch(x e){**  // Handling Code  **}**  **invalid**  **Two same catch block** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **} finally{**  //clean-up Code  **}**  **Valid** | **Try{**  // Risky Code  **}**  **finally{**  //clean-up Code  **}**  **Valid** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **}**  **Try{**  // Risky Code  **}Valid**  **finally**{  //clean-up Code  **}** |
| **Try{**  // Risky Code  **}**  **CE: Try without catch**  **invalid** | **catch(x e){**  // Handling Code  **}**  **CE: catch without try**  **invalid** | **finally{**  //clean-up Code  **}**  **CE: catch without try**  **invalid** | **Try{**  // Risky Code  **}finally{**  //clean-up Code  } **catch(x e){**  // Handling Code  **}**  **invalid**  **CE: catch without try** | **Try{**  // Risky Code  **}**  **SOP(“Hello);**  **catch(x e){**  // Handling Code  **}**  **invalid**  **CE: catch without try** |  |
| **Try{**  // Risky Code  **}**  **SOP(“Hello);**  **catch(x e){**  // Handling Code  **}**  **invalid**  **CE: catch without try**  **In Valid** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **}**  **SOP(“Hello);**  **catch(x e){**  // Handling Code  **}**  **invalid**  **CE: catch without try** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **}**  **SOP (“Hello);**  **finally{**  // Handling Code  **}**  **invalid**  **CE: finally without try** | **Try{**  **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **}**  **}catch(x e){**  // Handling Code  **}**  **Valid** | **Try{**  **Try{**  // Risky Code  **}**  **}catch(x e){**  // Handling Code  **}**  **Invalid**  **CE: catch without try** | **Try{**  **Try{**  // Risky Code  **}finally{**  // Handling Code  **}**  **}catch(x e){**  // Handling Code  **}**  Valid |
| **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **Try{**  // Risky Code  **}finally{**  // Handling Code  **}**  **}**  Valid | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **finally{**  // Handling Code  **}**  **}**  Invalid  **CE: finally without try** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **Try{**  // Risky Code  **}finally{**  // Handling Code  **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **}**  **}**  **}**  **Valid** | **Try{**  // Risky Code  **}catch(x e){**  // Handling Code  **} finally{**  //clean-up Code  **finally{**  //clean-up Code  **}**  **}**  Invalid  **CE: finally without try** | **Try**  **SOP(“hi”)**  **catch(x e){**  // Handling Code  **}**  **invalid curly brace is needed in try block** | **Try{**  **SOP(“hi”)**  **}catch(x e SOP(“hi”)**  **invalid curly brace is needed in catch block** |

**Note:**

1. In try-catch-finally block order Is important
2. Whenever we are writing try then compulsory we should write either catch or finally block otherwise we will get compile time error .i.e. try without catch or finally is invalid.
3. Whenever we are writing catch block compulsory try block must be required i.e. catch without try is invalid.
4. Whenever we are writing finally block compulsory we should write try block i.e. finally without try is invalid.
5. Inside try catch finally block we can declare try catch finally blocks i.e. nesting of try catch finally is valid and allowed.
6. For try catch and finally blocks curly braces are mandatory.
7. **Difference between throw and throws**

**throw Statement / Keyword / class:**

|  |  |
| --- | --- |
|  | The throw concept is very similar to throwing a ball and another side someone is there to catch. In the similar way here **programmer throw an exception object** using **throw** keyword and another side JVM catches this exception object.  Sometimes programmer creates an exception object and handover it to the JVM programmatically, for that **throw** key word is required. |

Example:

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **public class** TestException {  **public static void** main(String[] args) {  System.*out*.println(10/0);  }  In this case main method is responsible to create exception object and handover to the JVM. | Here in this programme an Exception( ArithmeticException ) is raised and an Exception object gets created by the method where this exception is raised and handover this object to JVM for default exception handling and terminates the program abnormally and default handler Exception print the exception on the console This all things happens internally (implicitly). Now if the programmer wants to create an exception explicitly: See Next Example. |
| **package** com;  **import**java.io.IOException;  **publicclass** ThrowExample {  **publicstaticvoid** main(String[] args) {  System.*out*.println(**new** ArithmeticException("/ by zero created by arun").toString());  **thrownew** ArithmeticException("/ by zero Explicitly");  }} | OUTPUT:  java.lang.ArithmeticException: / by zero created by arun  Exception in thread "main" java.lang.ArithmeticException: / by zero Explicitly at com.ThrowExample.main(ThrowExample.java:6)  Here in this example programmer created an ArithmatixException by using (**new**ArithmeticException ()) because ArithmeticException is a java class and after that we use the **throw** keyword to handover the exception to JVM manually.  **Note:** In this example also we have not written any exception handling code so the JVM will terminate the program abnormally and default handler exception will print the exception detail on console. |
| **new** ArithmeticException ("/by zero Arun"): | This will create an AE and the argument will be printing on the console. |
| **throw** | handover our created object to the JVM manually |

**Note:**The result of both program is exactly same i.e. in the both program JVM terminates the program abnormally but difference between them is in first program everything happens internally (implicitly), while in next program, programmer created an exception object explicitly and then handover this object to the JVM manually (explicitly)

Best use of throw key word is for user defined exception or customized exceptions

**Definition:**

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

|  |
| --- |
| **Syntax:thrownew** ArithmeticException ("/ by zero explicitly"); |

Hence the main objective of the throw key word is to handover our created exception object to JVM manually. Using throw programmer can create user-defined customized exception to wherever is required.

|  |  |
| --- | --- |
| Example: Suppose we have to withdraw 10000 rupees form ATM and in my account have less than 10000 then in this situation we can use customized user defined exception to handle this case. | **package** com;  **import**java.io.IOException;  **publicclass** ThrowExample {  **publicstaticvoid** main(String[] args) {  System.*out*.println(**“Welcome to Bank”**);  **thrownew**InsufficientFundException();  }} |

**Example of User defined or customized exception Pending........**

**---------------------------------------------------------------Case 1- ------------------------------------------------------------**

**Case 1: throw e: If e refers null then we will get null pointer exception (NPE)**

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** ThrowExample {  **static** ArithmeticException *ae* = **new** ArithmeticException();  **publicstaticvoid** main(String[] args) {  **throw***ae*;  }  } | Here in this example we have already created exception object and then throwing ,Will get runtime ArithmeticException:  Exception in thread "main" java.lang.ArithmeticExceptionat com.ibm.testspring.ThrowExample.<clinit>(ThrowExample.java:4)at java.lang.J9VMInternals.initializeImpl(Native Method)at java.lang.J9VMInternals.initialize(J9VMInternals.java:233) |
| **package** com.ibm.testspring;  **publicclass**ThrowExample {  **static** ArithmeticException *e*;  **publicstaticvoid** main(String[] args) {  **throw***e*;  }  } | In this example we did not create exception object just we have declared object and then throwing the same, We will get Null Pointer Exception. Because for static variable the default value is null so declared object will be null and we are throwing null so we also will get NPE.  Exception in thread "main" java.lang.NullPointerExceptionat com.ibm.testspring.ThrowExample.main(ThrowExample.java:13) |

**--------------------------------------------------------------------Case 2 --------------------------------------------------------------**

**Case 2:After throw statement we are not allowed to write any statement directly otherwise we will get compile time error saying “unreachable statement”**

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** ThrowExample {  **publicstaticvoid** main(String[] args) {  System.*out*.println(10/0);  }  } | Here we will get Run time ArithmeticException saying / by zero |
| **package** com.ibm.testspring;  **publicclass** ThrowExample {  **publicstaticvoid** main(String[] args) {  **thrownew** ArithmeticException("/ by zero");  System.*out*.println("Hello");  }  }  **CE : unreachable code atsysout** | But here as per case 2 we will get compile time exception or error, because After throw statement we cannot write any statement directly. We can write in if –else condition  If (condition) (**thrownew** ArithmeticException("/ by zero");}  Else{ System.*out*.println("Hello");} |

**--------------------------------------------------------------------Case 3 --------------------------------------------------------------**

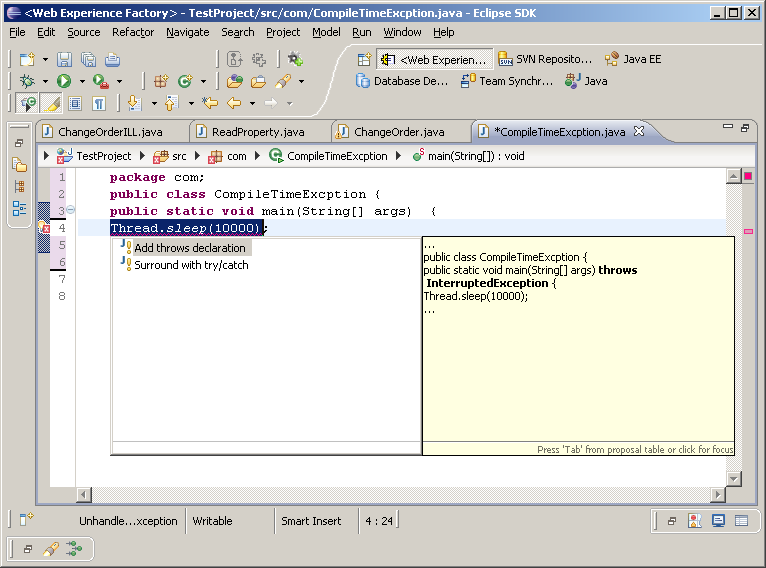
**Case 3**We can use throw key word only for Throwable types (Exception , RunTimeException) ,but if we are trying to use for normal java objects we will get compile time error saying incompatible types

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** ThrowExample {  **publicstaticvoid** main(String[] args) {  **thrownew** ThrowExample();  System.*out*.println("Hello");  }  } | Here in this example we will get compile time exception or error saying incompatible types because the class to which object we are throwing is not Throwable type class like Exception, RunTimeException.  Exception in thread "main" java.lang.Error: Unresolved compilation problem:  No exception of type ThrowExample can be thrown; **an exception type must be a subclass of Throwable** |
| **package** com.ibm.testspring;  **publicclass**ThrowExample**extends** RuntimeException {  **publicstaticvoid** main(String[] args) {  **thrownew** ThrowExample();  }  } | To resolve the above compile time exception or error we have to make this class **Throwable type. To make it Throwable type we have extends Throwable class RuntimeException.**  **But we will get run time exception** |

**Throws key word:**

In our program if there is a possibility of rising checked exception then compulsory we should handle that checked exception otherwise we will get compile time error saying “ un reported exception xyz must be caught or declared to be thrown” .

|  |  |
| --- | --- |
| **Example1**  **package** com;  **Import** java.io.\*;  **Publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **PrintWriter pw = newPrintWriter("abc.txt");**  pw.println("Hello");  }  }  There is a chance of checked exception and we did not handle soFileNotFoundExceptioncompile time error we will get. | When we try to run this program we will get compile time exception FileNotFoundException because here compiler doesn’t know that that abc.txt file is already available or not.  Exception in thread "main" java.lang.Error: Unresolved compilation problem:  Unhandled exception type FileNotFoundExceptionat com.RunTimeStack.main(RunTimeStack.java:7) |
| **Example2**  **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) {  **Thread.sleep(10000);**  }  } | Here in this program we are making sleep of main thread , so if main thread is in sleep then there may be a chance of other thread to be interrupted as these threads are dependent on main thread so we will get compile time checked exception and since we are handling the exception so we will get compile time exception saying “ un reported exception java.lang.inturrputedException must be caught or declared to be thrown” |



We can handle this compile time error by using the fallowing two ways.

|  |  |
| --- | --- |
| **First Way: By using try catch.** | **Second Way: By using throws Keyword** |
| **package com;**  **publicclass CompileTimeExcption {**  **publicstaticvoid main(String[] args) {**  **try {**  **Thread.*sleep*(10000);**  **System.*out*.println("Hiiiiiiiiii");**  **}catch (InterruptedException e) {**  **// TODO Auto-generated catch block**  **e.printStackTrace();**  **}**  **}**  **}**  **Now the program will be executing and main thread will go to sleep till 10 sec and after 10 sec next line will be executing.** | **package com;**  **publicclass CompileTimeExcption {**  **publicstaticvoid main(String[] args) throws InterruptedException {**  **Thread.*sleep*(10000);**  **}**  **}**  **Here we are handling the checked or compile time exception by using throws keyword. The throws keyword doing nothing but it just delegate the handling responsibility to the caller of main method JVM or some other method from where the method is being called. It means throws does not do any handle any checked exception acidity** |
| **package com;**  **import java.io.FileNotFoundException;**  **import java.io.PrintWriter;**  **publicclass CompileTimeExcption {**  **publicstaticvoid main(String[] args) {**  **try {**  **Thread.*sleep*(10000);**  **} catch (InterruptedException e1) {**  **e1.printStackTrace();**  **}**  **PrintWriter pw;**  **try {**  **pw = new PrintWriter("D:\\Arun\\abc.txt");**  **pw.println("Hello");**  **} catch (FileNotFoundException e) {**  **e.printStackTrace();**  **}**  **System.*out*.println("HIIIIIIIII");**  **}}**  **Handling both exception using try catch block** | **package com;**  **import java.io.FileNotFoundException;**  **import java.io.PrintWriter;**  **publicclass CompileTimeExcption {**  **publicstaticvoid main(String[] args) throws*InterruptedException, FileNotFoundException* {**  **Thread.*sleep*(10000);**  **PrintWriter pw = new PrintWriter("D:\\Arun folder\\Study\\abc.txt");**  **pw.println("Hello");**  **System.*out*.println("HIIIIIIIII");**  **}**  **}**  **Throws two exceptions as shown above.** |
| **package com;**  **import java.io.FileNotFoundException;**  **import java.io.PrintWriter;**  **importjava.io.\*;**  **publicclass CompileTimeExcption {**  **publicstaticvoid main(String[] args) throws InterruptedException {**  **Thread.*sleep*(10000);**  **PrintWriter pw;**  **try {**  **pw = new PrintWriter("D:\\Arun folder\\Study\\abc.txt");**  **pw.println("Hello");**  **} catch (FileNotFoundException e) {**  **// TODO Auto-generated catch block**  **e.printStackTrace();**  **}System.*out*.println("HIIIIIIIII");**  **}}**  **InterruptedException is handled by using throws keyword and FileNotFoundException is handled by try catch block.** | |

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **publicclass** TestException {  **publicstaticvoid** main(String[] args) {d*oStuff*();  System.*out*.println(10/0);  }  **Privatestaticvoid** doStuff() {  *doMoreStuff*();  System.*out*.println(“Hi”);  }  **Privatestaticvoid** doMoreStuff() {  Thread.sleep(10000);  }} | **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) **throws** InterruptedException {  *doStuff*();}  **privatestaticvoid** doStuff() **throws** InterruptedException {  *doMoreStuff*();  System.*out*.println("Hello");  }  **Privatestaticvoid** doMoreStuff() **throws** InterruptedException {Thread.*sleep*(10000);}  } |
| Here in the above program in the doMoreStuff () method we are calling Thread.sleep (10000).So there may be chance of InterruptedException checked exception. So handle we have written the program above in the second box   1. **Here first of all exception will rise in** doMoreStuff () so to handle this checked exception we have used **throws** InterruptedException i.e. doMoreStuff () method just delegates the exception object to its caller method doStuff () to handle the exception. | 1. **Now JVM will check in** doStuff () method and it founds that this method also delegates the checked exception to its caller method main () to handle the exception. 2. **Now JVM will check in** main() and here also JVM found that main delegates the checked exception to its caller which is the nothing but JVM only so finally JVM will call default exception handler to handle the exception and finally program compile time error will be gone. |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) **throws** InterruptedException {  *doStuff*();}  **privatestaticvoid** doStuff(){  *doMoreStuff*();  System.*out*.println("Hello");  }  **Privatestaticvoid** doMoreStuff()  {Thread.*sleep*(10000);}  } | **Note: Given program will compile or not?**  **Ans: No**  As we can see that exception will rise indoMoreStuff () method so the JVM will check whether handle the exception or not. Now doMoreStuff()method will tell the JVM that exception is already handled in main() method, then JVM will tell I know exception is handled in main but you have not delegate your exception object to your caller method doStuff() and main() so I will not compile your code and hence we will get compile time checked exception CE : java. Lang. InterruptedException has to caught or thrown in doMoreStuff () method. So in the left program if we remove at least throws statement then code won’t compile. |
| **package** com;  **publicclass** CompileTimeExcption {  **publicstaticvoid** main(String[] args) **throws** InterruptedException {  Thread.*sleep*(10000);  *doStuff*();  }  **Privatestaticvoid** doStuff() {  *doMoreStuff*();  System.*out*.println("Hello");  }  **Privatestaticvoid** doMoreStuff() {  System.*out*.println("Hello");}} | **Note: Given program will compile or not?**  **Ans: Yes**  **Because the** InterruptedException will rise in main () method **a**nd exception object is delegating to its caller JVM so JVM will compile the program without any issue. |

**Main Purpose or conclusion of throws keyword**

1. 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.
2. **Throws** keyword required only of checked exception and uses of throws keyword for un-checked exception, there is no use or no impact.
3. **Throws** keyword required only to convince compiler and uses of throws keyword does not prevent abnormal termination of the program because while using throws keyword ,it just delegate the responsibility of handling exception to JVM and JVM uses the default handler exception to handle the exception and in result abnormal termination happens.

**Note:** It is recommended to use try-catch instead of using throws keyword

|  |  |
| --- | --- |
| **Case 1-** We can use throws keyword for methods and constructors but not for classes because method and constructer can be called by another method of constructor but class cannot be called. So at class level throws keyword is invalid | **Class Test throws Exception** { // Invalid  **Test () throws Exception** { // Valid  System.*out*.println ("Hello");  }  **Privatestaticvoid** doMoreStuff ()**throws Exception**  { // Valid  System.*out*.println ("Hello");  }} |
| **Case 2 -** We can use throws keyword only for Throwable types. If we are trying to use for normal java classes (i.e. not Throwable classes like Exception etch) then we will get compile time error saying incompatible types. But if we extend the Throwable or any RunTimeException class in the main Test class then we won’t get any compile time error. Here in the right side example we have extends Exception Class which is the child of Throwable Class. Hence now we can use throws keyword at constructor Test()  **Case3-** | **Class Test extends Exception** { // valid  **Test () throws Exception** { // Valid  System.*out*.println ("Hello");  }  **Privatestaticvoid** doMoreStuff ()**throws Exception**  { // Valid  System.*out*.println ("Hello");  }}  ---------------OR-------------------  **package** com;  **publicclass**CompileTimeExcption**extends** Throwable{  **void** Test () **throws** Exception {// Valid  System.*out*.println ("Hello");}  **Privatestaticvoid**doMoreStuff ()**throws** Exception { // Valid  System.*out*.println ("Hello");}} |
| **Class Test{**  **public static void** main(String[] args){  **throw new Exception();**  **}**  **}**  **Here we will get compile time exception saying CE: unreported exception java.lang.Exception must be caught or declared to be thrown :**  **Here we get the exception because Exception is a checked Exception that is why it occurs at compile time** | **Class Test{**  **public static void** main(String[] args){  **throw new Exception();**  **}**  **}**  **Here we will get run time exception saying RD: exception in thread main java.lang.Errorat Test main();**  **Here we get the exception because Exception is a un checked Exception that is why it occurs at run time** |

**Special Note:**

Within the try block if there is no chance of raising an exception the n we can’t write catch block for that exception otherwise we will get compile time error saying “Exception XYZ is never thrown in body of corresponding try statement. **But this rule is applicable only for fully checked exceptions.**

**Exception Handling Key words summary: Difference between throw vs throws.**

We have covered up till five key word related the Exception Handling (try,catch,finally, throw and throws)

|  |  |
| --- | --- |
| Try | To maintain risky code |
| catch | To maintain raised exception handling code |
| finally | To maintain clean up code like closing the resource which are opened in try block |
| throw | To handover our created exception objects to the JVM manually. |
| throws | To delegate the responsibility of exception handling to the caller. |

**Various possible compile time error in exception handling**

1. Unreported exception xyz ; must be caught or declared to be thrown
2. Exception xyz has already been caught.
3. Exception xyz is never thrown in body of corresponding try statement or block.
4. Unreachable statement.
5. Incompatible types: found in Test: required java.lang. Throwable.
6. Try without catch or finally.
7. Catch without try.
8. Finally without try.

**Customized or user defined exceptions**

Sometimes to meet programming requirements, we can define our own exceptions. Such types of exception are called customized or user defined exceptions.

Example: TooYoungException, TooOldYException, InSufficientFundsExceptions etc.

**Note1- :** throw key word is best suitable for user defined or customized exception but not for pre-defined exceptions.

**Note2-:** It is highly recommended to define customized exceptions as unchecked i.e. we have to extendRunTimeException but not Exception.

**Note3-:**super(s) is required to make our description available to default exception handler.

**Example of Defining Customized Exception:**

|  |  |
| --- | --- |
| **package** com;  **publicclass**TooYoungException**extends** RuntimeException{  **public** TooYoungException(String s) {  **super**(s);  }  } | **package** com;  **publicclass**TooOldYException**extends** RuntimeException{  **public** TooOldYException(String s) {  **super**(s);  }  } |
| **How to use or call our own customized exception:**  **package com;**  **publicclass CustExceptions {**  **publicstaticvoid main(String[] args) {**  **int age = Integer.*parseInt*("90");**  **//int age = Integer.*parseInt*("14");**  **//int age = Integer.*parseInt*("18");**  **//int age = Integer.*parseInt*("60");**  **if(age>60){**  **thrownew TooYoungException("Your age is already crossed to get matches ");**  **}elseif (age<18){**  **thrownew TooOldYException("you are too young to get matches ");**  **}else{**  **System.*out*.println("Soon You will get your match detail from email");**  }}} | |
| **Output : Will Get our customized Exception as shown below**  **When age > 60 means age is equal to 90 then we get the exception like below:**  Exception in thread "main" com.TooYoungException: ***Your age is already crossed to get matches***at com.CustExceptions.main(CustExceptions.java:7) | |
| **Output :**  **When age < 18 means age is equal to 14 then we get the exception like below:**  Exception in thread "main" com.TooOldYException: ***you are too young to get matches***at com.CustExceptions.main(CustExceptions.java:9) | |
| **Output : Will Get our customized Exception as shown below**  **When age is equal to 18 or 60**  Soon You will get your match detail from email | |

**Note1- :** throw key word is best suitable for user defined or customized exception but not for pre-defined exceptionslike (NullPointerException or ArithmeticException).

**Explanation:**

Let us assume above example for X Matrimonial sites where age limit range should be 28 to 60. Now let us assume for Y Matrimonial site where the age limit range should be 21 to 70 then in the if condition instead of 60 we will have to put 70 and instead of 17 we will have to put 18 in another if condition. So here, when our own customized exception has to raised, it depends on our programming condition.

Similarly ArithmeticException has to be raised when condition is divide by zero (/ by 0). And NullPointerException has to be raised when we are performing any operation on NULL. In such type exception JVM will decided when these exception has to raised but in case of customized exception, based on our programming conditions we will have to raise the exception. JVM does not play any role in raising customized exception.

So here we have to decide or responsible to rise an exception based on our programming requirements and to rise this exception compulsory some keyword is required and that is nothing by but (throw) keyword.

**Note2-:** It is highly recommended to define customized exceptions as unchecked i.e. we have to extend **RunTimeException** but not Exception.

**Explanation:**

**If we are throwing unchecked exception the compiler will never check for unchecked exception whether you are handling or not. But if you are using checked exception then compiler will check or ask to handle that exception (either we have to use try-catch block or throws keyword to handle) and this will be meaningless because he programmer is going to raise exception so not required to handle i.e. It is highly recommended to define customized exceptions as unchecked i.e. we have to extend RunTimeException but not Exception. So instead of(unchecked) RunTimeExceptionif we use(Checked) Exceptionthen immediately compiler will object to handle the exception.**

**Note3-:super(s) is required to make our description available to default exception handler. Here is a string that is nothing but the description what we are passing in constructor as an exception.**

**Explanation:**

**As in the above example we can see that one string is getting passed as description to the constructor in the throw call statement. This string value will be passed to defined constructor and further this string value will be passed in super(s):**

**So the question arises here is what is the need of super(s)?**

**Ans:**

**Suppose in the programme in the first if condition if (age>60) then we will get customized exception saying “your age is already crossed to get matches”. And default exception handler will responsible to print this message on the console.**

**Default exception handler, internally will use printStackTrace () method and printStackTrace () method will be available in Throwable class. In other word.**

**TooYoungException is the child class of RuntimeException.RuntimeException is the child class of Exception and Exception is the child class of Throwable where the printStackTrace () method will be available and this method is used by Default Exception Handler to print the message in the console.**

So now one more question is how we will make available this (Description) customized exception to Default Exception Handler.

Ans: We have a chain of throwing this customized exception as given below.

|  |  |
| --- | --- |
| **Exception calling heirarchy** | **Exception throwing heirarchy** |
| **thrownew** TooYoungException (“Description")  **public** TooYoungException(String s)  **super**(s); | **Throwable** (printStackTrace() used by DEH to  print message on the console)  **Exception**– Parent class  **RuntimeException** – Parent class  **public** TooYoungException(String s)  **super**(s);// Exception raised here |

So super(s) is required inside the constructor definition to make our description available to default exception handler.

1. **Top-10 Exceptions in java**

As we know that first of all the exception are divided into two categories.

1. Checked Exception
2. Unchecked Exception

Now question that who is rising these exception so based on the person who is rising an exception, all exceptions are divided into two categories.

1. JVM Exceptions.
2. Programmatic Exceptions.

**JVM Exceptions:**

The Exceptions which are raised automatically by JVM whenever a particular events occurs are called JVM exceptions.

Example: ArithmatixException, NullPointerException etc.

**Programmatic Exceptions OR Customized Exception OR User Defined Exception:**

The exceptions which are raised explicitly either by programmer or by API developer to indicate that something goes wrong are called Programmatic Exceptions.

Example: TooOldYException, IllegalArgumentExceptions etc.

**Note:** There is one more third person is there who rise the exception “API Developer” this is also nothing but programmatic exception: Example: String Class, Thread Class etc. These classes is written by API developer only and write customized exception only using **throw**keyword.

**Example: Exception raised by API Developer (set priority example in Thread Class example)**

As we know in java thread have some priority (1 to 10) and these priority is set by one mothed present in thread class (setPriority (int priority)

|  |  |
| --- | --- |
| **package** com;  **publicclass** SetPriority {  **publicstaticvoid** main(String[] args) {  Thread t = **new** Thread();  t.setPriority(5);  }  } | **package** com;  **publicclass** SetPriority {  **publicstaticvoid** main(String[] args) {  Thread t = **new** Thread();  t.setPriority(15);  }  } |
| **As priority valid range is from 1 to 10 so will not get any exception as we have given 5 priority** | Exception in thread "main" java.lang.IllegalArgumentException  java.lang.Thread.setPriority(Thread.java:825) |

**Q: Now Question arise that how they implement internally?**

|  |  |
| --- | --- |
| **package** com;  **publicclass** SetPriority {  **publicfinalstaticint***MAX\_PRIORITY*=10;  **publicfinalstaticint***MIN\_PRIORITY*=1;  **publicstaticvoid** main(String[] args) {  Thread t = **new** Thread();  t.setPriority(15);  }  **publicfinalvoid** setPriority(**int** newPriority){  **if**(newPriority>*MAX\_PRIORITY*  || newPriority<*MIN\_PRIORITY*){  **thrownew** IllegalAccessException();  }  }  } | **package** com;  **publicclass** IllegalAccessException **extends** RuntimeException{  **public** IllegalAccessException() {  **super**();  } |
| **OUT PUT:**Exception in thread "main" java.lang.IllegalArgumentException  at java.lang.Thread.setPriority(Thread.java:825)  at com.SetPriority.main(SetPriority.java:7) | |

**Top Ten Exceptions:**

1. **ArrayIndexOutOfBoundsException:**

* It is the child class of **RuntimeException** and hence it is **unchecked**.
* **Raised by JVM**automatically whenever we are trying to access array element **with out of range index**.
* Example:

|  |  |
| --- | --- |
| **package** com;  **publicclass**ArrayIndexOutOfBoundException {  **publicstaticvoid** main(String[] args) {  **int** [] x = **newint** [4];  System.*out*.println(x[1]);  }  } | **package** com;  **publicclass** ArrayIndexOutOfBoundException {  **publicstaticvoid** main(String[] args) {  **int** [] x = **newint** [4];  System.*out*.println(x[5]);  }} |
| **Out Put : 0**  **Correct Output because we are trying to get element within the range (1- 3).** | Exception in thread "main" java.lang. ArrayIndexOutOfBoundsException: Array index out of range: 5 |

1. **NullPointerException:**

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

|  |
| --- |
| **package** com;  **publicclass**NullPointerExampler {  **publicstaticvoid** main(String[] args) {  String s = **null**;  System.*out*.println(s.length());  }  } |
| Exception in thread "main" java.lang. NullPointerExceptionat com.ArrayIndexOutOfBoundExceptions.main(NullPointerExampler.java:5) |

1. **ClassCastException:**

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

|  |  |
| --- | --- |
| **package** com;  **publicclass** ClassCastExceptions {  **publicstaticvoid** main(String[] args) {  String str = **new** String("Arun");  System.*out*.println(str.length());  Object obj = (Object)str;  System.*out*.println(obj);  }  }**As we can typecast child class String to its parent class Object that is why wedon’t get ClassCastException** | **package** com;  **publicclass** ClassCastExceptions {  **publicstaticvoid** main(String[] args) {  Object obj = **new** Object();  String s = **new** String("Arun");  String s1 = (String)obj;  System.*out*.println(s1.length());  }  }  **As we cannot typecast parent class object to child class object String that is why we get ClassCastException** |
| **Output: 4**  **Arun**  **Here we have converted child class object to parent class object** | Exception in thread "main" java.lang.ClassCastException: java.lang.Object incompatible with java.lang.String at com.ClassCastExceptions.main(ClassCastExceptions.java:6) |
| **package** com;  **publicclass** ClassCastExceptions {  **publicstaticvoid** main(String[] args) {  Object obj = **new** String("Arun");  String s1 = (String)obj;  System.*out*.println(s1.length());  }  } | Output : 4 :  Here internal object is child class object that is why type casting is allowed but if internal object is parent object then typecasting is not allowed. |

1. **StackOverFlowError:**

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

As we know that for every thread JVM is going to create on RunTime Stack. In the below program we have only one thread i.e. main thread. Now each and every method call entry performed by main thread will be stored in run time stack. Main thread first of all will call main () method so first entry in the RTS will for main () method. Main () internally call m1 () method so the next entry will for m1 () method. M1 () method internally calls m2 () method so the next entry in RTS will for m2 () method. M2 () internally calls M1 () method. Again M1 () calls M2 () then M2 () calls M1 () and so on........ After a certain point the stack will be overflow and that time we will get runtime exception **StackOverFlowError**

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **public** **class** StackOverFlowExample {  **public** **static** **void** main(String[] args)  *m1*();  }  **private** **static** **void** m1() {  *m2*();  }  **private** **static** **void** m2() {  *m1*();  }  } | -  -  -  M2()  M1 ()  Main() |

1. **NoClassDefFoundError:**

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

**Example: if we try to run: java test:**

**If test .class file not available then we will get run time exception saying NoClassDefFoundError: test**

1. **ExceptionInInitializerError:**

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

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **public** **class** StackOverFlowExample {  **static** **int** *x* =10/0;  } | Trying to execute static variable assignment  **ExceptionInInitializerError** division by zero |
| **package** com.ibm.testspring;  **public** **class** StackOverFlowExample {  **static** {  String s= **null**; System.*out*.println(s.length());  }  } | Trying to execute static blocks and in this block we are trying to execute one operation on null string so.  **ExceptionInInitializerError** caused by NullPointerException |

1. **IllegalArgumentExceptions:**

* It is the child class of **RunTimeException** and hence it is **unchecked**.
* **It is raised by explicitly** either by programmer or by API developer to indicate that a method has been invoked with illegal argument.

|  |  |
| --- | --- |
| **package** com;  **public class** SetPriority {  **public final static int** *MAX\_PRIORITY*=10;  **public final static int** *MIN\_PRIORITY*=1;  **public static void** main(String[] args) {  Thread t = **new** Thread();  t.setPriority(15);  }  **Public final void** setPriority(**int** newPriority){  **if**(newPriority>*MAX\_PRIORITY*  || newPriority<*MIN\_PRIORITY*){  **throw new** IllegalAccessException();  }  }} | **OUT PUT:**Exception in thread "main" **java.lang.IllegalArgumentException** at java.lang.Thread.setPriority(Thread.java:825) at com.SetPriority.main(SetPriority.java:7)  The valid range of thread priorities is 1 to 10. But if we are trying set priority with any other values then we will get run time exception saying **IllegalArgumentException**. |

1. **NumberFormatException:**

* It is the direct child class of IllegalArgumentExceptions and IllegalArgumentExceptions is the child class of RuntimeException and hence it is **unchecked**.
* **It is raised by 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.

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **public** **class** NumberFormatExceptionClass  **public** **static** **void** main(String[] args)  **int** i = Integer.*parseInt*("10");  System.*out*.println(i);  }  } | Output: 10:  In the below example we are passing IllegalArgument “ten” which in not in proper format so we will get **NumberFormatException** which is the child class of IllegalArugment |
| **package** com.ibm.testspring;  **public** **class** NumberFormatExceptionClass  **public** **static** **void** main(String[] args)  **int** i = Integer.*parseInt*("ten");  System.*out*.println(i);  }  } | Exception in thread "main" java.lang.**NumberFormatException**: **For input string: "ten"**  Atjava.lang.NumberFormatException.forInputString(NumberFormatException.java:59)at  com.ibm.testspring.**NumberFormatExceptionClass**.main(NumberFormatExceptionClass.java:12) |

1. **IllegalStateException:**

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

**Example: after starting of thread we are not allowed to re-start the same thread once again otherwise we will get runtime exception saying IllegalThreadStateException**

**Behavioural Example:**

Suppose a person born and after 30 years he realizes that he did not get good marks in 10 even though he has B.tech degree that is why he is not getting good job. Now imagine he called Brahma to restart his life once again so that he can complete his 10 properly. So immediately an answer will come from Brahma saying IllegalStateException- that i am responsible to start your life once now i cannot restart your life again

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **public** **class** IllegalStateException {  **public** **static** **void** main(String[] args) {  **int** i = Integer.*parseInt*("10");  System.*out*.println(i);  Thread t = **new** Thread();  **t.start();**  System.*out*.println("Thread started");  **t.start();**  }  } | 10  Thread started  Exception in thread "main" java.lang.**IllegalThreadStateException**: Thread is already started at java.lang.Thread.start(Thread.java:924)  at **com.ibm.testspring.IllegalStateException.main**  (IllegalStateException.java:17) |

In the above example we are trying to start the thread which is already started so we will get **IllegalThreadStateException**

1. **AssertionError**

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

**If x is not greater than 10 then we will get RuntimeException saying assertion error.**

|  |  |
| --- | --- |
| **package** com.ibm.testspring;  **public** **class** NumberFormatExceptionClass {  **public** **static** **void** main(String[] args)  **int** i = Integer.*parseInt*("1");  System.*out*.println(i);  **assert(i>10);**  }  } |  |
| * ArrayIndexOutOfBoundsException: * NullPointerException: * ClassCastException: * StackOverFlowError: * NoClassDefFoundError: * ExceptionInInitializerError: | These exceptions are raised automatically by JVM so these exceptions are **JVM exceptions**. |
| * IllegalArgumentExceptions: * NumberFormatException: * IllegalStateException: * AssertionError | **It is raised by explicitly** either by programmer or by API developer so these exceptions are **programmatic Exceptions** |

1. **1.7 version enhancement**
2. **Try with Resources:**
3. **Multi catch**

Until 1.6 java version whatever resources we opened as a part of try block those we have to close inside finally block

|  |  |
| --- | --- |
| **package** com;  **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.IOException;  **public class** TryWithoutResources {  **public static void** main(String[] args) {  BufferedReader br =**null**;  **try** {  System.*out*.println("inside");  // Read the data present in Input.txt file  **br = new BufferedReader(new FileReader("D:\\My Own Document.txt"));**  System.*out*.println("inside2"+br.toString());  // Use the br based on our requirements.  // Read line by line from the given fi  System.*out*.println(br.readLine());  } **catch** (IOException e) {  // **TODO**: handle exception  e.printStackTrace();  }**finally**{  // Closing the resources opened in try block  **if**(br!=**null**){  **try** {  br.close();  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  }  }  **This simple program until java 1.6 Version** | **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.IOException;  **publicclass** TryWithResource {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  BufferedReader br =**null**;  // Try with resources  **try (br = newBufferedReader (newFileReader ("D:\\My Own Document.txt")))**  {  // use br based on our requirement  // br will be closed automatically once control reaches end of the try block either normally or abnormally  // and programmer is not responsible to close explicitly  }**catch**(IOException e){  e.printStackTrace();  }}}  **Syntax:---------------------------**  // Try with resources  **try(**//**open the database or resources)**  {// use opened resources based on our requirement  // Resources will be closed automatically once control reaches end of the try block either normally or abnormally and programmer is not responsible to close explicitly  }**catch**(IOException e){  e.printStackTrace();  }}}  **Introduced in Java 1.7 Version** |
| **Drawback:** -Whatever the resources we opened, compulsory we have to close in the finally block.  -It is like burden for the programmer to close the resources.  -Finally block is mandatory to close all the resources so it increase the code length and code readability also decreases. | **Advantages: -**Whatever resources we opened in the try block will be closed automatically i.e. once the  -It is not mandatory for the programmer to close the resources in the final block so -it decreases the code length and also improves the code readability. |

1. **We can declare multiple resources but these resources should be separated with (;) semicolon**

**Syntax:**

|  |  |
| --- | --- |
| // Try with resources  **try(R1;R2;R3){**  }**catch**(IOException e){  e.printStackTrace();  }}} | **try (br = new BufferedReader (new FileReader ("D:\\My Own Document. txt");br = new BufferedReader(new FileReader("D:\\My Own Document.txt");R3){**  }**catch**(IOException e){  e.printStackTrace();  }}} |

1. All resources should be AutoClosable resources. A resource is said to be AutoClosable if and only if corresponding class implements java.lang.AutoClosable.
2. All IO, Database and network related resources are already implemented AutoClosable Interface.
3. Being a programmer we are not required to do anything just should aware the point.
4. AutoClosable Interface came in java 1.7 version and it contains only one method close ().

I.e. Public void close ().

1. All resource reference variables are implicitly final and hence within the try block we cannot perform re-assignment, otherwise we will get compile time error.

|  |  |
| --- | --- |
|  | **If we try to run this program then auto-closable resources may not be re-assigned** |

1. **Until 1.6 version try should be associated with either catch or finally but from 1.7 version onwards we can take only try with resources without catch or finally**

|  |  |
| --- | --- |
| Try{  }Catch(){  }  ---------------------------------or------------------------  Try{  }finally{  }  Until java 1.6 version : Valid syntax but only try is invalid  Try{  } | But form java 1.7 version only try with resource is valid syntax:  Try(resources)  {  ……  }  Valid syntax |

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

1. **Multi Catch Block:**

Until 1.6 versions even though multiple different exceptions 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.

|  |  |
| --- | --- |
| **package** com;  **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.FileWriter;  **import** java.io.IOException;  **public class** TryWithoutResources {  **public static void** main(String[] args) **throws** InterruptedException {  BufferedReader br =**null**;  FileWriter fw=**null**;  **try** {  System.*out*.println("inside");  **// Read the data present in Input.txt file**  br = **new** BufferedReader(**new** FileReader("D:\\Demo.txt"));  fw= **new** FileWriter("D:\\Demo1.txt");  fw.write("Hi Arun");  } **catch** (IOException e) {  e.printStackTrace();  }**catch** (ArithmeticException e) {  e.printStackTrace();  } **catch** (NullPointerException e) {  e.getMessage();  } **catch** (InterruptedException e) {  e. getMessage();  }}} | **package** com;  **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.FileWriter;  **import** java.io.IOException;  **public class** TryWithoutResources {  **public static void** main(String[] args) **throws** InterruptedException {  BufferedReader br =**null**;  FileWriter fw=**null**;  **try** {  System.*out*.println("inside");  **// Read the data present in Input.txt file**  br = **new** BufferedReader(**new** FileReader("D:\\Demo.txt"));  fw= **new** FileWriter("D:\\ Demo1.txt");  fw.write("Hi Arun");  }**catch**(**IOException | ArithmeticException** e){  e.printStackTrace();  }**catch**(**NullPointerException** **|InterruptedException** e ){  e. getMessage ();  }  }  } |
| As in the above program we can see that multiple different type of exception having same handling code in the catch block, even though having same handing code, compulsory we have to write separate catch block having the same handling code for each of different exception. We cannot combine those catch block having same handling code. Until java 1.6V we have to write catch block for different exception and hence it is increasing code length and reducing code readability | From Java 1.7v onwards we have the concept of multi catch block. Here in this concept we have to write only one catch block for different exception having the same handling code i.e. those exception having the same handling code in the catch block can be combined in one catch block. |

Until 1.6 versions 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. To overcome this problem sun people introduces multi catch block in 1.7 versions.

According to this we can write a single catch block that can handle multiple different types of exceptions. The main advantage of this approach is length of the code will be reduced and readability will be improved.

**Example:**

|  |  |
| --- | --- |
| **Import** java.io.\*;  class MultiCatchBlock{  **public static void** main(String args[]){  **try**{  System.out.println(10/0);  String name = null;  System.out.println(name.length());  }**catch**(ArithmeticException | NullPointerException e){  e.printStackTrace();  }}} | **Import** java.io.\*;  class MultiCatchBlock{  **public static void** main(String args[]){  **try**{  //System.out.println(10/0);  String name = null;  System.out.println(name.length());  }**catch**(ArithmeticException | NullPointerException e){  e.printStackTrace();  }}} |
| **While executing will get ArithmeticException divide by zero(/ by 0)** | **While executing will get NullPointerException** |

In the above example whether raised exception is either ArithmaticException or NullPointerException, 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 we will get compile time error.

|  |  |
| --- | --- |
| **Import** java.io.\*;  class MultiCatchBlock{  **public static void** main(String args[]){  **try**{  System.out.println(10/0);  String name = null;  System.out.println(name.length());  }**catch**(ArithmeticException | Exception e){  e.printStackTrace();  }}} | Here we will get CE:  Alternatives in a multi – catch statement cannot be related by subclassing (child) |

**Exception Propagation:**

Inside a method if an exception raised and if we are not handling that exception then exception object will be raised in the method where this exception occurs and this exception object will be propagated to caller then caller method is responsible to handle exception. This process is called exception propagation.

***Example***: Refer the Runtime Stack mechanism for understanding the exception propagation

**Re-throwing exception:**

We can use this approach to convert one exception type to another exception type

|  |  |
| --- | --- |
| **Import** java.io.\*;  class MultiCatchBlock{  **public static void** main(String args[]){  **try**{  System.out.println(10/0);  String name = null;  System.out.println(name.length());  }**catch**(ArithmeticException){  **throw** new NullPointerException();  }}} | Here in this example it will throw Arithmetic Exception but in the catch block again we are re-throwing the exception and converting the ArithmeticException into NullPointerException. This concept can be used where we have to convert each exception into one specific exception and based on that we have to do some activity or we have to show any web page to the user: |