**Exception Handling:**

* An unwanted and unexpected event that disturbs the flow of program is called exception handling
* 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, instead it means providing an alternative way to continue rest of the program normally

**Runtime Stack Mechanism:**

For every thread, JVM will create a runtime stack. Each and every method call performed by that thread will be stored in the corresponding thread, each entry in the stack frame or called 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.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  | doMoreStuff |  |  |  |
|  |  | doStuff() | doStuff() | doStuff |  |  |
|  | main() | main() | main() | main() | main() |  |

Runtime Stack for main thread

**Default Exception Handling:**

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.

->Name of Exception

->Description of Exception

->Location at which Exception occurs [Stack Trace]

After creating exception object, method handovers that object to the JVM. JVM will check weather 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 the corresponding entry from stack. Then JVM identifies caller method and checks weather caller method contains any handling code or not. If the caller method doesn’t contain handling code, then JVM terminates that caller method also abnormally and remove corresponding entry from the stack. This process will be continued until main method, and if main method also doesn’t contain handling code, then JVM also terminated the main method and remove corresponding method abnormally and removes corresponding entry from stack. Then JVM handovers exception handling to default Exception Handler which is the part of JVM. Default Exception Handler prints the Exception information in the following format and terminates the program abnormally.

Exception in thread “aaaa” Name of Exception: Description Stack Trace

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

*doStuff*();

}

**public** **static** **void** doStuff() {

*doMoreStuff*();

}

**public** **static** **void** doMoreStuff() {

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

}

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

at ExceptionHandling/DefaultExceptionHandling.Test.doMoreStuff(Test.java:12)

at ExceptionHandling/DefaultExceptionHandling.Test.doStuff(Test.java:9)

at ExceptionHandling/DefaultExceptionHandling.Test.main(Test.java:6)

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

*doStuff*();

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

}

**public** **static** **void** doStuff() {

System.***out***.println("Doing stuff");

*doMoreStuff*();

}

**public** **static** **void** doMoreStuff() {

System.***out***.println("Doing More Stuff");

}

Doing stuff

Exception in thread "main" Doing More Stuff

java.lang.ArithmeticException: / by zero

at ExceptionHandling/DefaultExceptionHandling.Test.main(Test.java:7)

In a program, even if one method terminates abnormally then the program termination is abnormal. If all method terminates normally, then only program termination is normal termination.

**Exception Hierarchy:** Throwable class access root of java exception hierarchy. Throwable class defines two child classes. (i) Exception and (ii) Error

**Exception:** Most of the times, Exceptions are caused by our program and these are recoverable for example: If out programming requirement is to read data from remote file locating at London, at runtime if remote file is not available, then we get runtime exception saying file not found exception. If file not found exception occurs, we can provide local file and continue rest of program normally.

**try** {

//Read data from remote file in a london directory

}

**catch**(FileNotFoundException e) {

//Use local file and continue rest of program normally

}

**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. For example, if out of memory error occur, being programmer we can’t do anything and the program will be terminated abnormally. And the system admin or server admin is responsible to increase heap memory.

**Checked Exceptions vs Unchecked Exceptions**

**Checked Exceptions:** The Exceptions which are checked by the compiler for smooth execution of the program are called checked exceptions. Example: FileNotFoundException

If there is a chance of raising checked exceptions, then compulsory we should handle that checked exception (either by try-catch or throws keyword ),otherwise we will get compile time error.

**Example:**

**public** **static** **void** checkedDemo(){

BufferedWriter bw = **null**;

bw = **new** BufferedWriter(**new** FileWriter("abc.txt"));

bw.write("Hello");

bw.close();

}

Unhandled exception type IOException Test.java/ExceptionHandling/src/ExceptionTest

**Unchecked Exception:** The exceptions which are not checked by compiler weather programmer handling or not, such type of exceptions are called unchecked exceptions.

example: ArithmeticException, NullPointerException.

* **Note:** Weather it is checked or unchecked, every exception occurs at runtime only, there is no chance of occurring any exception yet compile time.
* Runtime exception and its child classes, Error and its child classes are unchecked. Expect this remaining are checked.

**Fully Checked and Partially Checked:**

* A checked exception is said to be fully checked if and only if all its child classes are also checked.
* Example: IOException, InterruptedException
* A checked exception is said to pe partially checked if and only if some of its child classes are unchecked.
* Example: Exception, Throwable
* The only possible partially checked exceptions in java are Exceptions, Throwable.

**Practice: Describe the nature of exceptions as checked or fully checked or partially checked?**

1. **IOException 🡪 Checked (Fully)**
2. **RuntimeException🡪unchecked**
3. **InterruptedException🡪Checked (Fully)**
4. **Error🡪 unchecked**
5. **Throwable🡪Checked (Partially)**
6. **ArithmeticException🡪unchecked**
7. **NullPointerException🡪unchecked**
8. **Exception🡪Checked (Partially)**
9. **FileNotFoundException** 🡪 **Checked (Fully)**

**Customize Exception Handling by using try catch**

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 try block and corresponding handling code, we have to define inside catch block. Without try catch block program may result in abnormal termination.

**try {**

**Risky Code**

**}**

**catch (Exception e){**

**}**

**Control Flow in try catch block**

try {

stat1;

stat2;

stat3;

}

catch (Exception e){

stat1;

}stat5;

**Case 1:** No Exception – 1,2,3,4,5 (Normal termination).

**Case2:** Exception at stat2 and Catch block matched with exception– 1,4,5(Normal termination).

**Case 3:** Exception at stat2 and corresponding catch block not matched –

1, abnormal termination.

**Case 4:** Exception raised at stat4 or stat5, then it is always a abnormal termination.

**Note:**

* Within the try block, if anywhere an exception raised, then rest of the try block won’t be executed even though we handle 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 be a chance of raising an 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.

**Methods to print Exception Information**

Throwable class defines the following methods to print exception information

|  |  |
| --- | --- |
| **METHOD** | **Print Format** |
| **printStackTrace ()** | Name of Exception: Description: Stack Trace |
| **toString ()** | Name of Exception: Description: |
| **getMessage ()** | Description of Exception |

**Example:**

//Methods to print Exception message

**try** {

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

}

**catch**(Exception e) {

System.***out***.println("Name of Exception-description-stack trace of where exception occured-Print Stack trace");

e.printStackTrace();

System.***out***.println("Name of exception and description - e.toString()");

System.***out***.println(e.toString());

System.***out***.println("get Message to print description alone");

System.***out***.println(e.getMessage());

}

**Output:**

Name of Exception-description-stack trace of where exception occured-Print Stack trace

java.lang.ArithmeticException: / by zero

at ExceptionHandling/ExceptionTest.Test.main(Test.java:26)

Name of exception and description - e.toString()

java.lang.ArithmeticException: / by zero

get Message to print description alone

/ by zero

**Best practice to handle exceptions:**

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

**Example:**

**public** **static** **void** multipleCatch(**int** n,String str) {

**try** {

// This code may throw various exceptions

**int**[] numbers = {1, 2, 3};

System.***out***.println(numbers[n]); // This will throw ArrayIndexOutOfBoundsException

System.***out***.println(str+" "+str.length()); // This will throw NullPointerException

} **catch** (ArrayIndexOutOfBoundsException e) {

e.printStackTrace();

} **catch** (NullPointerException e) {

e.printStackTrace();

} **catch** (Exception e) {

e.printStackTrace();

}

}

*multipleCatch*(5, **null**);

java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3

at ExceptionHandling/ExceptionTest.Test.multipleCatch(Test.java:23)

at ExceptionHandling/ExceptionTest.Test.main(Test.java:54)

*multipleCatch*(0,**null**);

1

java.lang.NullPointerException: Cannot invoke "String.length()" because "str" is null

at ExceptionHandling/ExceptionTest.Test.multipleCatch(Test.java:24)

at ExceptionHandling/ExceptionTest.Test.main(Test.java:55)

*multipleCatch*(0,"hello");

1

hello 5

**Conclusion:** With single catch block we can handle only one exception and perform all alternative operations at one place which is not recommended. One should perform alternative operations only if corresponding current operation raised exceptions. Hence when handling with multiple risky statements, you should go for multiple catch blocks.

**Order of Catch blocks:** If try with multiple catch blocks present, then the order of catch blocks is very important. We have to rake child first and then parent. Otherwise we will get compile time error saying -: **Exception xxx has already been caught**

**try** {

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

}

**catch**(Exception e) {}

**catch**(ArrayIndexOutOfBoundsException e) {}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unreachable catch block for ArrayIndexOutOfBoundsException. It is already handled by the catch block for Exception

at ExceptionHandling/ExceptionTest.Test.main(Test.java:63)

**Example 2:**

**try** {

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

}

**catch**(ArrayIndexOutOfBoundsException e) {}

**catch**(ArrayIndexOutOfBoundsException e) {}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unreachable catch block for ArrayIndexOutOfBoundsException. It is already handled by the catch block for ArrayIndexOutOfBoundsException

at ExceptionHandling/ExceptionTest.Test.main(Test.java:63)

**Conclusion:** We can’t declare two catch blocks for the same exception, otherwise we will get a compile time error. That’s why a parent class already handling the child class exception, it can’t get caught at the later catch block as it may already caught in parent catch block if any exception occurred.

**Final, finally, finalize**

**Final:** Final is a modifier applicable for classes, methods and variables.

* If a class declared as final, we can’t extend that class. We can’t create child class for that class. i.e Inheritance is not possible for final classes.
* If a method is final, we can’t override that method in the child class.
* If a variable declared final, we can’t perform reassignment for that variable.

**Finally:** (finally) is a block always associated with try catch to maintain clean up code.

try {

RiskyCode

}

catch {

Handling Code

}

finally{

//cleanup Code

}

The speciality of finally block is it will be executed always irrespective of whether exception is raised or not raised or whether handled or not handled.

**Finalize():**

* finalize is a method always invoked by garbage collector just before destroying an object to perform cleanup activities.
* Once finalize method completes, immediately GarbageCollector destroys that object.

**Conclusion:**

* Finally block is responsible to perform clean up activities related to try block i.e whatever resources we opened at try block will be closed by finally block.
* Whereas finalize () method is responsible to perform clean up activities related to object i.e whatever resources associated will be deallocated before destroying an object by using finalize method.

**Various Possible combinations 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 i.e: try without catch or finally is invalid.

* Whenever we are writing catch block, try block must be required i.e catch without try is invalid.
* Whenever we are writing finally block, compulsory we should write try block. i.e: finaly without try is invalid.
* Inside try catch finally blocks we can declare try catch and finally blocks. i.e: nesting of try catch finally is allowed.
* For try catch finally blocks, curly braces are mandatory.

|  |  |  |
| --- | --- | --- |
| **1** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e){} | VALID |
| **2** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **catch**(Exception e) {} | VALID |
| **3** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **catch**(ArrayIndexOutOfBoundsException e) {} | Invalid  Exception has already been caught |
| **4** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **finally** {} | Valid |
| **5** | **try** {  System.***out***.println("Risky operations");  }  **finally** {} | Valid |
| **6** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {} | Valid |
| **7** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **try** {  System.***out***.println("Risky operations");  }  **finally** {} | Valid |
| **8** | **try** {  System.***out***.println("Risky operations");  } | Invalid  Try without catch or finally error |
| **9** | **catch**(ArrayIndexOutOfBoundsException e) {} | Invalid record higher name |
| **10** | **finally** {} | Invalid  Finally without try |
| **11** | **try** {  System.***out***.println("Risky operations");  }  **finally** {}  **catch**(ArrayIndexOutOfBoundsException e) {} | Invalid  Catch without try |
| **12** | **try** {  System.***out***.println("Risky operations");  }  System.out.println("I will break the bond");  **catch**(ArrayIndexOutOfBoundsException e) {} | Invalid  try without catch or finally  catch without try |
| **13** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  System.out.println("I will break the bond");  **finally** {} | Invalid  finally without try |
| **14** | **try** {  **try** {}  **catch**(Exception e) {}  }  **catch**(ArrayIndexOutOfBoundsException e) {} | Valid |
| **15** | **try** {  **try** {}  }  **catch**(ArrayIndexOutOfBoundsException e) {} | Invalid |
| **16** | **try** {  **try** {}  **finally**{}  }  **catch**(ArrayIndexOutOfBoundsException e) {} | Valid |
| **17** | **try** {}  **catch**(ArrayIndexOutOfBoundsException e) {  **try** {}  **catch**(Exception t) {}  } | Valid |
| **18** | **try** {}  **catch**(Exception e) {  **finally** {}  } | Invalid |
| **19** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **finally** {  **try** {}  **catch**(Exception e) {}  } | Valid |
| **20** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **finally** {}  **finally** {} | Invalid |
| **21** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **finally** {  **finally** {}  } | Invalid |
| **22** | **try**  System.***out***.println("Risky operations");  **catch**(ArrayIndexOutOfBoundsException e) {}  **finally** {} | Invalid |
| **23** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e)  System.***out***.println(e.getMessage());  **finally** {} | Invalid |
| **24** | **try** {  System.***out***.println("Risky operations");  }  **catch**(ArrayIndexOutOfBoundsException e) {}  **finally**  resource.close(); | Invalid |

**Throw, Throws**

**Throw:**

JVM

Programmer

Exception Object

Sometimes we can create Exception object explicitly and handover it to JVM manually. For this we need to use throw keyword.

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

**//Exception in thread main, default exception creation**

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

}

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

**//Creation of exception object explicitly and**

**//handing it over to jvm manually using** throw **keyword**

**throw** **new** ArithmeticException(“message to be printed”);

}

The main purpose of throw keyword is to hand over the exception object to JVM manually. The best use of throws keyword is to create user defined exceptions.

**Case 1:**

If Exception object is null, we will get NullPointerException.

i.e: **static** ArithmeticException *nullAe*;

**static** ArithmeticException ae = **new** ArithmeticException();

**throw** ae;// ArithmeticException

**throw** nullAe;//NullPointerException

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

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

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

//Raises ArithmeticException

**throw** **new** ArithmeticException();

System.***out***.println("Hello");//Unreachable code

//Compile time error

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unreachable code

at ExceptionHandling/ExceptionTest.Test.main(Test.java:41)

**Case 3:** We can use throw keyword only for throwable types. If we are trying to use it for normal java objects, we will get compile time error saying Incompatible types.

**class** CustomException{

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

**throw** **new** CustomException();

}

}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

No exception of type CustomException can be thrown; an exception type must be a subclass of Throwable

**class** CustomException **extends** ArithmeticException{

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

**throw** **new** CustomException();

}

}

Exception in thread "main" ExceptionTest.CustomException

at ExceptionHandling/ExceptionTest.CustomException.main(Test.java:9)

at ExceptionHandling/ExceptionTest.Test.main(Test.java:41)

**Throws Keyword:** In our program, if there is a possibility of raising checked exception, the compulsory we should handle that checked exception. Otherwise, we would get compile time error saying unreported exception xxx must be caught or declared to be thrown.

**public** **static** **void** checkedDemo(){

BufferedWriter bw = **null**;

bw = **new** BufferedWriter(**new** FileWriter("abc.txt"));

bw.write("Hello");

bw.close();

}

Unhandled exception type IOException Test.java/ExceptionHandling/src/ExceptionTest

We can handle this compile time error by two ways

**Way-1:** Using try-catch block

**public** **static** **void** checkedDemo(){

BufferedWriter bw = **null**;

**try** {

bw = **new** BufferedWriter(**new** FileWriter("abc.txt"));

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

bw.write("Hello");

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

bw.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**Way 2:** By using throws keyword

**public** **static** **void** checkedDemo()**throws** Exception {

BufferedWriter bw = **null**;

bw = **new** BufferedWriter(**new** FileWriter("abc.txt"));

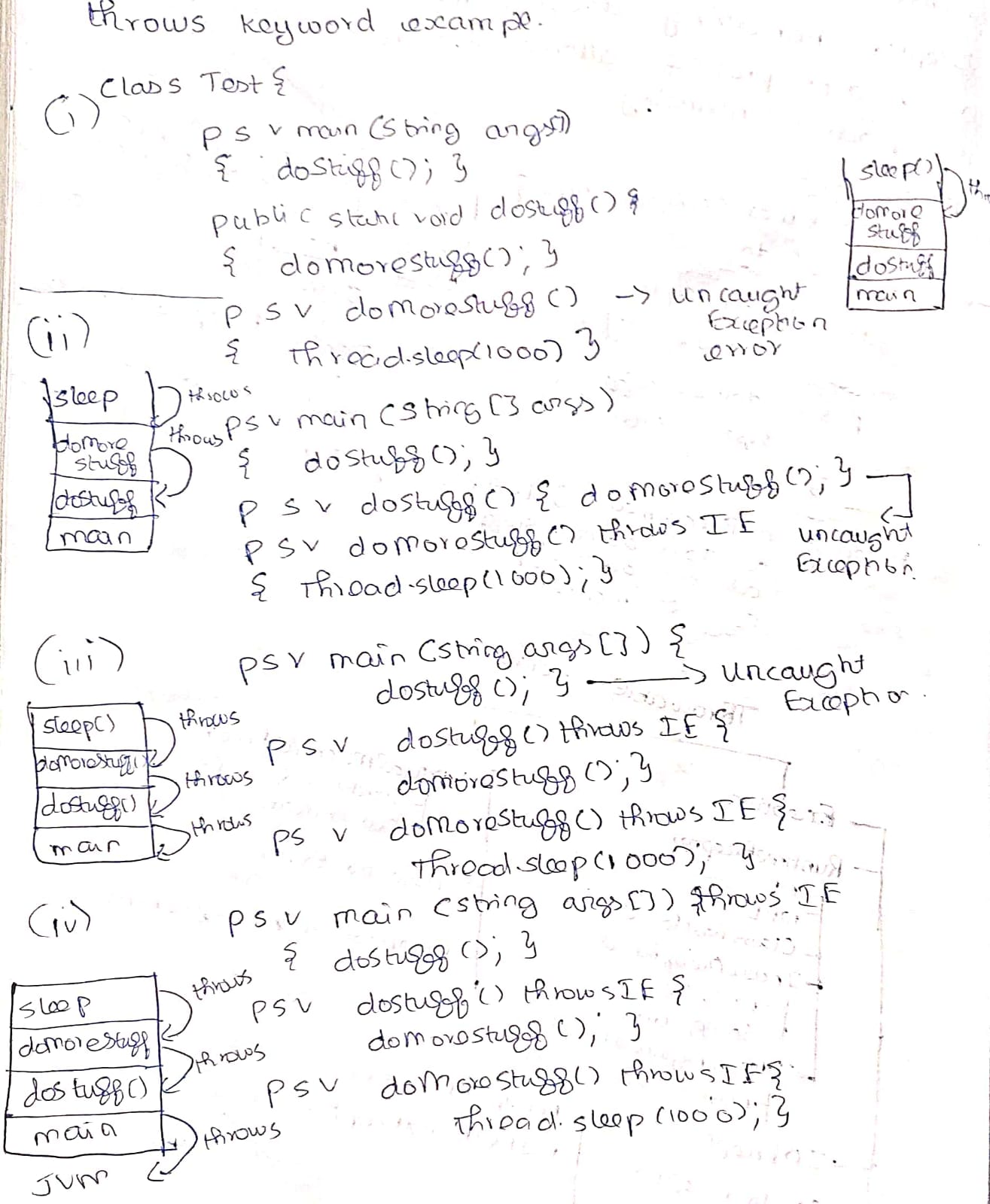
bw.write("Hello");

bw.close();

}

* We can use throws keyword to delegate the 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 is required only for checked exceptions. The usage of throw keyword on unchecked exceptions has no impact.
* Throws keyword require only to convince the compiler and usage of throws keyword doesn’t prevent abnormal termination of the program.

Example:



If we remove atleast one throws keyword from the above program, without delegation, we can’t handle the checked exceptions and it results in compile time error

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

*doStuff*();

}

**public** **static** **void** doStuff() throws InterruptedException {

*doMoreStuff*();

}

**public** **static** **void** doMoreStuff(){//Error uncaught exception

Thread.sleep(1000);

}

Note:

* Throws exception can be used at method level and constructor level, but can’t be used at class level.
* Throws keyword only throws Exception object. The object should be a children of exception class. Trying to use for normal java classes will give incompatible errors

Example:

**class** CustomException{

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

}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

No exception of type CustomException can be thrown; an exception type must be a subclass of Throwable

**class** CustomException **extends** ArithmeticException{

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

}

* Within try block if we can’t raise a exception we cannot catch the corresponding exception in catch block. This rule is applicable only for fully checked exceptions

|  |  |  |
| --- | --- | --- |
| 1. | **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println("Hello");  }  **catch**(Exception e) {}  } | Partially Checked Exception  No error |
| 2 | **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println("Hello");  }  **catch**(ArithmeticException e) {}  } | Unchecked Exception  No error |
| 3 | **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println("Hello");  }  **catch**(IOException e) {}  } | Fully Checked Exception  Error: Exception never thrown unreachable catch block |
| 4 | **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println("Hello");  }  **catch**(InterruptedException e) {}  } | Fully Checked Exception  Error: Exception never thrown unreachable catch block |
| 5 | **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println("Hello");  }  **catch**(Error e) {}  } | Unchecked Exception  No error |
|  | **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println("Hello");  }  **catch**(Exception e) {}  } |  |

**Summary of Exception Handling Keywords:**

**try->** To maintain Risky code

**catch->** To maintains exception handling code

**finally->** To maintain cleanup code

**throw->** To hand over created exception object to the JVM manually

**throws->** To delegate the responsibility of exception handling to caller method or main

**Various Errors expected in Exception Handling:**

* Unreported exception XXX; must be caught or declared to be thrown
* Exception XXX has already been caught
* Exception XXX is never thrown in body corresponding try statement
* Unreachable statement
* Incompatible types
* Try without catch or finally and vice versa

**CUSTOMIZED or USER DEFINED EXCEPTION**

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

**Example:** AgeNotValidException, MailNotValidException, InsufficientBankBalanceException

**package** customizedException;

**class** InvalidUserNameException **extends** RuntimeException{

**public** InvalidUserNameException(String message) {

**super**(message);

}

}

**class** InvalidPasswordException **extends** RuntimeException{

**public** InvalidPasswordException(String message) {

**super**(message);

}

}

**class** LoginDetails {

**private** String username;

**private** String password;

**public** LoginDetails(String username, String password)**throws** RuntimeException{

**this**.setUsername(username);

**this**.setPassword(password);

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) **throws** InvalidUserNameException {

**if**(username.isEmpty()||username.equals(**null**)) {

**throw** **new** InvalidUserNameException("Username cannot be empty");

}

**this**.username = username;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password)**throws** InvalidPasswordException {

**if**(password.isEmpty()||password.equals(**null**)) {

**throw** **new** InvalidPasswordException("Password cannot be empty");

}

**this**.password = password;

}

@Override

**public** String toString() {

**return** "CustomizedDemo1{" +

"username='" + username + '\'' +

", password='" + password + '\'' +

'}';

}

}

**public** **class** CustomizedDemo1{

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

**try** {

LoginDetails login = **new** LoginDetails("user123", "pass123");

System.***out***.println(login);

login.setUsername("newUser");

login.setPassword("newPass");

System.***out***.println(login);

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

**try** {

LoginDetails invalidLogin = **new** LoginDetails("", "password");

} **catch** (InvalidUserNameException | InvalidPasswordException e) {

System.***out***.println(e.getMessage());

}

**try** {

LoginDetails invalidLogin = **new** LoginDetails("username", "");

} **catch** (InvalidUserNameException | InvalidPasswordException e) {

System.***out***.println(e.getMessage());

}

}

}

**Note:**

* Throw keyword is best suitable for user-defined or customized exceptions but not for pre-defined exception.
* It is highly recommended to define customized exceptions as unchecked i.e we have to extend RuntimeException, but not Exception.
* We use super(message) to invoke super class constructor to pass the message description to Default Exception handlers.