OCJA

DURGA SOFTWARE SOLUTIONS

Java Exception Handling

For Certification & Interviews

Exception Handling

Introduction

- An Unwanted Unexpected Event that Disturbs Normal Flow of the Program is Called Exception.
  Eg: SleepingException, TyrePuncheredException, FileNotFoundException Etc…

- It is Highly Recommended to Handle Exceptions.
- The Main Objective of Exception Handling is Graceful Termination of the Program.
  i.e. we should Not Miss anything, we should Not Block any Resource.

Exception Handling:

- Exception Handling doesn't mean repairing an Exception.
- We have to define an Alternative Way to Continue Rest of the Program Normally.
- This Way of defining Alternative is nothing but Exception Handling.
- For Example if Our Programming Requirement is to Read Data from the File locating at London.
- At Runtime 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.

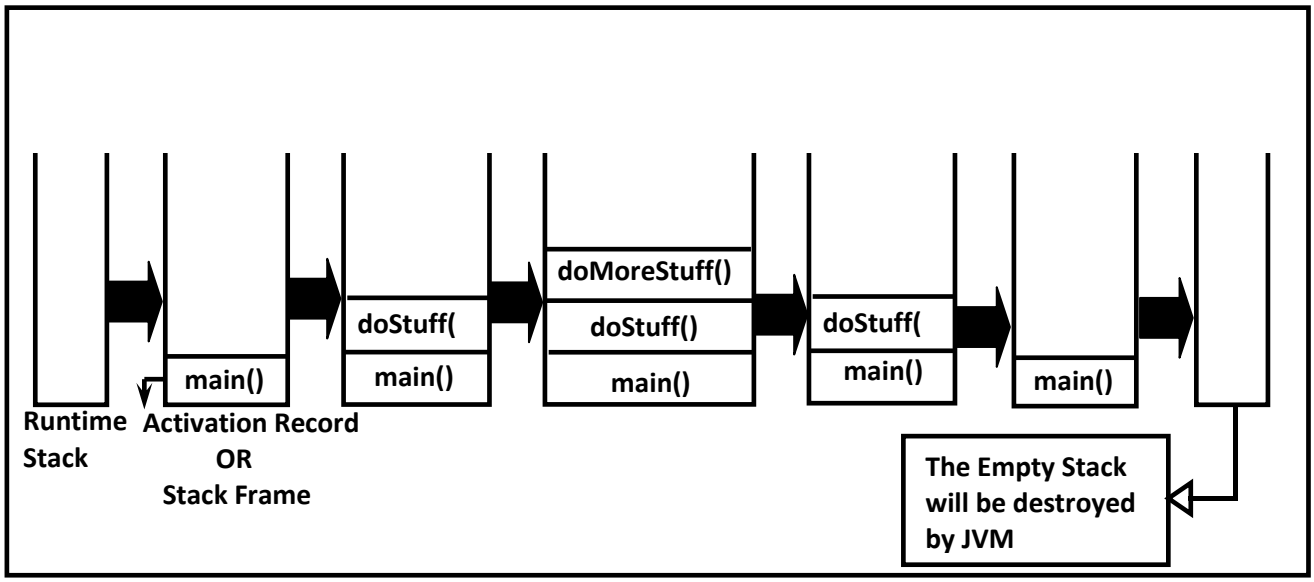
```
try {
        //Read Data from Remote File locating at London
}
catch (FileNotFoundException e) {
        //Use Local File and Continue Rest of the Program Normally
}
```

Q) What is an Exception?
Q) What is the Purpose of Exception Handling?
Q) What is the Meaning of Exception Handling?

Runtime Stack Mechanism

- For Every Thread JVM will Create a Runtime Stack.
- Every Method Call performed by that Thread will be stored in the Corresponding Stack.
- Each Entry in the Stack is Called *Activation Record* OR *Stack Frame*.
- After completing Every Method Execution JVM Removes the Corresponding Entry from the Stack.
- After completing all Method Calls the Stack become Empty and that Empty Stack will be Destroyed by the JVM and the Program will be terminated Normally.

```
class Test {
	public static void main(String[] args) {
		doStuff();
	}
	public static void doStuff() {
		doMoreStuff();
	}
	public static void doMoreStuff() {
		System.out.println("Hello"); //Hello
	}
}
```



Default Exception Handling in Java

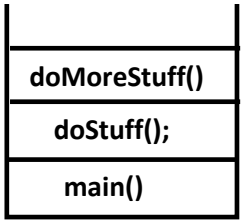
- In Our Java Program Inside a Method if an Exception raised, then that Method is Responsible to Create an Exception Object by including the following Information.
  - Name of Exception
  - Description of Exception
  - Location of Exception (Stack Trace)
- After creating Exception Object Method Handovers that Object to the JVM.
- JVM will Check whether Corresponding Method contain any Exception Handling Code OR Not.
- If the Method doesn't contain any Exception Handling Code then JVM Terminates that Method Abnormally and Removes Corresponding Entry from the Stack.
- JVM will Identify Caller Method and Check whether the Caller Method contain any Exception Handle Code OR Not.
- If Caller Method doesn't contain any Exception Handling Code then JVM Terminates that Caller Method and Removes Corresponding Entry from the Stack.
- This Process will be continued until main().
- If the main() also doesn't contain Exception Handling Code then JVM Terminates main() Abnormally and Removes Corresponding Entry from the Stack.

- Then JVM Handovers the Responsibility of Exception Handling to the *Default Exception Handler*, which is the Part of the JVM.
- Default Exception Handler Just Terminates the Program Abnormally and Prints Exception Information to the Console in the following Format.

Exception in thread: "Xxx" Name of the Exception: Description
Stack Trace

Examples:

```
class Test {
    public static void main(String[] args) {
        doStuff();
    }
    public static void doStuff() {
        doMoreStuff();
    }
    public static void doMoreStuff() {
        System.out.println(10/0);
    }
}
```



Runtime Stack

```
RE: Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Test.doMoreStuff(Test.java:9)
      at Test.doStuff(Test.java:6)
      at Test.main(Test.java:3)
```

```
class Test {
    public static void main(String[] args) {
            doStuff();
    }
    public static void doStuff() {
            doMoreStuff();
            System.out.println(10/0);
    }
    public static void doMoreStuff() {
            System.out.println("Hello");
    }
}
```

```
Hello
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Test.doStuff(Test.java:7)
      at Test.main(Test.java:3)
```

```
class Test {
    public static void main(String[] args) {
                doStuff();
                System.out.println(10/0);
    }
    public static void doStuff() {
                doMoreStuff();
                System.out.println("Hi");
    }
    public static void doMoreStuff() {
                System.out.println("Hello");
    }
}
```

```
Hello
Hi
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:4)
```

Note:

- In Our Program if all Methods Terminated Normally, then Only the Program will be Terminated Normally.
- In Our Program if at least One Method terminates Abnormally then the Program Termination is Abnormal Termination.

Exception Hierarchy

- Throwable Class Acts as Root for Exception Hierarchy.
- Throwable Class contains 2 Child Classes *Exception* and *Error*

Exception: Most of the Cases Exceptions are Caused by Our Program and these are Re-Coverable.

Eg:

- If Our Programming Requirement is to Read Data from the File locating at London.
- At Runtime if London File is Not Available then we get *FileNotFoundException*.
- If FileNotFoundException Occurs we can Provide Local File to Continue Rest of the Program Normally.
- Programmer is Responsible to Recover Exception.

Error:

- Most of the Cases Errors are Not Caused by Our Program and these are Due to Lack of System Resources.
- Errors are Non- Recoverable.

Eg:

- If *OutOfMemoryError* Occurs, being a Programmer we can't do anything and the Program will be terminated Abnormally.
- System Admin OR Server Admin is Responsible to Increase Heep Memory.

Checked Vs Unchecked Exception:

Checked Exceptions:

- The Exceptions which are Checked by the Compiler for Smooth Execution of the Program at Runtime are Called Checked Exceptions.
- Compiler Checks whether we are handling Checked Exceptions OR Not. If we are Not handling then we will get Compile Time Error.

Eg:

HallTicketMissingException, PenNotWorkingException, FileNotFoundException, Etc.

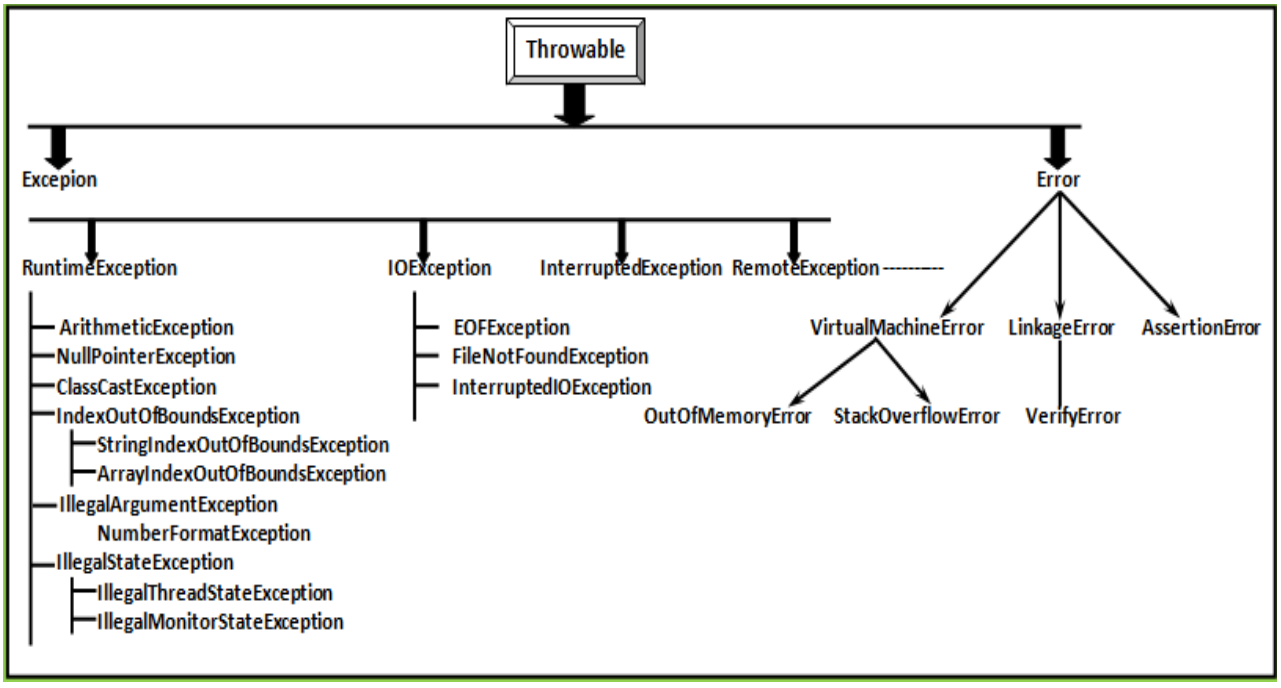
Unchecked Exceptions:

- The Exceptions which are Not Checked by the Compiler are Called Unchecked Exception.
- Compiler won't Check whether we are Handle OR Not Unchecked Exceptions.

Eg: ArithmeticException, NullPointerException, BombBlostException, Etc.

Note:

- Whether the Exception is Checked OR Unchecked Compulsory it will Occur at Runtime Only.
- There is No Chance of occurring any Exception at Compile Time.
- Runtime Exceptions and its Child Classes, Errors and its Child Classes are Unchecked Exceptions Excetp these all remaining are Considered as Checked Exceptions.



Fully Checked Vs Partially Checked:

A Checked Exception is Said to be Fully Checked if and Only if all its Child Classes also Checked.
Eg: IOException, InterruptedException, ServletException, Etc...

A Checked Exception is Said to be Partially Checked if and Only if Some of its Child Classes are Unchecked.
Eg: Throwable, Exception.

Note: The Only Possible Partially Checked Exceptions in Java are

- Throwable
- Exception

Describe the Behaviour of the following Exceptions

1) IOException → Fully Checked
2) RuntimeException → Unchecked
3) InterruptedException → Fully Checked
4) Error → Unchecked
5) Throwable → Partially Checked
6) ArithmeticException → Unchecked
7) NullPointerException → Unchecked
8) Exception → Partially Checked
9) FileNotFoundException → Fully Checked

Customized Exception Handling by using try - catch

- It is Highly Recommended to Handle Exceptions.
- The Code which May Raise an Exception is Called *Risky Code*.
- We have to Place that Risky Code Inside *try* Block and Corresponding Handle Code Inside catch Block.

```
try
{
        //Risky Code
}
catch (Exception e)
{
        //Handling Code
}
```

Without try - catch

```
class Test {
    public static void main(String[] args) {
        System.out.println("Statement 1");
        System.out.println(10/0);
        System.out.println("Statement 2");
    }
}
```

```
Statement 1
RE: Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:4)
```

Abnormal Termination

With try - catch

```
class Test {
    public static void main(String[] args) {
        System.out.println("Statement 1");
        try {
            System.out.println(10/0);
        }
        catch(ArithmeticException e) {
            System.out.println(10/2);
        }
        System.out.println("Statement 2");
    }
}
```

```
Statement 1
5
Statement 2
```

Normal Termination

Control Flow in try - catch:

```
try {
        Statement 1;
        Statement 2;
        Statement 3;
}
catch(X e) {
        Statement 4;
}
Statement 5;
```

Case 1: If there is No Exception → 1, 2, 3, 5, Normal Termination.

Case 2: If an Exception raised in Statement 2 and Corresponding catch Block Matched → 1, 4, 5, Normal Termination.

Case 3: If an Exception raised at Statement 2 and Corresponding catch Block Not Matched → 1 followed by Abnormal Termination.

Case 4: If an Exception raised at Statement-4 OR Statement-5 then it's Always Abnormal Termination.

Conclusions:

- Within the try Block if any where an Exception raised then 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 Hence Length of the try Block should be as Less as Possible.
- If there is any Statement which raises an Exception and it is Not Part of the try Block then it is Always Abnormal Termination.
- In Addition to try Block there May be a Chance of raising an Exception Inside catch and finally Blocks Also.

Methods to Print Exception Information

Throwable Class defines the following Methods to Print Exception Information.

Method	Printed Format
printStackTrace()	Name of the Exception: Description Stack Trace
toString()	Name of the Exception: Description
getMessage()	Description

```
class Test {
	public static void main(String[] args) {
		try {
			System.out.println(10/ 0);
		}
		catch (ArithmeticException e) {
			e.printStackTrace(); //RE: java.lang.ArithmeticException: / by zero
						at Test.main(Test.java:4)
			System.out.println(e); //RE: java.lang.ArithmeticException: / by zero
			System.out.println(e.getMessage()); /// by zero
		}
	}
}
```

Note: Internally Default Exception Handler Uses *printStackTrace()* to Print Exception Information to the Console.

try with Multiple catch Blocks

The way of handling an Exception is varied from Exception to Exception. Hence for Every Exception Type we have to define a Separate catch Block. Hence try with Multiple catch Blocks is Possible and Recommended to Use.

```
try {
        ::::::::::::
        ::::::::::::
}
catch (Exception e) {
        ::::::::::::
        ::::::::::::
}
```

Not Recommended

```
try {
        ::::::::::::::
        ::::::::::::::
}
catch (ArithmeticException e) {
        //Perform these an Alternative
ArithmeticException
}
catch (NullPointerException e) {
        //Handling Related to null
}
catch (FileNotFoundException) {
        //Use Local File Instead of Remote File
}
catch (SQLException e) {
        //Use MySQL DB Instead of Oracle
}
catch (Exception e) {
        Default Exception Handling
}
```

Recommended

Note:

- If try with Multiple catch Blocks Present then the Order of catch Blocks are Very Important. It should be from Child to Parent.
- By Mistake if we are trying to Take Parent to Child then we will get Compile Time Error Saying: exception XXX has already been caught

```
try {}
catch (Exception e) {}
catch (ArithmeticException e) {} //CE: exception ArithmeticException has already been caught
```

```
try {}
catch (ArithmeticException e) {}
catch (Exception e) {}
```

For any Exception if we are writing 2 Same catch Blocks we will get Compile Time Error.


```
try {}
catch(ArithmeticException e) {}
catch(ArithmeticException e) {} //CE: error: exception ArithmeticException has already been caught
```

finally Block

- It is Never Recommended to Define Clean-up Code Inside try Block. Because there is No Guaranty for the Execution of Every Statement Inside try Block.
- It is Never Recommended to Define Clean-up Code Inside catch Block. Because if there is No Exception then catch Block won't be executed.
- Hence we required Some Place to Maintain Clean-up Code which should be executed Always irrespective of whether Exception raised OR Not raised and whether Handled OR Not Handled. Such Type of Best Place is Nothing but finally Block.
- Hence the Main Objective of finally Block is to Maintain Clean-up Code.

```
try {
        //Risky Code
}
catch(X e) {
        //Handling Code
}
finally {
        //Clean Up Code
}
```

- The Specialty of finally Block is it will be executed Always irrespective of whether Exception raised OR Not raised and whether Exception Handled OR Not Handled.

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println ("try");
        }
        catch (Exception e) {
            System.out.println ("catch");
        }
        finally {
            System.out.println ("finally");
        }
    }
}
```

try
finally

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println ("try");
            System.out.println (10/0);
        }
        catch (Exception e) {
            System.out.println ("catch");
        }
        finally {
            System.out.println ("finally");
        }
    }
}
```

try
catch
finally


```
class Test {
	public static void main(String[] args) {
		try {
			System.out.println ("try");
			System.out.println (10/0);
		}
		catch (NullPointerException e) {
			System.out.println ("catch");
		}
		finally {
			System.out.println ("finally");
		}
	}
}
```

```
try
finally
Exception in thread "main"
java.lang.ArithmeticException: / by zero
	at Test.main(Test.java:5)
```

finally Vs return:

If return Statement Present Inside try OR catch Blocks 1st finally will be executed and after that Only return Statement will be Considered i.e. finally Block Dominates return Statement.

```
class Test {
	public static void main(String[] args) {
		try {
			System.out.println("try");
			return;
		}
		catch(Exception e) {
			System.out.println("catch");

		}
		finally {
			System.out.println("finally");

		}
	}
}
```

```
try
finally
```

If try-catch-finally Blocks having return Statements then finally Block return Statement will be Considered i.e. finally Block return Statement has More Priority than try and catch Block return Statements.

```
class Test {
	public static void main(String[] args) {
		System.out.println(m1()); //999
	}
	public static int m1() {
		try {
			return 777;
		}
		catch(Exception e) {
			return 888;
		}
		finally {
			return 999;
		}
	}
}
```

finally Vs System.exit(0):

There is Only One Situation where the finally Block won't be executed that is whenever we are System.exit(0).
Whenever we are using System.exit(0) then JVM itself will be Shutdown and hence finally Block won't be executed. That is System.exit(0) Dominates finally Block.

```
class Test {
	public static void main(String[] args) {
		try {
			System.out.println("try");
			System.exit(0);
		}
		catch(Exception e) {
			System.out.println("catch");
		}
		finally {
			System.out.println("finally");
		}
	}
}
```

```
try
```

System.exit(0);

- We can Use this Method to Exit (Shut Down) the System (JVM) Programmatically.
- The Argument Represents as Status Code.
- Instead of 0 we can Pass any Valid int Value.
- 0 Means Normal Termination, Non- Zero Means Abnormal Termination.
- So this status code internally used by JVM.

- Whether it is 0 OR Non- Zero Effect is Same in Our Program but this Number Internally used by JVM.

Difference between final, finally and finalize

final:

- final is a Modifier is Applicable for Classes, Methods and Variables.
- If a Class declared as final then we can't Create Child Class. That is Inheritance is Not Possible for final Classes.
- If a Method declared as final then we can't Override that Method in Child Classes.
- If a Variable declared as final then we can't Perform Re- Assignment for that Variable.

finally:

- finally is a Block Always associated with try-catch to Maintain Clean Up Code.
- The Specialty of finally Block is it will be executed Always Irrespective of whether Exception raised OR Not and whether Handled OR Not Handled.

finalize():

- finalize() is a Method Always Called by the Garbage Collector Just before Destroying an Object to Perform Clean Up Activities.
- Once finalize() Completes Automatically Garbage Collector Destroys that Object.

Note:

- finally() is Responsible to Perform Object Level Clean-Up Activities whereas finally Block is Responsible to Perform try Block Level Clean-Up Activities i.e. whatever Resources we Opened at the Time of try Block will be Closed Inside finally Block
- It is Highly Recommended to Use finally Block than finalize()because we can't Expect Exact Behavior of Garbage Collector. It is JVM Vendor Dependent.

Control Flow in try - catch - finally:

```
try {
        System.out.println("Statement 1");
        System.out.println("Statement 2");
        System.out.println("Statement 3");
}
catch(X e) {
        System.out.println("Statement 4");
}
finally {
        System.out.println("Statement 5");
}
System.out.println("Statement 6");
```

Case 1: If there is No Exception. 1, 2, 3, 5 and 6 → Normal Termination.

Case 2: If an Exception raised at Statement 2 and Corresponding catch Block Matched. then 1, 4, 5, 6 → Normal Termination.

Case 3: If an Exception raised at Statement 2 and Corresponding catch Block Matched. then 1 and 5 Abnormal Termination.

Case 4: If an Exception raised at Statement 4 then it is Always Abnormal Termination but before that finally Block will be executed.

Case 5: If an Exception raised at Statement 5 OR Statement 6 then it is Always Abnormal Termination.

Control Flow in Nested try - catch - finally:

```
try {
        System.out.println("Statement 1");
        System.out.println("Statement 2");
        System.out.println("Statement 3");
        try {
                System.out.println("Statement 4");
                System.out.println("Statement 5");
                System.out.println("Statement 6");
        }
        catch(X  e) {
            System.out.println("Statement 7");
        }
        finally {
           System.out.println("Statement 8");
        }
        System.out.println("Statement 9");
}
catch(X e) {
      System.out.println("Statement 10");
}
finally {
      System.out.println("Statement 11");
}
System.out.println("Statement 12");
```

Case 1: If there is No Exception then 1, 2, 3, 4, 5, 6, and 8, 9, 11, 12 Normal Termination.

Case 2: If an Exception raised at Statement 2 and Corresponding catch Block Matched 1, 10, 11, and 12 Normal Terminations.

Case 3: If an Exception raised at Statement 2 and Corresponding catch Block is Not Matched 1 and 11 Abnormal Termination.

Case 4: If an Exception raised at Statement 5 and Corresponding Inner catch Block has Matched 1, 2, 3, 4, 7, 8, 9, 11, 12 Normal Termination.

Case 5: If an Exception raised at Statement 5 and Inner catch Block has Not Matched but Outer catch Block has Matched. 1, 2, 3, 4, 8, 10, 11, and 12 Normal Termination.

Case 6: If an Exception raised at Statement 5 and Both Inner and Outer catch Blocks are Not Matched then 1, 2, 3, 4, 8, and 11 Abnormal Termination.

Case 7: If an Exception raised at Statement 7 and Corresponding catch Block Matched 1, 2, 3,.,.,., 8, 10, 11, and 12 Normal Termination.

Case 8: If an Exception raised at Statement 7 and Corresponding catch Block Not Matched 1, 2, 3,.,.,.,8, and 11 Abnormal Termination.

Case 9: If an Exception raised at Statement 8 and Corresponding catch Block has Matched 1, 2, 3,.,.,.,., 10, 11, and 12 Normal Termination.

Case 10: If an Exception raised at Statement 8 and Corresponding catch Block Not Matched 1, 2, 3,.,.,.,., 11 Abnormal Termination.

Case 11: If an Exception raised at Statement 9 and Corresponding catch Block Matched 1, 2, 3,.,.,.,., 8, 10, 11, and 12 Normal Termination.

Case 12: If an Exception raised at Statement 9 and Corresponding catch Block Not Matched 1, 2, 3,.,.,.,.,8, and 11 Abnormal Termination.

Case 13: If an Exception raised at Statement 10 is Always Abnormal Termination but before that finally Block 11 will be executed.

Case 14: If an Exception raised at Statement 11 OR 12 it is Always Abnormal Termination.

Note:

- We can Take try - catch - finally Inside try Block i.e. Nesting of try - catch - finally is Always Possible.
- More Specific Exceptions can be handled by Inner catch Block and Generalized Exceptions are handled by Outer catch Blocks.
- Once we entered into the try Block without executing finally Block the Control Never Comes Up.
- If we are Not entering into the try Block then finally Block won't be executed.

```
class Test {
    public static void main(String[] args) {
        try {
                System.out.println(10/0);
        }
        catch(ArithmeticException e) {
                System.out.println(10/0);
        }
        finally {
                String s = null;
                System.out.println(s.length());
        }
    }
}
```

1) CE
2) RE: ArihmeticException
3) RE: NullPointerException ✕
4) RE: ArihmeticException and RE: NullPointerException

Note: Default Exception Handler can Handle Only One Exception at a Time i.e. the Most Recently raised Exception.

Various Possible Combinations of try-catch-finally

- In try-catch-finally Order is Important.
- Inside try-catch-finally we can take try-catch-finally that is nesting of try-catch-finally is always possible.
- Whenever we are taking try Compulsory we have to Take either catch OR finally Blocks i.e. try without catch OR finally is Invalid.
- Whenever we are writing catch Block Compulsory we have to write try Block i.e. catch without try is Invalid.
- Whenever we are writing finally Block Compulsory we should write try i.e. finally without try is Invalid.
- For try-catch-finally Blocks Curly Braces are Mandatory.
- We can't write 2 catch Blocks for the Same Exception Otherwise we will get CE.

```
try {}
catch (X e) {}
```

```
try {}
catch (X e) {}
catch (Y e) {}
```

```
try {}
catch (X e) {}
catch (X e) {} // CE: exception ArithmeticException has already been caught
```

```
try {}
finally {}
```

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```


```
finally  {} //CE: 'finally' without 'try'
```

```
catch (X e) {} //CE: 'catch' without 'try'
```

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
System.out.println("Hello");
catch {} //CE: 'catch' without 'try'
```

```
try {}
catch (X e) {}
System.out.println("Hello");
catch (Y e) {} //CE: 'catch' without 'try'
```

```
try {}
catch (X e) {}
System.out.println("Hello");
finally {} //CE: 'finally' without 'try'
```

```
try {}
finally {}
catch (X e) {} //CE: 'catch' without 'try'
```

```
try {}
catch (X e) {}
try {}
finally {}
```

```
try {}
catch (X e) {}
finally {}
finally {} //CE: 'finally' without 'try'
```

```
try {}
catch (X e) {
                try {}
                catch (Y e1) {}
}
```

```
try {}
catch (X e) {}
finally {
            try {}
            catch (Y e1) {}
             finally {}
}
```

```
try {
        try {} //CE: 'try' without 'catch', 'finally' or resource declarations
}
catch (X e) {}
```

```
try //CE: '{' expected
      System.out.println("Hello");
catch (X e1) {} //CE: 'catch' without 'try'
```

```
try {}
catch (X e) //CE:'{' expected
     System.out.println("Hello");
```

```
try {}
catch (NullPointerException e1) {}
finally //CE: '{' expected
        System.out.println("Hello");
```

throw Keyword:

- Sometimes we can Create Exception Object Explicitly and we can Handover Our Created Exception Object to the JVM Manually. For this we have to Use throw key Word.

  Eg:

```
throw new ArithmeticException("/by zero");
```

throw Key Word Handover Our Created Exception Objet to the JVM Manually

Creation of Exception Object Explicitly

- In General we can Use throw Key Word for Customized Exceptions but Not for pre-defined Exceptions.

The Result of following 2 Programs is Exactly Same.

Case – 1

```
class Test {
      public static void main(String[] args) {
            System.out.println(10/0);
      }
}

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

Case – 2

```
class Test1 {
    public static void main(String[] args) {
          throw new ArithmeticException("/ by zero Explicitly");
    }
}

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

In the Case – 1 main() is Responsible to Create Exception Object and Handover to the JVM. This Total Activity will be performed Internally.

In the Case – 2 Programmer creating Exception Object Explicitly and Handover to the JVM Manually.

Hence the Main Purpose of throw Key Word is to Handover Our Created Exception Object to the JVM Manually.

Case 1: throw e;
If 'e' Refers 'null' then we will get *NullPointerException*.

```
class Test {
      static ArithmeticException e = new ArithmeticException();
      public static void main(String[] args) {
            throw e;
      }
}

RE: Exception in thread "main" java.lang.ArithmeticException
```

```
class Test {
        static ArithmeticException e;
        public static void main(String[] args) {
              throw e;
        }
}

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

Case 2: After throw Statement we are Not allowed to write any Statements Directly Otherwise we will get Compile Time Error Saying unreachable statement.

```
class Test {
       public static void main(String[] args) {
              System.out.println(10/0);
              System.out.println("Hello");
       }
}

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

```
class Test {
      public static void main(String[] args) {
            throw new ArithmeticException("/by zero");
            System.out.println("Hello"); //CE: unreachable statement
      }
}
```

Case 3: We can Use throw Key Word Only for Throwable Types. Otherwise we will get Compile Time Error Saying incompatible types.

```
class Test {
        public static void main(String[] args) {
                throw new Test();
        }
}
```

CE: incompatible types
required: Throwable
found: Test

```
class Test extends RuntimeException {
        public static void main(String[] args) {
                throw new Test();
        }
}
```

RE: Exception in thread "main" Test

throws Key Word

In Our Program if there is any Chance of raising Checked Exception then Compulsory we should Handled that Checked Exception Otherwise we will get Compile Time Error Saying unreported exception XXX; must be caught or declared to be thrown.

```
import java.io.PrintWriter;
class Test {
        public static void main(String[] args) {
                PrintWriter out = new PrintWriter("Abc.txt");
                out.println("Hello");
        }
}
```

CE: unreported exception FileNotFoundException; must be caught or declared to be thrown

```
class Test {
        public static void main(String args[]) {
                Thread.sleep(5000);
        }
}
```

CE: unreported exception InterruptedException; must be caught or declared to be thrown

We can Handle this Compile Time Error in 2 Ways.

1) By Using try-catch
2) By Using throws Key Word

1st Way: By Using try - catch Block

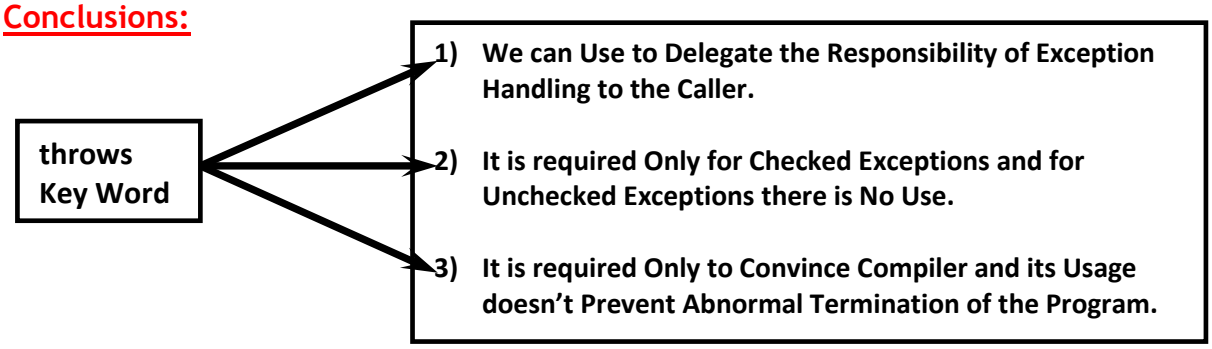
```
class Test {
        public static void main(String args[]) {
                try {
                        Thread.sleep(5000);
                }
                catch (InterruptedException e) {}
        }
}
```

2nd Way: By Using throws Key Word

- We can use throws Key Word to Delegate the Responsibility of Exception Handling to the Caller Method (It May be Another Method OR JVM). Then Caller is Responsible to Handle that Checked Exception.
- throws Key Word required Only for Checked Exceptions.
- Usage of throws Key Word for Unchecked Exceptions there is No Use.
- throws Key Word required Only to Convince Compiler and it doesn't Prevent Abnormal Termination of the Program.
- Hence Recommended to Use try- catch- finally Over throws Key Word.

```
class Test {
      public static void main(String[] args)throws InterruptedException {
                  Thread.sleep(5000);
      }
}
```

Conclusions:




```
class Test {
      public static void main(String[] args)throws InterruptedException {
                  doStuff();
      }
      public static void doStuff()throws InterruptedException {
                  doMoreStuff();
      }
      public static void doMoreStuff()throws InterruptedException {
                  Thread.sleep(5000);
      }
}
```

In the Above Program if we Remove at-least One throws Statement then the Code won't Compile. We will get CE: *unreported exception InterruptedException; must be caught or declared to be thrown*

Case 1: We can Use throws Key Word Only for Methods and Constructors but Not for Classes.


```
class Test throws Exception ✕{
        Test() throws Exception {} √
        public static void m1() throws Exception {} √
}
```

Case 2:

We can Use throws Key Word Only for Throwable Types but Not for Normal Java Classes. Otherwise we will get Compile Time Error Saying *incompatible types required: java.lang.Throwable found: Test*

```
class  Test {
      public static void main(String[] args) throws Test {}
}
```

CE: incompatible types
required: Throwable
found: Test

```
class  Test extends RuntimeException/ Exception/ Throwable {
        public static void main(String[] args) throws Test {}
} √
```

Case 3:

```
class Test {
      public static void main(String[] args) {
              throw new Exception();
      }
}
```

Checked Exception

CE: unreported exception Exception; must be caught or declared to be thrown

```
class Test {
      public static void main(String[] args) {
              throw new Error();
      }
}
```

Unchecked Exception

RE: Exception in thread "main" java.lang.Error
at Test.main(Test.java:3)

Case 4:

Inside try Block, if there is No Chance of raising an Exception then we can't write catch Block for that Exception. Otherwise we will get Compile Time Error Saying CE: *exception XXX is never thrown in body of corresponding try statement*. But this Rule is Applicable Only for Fully Checked Exceptions.

```
class Test {
      public static void main(String[] args) {
            try {
                  System.out.println("Hello");
            }
            catch (ArithmeticException e) {}
      }
}
```

Unchecked Exception

Output: Hello

```
class Test {
      public static void main(String[] args) {
            try {
                  System.out.println("Hello");
            }
            catch (Exception e) {}
      }
}
```

Partially Checked Exception

Output: Hello


```
import java.io.IOException;
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (IOException e) {}
    }
}
```

Fully Checked Exception

CE: exception IOException is never thrown in body of corresponding try statement

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (InterruptedException e) {}
    }
}
```

Fully Checked Exception

CE: exception InterruptedException is never thrown in body of corresponding try statement

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (Error e) {}
    }
}
```

Unchecked Exception

Output: Hello

Summary of Exception Handling Key Words

1) try → To Maintain Risky Code
2) catch → To Maintain Handling Code
3) finally → To Maintain Clean Up Code
4) throw → To Hand-Over Our Created Exception Object to the JVM Manually
5) throws → To Delegate Responsibility of Exception Handling to the Caller Method

Various Possible Compile Time Errors in Exception Handling

1) unreported exception Exception; must be caught or declared to be thrown
2) Exception XXX has already been caught
3) Exception XXX is never thrown in body of corresponding try statement
4) unreachable statement
5) incompatible types
   required: java.lang.Throwable
   found: Test

6) try without catch or finally

7) catch without try

8) finally without try

Customised OR User defined Exceptions

Sometimes to Meet Programming Requirement we have to Create Our Own Exceptions which are Nothing but Customized Exceptions.
Eg: TooYoungException, TooOldException, InsufficentFundsException, Etc.....

```
class TooYoungException extends RuntimeException {
        TooYoungException(String s) {
                super(s);
        }
}
class TooOldException extends RuntimeException {
        TooOldException(String s) {
                super(s);
        }
}
class CustomizedException {
      public static void main(String[] args) {
              int age = Integer.parseInt(args[0]);
              if(age > 60) {
                      throw new TooYoungException("Please Wait Some More Time U will get Best Match");
              }
              else if(age < 18) {
    throw new TooOldException("Your Age Already Crossed Marriage Age No Chance of getting Match");
                    }
                    else {
                            System.out.println("U will get Match Details Soon by Email...!");
                    }
        }
}
```

```
java CustExceptionDemo 60
U will get Match Details Soon by Email...!

java CustExceptionDemo 70
Exception in thread "main" TooYoungException: Please Wait Some More Time U will get Best
Match
      at CustExceptionDemo.main(CustExceptionDemo.java:15)

java CustExceptionDemo 50
U will get Match Details Soon by Email...!

java CustExceptionDemo 15
Exception in thread "main" TooOldException: Your Age Already Crossed Marriage Age No Chance
of getting Match
      at CustExceptionDemo.main(CustExceptionDemo.java:18)
```

Note:

- throw Key Word is Best Suitable for Customized Exceptions but Not for Pre-defined Exceptions.
- It is Highly Recommended to define Customized Exceptions as Unchecked i.e. we have to extends *RuntimeException* but Not *Exception*.
- To Make Description Available to Parent Class, from which Default Exception Handler get these Description.

Top 10 Exceptions

Based on the Person who is raising Exception, all Exceptions are divided into 2 Types.

1) JVM Exceptions
2) Programmatic Exceptions.

JVM Exceptions: The Exceptions which are raised Automatically by the JVM whenever a Particular Event Occurs Such Type of Exceptions are Called JVM Exceptions.
Eg: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException Etc.

Programmatic Exceptions:
The Exceptions which are raised Explicitly either by Programmer OR by API Developer are Called Programmatic Exceptions.
Eg: IllegalArgumentException, TooYoungException, Etc.

2. **ArrayIndexOutOfBoundsException:**

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Automatically by the JVM Whenever we are trying to Access Array Element with Out of Range Index.

Eg:

```
int[] a = new int[10];
System.out.println(a[0]); //0
System.out.println(a[100]); //RE: java.lang.ArrayIndexOutOfBoundsException: 100
System.out.println(a[-100]); //RE: java.lang.ArrayIndexOutOfBoundsException: -100
```

NullPointerException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Automatically by the JVM whenever we are trying to Perform any Method Call on null Reference.

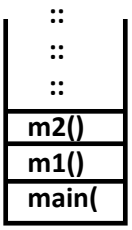
Eg:

```
String s = null;
System.out.println(s.length());//RE: Exception in thread "main" java.lang.NullPointerException
```

StackOverFlowError:
It is the Child Class of Error and Hence it is Unchecked.
Raised Automatically by the JVM whenever we are trying to Perform Recursive Method Call.

```
class Test {
    public static void m1() {
        m2();
    }
    public static void m2() {
        m1();
    }
    public static void main(String[] args) {
        m1(); //RE:  java.lang.StackOverflowError
    }
}
```

Recursive Method Call



Runtime Stack

ClassCastException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Automatically by the JVM whenever we are trying to Type Cast Parent Object to Child Type.

Eg:

```
String s = new String("abc");
Object o = (Object)s; //√
```

```
Object o = new Object();
String s = (String)o;

RE: java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.String
```

```
Object o = new String("abc");
String s = (String)s;//valid//√
```

NoClassDefFoundError:

- It is the Child Class of *Error* and Hence it is Unchecked.
- Raised Automatically by the JVM whenever JVM Unable to find required *.class* File.

Eg: java Test

If *Test.class* is Not Available then we will get RuntimeException Saying java.lang.NoClassDefFoundError: Test

ExceptionInInitializerError:

- It is the Child Class of Error and Hence it is Unchecked.
- Raised Automatically by the JVM if any Exception Occurs while executing Static Variable Assignments and Static Blocks.

```
class Test {
    static int i = 10/0;
}
```

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



```
class Test {
    static {
        String s = null;
        System.out.println(s.length());
    }
}
```

```
Exception in thread "main"
java.lang.ExceptionInInitializerError
Caused by: java.lang.NullPointerException
```

IllegalArgumentException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Explecitly by the Programmer OR by API Developer to Indicate that a Method has been invoked with Illegal Argument.

```
class Test {
    public static void main(String[] args) {
        Thread t = new Thread();
        t.setPriority(10); √
        t.setPriority(100);
    }
}
```

```
RE: Exception in thread "main" java.lang.IllegalArgumentException
    at java.lang.Thread.setPriority(Thread.java:1125)
    at Test.main(Test.java:5)
```

- The Valid Range of Thread Priorities is 1 to 10. If we are trying to Set the Priourity with any Other Value we will get Runtime Exception Saying IllegalArgumentException.

NumberFormatException:

- It is the Direct Child Class of *IllegalArgumentException*, which is Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Explecitly either by Programme OR by API Developer to Indicate that we are trying to Convert String to Number but the String is Not Properly Formatted.

Eg:

```
int i = Integer.parseInt("10"); √
int i = Integer.parseInt("Ten"); //RE: java.lang.NumberFormatException: For input string: "Ten"
```

IllegalStateException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Explecitly either by Programmer OR by API Developer to Indicate that a Method has been invoked at Wrong Time.

Examples:

1) After Starting a Thread we are Not allowed to Re- Start the Same Thread Once Again. Otherwise we will get Runtime Exception Saying *IllegalThreadStateException*.

```
Thread t = new Thread();
t.start();
t.start(); //RE: java.lang.IllegalThreadStateException
```

2) Once Session Expires we are Not allow to Call any Method on that Session Object. If we are trying to Call we will get Runtime Exception Saying *IllegalStateException*.


```
HttpSession session = req.getSession();
System.out.println(session.getId()); //Valid
session.invalidateI();
System.out.println(session.getId()); //RE: IllegalStateException
```

AssertionError:

It is the Child Class of Error and Hence it is Unchecked.
Raised Explecitly to Indicate that assert Statement Fails.
Eg: assert(x > 10);
if(x !> 10) then we will get Runtime Exception Saying *AssertionError*.

Exception/ Error	Raised By
1) ArrayIndexOutOfBoundsException	Raised by JVM and Hence these are JVM Exceptions
2) NullPointerException	
3) StackOverflowError	
4) ClassCastException	
5) NoClassDefFoundError	
6) ExceptionInInitializerError	
7) IllegalArgumentException	Raised by Explicitly either by Programmer OR by API Developer and Hence they are Programmatic Exceptions
8) NumberFormatException	
9) IllegalStateException	
10) AssertionError	

1.7 Version Enhancements

In 1.7 Version as the Part of Exception Handling the following 2 Concepts Introduced.

- try with Resources
- Multi catch Block

try with Resources: Until 1.6 Version it is Highly Recommended to write finally Block to Close All Resources which are Opened as the Part of try Block.

```
BufferedReader br = null;
try {
        br = new BufferedReader(new FileReader("Input.txt");
        //Use br based on Our Requirement
}
catch (InterruptedException e) {
        //Handling Code
}
finally {
            if (br != null) {
                br.close();
            }
}
```

The Problems in this Approach are:

- Compulsory Programmer is required to Close All Opened Resources in finally Block. It Increases Length of the Code and Reduces Readability.
- It Increases the Complexity of the Programming.

To Overcome these Problems SUN People Introduce *try with Resources* in 1.7 Version.

```
try (BufferedReader br = new FileReader("Input.txt")) {        Resource


//Use br based on Our Requirements. br will be Closed Automatically Once the Control
        Reaches End of try either Normally OR Abnormally
}
catch (InterruptedException e) {}
```

- The Main Advantage of try with Resources is the Resources which are Opened as the Part of try Block will be Closed Automatically and we are Not required to Close Explicitly. It Reduces Complexity of the Programming.
- It is Not required to write finally Block Explicitly and Hence Length of the Code will be Reduced and Readability will be Improved.

Conclusions:

- We can Declare Multiple Reasons and All these Resources should be Separated with ';'.

  Syntax: try (R1; R2; R3) {

  }

  Eg: try (FileWriter fw = new FileWriter("Output.txt");
  FileWriter fw = new FileWriter("Input.txt");) {

  }

- All Resources should be AutoClosable Resources.
- A Resource is Said to be AutoClosable
  ➔ Corresponding Class Implements java.lang.AutoClosable Interface either Directly OR In- Directly.

- This Interface introduced in 1.7 Version and it contains Only One Method
  public void close();

- All Network OR Database Related OR File IO Related Resources Implements AutoClosable Interface. Being a Programmer we are Not required to do anything.
- All Resource Reference Variables are Implicitly final. Hence within the try Block we can't Perform Re- Assignment.

```
import java.io.*;
class TryWithResources {
      public static void main (String args[]) throws Exception {
            try (BufferedReader br = new BufferedReader (new FileReader ("abc.txt"))) {
                  br = new BufferedReader (new FileReader ("Input.txt"));
            }
      }    //CE: auto-closeable resource br may not be assigned
}
```

- Until 1.6 Version try should be followed by either catch OR finally but from 1.7 onwards we can Take Only try with Resources without catch and finally Blocks.
  Eg: try (R) {----------}
- The Main Advantage of try with Resources is finally Block will become Dummy because we are required to Close the Resources Explicitly.

> Until 1.6 Version finally Block is Hero.
> But 1.7 Version onwards Zero.

Multi Catch Block (Catch Block with Multiple Exceptions)

Until 1.6 Version even though Multiple Exceptions having Same Handling Code Compulsory we have to write a Separate catch Block for Every Exception.

```
try {
        -------
}
catch (ArithmeticException e) {
        e.printStackTrace();
}
catch (NullPointerException e) {
        e.printStackTrace();
}
catch (ClassCastException e) {
        System.out.println(e.getMessage());
}
catch (IOException e) {
        System.out.println(e.getMessage());
}
```

- **The Problem in this Approach is it Increases Length of the Code and Reduces Readability.**
- **To Overcome this Problem SUN People Introduced Multi Catch Block in 1.7 Version.**
- **In this Approach we can write a Single Catch Block which can Handle Multiple Exceptions of different Types.**

```
try {
        -------
}
catch (ArithmeticException | NullPointerException e) {
        e.printStackTrace();
}
catch (ClassCastException | IOException e) {
        System.out.println(e.getMessage());
}
```

```
class MultiCatchBlock {
    public static void main(String[] args) {
        try {
                //System.out.println(10/ 0);
                   String s = null;
                   System.out.println(s.length());
        }
        catch (ArithmeticException | NullPointerException e) {
                System.out.println(e);  //java.lang.NullPointerException
        }
    }
}
```

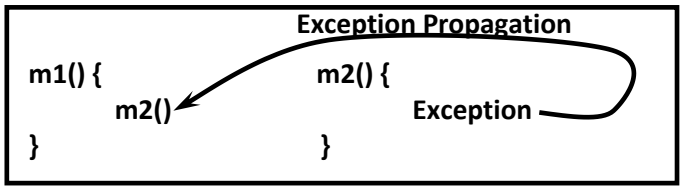
- If Mutli Catch Block there should Not be any Relation between Exception Types (Like Parent to Child OR Child to Parent OR Same Type) Otherwise we will get Compile Time Error.

```
catch (ArithmeticException | NullPointerException | ClassCastException e) {}
catch (ArithmeticException | Exception e) {}

CE: Alternatives in a multi-catch statement cannot be related by subclassing
    Alternative ArithmeticException is a subclass of alternative Exception
```

Exception Propagation:

- Within a Method if an Exception raised and if we are Not Handle that Exception then that Exception Object will be propagated to Automatically to the Caller Method.
- Then Caller Method is Responsible to Handle that Exception.
- This Process is Called *Exception Propagation*.



Re-Throwing Exception

We can Use this Approach to Convert One Exception Type to Another Exception Type.

```
try {
        System.out.println(10/ 0);
}
catch (ArithmeticException e) {
        throw new NullPointerException();
}
```

Q1. Given the code fragment:

```
1)  class X
2)  {
3)     public void printFileContent()
4)     {
5)        //Line-1
6)        throw new IOException();//Line-2
7)     }
8)  }
9)  public class Test
10) {
11)    public static void main(String[] args)//Line-3
12)    {
13)       X x= new X();
14)       x.printFileContent();//Line-4
15)       //Line-5
16)    }
17) }
```

Which two modifications required to compile code successfully?

A. Replace Line-3 with public static void main(String[] args) throws Exception
B. Replace Line-4 with:

```
1)    try
2)    {
3)     x.printFileContent();
4)    }
5)    catch (Exception e){}
6)    catch (IOException e){}
```

C. Replace Line-3 with public static void main(String[] args) throws IOException
D. Replace Line-2 with throw IOException("Exception Raised");
E. At Line-5 insert throw new IOException();

Answer: A, C

Q2. Given the code Fragment:

```
1)  public class Test
2)  {
3)     void readCard(int cno) throws Exception
4)     {
5)        System.out.println("Rearding Card");
6)     }
7)     void checkCard(int cno) throws RuntimeException//Line-1
8)     {
```

```
9)      System.out.println("Checking Card");
10)   }
11)   public static void main(String[] args)
12)   {
13)      Test t = new Test();
14)      int cardNo=1234;
15)      t.checkCard(cardNo);//Line-2
16)      t.readCard(cardNo);//Line-3
17)   }
18) }
```

What is the result?

A. Checking Card
Reading Card

B. Compilation Fails at Line-1
C. Compilation Fails at Line-2
D. Compilation Fails at Line-3
E. Compilation Fails at Line-2 and Line-3

Answer: D

Q3. Given the following code for the classes MyException and Test:

```
1)  public class MyException extends RuntimeException
2)  {
3)  }
4)
5)  public class Test
6)  {
7)     public static void main(String[] args)
8)     {
9)        try
10)       {
11)          m1();
12)       }
13)       catch (MyException e)
14)       {
15)          System.out.print("A");
16)       }
17)    }
18)    public static void m1()
19)    {
20)       try
21)       {
22)          throw Math.random() > 0.5 ? new Exception():new MyException();
23)       }
```

```
24)       catch (RuntimeException e)
25)       {
26)          System.out.println("B");
27)       }
28)    }
29) }
```

What is the result?

A. A
B. B
C. Either A or B
D. AB
E. Compilation Fails

Answer: E

Q4. Given the code fragment:

```
1)  String[] s= new String[2];
2)  int i=0;
3)  for(String s1: s)
4)  {
5)     s[i].concat("element"+i);
6)     i++;
7)  }
8)  for(i=0; i<s.length;i++)
9)  {
10)    System.out.println(s[i]);
11) }
```

What is the result?

A. element 0
element 1
B. null element 0
null element 1
C. null
null
D. A NullPointerException is thrown at runtime

Answer: D

Q5. Given the code fragment:

```
1)  public class Test
2)  {
3)     public static void main(String[] args)
4)     {
5)        String[] names={"Thomas","Bunny","Chinny"};
6)        String[] pwds=new String[3];
7)        int i =0;
8)        try
9)        {
10)          for (String n: names)
11)          {
12)             pwds[i]=n.substring(2,6);
13)             i++;
14)          }
15)       }
16)       catch (Exception e)
17)       {
18)          System.out.println("Invalid Name");
19)       }
20)       for(String p: pwds)
21)       {
22)          System.out.println(p);
23)       }
24)    }
25) }
```

What is the result?

A.
Invalid Name
omas
null
null

B.
Invalid Name

C.
Invalid Name
omas

D.
Compilation Fails

Answer: A

Q6. Given the code fragment:

```
1)  import java.util.*;
2)  public class Test
3)  {
4)     public static void main(String[] args)
5)     {
6)        ArrayList l = new ArrayList();
7)        String[] s;
8)        try
9)        {
10)          while(true)
11)          {
12)             l.add("MyString");
13)          }
14)       }
15)       catch (RuntimeException e)
16)       {
17)          System.out.println("Caught a RuntimeException");
18)       }
19)       catch (Exception e)
20)       {
21)          System.out.println("Caught an Exception");
22)       }
23)       System.out.println("Ready to use");
24)    }
25) }
```

What is the result?

A. Caught a RuntimeException printed to the console
B. Caught an Exception printed to the console
C. A runtime error is thrown at runtime
D. Ready to use printed to the console
E. The code fails to compile because a throws keyword required

Answer: C

Q7. Which three are advantages of the Java Exception Mechanism?

A. Improves the program structure because the error handing code is separated from the normal program function.

B. Provides a set of standard exceptions that covers all possible errors

C. Improves the program structure beacuase the programmer can choose where to handle exceptions

D. Imporves the program structure because exceptions must be handled in the method in which they occurred.
E. Allows the creation of new exceptions that are tailored to the particular program being created.

Answer:A,C,E

Q8. Which 3 statements are true about exception handling?

A. Only unchecked exceptions can be rethrown
B. All Subclasses of the RuntimeException are recoverable
C. The parameter in catch block is of throwable type
D. All subclasses of RuntimeException must be caught or declared to be thrown
E. All Subclasses of the Exception except RuntimeException class are checked exceptions
F. All subclasses of the Error class are checked exceptions and are recoverable

Answer: B,C,E

Q9. Which two statements are true?

A. Error class is unextendable
B. Error class is extendable
C. Error is a RuntimeException
D. Error is an Exception
E. Error is a Throwable

Ans: B,E

FAQ's

1) What is An Exception?

2) What is The Purpose of Exception Handling?

3) What is The Meaning of Exception Handling?

4) Explain Default Exception Handling Mechanism in Java?

5) What is The Purpose of try?

6) What is The Purpose of catch Block?

7) Is try With Multiple catch Block Possible?

8) If try With Multiple catch Block Present, Is Order of catch Blocks Important in Which Order We Have To Take?

9) What Are Various Methods To Print Exception Information? And Differentiate Them.

10) If An Exception Raised Inside catch Block Then What Will Happen?

11) Is it Possible To Take try, catch Inside try Block?

12) Is it Possible To Take try, catch Inside catch Block?

13) Is it Possible To Take try Without catch?

14) What is The Purpose of Finally Block?

15) Is Finally Block Will Be Execute Always?

16) In Which Situation Finally Block Will Not Executed?

17) If Return Statement Present Inside try, is Finally Block Will Be Executed?

18) What is The Difference Between final, finally And finalize ()?

19) Is it Possible To Write Any Statement Between try-catch And finally?

20) Is it Possible To Take 2 finally Blocks For The Same try?

21) Is Syntax try-finally-catch is Valid?

22) What is The Purpose of throw?

23) Is it Possible To throw An Error?

24) Is it Possible To throw Any Java Object?

25) After throw is it Allow To Take Any Statement Directly?

26) What is The Purpose Of throws?

27) What is The Difference Between throw And throws?

28) What is The Difference Between throw And thrown?

29) Is it Possible To Use throws Keyword For Any Java Class?

30) If We Are Taking catch Block For An Exception But There is No Chance of Rising That Exception in try Then What Will Happen?

31) Explain Exception Handling Keywords?

32) Which Class Act As Root For Entire Java Exception Hierarchy?

33) What is The Difference Between Error And Exception?

34) What is Difference Between Checked Exception And Unchecked Exception?

35) What is Difference Between Partially Checked And Fully Checked Exception?

36) What is A Customized Exception?

37) Explain The Process of Creating The Customized Exception.

38) Explain Control Flow in try, catch, finally.

39) Can You Give The Most Common Occurred Exception in Your Previous Project?

40) Explain The Cases Where You Used Exception Handling in Your Previous Project?