## Exception Handling

- Exception is an object/instance, which is used to send notification to the end user if exceptional situation occurs in the program.
- We should handle exception
  1. To manage runtime errors centrally(inside main method )
  2. To avoid resource leakage.
- Operating System Resources
  1. Memory
  2. File
  3. Thread
  4. Socket
  5. Nework Connection
  6. IO devices.
- If we want to handle exception then we should use five keywords:
  1. try
  2. catch
  3. throw
  4. throws
  5. finally
- AutoCloseable is interface declared in java.lang package.
- "void close() throws Exception" is a method of java.lang.AutoCloseable
- Closeable is interface declared in java.io package.
- "void close() throws IOException" is a method of java.io.Closeable interface.

**Resource**

- An instance, whose type implements AutoCloseable/Closeable interface is called resource.

```
class Test implements AutoCloseable
{
    @Override
    public void close() throws Exception
    {   }
}
class Program
{
    public static void main(String[] args)
    {
        Test t = new Test(); //resource
        }
    }
```

### Exception class hierarchy

- java.lang.Throwable is a super class of all errors and exceptions in java lanaguage.
- If runtime error gets generated due to runtime enviroment then it is considered as Error in context of exception handling.
- We can not recover from error.
- We can write try catch block to handle errors. But we can not recover from error hence it is not recommended to try try catch block to handle errors.
- Example:
    1. StackOverflowError
    2. VirtualMachineError
    3. OutOfMemoryError
- If runtime error gets generated due to application then it is considered as Exception in context of exception handling.
- We can recover from exception.
- Since it is possible to recover from exception, it is recommended to write try catch block to handle exception.
- Example:
    1. NullPointerException
    2. ClassCastException
    3. ClassNotFoundException

### Types of exception

```
1. Checked Exception
2. Unchecked Exception
```

- Above types of exception are designed for java compiler.

#### Unchecked Exception

- java.lang.RuntimeException and all of its sub classes are considered as Unchecked exception.
- Handling unchecked exception is optional.
- Example:
    1. NumberFormatException

2. NullPointerException
3. NegativeArraySizeException
4. ArrayIndexOutOfBoundsException
5. ClassCastException

#### Checked Exception

- java.lang.Exception and all its sub classes except java.lang.RuntimeException(and its sub classes ) are considered as checked exception.
- It is mandatory to handle checked exception.
- Example:
    1. CloneNotSupportedException
    2. InterruptedException
    3. ClassNotFoundException
    4. FileNotFoundException
    5.

### Throwable

- It is a class declared in java.lang package.
- Only objects that are instances of Throwable class (or one of its subclasses) are thrown by the JVM or can be thrown by the Java throw statement.
- Similarly, only Throwable class or one of its subclasses can be the argument type in a catch clause
- Constructor(s):

1. public Throwable()

```
Throwable t = new Throwable( );
```

2. public Throwable(String message)

```
Throwable t = new Throwable( "Exception" );
```

3. public Throwable(Throwable cause)

```
String msg = "Exception";
Throwable cause = new Throwable( msg);
Throwable t = new Throwable( cause);
```

4. public Throwable(String message, Throwable cause)

```
String msg = "Exception";
Throwable cause = new Throwable();
Throwable t = new Throwable( msg, cause);
```

- Method(s)

1. public String getMessage()
2. public Throwable getCause()
3. public void printStackTrace()

**try**

- It is keyword in java
- It is used to inspect exception.
- In java, try block must have at least one catch block, finally block or resource.

**catch**

- It is keyword in java
- It is used to handle exception.
- For single try block we can provide multiple catch block.
- In single catch block, we can handle multiple specific exceptions. such catch block is called multi catch block.

```
try
{   }
catch( ArithmeticException | InputMismatchException ex )
{
    //TODO
}
```

- NullPointerException is a unchecked exception.

```
NullPointerException ex = new NullPointerException();   //OK

RuntimeException ex = new NullPointerException();   //OK

Exception ex = new NullPointerException();//OK
```

- Interrupted Exception is a checked exception.

```
Interrupted ex = new InterruptedException();//OK
Exception ex = new InterruptedException();//OK
```

- java.lang.Exception class reference variable can contain reference of any checked as well as unchecked exception. Hence to write generic catch block we should use Exception class.

- Syntax:

```
try
{
    //TODO
}
catch( Exception ex )//Generic catch block
{
    ex.printStackTrace();
}
```

- If child/parent relation is exist between exception types then we must handle child type exceptions first.

```
try
{
    //TODO
}
catch (ArithmeticException ex)
{   }
catch (RuntimeException ex)
{   }
catch (Exception ex)
{   }
```

**throw**

- It is keyword in java.
- It is used to generate new exception
- using throw keyword, we can throw instance of sub class of java.lang.Throwable class only.
- throw statement is jump statement.

```
try
{
    System.out.print("Num1  :   ");
    int num1 = sc.nextInt();
    System.out.print("Num2  :   ");
    int num2 = sc.nextInt();
    if( num2 == 0 )
        throw new ArithmeticException("Divide by zero exception");
    int result = num1 / num2;
    System.out.println("Result  :   "+result);
}
catch (ArithmeticException ex)
{
    System.out.println(ex.getMessage());
}
```

#### finally

- It is keyword in java.
- If we want to release local resources then we should use finally block.
- JVM always execute finally block.
- for try block we can provide only one finally block.
- If we write System.exit(0) inside try and catch block then JVM do not execute finally block.

#### throws

- If we want to delegate exception(checked/unchecked) from one method to another method then we should use throws clause.

```
public static void printRecord( ) throws InterruptedException
{
    for( int count = 1; count <= 10; ++ count )
    {
        System.out.println("Count: "+count);
        Thread.sleep(250);
    }
}
public static void main(String[] args)
{
    try
    {
        Program.printRecord();
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

#### Custom Exception

- JVM can understand exceptional conditions that is occurred in business logic. If we want to handle such situations then we should write custom exception class.
- If we want to define custom unchecked exception class then we should extend the class from java.lang.RuntimeException class.

```
class StackOverflowException extends RuntimeException
{   }
```

- If we want to define custom checked exception class then we should extend the class from java.lang.Exception class.

```
class StackOverflowException extends Exception
{   }
```

#### Exception Chaining

- Generally exceptions are handled by throwing new type of exception. It is called exception chaining
- If we want trace any application then we should use exception chaining.

#### Bug

- If runtime error gets generated due to application developer's mistake then it is considered as bug.
- Example:
    1. NullPointerException
    2. ArrayIndexOutOfBoundsException
    3. ClassCastException
- We should not provide try catch block to handle bug rather we should find out cause of the bug.

#### Exception

- If runtime error gets generated due to end users mistake then it is considered as Exception.
- Example:
    1. ClassNotFoundException
    2. FileNotFoundException
- We should provide try catch block to handle exception

#### Error

- If runtime error gets generated due to enviromental condition then it is considered as error.