Day 7

Exception Handling

```
- Exception is an instance that is used to send notification to the end
user of the system if any exceptional situation occurs in the program.
- In java to handle exception, we should use five keywords
    1. try
        - It is keyword in java
        - If we want to inspect group of statements for exception then we
should use try block/handler
        - Try block must have, at least one catch, finally or resource.
    2. catch
        - It is a keyword.
        - If we want to handle exception then we should use catch
block/handler.
        - Catch block can handle exception thrown from try block.
        - try block may have multiple catch block.
        - In java, in single catch block, we can handle multiple specific
exceptions. Such catch block is called multi-catch block.
```

```java
                try
                {

                        Scanner sc = new Scanner(System.in);
                        System.out.print("Num1  :       ");
                        int num1 = sc.nextInt();
                        System.out.print("Num2  :       ");
                        int num2 = sc.nextInt();

                        int result = num1 / num2;
                        System.out.println("Result      :       "+result);
                }
                catch( ArithmeticException | InputMismatchException ex )
                {
                        System.out.println("Exception");
                        //ex.printStackTrace();
                }
```

```
    - A  catch block, that handles all exceptions is called generic catch
block.
    - Consider following code:
        1. NullPointerException ex = new NullPointerException()
        2. RuntimeException ex = new NullPointerException();
        3. Exception ex = new NullPointerException();
    - Consider following code:
        1. InterruptedException ex = new InterruptedException()
        2. Exception ex = new InterruptedException();
    - Exception class reference variable can contain reference  of
instance of checked as well as unchecked exception.  Hence to write
generic catch block we should use java.lang.Exception class.
```

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

```
     - In case of exception handling, if child-parent relationship is
exist between exceptions then it is mandatory to handle sub class
exception first.
```

```
                try
                {

                        Scanner sc = new Scanner(System.in);
                        System.out.print("Num1  :       ");
                        int num1 = sc.nextInt();
                        System.out.print("Num2  :       ");
                        int num2 = sc.nextInt();

                        int result = num1 / num2;
                        System.out.println("Result      :       "+result);
                }
                catch( ArithmeticException ex )
                {
                        System.out.println(ex.getClass().getName());
                }
                catch( RuntimeException ex )
                {
                        System.out.println(ex.getClass().getName());
                }
                catch( Exception ex )
                {
                        System.out.println(ex.getClass().getName());
                }
```

```
    3. throw
        - It is keyword in java
        - To generate new exception we should use throw keyword
        - throw statement is jump statement.
        - Using throw keyword we can throw instance of sub class of
Throwable.
    4. throws
        - It is keyword in java.
        - If we want to delegate exception from one method to another
method then we should use throws keyword/clause
```

```
public class Program
{
        /*public static void print( )
        {
                try
                {
                        for( int count = 1; count <= 10; ++ count )
                        {
                                System.out.println("Count       :
"+count);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
        }*/
        public static void print( ) throws InterruptedException
        {
                for( int count = 1; count <= 10; ++ count )
                {
                        System.out.println("Count       :       "+count);
                        Thread.sleep(1000);
                }
        }
        public static void main(String[] args) //throws
InterruptedException
        {
                try
                {
                        Program.print();
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
        }
}
```

~~~
```     –
        5. finally
            – It is keyword in java
            – If want to relase local resources then we should use finally
block
            – For try block we can provide only one finally block.
            – It must appear after all catch block.
            – JVM always execute finally block.
            – If we write "System.exit(0)" inside try and catch block then
JVM do not execute finally block.
    – What is need to handle exception
        1. To handle all the runtime errors centrally so that we can
reduce maintenance of system.
        2. To handle OS resources carefully.
    – Following are OS resources
        1. File
        2. Thread
        3. Socket
        4. Network Connection
        5. IO Devices
    – Throwable is non final and concrete class declared in
java.lang package.
    – It is super class of Error and Exception in java language.
        **Error**
            – Error gets generated due to runtime environment
            – we can not recover from error.
            – We can write try–catch block to handle error but it is
useless.
        **Exception**
            – Exception gets generated due to application
            – We can recover from exception.
            – We can handle exception using try catch block.
    – Only objects that are instances of Throwable (or one of its
subclasses) are thrown by the JVM or can be thrown by the Java throw
statement.
    – Similarly, only Throwable or one of its subclasses can be the
argument type in a catch clause.

```java
class MyException
{       }
public class Program
{
        public static void main(String[] args)
        {
                int num1 = 10;
                int num2 = 0;
                if( num2 != 0 )
                {
                        int result = num1 / num2;
                        System.out.println(result);
                }
                else
                        throw new MyException( );
~~~

```
                //Error : MyException is non sub class of Throwable
        }
}
```

```
- To run this program, we should extend MyException from Throwable class.
```

```
class MyException extends Throwable
{       }
public class Program
{
        public static void main(String[] args)
        {
                int num1 = 10;
                int num2 = 0;
                if( num2 != 0 )
                {
                        int result = num1 / num2;
                        System.out.println(result);
                }
                else
                        throw new MyException( );       //OK
        }
}
```

```
- Members of java.lang.Throwable class
    - Constructor
        1. public Throwable()
            Throwable th = new Throwable( );
        2. public Throwable(String message)
            Throwable th = new Throwable( "Exception" );
        3. public Throwable(Throwable cause)
            Throwable th1 = new Throwable( "Exception" );
            Throwable th2 = new Throwable( th1 );
        4. Throwable(String message, Throwable cause)
            Throwable th1 = new Throwable(  );
            Throwable th2 = new Throwable( "Exception", th1 );
    - Methods
        1. public String getMessage();
        2. public Throwable getCause();
        3. public void printStackTrace();
        4. public void printStackTrace(PrintStream s);
    - Types of Exception
        1. Checked Exception
        2. Unchecked Exception
        - These are types of exception designed for java compiler.
```

Unchecked Exception

```
- java.lang.RuntimeException and all its sub classes are considered as
unchecked exception.
- Handling unchecked exception is not mandatory. In other words, to handle
unchecked exception, java compiler do not force us to write try catch
block.
- Example:
    1. NumberFormatException
    2. NullPointerException
    3. NegativeArraySizeException
    4. ArrayIndexOutOfBoundsException
    5. IllegalArgumentException
    6. ClassCastException
```

Checked Exception

```
- java.lang.Exception and all its sub classes except
java.lang.RuntimeException(and its sub classes) are considered as checked
exceptions.
- Handling checked exception is mandatory. In other words, to handle
checked exception, java compiler force us to write try catch block.
- Example:
    1. InterruptedException
    2. CloneNotSupportedException
    3. IOException
    4. SQLException
    5. ClassNotFoundException
```

Resource

```
- AutoCloseable is interface declared in java.lang package.
- "void close() throws Exception" is a method of AutoCloseable interface
- Closeable is sub interface of AutoCloseable interface
- "void close() throws IOException" is method of closeable interface.
- In context of excetion handling any instance is resource if its type
implements either java.lang.AutoCloseable or java.io.Closeable interface.
```

```
class Abc implements AutoCloseable
{
        public void close() throws Exception
        {       }
}
public class Program
{
        public static void main(String[] args)
        {
                Abc obj = new Abc();//here new Abc(); is resource
        }
}
```

```
- Consider code of try with resource
```

```
try( Scanner sc = new Scanner(System.in);)
                {

                        System.out.print("Num1  :       ");
                        int num1 = sc.nextInt();
                        System.out.print("Num2  :       ");
                        int num2 = sc.nextInt();
                        int result = num1 / num2;
                        System.out.println("Result      :       "+result);
                }
                catch (Exception ex)
                {
                        ex.printStackTrace();
                }
```

```
- In java, we can write try-catch block inside another try, catch  and
finally block. It is called nested try catch block.
```

Custom Exception

```
- JVM can not understand exceptional situation occurs in business logic.
To handle such situation, we should define custom exception.
- If we want to define custom checked exception class then we should
extend it from java.lang.Exception class
```

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

```
- If we want to define custom unchecked exception class then we should
extend it from java.lang.RuntimeException class
```

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

Exception Chaining

```
- We can handle exception by throwing new type of exception. It is called
exception chaining
```

```
abstract class A
{
        public abstract void print();
}
class B extends A
{
        @Override
        public void print() throws RuntimeException
        {
                try
                {
                        for( int count = 1; count <= 10; ++ count )
                        {
                                System.out.println("Count       :
"+count);
                                Thread.sleep(250);
                        }
                }
                catch (InterruptedException cause)
                {
                        throw new RuntimeException( cause );
//Exception Chaining
                }
        }
}
```

AutoBoxing and AutoUnBoxing

```
- Boxing is a process of converting state of instance of value type into
reference type.
- example
```

```
  int number = 10;
  String strNumber = String.valueOf( number );//Boxing
- If boxing is done implicitly then it is called auto-boxing
- example
  int number = 10;
    Object obj = number;  //AutoBoxing
- Unboxing is a process of converting state of instance of reference type
into value type.
- example
  String str = "125";
  int number = Integer.parseInt( str ); //UnBoxing
- If unboxing is done implicitly then it is called    auto-unboxing.
- example
  Integer n1 = new Integer(125 );
    //int n2 = n1.intValue();    //UnBoxing
    int n2 = n1;  //Auto-UnBoxing
```

Generics

```
- If we want to write generic code in java then we should use generics
- We can write generic code using
  1. java.lang.Object class
  2. Generics
```

Generic code without generics

```
class Box
{
    private Object object;
    public Object getObject()
    {
        return object;
    }
    public void setObject(Object object)
    {
        this.object = object;
    }
}
```

```
public static void main(String[] args)
    {
        Date date = new Date();
        Box b3 = new Box();
        b3.setObject( date );  //Upcasting
```

```
                date =  (Date) b3.getObject();  //Downcasting
                System.out.println(date.toString());
        }
        public static void main2(String[] args)
        {
                int number = 10;
                Box b2 = new Box();
                b2.setObject(number);   //Auto-Boxing

                Integer n1 =  (Integer) b2.getObject(); //Downcasting
                number = n1;    //Auto-UnBoxing
                System.out.println(number);
        }
        public static void main1(String[] args)
        {
                Box b1 = new Box();
        }
```

```
- Using Object class, we can not write type safe generic code.
- If we want to write type safe generic code then we should use generics.
```

Generic code using generics

```
class Box<T>    //T -> Type Parameter
{
        private T object;
        public T getObject()
        {
                return object;
        }
        public void setObject(T object)
        {
                this.object = object;
        }
}
```

```
public static void main1(String[] args)
{
    Date date = new Date();
    Box<Date> b1 = new Box<Date>();
    b1.setObject( date );
    date=  b1.getObject();
}
```

Why Generics

```
1. It gives us stronger type checking at compile time. In other words, we
can write typesafe code.
2. It allows to implement generic data structure and algorithm
3. It completly eleminates explicit type casting.
```

Type inference

```
- An ability of compiler to detect type of argument at compile time is
called type inference.
    Box<Date> b1 = new Box<Date>(); //Ok
    Box<Date> b2 = new Box<>(); //Ok : Type will be inffered from left
side.
- During instantiation of generic type, type argument must be reference
type.
    Box<int> b1 = new Box<int>();   //Not Ok
        Box<Integer> b1 = new Box<Integer>();   //Ok
        Box<Integer> b2 = new Box<>();  //Ok
- If we instantiate generic type without type argument then type is called
raw type.
Box b3 = new Box();        //Box is raw type
//Box<Object> b3 = new Box<>();
```

Commonly used type parameter names in java

T - Type E - Element N - Number K - Key V - Value U,S - Second Type Parameters

Interface