Day 14

Exception Handling

Custom/User Defined Exception

- JVM can not understand exceptional conditions of business logic. If we want to handle it then we should define user defined exception class.
- Custom unchecked exception class

```
class StackOverflowException extends RuntimeException{
    //TODO
}
```

- Custom checked exception class

```
class StackOverflowException extends Exception{
    //TODO
}
```

- Stack is a linear data structure/collection in which, we can store elements in Last In First Out manner(LIFO).
- We can perform following operations on Stack:
    1. boolean empty( ) //index == -1
    2. boolean full( ) //if index == size - 1
    3. void push(int element) //To insert element into stack
    4. int peek( ); //To read topmost value from stack //readonly
    5. void pop( ); //To remove element

Exception Chaining

```
package test;

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(500);
                if(count  == 5 )
```

```
                    Thread.currentThread().interrupt();
                }
            } catch (InterruptedException cause) {
                throw new RuntimeException(cause);  //Exception Chaining
            }
        }
    }
    public class Program {
        public static void main(String[] args) {
            try {
                A a = new B( ); //Upcasting
                a.print();
            } catch (RuntimeException e) {
                Throwable cause = e.getCause();
                System.out.println(cause);
                //e.printStackTrace();
            }
        }
    }
```

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

In the context of exception handling:

1. Bug
    - If runtime errors gets generated due to developers mistake then it is considered as bug in java application.
    - Example:
        1. NullPointerException
        2. ArrayIndexOutOfBoundsException
        3. ClassCastException
    - We should not write try catch block to handle bug.
2. Exception
    - If runtime errors gets generated due to end users/clients mistake then it is considered as exception in java application.
    - Example
        1. ClassNotFoundException
        2. FileNotFoundException
        3. ArithmeticException
    - We should write try catch block to handle exception.
3. Error
    - If runtime errors gets generated due to environmental condition then it is considered as error in java application.
    - Example
        1. OutOfMemoryError
        2. StackOverflowError
        3. InternalError
        4. VirtualMachineError

- It is not recommended to write try-catch block to handle error.

Function Activation Record(FAR)

- Program in execution is called process/running instance of a program is called process.
- Process may contain one or more threads.
- JVM maintains runtime stack per thread. This stack contains stack frame.
- Stack Frame contains data of called function:
  1. Return address
  2. Method Parameters
  3. Method Local variable
  4. Other method calls
  5. temp space of computing.
- This information stored inside stack frame is called function activation record.

Exception Propagation

- During execution, if exception occurs then, first call stack gets destroyed and then control is retured to the calling method. This process is called exception propagation.

Assertion : Home Work

Boxing

- Consider example
  - int is primitive/value type
  - String is class hence it is non primitive/reference type

```
int number = 10;
String strNumber = String.valueOf( number );    //Boxing
```

- Process of converting, value of variable of primitive type into non primitive type is called boxing.

Auto-Boxing

- If boxing is done implicitly then it is called auto-boxing.

```
int number = 10;
Object obj = number;    //OK : Auto-Boxing

//Integer i = Integer.valueOf( number );    //Auto-Boxing
//Object obj = i;   //Upcasting
```

UnBoxing

```
String strNumber = "125";
int number = Integer.parseInt( strNumber ); //Unboxing
```

- Process of converting, state of instance of non primitive type into primitive type is called unboxing.

Auto-UnBoxing

- If unboxing is done implicitly then it is called as auto-unboxing.

```
Integer i1 = new Integer(125);
//int i2 = i1.intValue();   //UnBoxing
int i2 = i1;    //Auto-UnBoxing
```

Generic Programming

- If we want to write generic code in java then we can use:
    1. java.lang.Object class
    2. Generics
- Generic code using java.lang.Object

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

- Storing primitive value inside Box instane.

```
public static void main2(String[] args) {
    Box box = new Box();    //OK
    box.setObject(125); //box.setObject(Integer.valueOf(125));
}
```

- Storing non primitive value inside Box instane.

```
public static void main3(String[] args) {
    Box box = new Box();    //OK
    box.setObject( new Date( ) );   //OK
}
```

- Using Object class we can write generic code but we can not write typesafe generic code.

```
public static void main(String[] args) {
    Box box = new Box();    //OK
    box.setObject( new Date( ) );   //OK
    String str =  (String) box.getObject(); //Downcasting :
ClassCastException
}
```

- If we want to write type-safe generic code then we should use generics.

Generics

- By passing data type as a argument, we can write generic code in java hence parameterized type is also called generics.

```
//Parameterized Class/Type
class Box<T>{   //T =>  Type Parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
        this.object = object;
    }
}
public class Program {
    public static void main(String[] args) {
        Box<Date> box = new Box<Date>( );   //Date  => Type Argument
        box.setObject(new Date());
        Date date = box.getObject();
        System.out.println(date);
    }
}
```

- Why Generics?
    1. It gives us stronger type checking at compile time. In other words, we can write type safe code in Java.
    2. It completly eleminates need explict type casting.
    3. We can write generic algorithm and data structure which reduces develpers effort.
- Type Inference:

```
Box<Date> box = new Box<Date>( );   //OK
Box<Date> box = new Box<>( );   //Type Inference
//Type argument of instane will be infered from reference.
```

- An ability of Java compiler to decide type argument of instance by looking toward type argument of reference is called type inference.
- Raw Type:

```
Box box = new Box( );   //Box=> Raw Type
//Box<Object> box = new Box<>( );
```

- If we use parameterized type w/o passing type argument then it is called raw type. In this case java.lang.Object is considered as default type argument.
- During instantiation of parameterized type, type argument must be reference type.
    - int : primitive type / value type
    - Integer : non primitive type / reference type

```
//Box<int> box = new Box<>( );  //int => NOT OK
Box<Integer> box = new Box<>( );    //Integer => OK
```

- If we want to store primitive values inside instance of parameterized type then type argument must be Wrapper Class.
- Use of Wrapper:
    1. For parsing(Converting state of string into primitive values)
        - parseXXX( );
    2. To use as a type argument in parameterized type.
        - Box box = null;
- Wrapper Class Hierarcy:
    - java.lang.Object
        - java.lang.Boolean
        - java.lang.Character
        - java.lang.Number
            - java.lang.Byte
            - java.lang.Short
            - java.lang.Integer
            - java.lang.Long
            - java.lang.Float
            - java.lang.Double
- Commonly used type parameter names in generics:
    1. T : Type
    2. N : Type of Number
    3. E : Type of Element

4. K : Type of Key
5. V : Type Of Value
6. S,U, V : Second Type Parameters

- Specifying type parameter is a job of Type/class implementor and mentioning Type argument is a job of class user.
- It is possible to pass multiple type arguments:

```
interface Pair<K, V>{
    K getKey( );
    V getValue( );
}
class Dictionary<K, V> implements Pair<K,V>{
    private K key;
    private V value;
    public Dictionary(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    @Override
    public V getValue() {
        return this.value;
    }
}
public class Program {
    public static void main(String[] args) {
        Pair<Integer, String> p = new Dictionary<>(1, "DAC");
        System.out.println("Key :   "+p.getKey());
        System.out.println("Value   :   "+p.getValue());
    }
}
```

Bounded Type Parameter

- If we want to put restriction on data type that can be used as type argument then we should specify bounded type parameter.
- Class implementor is responsible for mentioning bounded type parameter.

```
class Box<T extends Number >{   // T is bounded type parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
```

```
        this.object = object;
    }
}
public class Program {
    public static void main(String[] args) {
        Box<Number> b1 = new Box<>();   //OK
        Box<Integer> b2 = new Box<>();  //OK
        Box<Double> b3 = new Box<>();   //OK
        Box<Boolean> b4 = new Box<>();  //Not OK
        Box<String> b5 = new Box<>();   //Not OK
        Box<Date> b6 = new Box<>();     //Not OK
    }
}
```

- ArrayList is resizable array declared in java.util package

```
public class Program {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>( );
        list.add(10);
        list.add(20);
        list.add(30);
        for( Integer element : list )
            System.out.println(element);
    }
}
```

```
    public static ArrayList<Integer>getIntegerList( ){
        ArrayList<Integer> list = new ArrayList<>( );
        list.add(10);
        list.add(20);
        list.add(30);
        return list;
    }
    public static ArrayList<Double>getDoubleList( ){
        ArrayList<Double> list = new ArrayList<>( );
        list.add(10.1);
        list.add(20.2);
        list.add(30.3);
        return list;
    }
    public static ArrayList<String>getStringList( ){
        ArrayList<String> list = new ArrayList<>( );
        list.add("DAC");
        list.add("DMC");
        list.add("DESD");
        return list;
    }
```

Wild Card

- In generics, "?" is called wild card which represent unknown type.
- Types of wild card:
    1. Unbounded Wild Card
    2. Upper bounded Wild Card
    3. Lower bounded Wild Card

Unbounded Wild Card

```
    private static void printList(ArrayList<?> list) {
        for( Object element : list )
            System.out.println(element);
    }
```

- In above code, list will contain reference of ArrayList which can contain unknown/any type of element.

```
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printList( integerList );   //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printList( doubleList );    //OK

    ArrayList<String> stringList = Program.getStringList();
    Program.printList(stringList);  //OK
```

Upper bounded Wild Card

```
private static void printList(ArrayList<? extends Number> list) {
    for( Number element : list )
        System.out.println(element);
}
```

- In above code, list will contain reference of ArrayList which can contain Number and its sub type of elements only.

```
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printList( integerList );   //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printList( doubleList );    //OK

    ArrayList<String> stringList = Program.getStringList();
    Program.printList(stringList);  //NOT OK
```

Lower bounded Wild Card

```
private static void printList(ArrayList<? super Integer> list) {
    for( Object element : list )
        System.out.println(element);
}
```

- In above code, list will contain reference of ArrayList which can contain Integer and its super type of element.

```
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printList( integerList );   //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printList( doubleList );    //NOT OK

    ArrayList<String> stringList = Program.getStringList();
    Program.printList(stringList);  //NOT OK
```

Generic Method:

```
    public static <T> void print( T obj ) {
        System.out.println(obj);
    }
    public static void main(String[] args) {
        Program.print(true);    //OK
        Program.print('A');    //OK
        Program.print(123);    //OK
        Program.print(456.78);    //OK
        Program.print("Java");    //OK
        Program.print(new Date());    //OK
    }
```

```
    public static <T extends Number> void print( T obj ) {
        System.out.println(obj);
    }
    public static void main(String[] args) {
        Program.print(true);    //Not OK
        Program.print('A'); //Not OK
        Program.print(123); //OK
        Program.print(456.78);  //OK
        Program.print("Java");  //Not OK
```

```
        Program.print(new Date());  //Not OK
    }
```

Restrictions on Generics

Link : https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

Type Erasure

Link : https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Assignment:

```
- Write a LinkedList in Java.
```