Day 6

- Override is annotation declared in java.lang package.
- It is introduced in jdk1.5
- It helps developer to override method in subclass.
- Annotation always begins with @.

```
class A
{
        public void print( double a )
        {
                System.out.println("Super class");
        }
}
class B extends A
{
        @Override
        public void print( double a )
        {
                System.out.println("Sub class");
        }
}
```

Equals method

- If we want to compare state of variable/instance of value type then we should use operator ==.

```
int num1 = 10;
int num2 = 10;
if( num1 == num2 )
    System.out.println("Equal");
else
    System.out.println("Not Equal");
//Equal
```

- We can use operator == with variable of reference type.
- If we want compare state of references then we should use operator ==.

```
Employee emp1 = new Employee("Abc", 12, 25000);
Employee emp2 = new Employee("Abc", 12, 25000);
if( emp1 == emp2 )
    System.out.println("Equal");
else
    System.out.println("Not Equal");
//Not Equal
```

- If we want to compare state of instances then we should use equals method.
- "equals" is non final method of java.lang.Object class.
- Syntax: public boolean equals(Object obj);
- If we do not override equals method in sub class then its super class's equals method gets call.
- Equals method of java.lang.Object do not compare state of instances. It compares state of refereces.

```
class Object
{
    public boolean equals(Object obj)
    {
        return (this == obj);
    }
}
```

- If we want to compare state of instances then we should ovveride equals method in sub class.

```
@Override
public boolean equals( Object obj )
{
    if( obj != null )
    {
        Employee other = (Employee) obj;
        if( this.empid == other.empid )
            return true;
    }
    return false;
}
```

- In java, primitive types are not classes.

```
int num1 = 10;
int num2 = 10;
if( num1.equals( num2 ) )   //Not OK
    System.out.println("Equal");
else
    System.out.println("Not Equal");
//Compiler Error
```

- We can not use equals method with variable/instance of value type.

Boxing & AutoBoxing

- Process of converting state of instance of value type into reference type is called boxing.

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

```
int number = 10;
String strNumber = Integer.toString(number);
```

```
int number = 10;
Integer i = Integer.valueOf(number);
```

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

```
int number = 10;
Object obj = number; //AutoBoxing
```

UnBoxing & AutoUnBoxing

- Process of converting state of instance of reference type into value type is called unboxing.

```
String str = "125";
int number = Integer.parseInt(str);
```

```
Integer n1 = new Integer(125);
int n2 = n1.intValue();
```

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

```
Integer n1 = new Integer(125);
int n2 = n1;
```

Generics

- In java, if we want to write generic code then we should use generics.
- Generic Code without generics

```
class Box
{
        private Object object;
```

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

```
Object obj = new String();//Upcasting : OK
String str = (String)obj;//Downcasting : OK
```

```
Object obj = new Date();//Upcasting : OK
Date dt = (Date)obj;//Downcasting : OK
```

```
Object obj = new Date();//Upcasting : OK
String str = (String)obj;//Downcasting
//ClassCastException
```

```
Box b1 = new Box();
b1.setObject( new Date( 119, 10, 6 ));
String str = (String) b1.getObject();
//Output : ClassCastException
```

- Using java.lang.Object class we can not write type safe generic code. If we want to write typesafe generic code then we should use generics.
- By passing, datatype / type as argument, we can write generic code in java. Hence parameterized type is called generics.
- Generic code using generics:

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

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

Commonly use type parameter names:

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. U,S : Second Type Parameters

Type Inference:

- An ability of compiler to detect type of argument at compile time and use it as a type argument is called type inference.

```
Box<Date> b1 = new Box<Date>(); //OK
Box<Date> b2 = new Box<>(); //OK
```

Raw Type:

- If we instantiate generic/paramerized type without type argument then parameterized type is called Raw type.

```
Box b1 = new Box(); //Box -> Raw type
//Box<Object> b1 = new Box<>();
```

- If we want to instantiate parameterized type then type argument must be reference type.

```
Box<int> b1 = new Box();    //Not OK
Box<Integer> b1 = new Box();    // OK
```

Need of Wrapper class

1. If we want to convert String into numeric type.

2. If we want to store numeric values inside instance of parameterized type then type argument must be wrapper class

- It is possible to specify multiple type parameters for the class/interface.

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

        }
}
```

Why Generics?

- It gives us stronger type checking at compile time. In other words, it helps us to write type safe code.
- It completly eliminates explict type casting
- It helps us to implement generic algorithm and data structure.

Bounded Type Parameter

- If we want to put restriction on type / datatype that can be used as type argument then we must specify bounded type parameter

```
class Box<T extends Number >
{   }
```

```
//T extends Number : Bounded type parameter

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<String> b4=new Box<>(); //Not OK
                Box<Date> b5 = new Box<>(); //Not Ok
        }
}
```

- Specifying bounded type parameter is a job of class implementor.

ArrayList

- It is resizable array.
- It is a part of collection framework

```
ArrayList<Integer> list = null;
list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);

for( Integer element : list )
{
    System.out.println(element);
}
```

- On the basis of diffrent type argument we can not overload method.

Wild card

- In java, '?' 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 print(ArrayList<?> list)
{
    for( Object element : list )
```

```
        System.out.println(element);
}
```

- In above code, list can contain refernce of ArrayList which can contain unknown type of element.

Upper bounded wild card

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

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

Lower bounded wild card

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

- In above code, list can contain reference of ArrayList which can contain elements of Integer and its super type.
- In type argument, we can not use inheritance.

```
private static void print(
    ArrayList<Integer> list)
{
    //TODO
}
ArrayList<Integer> intList = Program.getIntegerList( );

Program.print( intList ); //OK
```

```
private static void print(
    ArrayList<Number> list)
{
    //TODO
}
```

```
ArrayList<Integer> intList = Program.getIntegerList( );

Program.print( intList ); //Not OK
```

Generic Method

- generic method without generics:

```
public static void print( Object obj )
{
    System.out.println(obj.toString());
}
```

- generic method using generics:

```
public static <T> void print( T obj )
{
    System.out.println(obj.toString());
}
```

- Generic method with bounded type parameter

```
public static <T extends Number>
void print( T obj )
{
    System.out.println(obj.toString());
}
```

Restrictions on generics

- During instantation of parameterized type, type argument must be reference type.
- On the basis of only different type argument, we can not overload method.
- We can not instantiate type parameter

```
public static <T > void print( T obj )
{
    T t = new T(); //Not Ok
    //TODO
}
```

- we can not declare, parameterized type fields static.

```
class Box<T>
{
        private static T object;
}
```

- We can not use instanceof operator with parameterized type.

```
List<Integer> list = new ArrayList<>();
if( list instanceof ArrayList<Integer>)
//Not OK
{   }
```

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.