Day 7

Generics

- If we want to write generic program then we should use generics.
- If we want write generic code then we can use
    1. java.lang.Object class
    2. Generics
- Consider following hierarchy
    - java.lang.Object
        - java.lang.String
        - java.util.Date

```
Object obj = new Date( );   //Upcasting
Date date = ( Date )obj;    //Downcasting
```

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

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

```
int number = 10;        //Value Type
Object obj1 = number;   //OK
//Object obj1 = Integer.valueOf(number);

String str = new String();      //Reference Type
Object obj2 = str;      //Upcasting
```

Generic programming without generics

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

```
                this.object = object;
        }
}
```

```
public static void main(String[] args)
{
    Box b4 = new Box();
    b4.setObject( new Date() );
    String str =  (String) b4.getObject(); //ClassCastException
}
public static void main3(String[] args)
{
    Box b3 = new Box();
    b3.setObject(10);   //b3.setObject( Integer.valueOf(10) );

    /*Object object = b3.getObject();
    Integer i1 = (Integer) object;
    System.out.println(i1.intValue()); */

    Integer i1 =  (Integer) b3.getObject();
    System.out.println(i1.intValue());
}
public static String getString( Date date )
{
    String pattern = "dd/MMMM/yyyy";
    SimpleDateFormat sdf = new SimpleDateFormat(pattern);
    return sdf.format(date);
}
public static void main2(String[] args)
{
    Box b2 = new Box();
    b2.setObject(new Date());

    /*Object object = b2.getObject();
    Date date = (Date) object;*/

    Date date = (Date) b2.getObject();
    String strDate = Program.getString(date);
    System.out.println(strDate);
}
public static void main1(String[] args)
{
    Box b1 = new Box();
}
```

- 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.

Generic Programming using generics

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

```
Box<Date> b1 = new Box<Date>(  ); //Date : Type Argument
b1.setObject(new Date());
Date date =  b1.getObject();
```

Why Generics:

1. It gives us stronger type checking at compie time. In other words we can write type safe code.
2. It completly eliminates need of explicit typecasting.
3. It helps developer to write generic algorithm and data structure.

Type Inference

- An ability of compiler to detect type of argument automatically is called type inference.

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

- If we use parameterized type without type argument then such type is called raw type. In this case java.lang.Object is considered as default type argument.

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

- In Generics, type argument must be reference type.
- If we want to store numeric value inside instance of parameterized type then type argument must be wrapper class.

```

```

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

- In Type argument, we can not use inheritance.

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

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

Commonly used parameter Type Names in java

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

- We can specify multiple type parameters for parameterized type.

```
class Pair<K, V>
{
        private K key;
        private V value;
        public Pair()
        {       }
        public Pair(K key, V value)
        {
                this.key = key;
                this.value = value;
        }
        public K getKey()
        {
                return key;
        }
        public V getValue()
        {
                return value;
        }
}
```

```
Pair<Integer, String> p = new Pair<>( 1, "DAC" );
Integer key = p.getKey();
String value = p.getValue();
System.out.println(key+"        "+value);
```

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 Box<T extends Number > //T Bounded Type Parameter
{
        //TODO
}
```

```
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.
- On basis of only different type argument we can not overload method.

Wild Card

- In generics "?" is called wild card, which represents 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 reference of ArrayList which can contain unknown/any type of element.

Upper Bounded Wild Card

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

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

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 only.

Generic Method

```
//Generic Method without generics
public static void show( Object object )
{
    System.out.println( object);
}
//Generic Method using generics
public static <T> void display( T object )
{
    System.out.println( object);
}
public static <T extends Number > void print( T object )
{
    System.out.println( object);
}
public static void main(String[] args)
{
    //Program.print( 'A' );     //Not OK
    Program.print( 10 );
    Program.print( 3.14 );
    //Program.print( "KDAC" );  //Not OK
    //Program.print( new Date() );      //Not OK
}
```

Restrictions on generics

1. During instantiation of parameterized type, type argument must be reference type.

```
Box<int> b1 = new Box();     //Not OK
Box<Integer> b3 = new Box();     //OK
```

2. We can not use inheritance in type argument

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

3. On basis of only different type argument we can not overload method.

```
private static void print(ArrayList<Integer> list) ; //Not OK
private static void print(ArrayList<Double> list);  //Not OK
private static void print(ArrayList<String> list);  //Not OK
private static void print(ArrayList<?> list);  //OK
```

4. We can not instantiate type parameter

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

5. We not declare Type parameter field static.

```
class Box<T>
{
        private static T object;     //Not OK
}
```

6. We Cannot perform instanceof check against parameterized type

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

Fragile Base Class Problem

- If we make changes in method of super class then it is necessary to recompile all the sub classes. It is called fragile base class problem.

Interface

- Set of rules is also called as specification / Standard.
- If we want to define rules/specification for vendors/sub classes then we should use interface.
- Advantages of interface
    1. It helps us to build trust between service provider and service consumer.
    2. It helps to minimize vendor dependancy(loose coupling)
- interface is keyword in java.
- interface is non primitive / reference type in java.
- Interface can contain
    1. Nested interface
    2. Constant / Final field
    3. Abstract Method
    4. Default Method
    5. Static Method
- We can not instantiate interface but we can create reference of it.
- We can not define constructor inside interface.
- Interface fields are by default public, static and final.
- Interface methods are by default public and abstract.

```
interface A
{
        int num1 = 10;
        //public static final int num1 = 10;
        void print( );
        //public abstract void print( );
}
```

- If we want implement rules of interface then we should use implements keyword.
- It is mandatory to override all the abstract methods of interface otherwise sub class can be considered as abstract.
- First Solution

```
abstract class B implements A
{       }
```

- Second Solution

```
//Interface implementation class
class B implements A
{
        @Override
```

```
        public void print()
        {
                System.out.println("Num1        :       "+A.num1);
        }
}
```

- How to use interface

```
A a =  new B(); //Upcasting
a.print();//DMD
```

Class and Interface Syntax

- Interfaces : I1, I2, I3
- Classes : C1, C2, C3

1. I2 implements I1; //Not OK
2. I2 extends I1; //OK : Interface Inheritance
3. I3 extends I1 ,I2; //OK : Multiple Interface Inheritance
4. I1 extends C1; //Not OK
5. C1 extends I1; //Not OK
6. C1 implements I1; //OK : Interface Implementation Inheritance
7. C1 implements I1,I2; //OK : Multiple Interface Impl Inheritance
8. C2 implements C1; //Not OK
9. C2 extends C1; //OK : Implementation Inheritance
10. C3 extends C1,C2;//Not OK : Multiple Implementation Inheritance
11. C2 implements I1 extends C1; //Not OK
12. C2 extends C1 implements I1; //OK

- Abstract helper class which allows us to override some of the methods of interface is called adapter class.

Commonly used interfaces in core java

1. java.lang.AutoCloseable
2. java.io.Closeable
3. java.lang.Cloneable
4. java.lang.Comparable
5. java.util.Comparator
6. java.lang.Iterable
7. java.util.Iterator
8. java.io.Serializable

Cloneable Implementation

```
Date dt1 = new Date();
Date dt2 = dt1; //Shallow Copy of reference.
```

- If we want to create new instance from existing instance then we should use clone() method.
- clone() is native, non final method of java.lang.Object class
- Syntax: protected Object clone() throws CloneNotSupportedException
- Inside clone method, if we want to create shallow copy of current instance then we should use "super.clone()" method.
- Without implementing Cloneable interface, if we try to create clone of instance then "clone()" method throws CloneNotSupportedException.

Marker Interface

- Empty interface is also called as marker interface / tagging interface.
- Marker interface is used to generate metadata for compiler as well as jvm.
- Example
    1. java.lang.Cloneable
    2. java.util.EventListener
    3. java.util.RandomAccess
    4. java.io.Serializable
    5. java.rmi.Remote

Functional Programming

Default Method

- At runtime, if we want to modify interface then we should use default method.
- It is optional to override default method in sub class but it must contain body.
- We can call interafce default method in sub class. Syntax is : "InterfaceName.super.defaultMethodName();"

Static Interface Method

- It is a helper method that we can use in default method as well as sub class.
- We can not override static interface method.

Functional Interface

- If interface contains only one abstract method then it is called functional interface.
- Since functional interface Contains Single Abstract Method, it is also called as SAM interface.
- Functional interface can contain multiple default methods and static methods but it must contain one abstract method.
- FunctionalInterface is annotation declared in java.lang package.
- @FunctionalInterface annotation help to decide wheather interface is functional or not.

```
@FunctionalInterface
interface A
{
        void f1();  //Method Descriptor
}
```

- Abstract method declared in functional interface is called method descriptor.
- java.util.function package contains all Oracle supplied functional interfaces.
- Following are some of the functional interfaces declared in java.util.function package.

1. Predicate
    - boolean test(T t)
2. Supplier
    - T get()
3. Consumer
    - void accept(T t)
4. Function<T,R>
    - R apply(T t)
5. UnaryOperator
    - It is sub interface of Function<T,R> interface

- If we want to implement functional interface then we should use lambda expression and method reference.

Lambda Expression

- If expression contains lambda operator then such expression called lambda expression
- operator "->" is called lambda operator.
- Syntax: (Input Parameters) -> Lambda Body;
- Lambda body can contain one or multiple statements. If lambda body contains multiple statements then it is mandatory to provide curly braces.
- Lambda expression is also called as anonymous method.
- If we want to define labda expression then we should take help of method descriptor.
- Example 1

```
@FunctionalInterface
interface Printable
{
        void print( );  //Method Descriptor
}
```

```
Printable p = ( )-> System.out.println("Hello Lambda");
p.print();
```

- Example 2

```
@FunctionalInterface
interface Math
{
```

```
        int sum( int num1, int num2 );
}
```

```
//Math m = ( int num1, int num2 )-> num1 + num2; //or
//Math m = ( int x, int y )-> x + y; //or
Math m = (  num1,  num2 )-> num1 + num2;
int result = m.sum(10, 20);
System.out.println("Result      :       "+result);
```

- Example 3

```
@FunctionalInterface
interface Math
{
        int square( int number );
}
```

```
//Math m = ( int number )-> number * number;
//Math m = (  number )-> number * number;
Math m = number -> number * number;
int result = m.square(5);
System.out.println("Result      :       "+result);
```

- Example 4:

```
@FunctionalInterface
interface Math
{
        int factorial( int number );
}
```

```
Math m =  number -> {
int result = 1;
for( int count = 1; count <= number; ++ count )
    result = result * count;
return result;
};

int result = m.factorial(5);
System.out.println("Result      :       "+result);
```

Method Reference

```
@FunctionalInterface
interface Printable
{
        void print( );
}
```

```
public static void showRecord( )
{
    System.out.println("Inside showRecord");
}
public void displayRecord( )
{
    System.out.println("Inside displayRecord");
}
public static void main(String[] args)
{
    //Printable p = ( )->System.out.println("hello");
    //Printable p = Program::showRecord;

    Program program = new Program();
    Printable p = program::displayRecord;
    p.print();
}
```