Object Oriented Programming -2 [Java]

**Multiple Inheritance**

When one class has more than one super classes then it is called multiple inheritance. For e.g.

Teacher Student   
TA

Here a TA is both a teacher and a student, it has two parent classes. Java doesn’t allow multiple inheritence.

**Interfaces**

Interfaces in java are pure abstract i.e all methods in an interface are public and abstract and fileds are public, final and static by default. E.g :

public interface VehicleInterface { int a = 9; public String getType(); }

A class “implements” an interface. And a class implementing a interface must implement all its methods otherwise the class will have to be declared abstract. Also a class can implement multiple interfaces and if it does so then it will have to implement all the methods in the interfaces that it is implementing.

**Multiple inheritance in Java by interface**

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

interface Printable{

void print();

}

interface Showable{

void show();

}

class sample implements Printable,Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

sample obj = new sample();

obj.print();

obj.show();

}

}

Output:

Hello

Welcome

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

interface Printable{

void print();

}

interface Showable{

void print();

}

class TestTnterface1 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestTnterface1 obj = new TestTnterface1();

obj.print();

}

}

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interfaces serve as a contract, If a non abstract class is implemeting an interface then without looking into the code of the class we can be sure that the class must have implemented the methods in the interface, else the class would have been abstract.

**Some points about interface :**

1. We cannot instantiate an interface.
2. An interface does not contain any constructors.
3. All of the methods in an interface are abstract.
4. An interface cannot contain instance fields i.e non static fields. The only fields that can appear in an interface must be declared both static and final.
5. An interface is not extended by a class; it is implemented by a class.
6. An interface can extend multiple interfaces.

**Generics**

Suppose we make a pair class to store two ints.

public class Pair {

int first;

int second;

}

Now if we want to have a pair of two chars/strings/double then we will have to create separate pair classes for each of them. Generics allow us to create a single Pair class that will work for different types. Creating a generic class Syntax of a generic Pair class is :

public class Pair<T> {

T first;

T second;

}

Here represents the type parameter and “T” is an identifier that specifies a generic type name, it could have been any other letter. Now if we want to have pair of two ints, then syntax will be :

Pair<Integer> pInts = new Pair<Integer>();

Pair<String> pStrings = new Pair<Strings>(); // Pair of two

// Strings

Note that type parameters can represent only reference types, not primitive types. So for primitives int,char etc java has corressponding Wrapper classes Integer, Character etc. A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.

**Multiple Type Parameters**

We can have multiple type parameters as well i.e we can create a pair class where “first” and “second” can be of different types unlike the Pair class defined above where both have to be of same type.

public class Pair<T,S> {

T first;

S second;

}

Its instance can be created as follows :

Pair<Integer,String> pair = new Pair<Integer, String>();

So here pair.first is an Integer and pair.second is a String.

**Multilayer Generic Parameters**

The genric parameters can be multilayered.

Pair<Pair<Integer>> pLayered = new Pair<>();

Here pLayered.first and pLayered.second are themselves pair of Integers.

**Generic Methods**

Just like we saw generic classes above, we can make methods also accept generic parameters. We can make a method generic even when the class isn’t generic. Here is a sample printArray method that works for generic inputs.

public static<T> void printArray(T input[]){

for(int i = 0; i < input.length; i++){

System.out.print(input[i] +" ");

}

}

**Bounded Type Parameters:**

Many a times when you might want to restrict the kinds of types that are allowed to be passed to a type parameter. Say we want to create a generic sort function. In order to sort elements we will have to compare them. The “<” or “>” are not defined for non–primitives. So instead we will have to use compareTo() method (in Comparable interface) which compares two objects and returns an int based on result. Now in our sort function we should allow only those non-primitives who have compareTo() method defined for them or in other terms who have implemented the Comparable interface (as interface serves as a contract, so if a non-abstract class has implemeted has implemented Comparable method then we can be sure that it has compareTo() method ). We can do this as shown below :

public static<T extends Comparable<T>> void sort(T input[]){

T temp;

for(int i = 0; i < input.length; i++){

for(int j = 0; j < input.length - i - 1; j++){

if(input[j].compareTo(input[j+1])==1){

temp = input[j+1];

input[j+1] = input[j];

input[j] = temp;

}

}

}

}

When we write

<T extends Comparable<T>>

this means that only those parameters are allowed who have implemented Comparable Interface.

CompareTo() compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

**Java ArrayList class**

1. Java ArrayList class uses a dynamic array for storing the elements.It extends AbstractList class and implements List interface.
2. Java ArrayList class can contain duplicate elements.
3. Java ArrayList class maintains insertion order.
4. Java ArrayList class is non synchronized.
5. Java ArrayList allows random access because array works at the index basis.
6. In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

**Java Non-generic Vs Generic Collection**

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

ArrayList al=new ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

ArrayList<String> al=new ArrayList<String>();//creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

Note:

1. When we create an ArrayList in this way, default constructor is invoked and will internally create an array of Object with default capacity, which is 10.
2. If we display al.size() it will show 0 because nothing is added in ArrayList.
3. When the size reaches initial capacity i.e 10, a new array is created with NewCapacity = 1.5\*(InitialCapacity) and all elements are copied to new array (All this work is done internally ). Hence ArrayList is also called dynamic array.

ArrayList<String> arr=new ArrayList<String>(5);

When we create an ArrayList in this way, constructor with an integer argument is invoked and will internally create an array of Object with the size, specified in the constructor argument, which happens to be 5 in this case.

Example of Java ArrayList class

import java.util.\*;

class TestCollection1{

public static void main(String args[]){

ArrayList<String> al=new ArrayList<String>();//creating arraylist

al.add("Ravi");//adding object in arraylist

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator itr=al.iterator();//getting Iterator from arraylist to traverse elements

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

**Important methods of ArrayList**

**void add(int index, Object element)**

Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 OR index > size()).

**boolean contains(Object o)**

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).

**int indexOf(Object o)**

Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

**Object get(int index)**

Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 OR index >= size()).

**Object remove(int index)**

Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range (index < 0 OR index >= size()).

BigInteger

import java.math.BigInteger;

public class rough {

public static void main(String[] args) {

BigInteger BI1 = new BigInteger("123456789000987654321");

BigInteger BI2 = new BigInteger("10000000000000000000000");

BigInteger addBI = BI1.add(BI2); //returns BI1 + BI2

BigInteger subBI = BI1.subtract(BI2); //returns BI1 - BI2

BigInteger div1BI = BI1.divide(BI2); //returns BI1 / BI2

BigInteger div2BI = BI2.divide(BI1); //returns BI2 / BI1

BigInteger mulBI = BI1.multiply(BI2); // returns BI1 \* BI2

BigInteger greaterBI = BI1.max(BI2);// returns greater BI

BigInteger powBI = BI1.pow(10); // returns BI1 ^ 10

System.out.println("addBI : " + addBI);

System.out.println("subBI : " + subBI);

System.out.println("div1BI : " + div1BI);

System.out.println("div2BI : " + div2BI);

System.out.println("mulBI : " + mulBI);

System.out.println("greaterBI : " + greaterBI);

System.out.println("powBI : " + powBI);

}

}

OUTPUT:

addBI : 10123456789000987654321

subBI : -9876543210999012345679

div1BI : 0

div2BI : 81

mulBI : 1234567890009876543210000000000000000000000

greaterBI : 10000000000000000000000

powBI : 822526259212904680838519874066278170250073167557769325147372914763874381365243060369656652865583330954816509453580245270199503960590584543758212402960183532585867869117016059804887005682741566940511201

Note:

1. Big*Integer.divide(Big*Integer2) will return a whole number.
2. Values of BI1 and BI2 will never change.
3. BigInteger is an **immutable class**. So whenever you do any arithmetic, you have to reassign the output to a variable.
4. BigInteger is implemented with **int[]** .
5. **Range of BigInteger** : There is no theoretical limit. The BigInteger class allocates as much memory as it needs to hold all the bits of data it is asked to hold. There are, however, some practical limits, dictated by the memory available.

Another way to declare BigInteger :

//BigInteger newBI1 = new BigInteger(StringFormatNum , Radix);

BigInteger newBI2 = new BigInteger("111101011111" , 2);

BigInteger newBI3 = new BigInteger("123421211114" , 7);

//newBI2 = 3935

//newBI3 = 2688198419