OOPDP

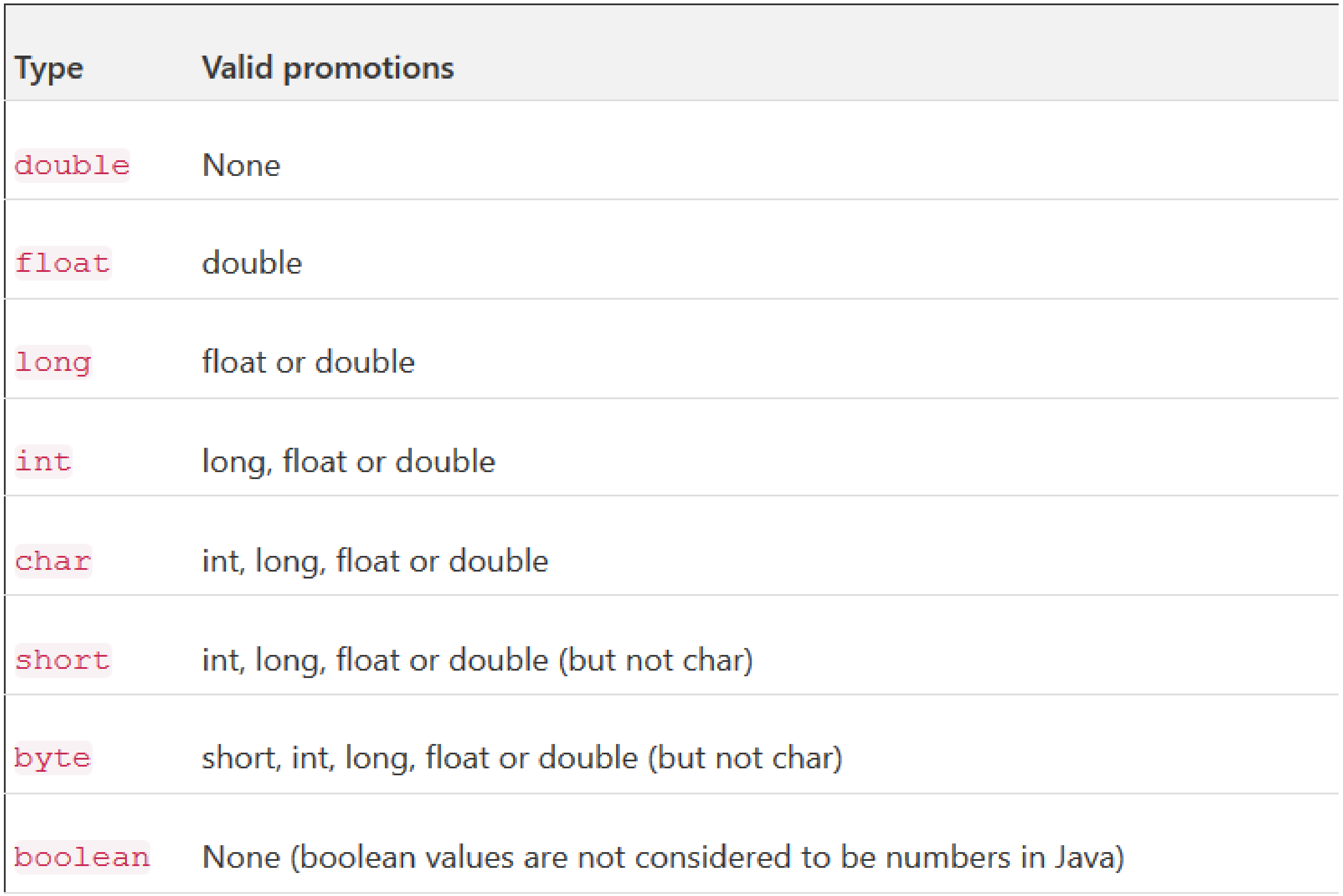
MODULE II &III

Argument Promotion and Casting

- converting an argument's value to the type that the method expects to receive in its corresponding parameter.
- For example, a program can call Math method sqrt() with an integer argument even though the method expects to receive a double argument (but not vice versa).

Widening or Automatic Type Conversion

- Widening conversion takes place when two data types are automatically converted.
- This happens when:
  - The two data types are compatible.
  - When we assign value of a smaller data type to a bigger data type.



Type	Valid promotions
`double`	None
`float`	double
`long`	float or double
`int`	long, float or double
`char`	int, long, float or double
`short`	int, long, float or double (but not char)
`byte`	short, int, long, float or double (but not char)
`boolean`	None (boolean values are not considered to be numbers in Java)

Example:

```
public class Mainclass
{
public static void main(String []args)
{
char c='a';
int a=c;
System.out.println(a);
  }}
```

- Output:

```
97
```

Narrowing or Explicit Conversion

- If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.
- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

```
public class Mainclass
{
public static void main(String []args)
{
 int c=12;
  char a=c;
System.out.println(a);
 }}
```

- Output:

```
Mainclass.java:6: error: incompatible types: possible lossy
   conversion from int to char char a=c;
^ 1 error
```

```
//Java program to illustrate explicit type conversion
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);

        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

- Output:

```
Double value 100.04
Long value 100
Int value 100
```

Scope

- Scope of a variable is the part of the program where the variable is accessible.
- Java scope rules can be covered under following categories.
  - **Member Variables (Class Level Scope)**
  - **Local Variables (Method Level Scope)**
  - **Loop Variables (Block Scope)**

Access Modifiers

- As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor , variable , method or data member.
- There are four types of access modifiers available in java:
  - Default – No keyword required
  - Private
  - Protected
  - Public

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Example1

```
class A
{
private int data=40;
  private void msg()
{System.out.println("Hello java");
}
 }
public class Simple
{
public static void main(String args[])
{     A obj=new A();
System.out.println(obj.data);//Compile Time Error
obj.msg();//Compile Time Error
 }  }
```

Example2

```
class A{
private A()
{}//private constructor
void msg()
{
System.out.println("Hello java");}
}
public class Simple
{
 public static void main(String args[])
{
  A obj=new A();//Compile Time Error
 }
}
```

Example3

```
//save by A.java
package p1;
 class A
{
    void msg()
    {
    System.out.println("Hello");
    }
}
```

```
//save by B.java
package mypack;
import p1.*;
public class B{
      public static void main(String args[])
     {
       A obj = new A();//Compile Time Error
       obj.msg();//Compile Time Error
      }
}
```

Example4

```
//save by A.java

package pack;
public class A{
public void msg(){
System.out.println("Hello");
}
}
```

```
package mypack;  import pack.*;
class B
{
public static void main(String args[])
{
A obj = new A();
  obj.msg();
}  }
```

- Output:

```
Hello
```

Example: Block Scope

```
public class Test
{
    public static void main(String args[])
    {
        {
            // The variable x has scope within
            // brackets
            int x = 10;
            System.out.println(x);
        }

        // Uncommenting below line would produce
        // error since variable x is out of scope.

        // System.out.println(x);
    }
}
```

Example:Loop block

```
class Test
{
    public static void main(String args[])
    {
        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}
```

In java, name of variable of inner and outer loop must be different.

```
class Test
{
    public static void main(String args[])
    {
        int a = 5;
        for (int a = 0; a < 5; a++) //error
        {
            System.out.println(a);
        }
    }
}
```

Example:

```
public class HelloWorld
{
int a;
public static void main(String args[])
 {
int a = 5;
 System.out.println(a);
HelloWorld ob=new HelloWorld();
   System.out.println(ob.a);
 } }
```

- Output

```
5
0
```

Commandline Arguments

- Command-line arguments in Java are used to pass arguments to the main program.
- If you look at the Java main method syntax, it accepts String array as an argument.
- When we pass command-line arguments, they are treated as strings and passed to the main function in the string array argument.
- The arguments have to be passed as space-separated values.
- We can pass strings and primitive data types as command-line arguments.
- The arguments will be converted to strings and passed into the main method string array argument.

Example1:

```
class CommandLineExample
{
    public static void main(String args[])
    {
    System.out.println("Your first argument is: "+args[0]);
    }
}
```

```
compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo
```

- Output:

```
Your first argument is: sonoo
```

Example2:

```
class A
{
    public static void main(String args[])
    {
     for(int i=0;i<args.length;i++)
    System.out.println(args[i]);
     }
}
```

```
compile by > javac A.java
run by > java A sonoo jaiswal 1 3 abc
```

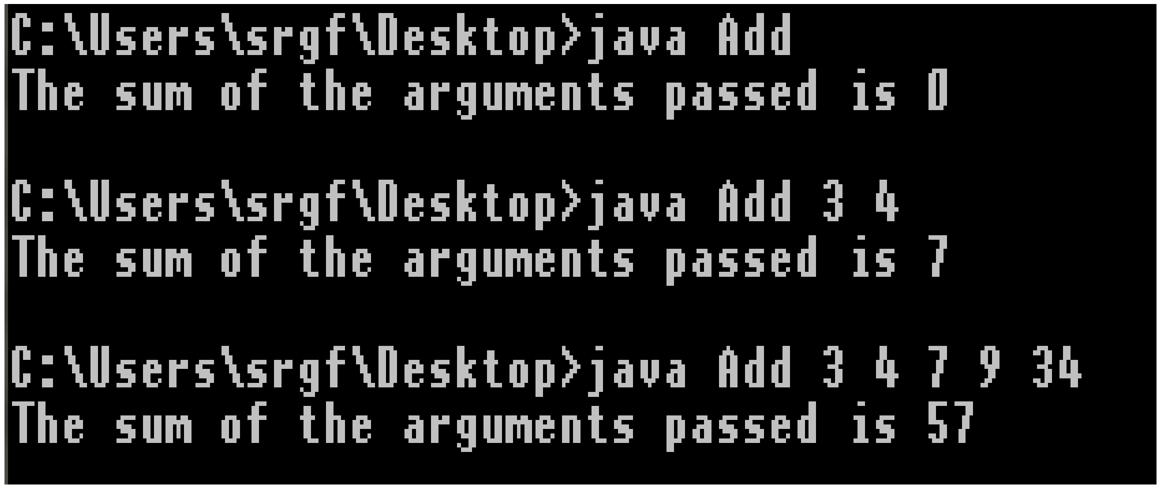
- Output:

```
sonoo

jaiswal

1

 3

abc
```

Example3:

```
public class Add {

        public static void main(String[] args)
{
            int sum = 0;
            for (int i = 0; i < args.length; i++)
{
                sum = sum + Integer.parseInt(args[i]);
            }

System.out.println("The sum of the arguments passed is " + sum);
        }
    }
```

Output:



```
C:\Users\srgf\Desktop>java Add
The sum of the arguments passed is 0

C:\Users\srgf\Desktop>java Add 3 4
The sum of the arguments passed is 7

C:\Users\srgf\Desktop>java Add 3 4 7 9 34
The sum of the arguments passed is 57
```

Arrays And ArrayList

- An **array** is a collection of items.
- Each slot in the **array** can hold an **object** or a primitive value.
- **Arrays** in **Java** are **objects** that can be **treated** just like other **objects** in the language.

Features of array

- In Java all arrays are dynamically allocated
- Since arrays are objects in Java, we can find their length using member length.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Declarations:

- int a[];
- MyClass ob[];

- Although the above first declaration establishes the fact that intArray is an array variable, **no array actually exists**.
- It simply tells to the compiler that this(intArray) variable will hold an array of the integer type.
- To link intArray with an actual, physical array of integers, you must allocate one using **new** and assign it to intArray.

Instantiating an Array in Java

- int[] intArray = new int[20];

- The elements in the array allocated by *new* will automatically be initialized to **zero** (for numeric types), **false** (for boolean), or **null** (for reference types).

Array Literal

- In a situation, where the size of the array and variables of array are already known, array literals can be used.
- int[] intArray = {1,2,3,4,5,6,7,8,9,10 };

Array of Objects

```
class Student
{
   public int roll_no;
   public String name;
   Student(int roll_no, String name)
   {
      this.roll_no = roll_no;
      this.name = name;
   }
}
```

(Cntd..)

```
Public class Mainclass
{
   public static void main (String[] args)
   {
      Student[] arr;
      arr = new Student[5];
      arr[0] = new Student(1,"aman");
      arr[1] = new Student(2,"vaibhav");
      arr[2] = new Student(3,"shikar");
      arr[3] = new Student(4,"dharmesh");
      arr[4] = new Student(5,"mohit");

      for (int i = 0; i < arr.length; i++)
         System.out.println("Element at " + i + " : " +
                     arr[i].roll_no +" "+ arr[i].name);
   }
}
```

- Output:

```
Element at 0 : 1 aman
Element at 1 : 2 vaibhav
Element at 2 : 3 shikar
Element at 3 : 4 dharmesh
Element at 4 : 5 mohit
```

ArrayIndexOutOfBoundsException

- JVM throws
- **ArrayIndexOutOfBoundsException** to indicate that array has been accessed with an illegal index.
- The index is either negative or greater than or equal to size of array.

Example:

```
class GFG
{
   public static void main (String[] args)
   {
      int[] arr = new int[2];
      arr[0] = 10;
      arr[1] = 20;

      for (int i = 0; i <= arr.length; i++)
         System.out.println(arr[i]);
   }
}
```

- **Output:**

```
10

20


Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 2 at
GFG.main(File.java:12)
```

Multidimensional Arrays

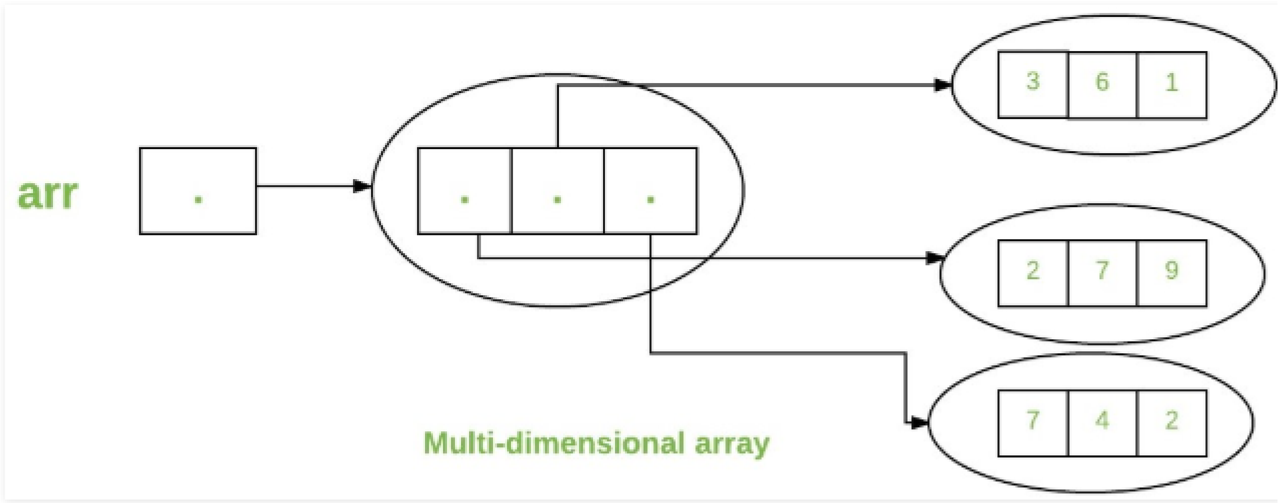
- Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other array.
- These are also known as Jagged Arrays.
- int[][] intArray = new int[10][20]; //a 2D array int[][][] intArray = new int[10][20][10]; //a 3D array

Example:

```
class multiDimensional
{
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++)
        {
            for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

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

- **Output:**

```
2 7 9
 3 6 1
7 4 2
```



Passing Arrays to Methods

```
class Test
{
   // Driver method
   public static void main(String args[])
   {
      int arr[] = {3, 1, 2, 5, 4};

      // passing array to method m1
      sum(arr);

   }

   public static void sum(int[] arr)
   {
      // getting sum of array values
      int sum = 0;

      for (int i = 0; i < arr.length; i++)
         sum+=arr[i];

      System.out.println("sum of array values : " + sum);
   }
}
```

Output:

```
sum of array values : 15
```

Returning Arrays from Methods

```
class Test
{
   // Driver method
   public static void main(String args[])
   {
      int arr[] = m1();

      for (int i = 0; i < arr.length; i++)
         System.out.print(arr[i]+" ");


   }

   public static int[] m1()
   {
      // returning  array
      return new int[]{1,2,3};
   }
}
```

Output:

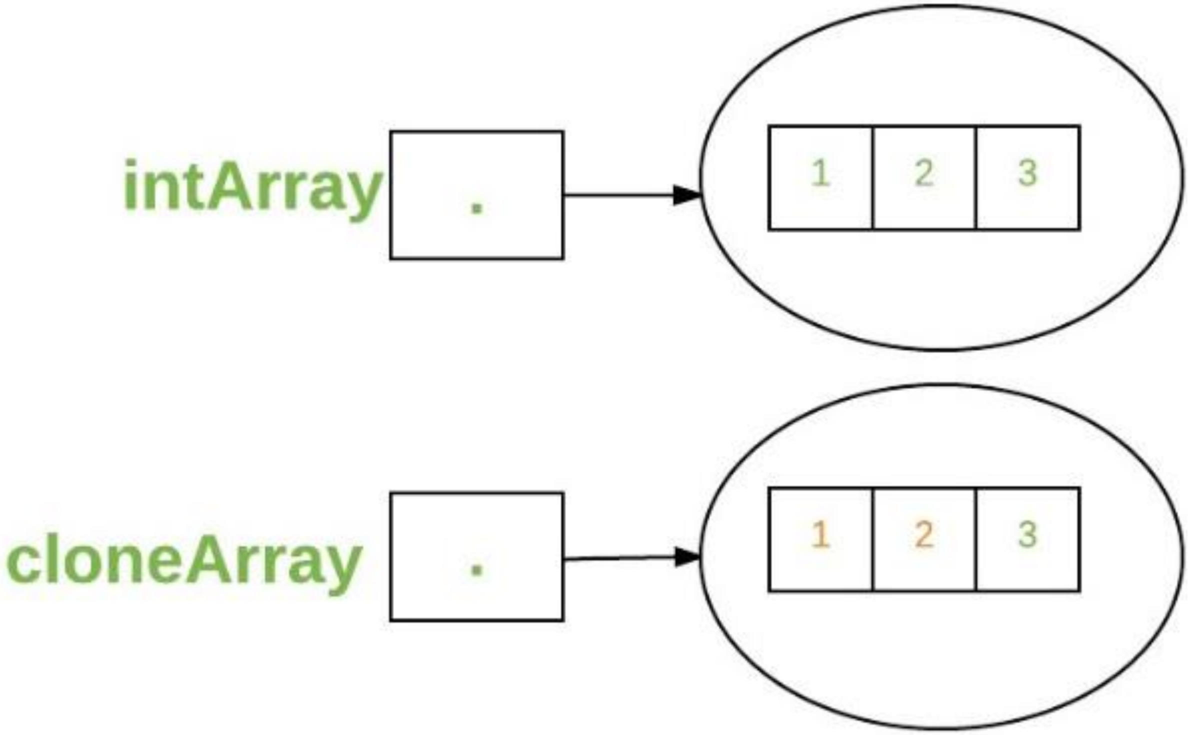
```
1 2 3
```

Cloning of arrays

```
class Test
{
   public static void main(String args[])
   {
      int intArray[] = {1,2,3};

      int cloneArray[] = intArray.clone();

      // will print false as deep copy is created
      // for one-dimensional array
      System.out.println(intArray == cloneArray);

      for (int i = 0; i < cloneArray.length; i++) {
         System.out.print(cloneArray[i]+" ");
      }
   }
}
```

Output:

```
false
1 2 3
```


intArray
1
2
3
cloneArray
1
2
3

cloning of multi-dimensional arrays

```
class Test
{
    public static void main(String args[])
    {
        int intArray[][] = {{1,2,3},{4,5}};

        int cloneArray[][] = intArray.clone();

        // will print false
        System.out.println(intArray == cloneArray);

        // will print true as shallow copy is created
        // i.e. sub-arrays are shared
        System.out.println(intArray[0] == cloneArray[0]);
        System.out.println(intArray[1] == cloneArray[1]);

    }
```

Output:

```
false
true
true
```

ArrayList

- ArrayList is a part of collection framework and is present in java.util package.
- **Collections in Java**
  - A Collection is a group of individual objects represented as a single unit.
  - Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
  - The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.

- It provides us dynamic arrays in Java.
- Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as similar to vector in C++.

Constructors in Java ArrayList

- **ArrayList()**:
- This constructor is used to build an empty array list
- **ArrayList(Collection c):**
- This constructor is used to build an array list initialized with the elements from collection c
- **ArrayList(int capacity):**
- This constructor is used to build an array list with initial capacity being specified

```
import java.io.*;
import java.util.*;

class arrayli
{
    public static void main(String[] args)
                        throws IOException
    {
        int n = 5;
         ArrayList<Integer> arrli = new ArrayList<Integer>(n);
        for (int i=1; i<=n; i++)
            arrli.add(i);

        // Printing elements
        System.out.println(arrli);

        // Remove element at index 3
        arrli.remove(3);

        // Displaying ArrayList after deletion
        System.out.println(arrli);

        // Printing elements one by one
        for (int i=0; i<arrli.size(); i++)
            System.out.print(arrli.get(i)+" ");
    }
}
```

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

```
import java.util.*;
public class GFG {

    public static void main(String[] args)
    {
        // create an ArrayList which going to
        // contains a list of Student names which is actually
        // string values
        ArrayList<String> students = new ArrayList<String>();

        // Add Strings to list
        // each string represents student name
        students.add("Ram");
        students.add("Mohan");
        students.add("Sohan");
        students.add("Rabi");
        System.out.println("list of Students:");

            students.forEach((n) -> print(n));
    }
    public static void print(String n)
    {
        System.out.println("Student Name is " + n);
    }
}
```

- **Output:**

```
list of Students:
 Student Name is Ram
Student Name is Mohan
Student Name is Sohan
 Student Name is Rabi
```

```
import java.util.ArrayList;

class GFG {
    public static void main(String[] args)
    {

        // creating an Empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(4);

             arr.add(1);
        arr.add(2);
        arr.add(3);
        arr.add(4);
          boolean ans = arr.contains(2);

        if (ans)
            System.out.println("The list contains 2");
        else
            System.out.println("The list does not contains 2");



    }
}
```

Output:

```
The list contains 2
```

Differences between Array and ArrayList

- 1. An array is basic functionality provided by Java.
- ArrayList is part of collection framework in Java.
- Therefore array members are accessed using [], while ArrayList has a set of methods to access elements and modify them.

- 2. Array is a fixed size data structure while ArrayList is not.
- One need not to mention the size of Arraylist while creating its object.
- Even if we specify some initial capacity, we can add more elements.

- 3.Array can contain both primitive data types as well as objects of a class depending on the definition of the array.
- However, ArrayList only supports object entries, not the primitive data types.

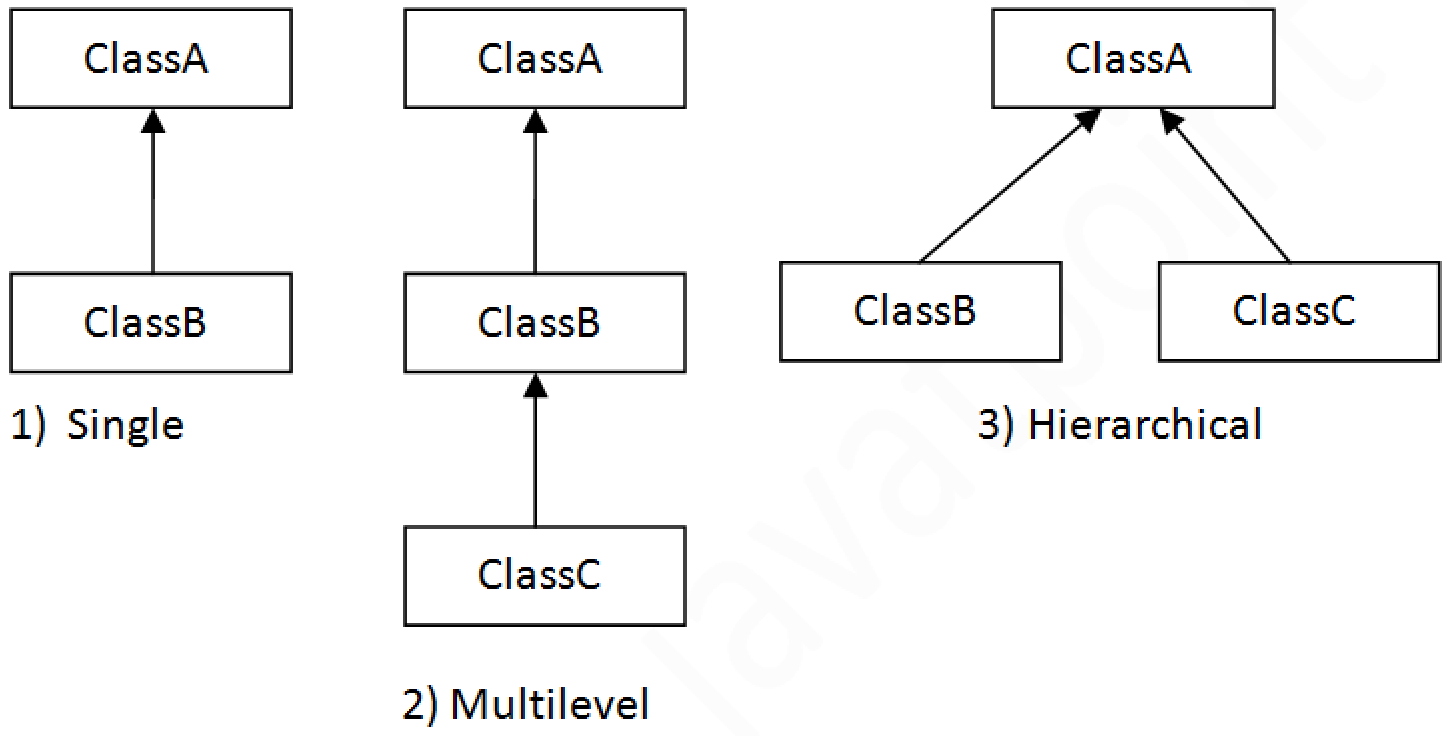
- Note: When we do arraylist.add(1); : it converts the primitive int data type into an Integer object.

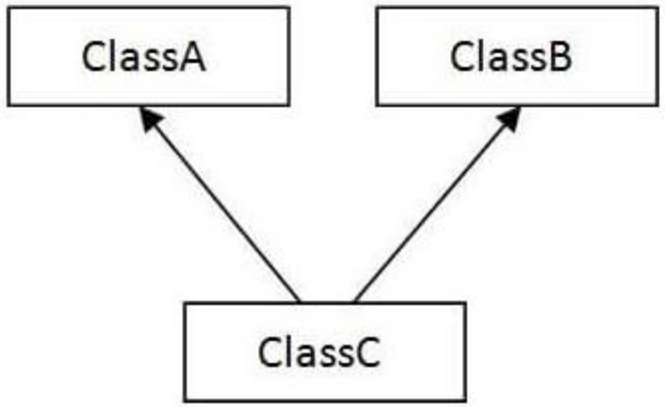
- 4. Since ArrayList can't be created for primitive data types, members of ArrayList are always references to objects at different memory locations .
- Therefore in ArrayList, the actual objects are never stored at contiguous locations.
- References of the actual objects are stored at contiguous locations. In array, it depends whether the arrays is of primitive type or object type.
- In case of primitive types, actual values are contiguous locations, but in case of objects, allocation is similar to ArrayList.

Inheritance

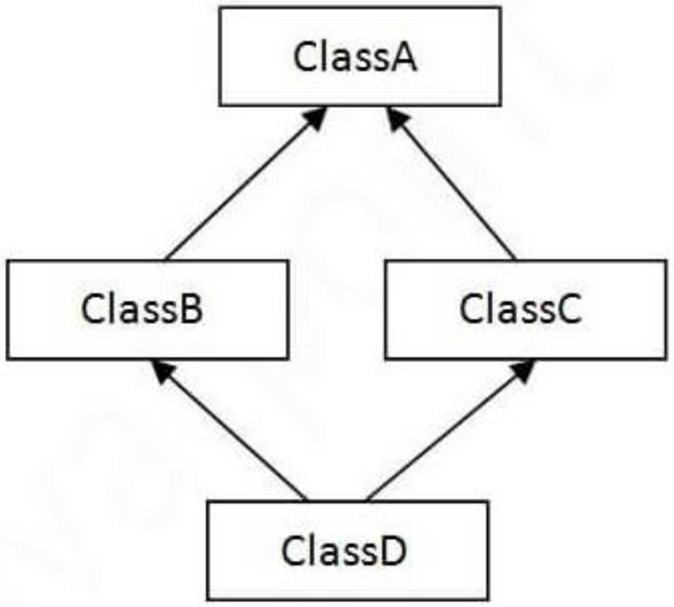
- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship

Types of Inheritance in Java





4) Multiple



5) Hybrid

Example1:Single Level

```
class Animal{
    void eat(){System.out.println("eating...");}
    }
    class Dog extends Animal{
    void bark(){System.out.println("barking...");}
      }
class TestInheritance{
    public static void main(String args[]){
    Dog d=new Dog();
    d.bark();
    d.eat();
    }
}
```

- Output:

```
barking…
eating…
```

Example2:Multiple Level

```
class Animal{
    void eat(){System.out.println("eating...");}
    }
    class Dog extends Animal{
    void bark(){System.out.println("barking...");}
     }
class BabyDog extends Dog{
    void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
    public static void main(String args[]){
    BabyDog d=new BabyDog();
    d.weep();
    d.bark();
    d.eat();
    }
}
```

- Output:

```
weeping…
barking…
eating…
```

Example3:Hierarchical Inheritance

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
    void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
    public static void main(String args[]){
    Cat c=new Cat();
    c.meow();
    c.eat();
    //c.bark();//C.T.Error
    }
}
```

Output:

```
meowing…
eating…
```

Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

Inheritance and constructors

- In Java, constructor of base class with no argument gets automatically called in derived class constructor.

```
class Base {
        Base() {
          System.out.println("Base Class Constructor Called ");
        }
}

class Derived extends Base {
        Derived() {
          System.out.println("Derived Class Constructor Called ");
        }
}

public class Main {
        public static void main(String[] args) {
          Derived d = new Derived();
        }
}
```

- Output:

```
Base Class Constructor Called

Derived Class Constructor Called
```

- if we want to call parameterized contructor of base class, then we can call it using super().
- The point to note is **base class constructor call must be the first line in derived class constructor**.

```
class Base {
        int x;
        Base(int _x) {
          x = _x;
        }
}
class Derived extends Base {
        int y;
        Derived(int _x, int _y) {
          super(_x);
          y = _y;
        }
        void Display() {
          System.out.println("x = "+x+", y = "+y);
        }
}
public class Main {
        public static void main(String[] args) {
          Derived d = new Derived(10, 20);
          d.Display();
        }
}
```

- Output:
  x = 10, y = 20

Abstract class

- An abstract class can have an abstract method without body and it can have methods with implementation also.
- abstract keyword is used to create a abstract class and method. Abstract class in java can’t be instantiated. An abstract class is mostly used to provide a base for subclasses to extend and implement the abstract methods and override or use the implemented methods in abstract class.

Abstract class in Java Important Points

- abstract keyword is used to create an abstract class in java.
- Abstract class in java can’t be instantiated.
- We can use abstract keyword to create an abstract method, an abstract method doesn’t have body.
- If a class have abstract methods, then the class should also be abstract using abstract keyword, else it will not compile.
- It’s not necessary for an abstract class to have abstract method. We can mark a class as abstract even if it doesn’t declare any abstract methods.
- If abstract class doesn’t have any method implementation, its better to use interface because java doesn’t support multiple class inheritance.
- The subclass of abstract class in java must implement all the abstract methods unless the subclass is also an abstract class.
- Java Abstract class can implement interfaces without even providing the implementation of interface methods.
- Java Abstract class is used to provide common method implementation to all the subclasses or to provide default implementation.
- It can have constructors.
- An abstract method can’t be static, private and final

Example1:

```
abstract class A
{
        abstract void m1();
         void m2()
      {
        System.out.println("This is a concrete method.");
      }
}
class B extends A
{
        void m1() {
        System.out.println("B's implementation of m2.");
      }
}

public class AbstractDemo
{
      public static void main(String args[])
      {
        B b = new B();
        b.m1();
        b.m2();
      }
}
```

- Output:

```
B's implementation of m2.
This is a concrete method.
```

Example2:

```
abstract class Shape{
    abstract void draw();
}
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
    void draw(){System.out.println("drawing circle");}
}

class TestAbstraction1{
    public static void main(String args[]){
    Shape s=new Circle()
    s.draw();
    }
}
```

- Output:

drawing circle

Example3:

```
abstract class Bike{
      Bike()
      {System.out.println("bike is created");}     //constructor
       abstract void run();
       void changeGear(){System.out.println("gear changed");}
 }

class Honda extends Bike
{
      void run(){System.out.println("running safely..");}
 }
  class TestAbstraction2{
        public static void main(String args[])
      {    Bike obj = new Honda();
        obj.run();
      obj.changeGear();   }
 }
```

- Output:

```
bike is created
running safely..
gear changed
```

Interfaces in Java

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .

Example1:

```
interface In1
{
          // public, static and final
          final int a = 10;

          // public and abstract
          void display();
}
class TestClass implements In1
{
            public void display()
          {
             System.out.println(“Hello");
          }
          public static void main (String[] args)
          {
             TestClass t = new TestClass();
             t.display();
             System.out.println(a);
          }
}
```

- Output:

```
Hello
10
```

New features added in interfaces in JDK 8

- Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces. Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.
- Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object.

Example2:

```
interface In1
{
    final int a = 10;
    static void display()
    {
        System.out.println("hello");
    }
}

// A class that implements the interface.
class TestClass implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        In1.display();
    }
}
```

Output :

```
hello
```

Example3:

```
interface In1
{
    final int a = 10;
    default void display()
    {
        System.out.println("hello");
    }
}
class TestClass implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        TestClass t = new TestClass();
        t.display();
    }
}
```

Output :

```
hello
```

New features added in interfaces in JDK 9

- From Java 9 onwards, interfaces can contain following also
  - Private methods
  - Private Static methods

```
public interface TempI {
    public abstract void mul(int a, int b);
public default void
    add(int a, int b)
    {
        sub(a, b);
        div(a, b);
        System.out.print("Answer by Default method = ");
        System.out.println(a + b);
    }
    public static void mod(int a, int b)
    {
        div(a, b); // static method inside other static method
        System.out.print("Answer by Static method = ");
        System.out.println(a % b);
    }
    private void sub(int a, int b)
    {
        System.out.print("Answer by Private method = ");
        System.out.println(a - b);
    }
    private static void div(int a, int b)
    {
        System.out.print("Answer by Private static method = ");
        System.out.println(a / b);
    }
}
```

```
class Temp implements TempI {

    @Override
    public void mul(int a, int b)
    {
        System.out.print("Answer by Abstract method = ");
        System.out.println(a * b);
    }
    public static void main(String[] args)
    {
       TempI in = new Temp();
        in.mul(2, 3);
        in.add(6, 2);
       TempI.mod(5, 3);
    }
}
```

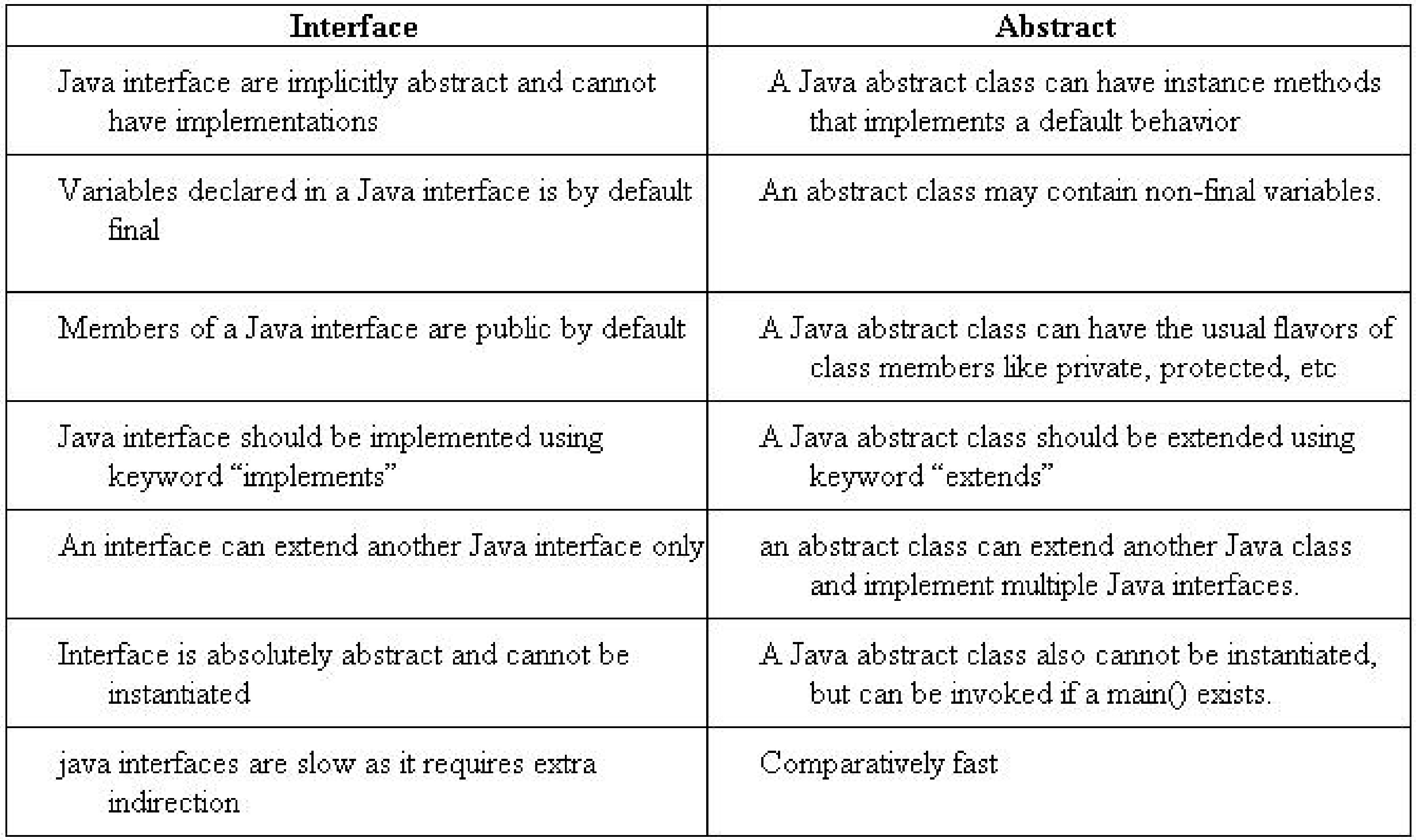
- OUTPUT :

```
Answer by Abstract method = 6
Answer by Private method = 4
Answer by Private static method = 3
Answer by Default method = 8
Answer by Private static method = 1
Answer by Static method = 2
```

Important points about interface

- We can’t create instance(interface can’t be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.

Abstract class Vs Interfaces



Interface	Abstract
Java interface are implicitly abstract and cannot have implementations	A Java abstract class can have instance methods that implements a default behavior
Variables declared in a Java interface is by default final	An abstract class may contain non-final variables.
Members of a Java interface are public by default	A Java abstract class can have the usual flavors of class members like private, protected, etc
Java interface should be implemented using keyword “implements”	A Java abstract class should be extended using keyword “extends”
An interface can extend another Java interface only	an abstract class can extend another Java class and implement multiple Java interfaces.
Interface is absolutely abstract and cannot be instantiated	A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
java interfaces are slow as it requires extra indirection	Comparatively fast