JAVA PART II

FUNCTIONS

- Block of statements used to perform task/operations.
- Modularity – splitting of larger programs into smaller programs
- Code readability- write once, run multiples

TYPES:

1. Built in functions- are provide by the language (library)
2. User defined functions- are created by programmers

- Built in library – common to all projects
- User defined library – specific to projects
- Third party library – common to projects

SYNTAX :

```
<access specifiers ><modifiers> return type functionname(args)
{
        statement1;
        statement2;
        <return values;>
    }
```

Notes :

1. All user defined functions must be defined inside the class body either bbefore the main method or after the main method.
2. Java does not allow you to create local functions in other defining a functions inside the body of the another functions is not allowed .

Program 1:

```
class mainclass
{
        static void fun() {

System.out.println("fun body started");
System.out.println("fun body ended");
        }
        public static void main(String[] args) {
        System.out.println("program started ");
        fun();// functioon calling or function invocation
        fun();
        System.out.println("program ended");
        }
}
```

Output:

```
program started
fun body started
fun body ended
fun body started
fun body ended
program ended
```

JAVA PART II

program 2:

```
class mainclass
{
	static void fun1() {

System.out.println("fun1() body started");
System.out.println("fun1() body ended");
	}
	static void fun2() {

		System.out.println("\tfun()2 body started");
		System.out.println("\tfun()2 body ended");
			}
	public static void main(String[] args) {
	System.out.println("program started ");
	fun1();// functioon calling or function invocation
	fun2();
	System.out.println("program ended");
	}
}
```

Output:

```
program started
fun1() body started
fun1() body ended
	fun()2 body started
	fun()2 body ended
program ended
```

Program 3:

```
class mainclass
{
	private static void fun(int arg1) {
		// TODO Auto-generated method stub
System.out.println("fun() started");
System.out.println("arg1 value: "+arg1);
System.out.println("fun ended");
	}


	public static void main(String[] args) {
	System.out.println("program started ");
	fun(12);
	System.out.println("program ended");
	}
}
```

Output :

```
program started
fun() started
arg1 value: 12
fun ended
program ended
```

JAVA PART II

Program 3

```
class mainclass
{
	private static void fun(int arg1) {
		// TODO Auto-generated method stub
System.out.println("fun() started");
System.out.println("arg1 value: "+arg1);
System.out.println("fun ended");
	}


	public static void main(String[] args) {
	System.out.println("program started ");
int x=12;// copy valiue of x into function arguments
	fun(x);
	System.out.println("program ended");
	}
}
```

Output :

```
program started
fun() started
arg1 value: 12
fun ended
program ended
```

Program 4:

```
class mainclass
{
	private static void fun(int arg1,double arg2) {
		// TODO Auto-generated method stub
System.out.println("fun() started");
System.out.println("arg1 value: "+arg1);
System.out.println("arg2 value: "+arg2);
System.out.println("fun ended");
	}


	public static void main(String[] args) {
	System.out.println("program started ");
fun(12,23.56);
	System.out.println("program ended");
	}
}
```

Output:

```
program started
fun() started
arg1 value: 12
arg2 value: 23.56
fun ended
program ended
```

JAVA PART II

OOPS

There are 4 types of JAVA

1. Class types
2. Interface typs
3. Enum type
4. Annotation type

1. Class type

```
class classname
{
//statements

}
```

2. Interface type

   Interface interfacename
   {

   }
3. Enum type
   Enum enumname
   {

   }
4. Annotation type
   Annotation annoatationame
   {
   ……..
   …….
   }

   JAVA CLASS DEFINITION

```
class classname
{
        declare variables;//member variable
        declare/define functions;//member function
        a. static member;
        b. non-static member/instance member
}
```

- Anything defined inside the class body is known as members of the class.
- We can define the variables as well as functions inside the body of the class

JAVA PART II

- The variables defined inside the class body is known as member variable . it is also known as data member or fields.
- The function defined inside the class body is known as member of functions

Member types:

The members of a class are defined as

1. Static member
2. Non-static member

- The static members are declared with the help of static keyword whereas non-static member are declared without static keyword.

Note:

- In java language , there is no concepts of global variables .
- Java has only two variables
  1. Local variables
  2. Member variables
- The typical class definition with members

```
class demo1
{
        static int x=10;//static variable
        int y=12;//non-static variable
        static void fun1()
        {
                int z=30;//local variable
        }
        static void fun2()
        {
                int k=23;//local variables
        }
}
class mainclass
{
        public static void main(String[] args) {
        System.out.println(demo1.x);

        }
        }
```

Note :

i. Members of a class can be accessed from another class. It can be restricted by using the access specifiers

ii. Java language provides 4 types of access specifiers
   a. Private
   b. Package level
   c. Protected
   d. Public

➔ How to access static members of a class

```
class demo
```

```
{
        static int x=23;
        static int y=23;
        static void fun()
        {
                System.out.println("Running fun()");
        }
}

class mainclass
{
        public static void main(String[] args) {
                System.out.println("main method started");
                System.out.println("x= "+demo.x);
                System.out.println("y = "+demo.y);
                demo.fun();
                System.out.println("main method ended");
        }
        }
```

Output:

```
main method started
x= 23
y = 23
Running fun()
main method ended
```

- ➔ How to access non-static members of a class
- Create object of a class/instance of a class
- Use new operator to create object of a class
- Syntax:

```
New classname();
```

- ➔ To access non static members of a class

```
New classname().membername;
```

```
class demo
{
        static int x=23;
        int y=34;
        void disp()
        {
                System.out.println("Running disp() method");
        }
}
class mainclass
{
        public static void main(String[] args) {
                new demo().disp();
                System.out.println("y value : "+new demo().y);
                System.out.println("x value: "+demo.x);
        }
}
```

Output:

```
Running disp() method
```

JAVA PART II

y value: 34
x value: 23

- The static members of the class are associated to class. They are loaded in the memory. One copy per class.
- This member can be accessed by using name of the class and with the help of dot operator.
- The non-static members are associated with the object with class until we create the object.
- It is not possible to access the non-static members of a class along with the object, the dot operator is used to access the non-static member.
- The non-static members are loaded one copy per object of the class.
- If we create no of object of a class then the non-static members are loaded n times in different locations.
- Since, the non-static members are associated with object. It is also known as object members or instance members.
- Since, the static member is associated to the class. It is also known as class member.

Variables :

1. Primitive variables
   - ✓ Used to store data
   - ✓ Are defined by data type
2. Non-primitive variables
   - ✓ Used to store user defined data types
   - ✓ Are declared using any 3 of types
     - i. Class type
     - ii. Interface type
     - iii. Enum type

```
package pack2;

class demo4
{
	int x=12;
	void test()
	{
		System.out.println("running test() method");

	}
}
class mainclass
{
	public static void main(String[] args) {
		System.out.println("main method started");
		new demo4().x=100;
		System.out.println(new demo4().x);
		System.out.println("main method ended");

	}
				}
```

OUTPUT:
main method started
12
main method ended

JAVA PART II

Primitive variable	Non-primitive variable
int id=1235;	Car bmw=new car()
Boolean status=true	Book classmate=new Book()
Double marks=89.99	Pencil Nataraj=new pencil()
Char grade='A'	Table studytable=new Table()

Declaration :

Classname vairiable_name;

1. Initialization:

Variablename=null;

- We can not use this variable
- In case, we use, we get null pointer exception

2. Initialize with Object
   Variablename=new classname();

- We reference variable to access object member.

```
package pack2;

class demo4
{
	int x=12;
	void test(){
		System.out.println("Running test() method");
	}
}
class mainclass
{
	public static void main(String[] args) {
		System.out.println("program started");
		demo4 d1;
		d1=new demo4();
		System.out.println(d1);
		System.out.println(d1.x);
		System.out.println("program ended");
	}
}
```

Output :

```
program started
pack2.demo4@15db9742
12
program ended
```

```
package pack2;
```

JAVA PART II

```
class Demo4
{
	int x=12;
	void test()
	{
		System.out.println("Running test() method");
	}
}
class mainclass
{
	public static void main(String[] args) {
		Demo4 d1=new Demo4();
		Demo4 d2=new Demo4();
		d1.x=34;
		System.out.println("first object x value="+d1.x);
		System.out.println("second object x value="+d2.x);
	}
}
```

OUTPUT:

first object x value=34
second object x value=12

SYNTAX:

Classname variable=new classname();

Variable- reference variable or object reference or object name
Classname()- object

```
package pack2;

class Demo6
{
	int p=34;
	double q=4.6;

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


	Demo6 d1=new Demo6();
	Demo6 d2=d1;
	d2.p=100;
	d2.q=12.45;
	System.out.println("p value= "+d1.p);
	System.out.println("q value= "+d1.q);

	}

}
```

OUTPUT:
p value= 100

JAVA PART II

q value= 12.45

- An object can be referred by multiple reference variable.
- The changes made by one reference will reflect in the another reference variable i.e if both are pointing to the same object.
- Ex. Google drive

OBJECT:

- Any entity having its own state and behavior is known as object.
- The state represents the property or information of the object , whereas behaviors represents the functionality of the object.
- A class is a definition block used to define the state and the behaviors of the object.
- The Non-static member variable is used to define the state of the object. Whereas the non-stati9c function is used to define the behaviors of the object.
- We can create multiple copy of the class by using new operator.
- Each copy of the class is known as object or instance.
- Class is a logical entity. Whereas object is physical entity.
- If multiple objects are having same properties, we must define class definition to create multiple object from the class definition.
- If we multiple objects are having different properties, we must define a class for each different object with that properties.

OBJECT :

Class classname

{

State of object

..

..

..

Behaviors of the object

}

Ex.

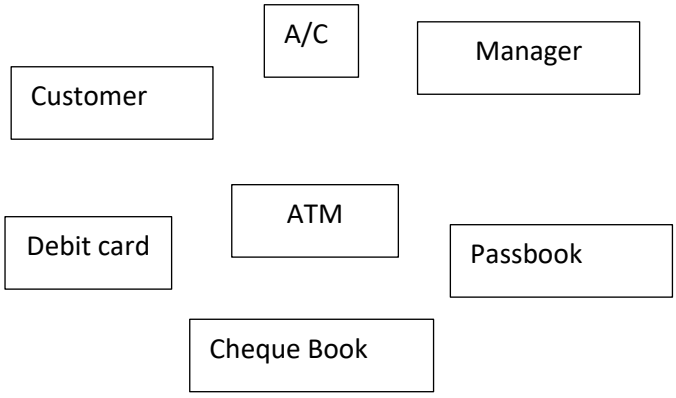
Notebook

1. Pages
2. Size
3. Price
4. Shape
5. Length
6. Soft/hard Biding
7. Ruled/unrolled

    1. Open()
    2. Close()
    3. Turn over ()

4. Tear pages()

BANKING SYTEM:



- In the above diagram, we need 7 class definition .

Circle.java

```
package pack2;

class circle
{
	double rad;
	final static double pi=3.14;
	void diameter()
	{
		double d=2*rad;
		System.out.println("diameter is "+d);
	}
	void area()
	{
		double a=pi*rad*rad;
		System.out.println("area is "+a);
	}
	void circumference()
	{
		double c=2*pi*rad;
		System.out.println("circumference of the circle is "+c);
	}
}
class mainclass
{
	public static void main(String[] args) {
		System.out.println("main method started");
		circle c1=new circle();
		c1.rad=23.8;
		c1.diameter();
		c1.area();
		c1.circumference();
		System.out.println(".....");
		circle c2=new circle();
		c2.rad=12.4;
```

```
                c2.diameter();
                c2.area();
                c2.circumference();

                System.out.println("main method ended");
        }
}
```

OUTPUT:

```
main method started
diameter is 47.6
area is 1778.6216
circumference of the circle is 149.464
.....................
diameter is 24.8
area is 482.8064
circumference of the circle is 77.872
main method ended
```

Note:

- The local variables must be initialized before using it in any operator otherwise compiler will throw error.
- The member variables can be initialized by the user or can be initialized by the compiler. If we don't initialize the member variables. the compiler provides default initialization.
- It provides 0 for integer 0.0 for floating point number , false for Boolean and \u000 for character type variable .

```
package pack2;

class Demo1
{
        static int x;
        static int y;
        static
        {
                System.out.println("Running Demo1()..");
                x=2*5+1;
                y=(1+5)*4;
        }
}
class mainclass{
        public static void main(String[] args) {
                System.out.println("running mainclass...");
                System.out.println("x value "+Demo1.x);
                System.out.println("y value "+Demo1.y);
        }
        }
        OUTPUT:
 running mainclass...
Running Demo1()..
x value 11
y value 24



package pack2;

class Demo1
```

JAVA PART II

```
{
        static int x=12;
        static int y=24;
        static
        {
                System.out.println("Running first static of  Demo1()....");
                x=56;
                y=34;
        }
        static
        {
                System.out.println("Running second static of Demo1()..");
                x=560;
                y=340;
        }
}
class mainclass{
        public static void main(String[] args) {
                System.out.println("running mainclass...");
                System.out.println("x value "+Demo1.x);
                System.out.println("y value "+Demo1.y);
        }
}
```

OUTPUT:

```
running mainclass...
Running first static of  Demo1()....
Running second static of Demo1()..
x value 560
y value 340
```

- Java provides blocks to initialize the member variable of a class.
- There are 2 types of blocks
  1. Static initialization blocks
  2. Non-static initialization blocks.

1. SIB:

- Static initialization is used to initialization, only the static member variable of the class. It can initialize the non-static member variable.
- The static blocks of the class are executed at the time of class loading. It is executed only omce per class.
- We can define multiple Blocks in a class.
- These multiple static classes are executed sequentially.
- If a class is having both main method and static method. JVM runs the static blocks first and then main method.

2. Instance Initialization:

- The IIB is also known as non static block, which is used to initialize the non static member variable of the class.
- It can also initialize both static and non-static members variable.
- The non-static blocks are executed only at the time of object creation.
- The non-static blocks are executed for each type of object creation.
- In a class, we can define multiple non-static blocks. These multiple non-static blocks are executed sequentially.
- If a class is having both static and non-static block, the static blocks are executed only once. Whereas non-static blocks are executed each time the object created.

JAVA PART II

```
package pack2;
class Demo4
{
	static int a;
	int x;
	int y;
	static
	{
		System.out.println("Running static blcok");
		a=23;
	}
	{
		System.out.println("Running non-static block");
		x=100;
		y=200;
	}
}
class mainclass
{
	public static void main(String[] args) {
		System.out.println("main methid started");
		System.out.println(" a value= "+Demo4.a);
		Demo4 d1=new Demo4();
		System.out.println("x value of 1st object is "+d1.x);
		System.out.println("y value of 2nd object is "+d1.y);
		Demo4 d2=new Demo4();
		System.out.println("\tx value of 2nd object is "+d2.x);
		System.out.println("\ty value of 2nd object is "+d2.y);
		System.out.println("main method ended");
	}
	}
```

Output:

```
main methid started
Running static blcok
 a value= 23
Running non-static block
x value of 1st object is 100
y value of 2nd object is 200
Running non-static block
	x value of 2nd object is 100
	y value of 2nd object is 200
main method ended
```

- In real time application development, if we need common initialization for all the object created for the class, we go for non-static block
- In the non-static block, we put the code, where it performs the common initialization.

```
package pack2;
class Demo7
{
	static int x;
	 int y;
	{
		x=34;
		y=67;
	}
```

```
}
class mainclass{
	public static void main(String[] args) {
		System.out.println("x value of Demo7 before object creation "+Demo7.x);
	//	System.out.println("y value of Demo7 before object creation "+Demo7.y);
		Demo7 d1=new Demo7();
	//	System.out.println("x value of Demo7 after object creation "+d1.x);
		System.out.println("y value of Demo7 after object creation "+d1.y);
		}
	}
```

OUTPUT:

x value of Demo7 before object creation 0
y value of Demo7 after object creation 67

```
 package pack2;
class Demo1
{
	int id;
	String name;
}
class mainclass
{
	public static void main(String[] args) {
		Demo1 d1=new Demo1();

		System.out.println(d1.id);
		System.out.println(d1.name);
	}
    }
```

OUTPUT:

0
null

Syntax:

String username=’hdb’;

TicketCounter.java

```
package pack1;

public class TicketCounter {
int no_tickets;
{
	no_tickets=100;
}
void issuetcikets(int n)
{
	System.out.println("issuing "+n+" Tickets");
	no_tickets-=n;
}
```

JAVA PART II

```
void returnTickets(int n)
{
	System.out.println("Returning "+n+" Tickets");
	no_tickets+=n;
}
void availableTickets()
{
	System.out.println("Total Tickets avaialable "+no_tickets);

}
}
```

Mainclass.java

```
package pack1;

public class mainclass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
TicketCounter tc1=new TicketCounter();
tc1.availableTickets();
tc1.issuetcikets(20);
tc1.availableTickets();
tc1.returnTickets(5);
tc1.availableTickets();
	}

}
```

OUTPUT:

```
Total Tickets avaialable 100
issuing 20 Tickets
Total Tickets avaialable 80
Returning 5 Tickets
Total Tickets avaialable 85
```

Assignment:

A customer of a bank can open an A/c to keep his money in the bank, basically customer can perform deposit, withdraw and wherever he wants, he can view his balance. Design a java program which makes the bank easy to implements the functionality.

bankAc.java

```
package pack1;

public class bankAc {
double bal;
{
	bal=25000.00;
}
void deposit(double amt)
{
```

```
		bal+=amt;
}
void withdraw(double amt)
{
		bal-=amt;
}
void viewbalance()
{
		System.out.println("avaialable balance = "+bal);
}
}
```

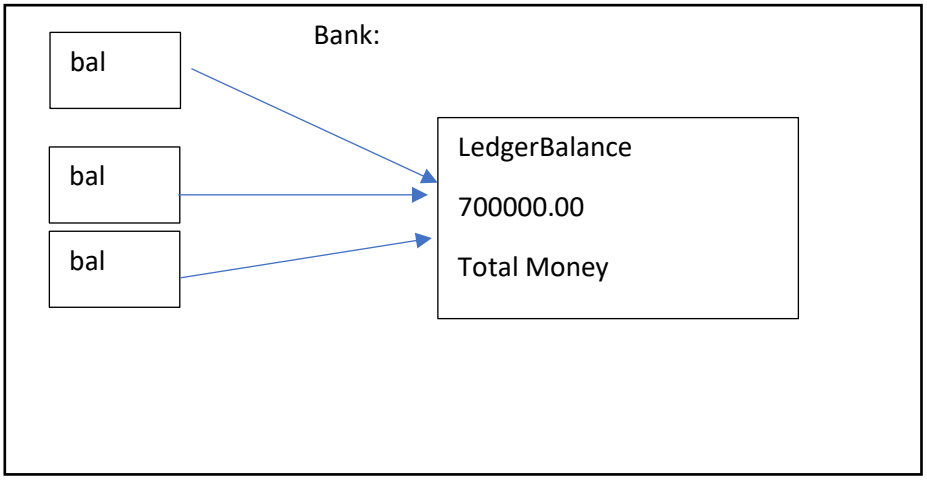
Mainclass.java

```
package pack1;

public class mainclass {

		/**
		 * @param args
		 */
		public static void main(String[] args) {
			// TODO Auto-generated method stub
bankAc b1=new bankAc();
b1.deposit(35000.00);
b1.viewbalance();
b1.withdraw(1500.00);
b1.viewbalance();

		}

}
```

OUTPUT:

```
avaialable balance = 60000.0
avaialable balance = 58500.0
```

➔ what is static member?



Account.java

JAVA PART II

```
package pack1;

public class Account {
	double acbal;
	void deposit(double amt)
	{
		acbal+=amt;
		Bank.ledgerbalamount+=amt;
	}
	void withdraw(double amt)
	{
		acbal-=amt;
		Bank.ledgerbalamount-=amt;
	}
	public void accountbal() {
		// TODO Auto-generated method stub
		System.out.println("account balance is "+acbal);
	}
}
```

Bank.java

```
package pack1;

public class Bank {
static double ledgerbalamount;

public static void ledgerbalamount() {
	// TODO Auto-generated method stub
	System.out.println("avaialable balance "+ledgerbalamount);
}
}
```

Mainclass.java

```
package pack1;

public class mainclass {
public static void main(String[] args) {
	Account a1=new Account();
	Account a2=new Account();
	Account a3=new Account();
	Bank.ledgerbalamount();
	a1.deposit(100000.00);
	a2.deposit(12000.00);
	a3.deposit(1000.00);
	Bank.ledgerbalamount();
	a2.withdraw(5000.00);
	a2.accountbal();
	Bank.ledgerbalamount();
}
}
```

OUTPUT:

```
avaialable balance 0.0
avaialable balance 113000.0
account balance is 7000.0
avaialable balance 108000.0
```

JAVA PART II

Note:

- The static members are used to whenever we want to share Data or functions to the object.

CONSTRUCTOR:

- Are special members/method of a class
- Are used to initialize state/properties of the object.

Syntax:

<access_specifiers>Constructorname(<args>)

{

//code to do initialization

}

Rules:

1. Constructor name must be same as class name
2. We should not specify any return type.

➔ There are two types of constructors

i. Compiler defined constructors or default constructors :- always No args constrcuors

ii. User defined constructor :-

a. No args constructors
b. Parameterized constructors(with args)

Demo1.java

```
public class Demo1 {
int x;
double y;
public Demo1(int x, double y) {

        this.x = x;
        this.y = y;
}
void display()
{
        System.out.println("x value "+x);
        System.out.println("y value "+y);
}
}
```

Mainclass.java

```
public class mainclass {

        /**
         * @param args
```

```
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Demo1 d1=new Demo1(4, 7.2);
d1.display();
	}

}
```

OUTPUT:

```
x value 4
y value 7.2
```

Demo2.java

```
public class Demo2 {

	int x;
	double y;
	public Demo2(int x, double y) {

		this.x = x;
		this.y = y;
	}
	void display()
	{
		System.out.println("x value = "+x+" y value = "+y);
	}
}
```

Mainclass.java

```
public class mainclass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Demo2 d1=new Demo2(12, 40.00);
d1.display();
	}

}
```

Output:

```
x value = 12 y value = 40.0
```

new classname(); - object creation

JAVA PART II

new- operator –

i. allocate memory
ii. create copy of class
iii. call constructor

classname() :- constructor of a class

i. initialize the object.

Scanner class in library:

next() :- to read String value

nextInt() :- to read int value

nextDouble :- to read double value

next().charAt(0) :- to read a character

step 1. Import java.util.Scanner;

- first start of source code

step 2: Scanner s1=new Scanner(System.in);

step 3: String name=s1.next();

int id=s1.nextInt();

double salary=s1.nextDouble();

mainclass1.java

```
import java.util.Scanner;


public class mainclass1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Scanner s1=new Scanner(System.in);
String name;
int age;
System.out.println("Enter your name");
name=s1.next();
System.out.println("enter your age");
age=s1.nextInt();
if(age>18)
{
	System.out.println(name+" is eligible for voting..");
}
else
{
	System.out.println(name+" is not eligible for voting...");
```

```
}
	}

}
```

OUTPUT:

```
Enter your name
hdb
enter your age
23
hdb is eligible for voting..
```

Pen.java

```
import java.util.Scanner;
public class Pen {
String color;
double price;
public Pen(String name, double price) {
	this.color = name;
	this.price = price;
}
void details()
{
	System.out.println("color = "+color+"\t price = "+price);
}
}
```

Mainclass4.java

```
import java.util.Scanner;


public class mainclass4 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Scanner s1=new Scanner(System.in);
String inkcolor;
double cost;
System.out.println("Enter color");
inkcolor=s1.next();
System.out.println("Enter price");
cost=s1.nextDouble();
Pen p1=new Pen(inkcolor,cost);
p1.details();

	}

}
```

OUTPUT:

JAVA PART II

```
Enter color
red
Enter price
12
color = red          price = 12.0
```

- constructors are special members(methods) of a class. Which is used to initialize the object at the time of object creation.
- The constructors initialize state of the object.
- The constructors can be defined either by the compiler or by the user.
- The constructors defined by the compiler are known as compiler defined constructors or default constructors.
- The default constructors will be always without arguments.
- User can define either no args constructors or parameterized constructors.
- The constructers defined with args are known as parameterized constructors.
- We can define any no. of arguments to the constructors.
- While defining the constructors , the constructors name must be same as class name and constructor should not specify any return type.
- The constructors are always called by the new operators at the time of object creation.
- The object creation always happens at run time.
- Every class must define a constructors either defined by the compiler or defined by the user.
- The compiler provides the constructors only when the class not having user defined constructors. If the class is already having user defined constructer, the compiler will not provide any constructors.
- The compiler provided constructors is known as default constructors

Demo5.java

```
public class Demo5 {
int x=12;
{
        System.out.println("Running non-static block");
        x=34;
}
public Demo5() {
        System.out.println("Running constructor");
}
void display()
{
        System.out.println("x value is "+x);
}
}
```

Mainclass5.java

```
public class mainclass5 {
public static void main(String[] args) {
        Demo5 d1=new Demo5();
        d1.display();
}
}
```

JAVA PART II

OUTPUT:

Running non-static block
Running constructor
x value is 34

- If a class is having non-static block and constructors, during object creation of that class, first non-static block will be executed then the constructors will be executed.
- Finally, the object will have the initialization provided by the constructor.

Demo6.java

```
public class Demo6 {
int x;
double y;
//constructor overloading
Demo6(int arg1)
{
	x=arg1;
}
Demo6(double arg1)
{
	y=arg1;
}
Demo6(int arg1,double arg2)
{
	x=arg1;
	y=arg2;
}
void display()
{
	System.out.println("x value = "+x+" \t y value ="+y);
}
}
```

Minclass6.java

```
public class maincass6 {
public static void main(String[] args) {
	Demo6 d1=new Demo6(23);
	Demo6 d2=new Demo6(4.7);
	Demo6 d3=new Demo6(12,23.60);
	d1.display();
	d2.display();
	d3.display();
}
}
```

OUTPUT:

x value = 23 y value =0.0
x value = 0 y value =4.7
x value = 12 y value =23.6

JAVA PART II

- Defining more than one constructors with different argument is known as constructor overloading.
- The arg list must differ either in the type of arg or in the type of length of the arg.
- Any two-overloaded constructor should not have same arg type.
- The overloaded constructors are identified on the basis of arguments.
- If class provides overloaded constructors, we can create object of that class with different initialization.
- Constructor of class always must be non-static.
- We cannot declare static, final or abstract.

Note:

- Constructor of a class returns the address where the object is created and initialized.
- We should not specify the return type since it always returns the address.

Assignment:

Student should be enrolled a course. To enroll the student, student name and contact number is required. Design a java program , where the student can enroll by specifying the name and the contact . your design shied also be have to enroll the student by specifying the only name. Write interactive java program.

Demo8.java

```
public class Demo8 {
final static int x;
final int y;
static{
        x=34;
}
public Demo8(int y) {
        this.y = y;
}


}
```

Mainclass8.java

```
public class mainclass8 {
public static void main(String[] args) {
        System.out.println("x value "+Demo8.x);
        Demo8 d1=new Demo8(34);
        System.out.println("y value is "+d1.y);

}
}
```

Output:

```
x value 34
y value is 34
```

JAVA PART II

Employee.java

```
public class Employee {
final int id;

public Employee(int id) {
        this.id = id;
}
void display()
{
        System.out.println("Id = "+id);
}
}
```

Empclass.java

```
public class Empclass {
public static void main(String[] args) {
        Employee e1=new Employee(12);
        System.out.println("e1 value ");
        e1.display();
        Employee e2=new Employee(24);
        System.out.println("e2 value");
        e2.display();
}
}
```

OUTPUT:

```
e1 value
Id = 12
e2 value
Id = 24
```

this keyword :

- Java provides a special keyword by the name this. Which is used to refer the current object.
- The object or a reference on which the functions involved is known as current object.
- this keyword always points to the current object.
- this keyword must be used only inside the non-static method body or constructor body. It should not be used in static method body.
- If the member variable and local variables names are same. The member variable is differentiated by this keyword.

ARRAYS

Storage :

Level	Types
1	Variables
2	Arrays
3	Collection or DS

4	Files
5	Database

- To store multiple values of same type.

Array Declaration :-

Arraytype [] arrayname[];

Array initialization

Arrayanme=new arrayytype[n];

n- size of array

DATE: 8/19/17

➔ There are two types of arrays in java
1. Array of primitive types
2. Array of non-primitive types

- In this array, we can store only primitive types values. The array of non-primitive types is declared using class type.
- In this array, we can store only the objects of the class declared with the array types.
- Whenever we create any array in java, the elements are initialed with default values.
- The array of primitive types will have default value according to the data types.
- In case of array of non-primitive types, the elements will be initialized to Null by defaults.

Syntax:

Array_type[] arrayname=new arraytype[n];

a. Int[] a1=new int[10];
b. Pen[] p1=new pen[5];

Int[] a1=new int[5];

0	0	0	0
0	1	2	3

Sop(a[0]);

A[0]=24;;

A[1]=23;

A[2]=45;

A[3]=12;

A[4]=14;

for(int i=0;i<+4;i++)

{

Sop(A[i]);

}

- If the index is exceeding the value, we get ArrayIndexOutofBounds.
- In array, the size can be accessed by using a property known as length.
- In array, length is a property. it is not a function.
  Note:
- In string, length is a function not property.
  Program:

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

	int[] a1=new int[9];
	a1[0]=23;
	a1[1]=26;
	a1[2]=20;
	a1[3]=22;
	a1[4]=27;

	System.out.println("array size:="+a1.length);
	System.out.println("array elements");
	for(int i=0;i<=a1.length-1;i++)
	{
		System.out.println(a1[i]);

	}
	}
	}
```

Program 2:

```
class pen
{
	String color;
	double price;
	pen(String color,double price)
	{
		this.color=color;
		this.price=price;
	}
	void details()
	{
		System.out.println("ink color: "+color);
		System.out.println("ink price: "+price);
	}

}
```

JAVA PART II

```
class mainclass
{
	public static void main(String[] args) {
	pen[] p1=new pen[5];
	//store of objects of pens
	p1[0]=new pen("Black",17.24);
	p1[1]=new pen("Red",19.24);
	p1[2]=new pen("blue",17.24);
	p1[3]=new pen("yellow",14.24);
	p1[4]=new pen("orange",11.24);
	System.out.println("array size: "+p1.length);
	System.out.println("array elements");
	for(int i=0;i<p1.length-1;i++)
	{
		System.out.println(p1[i]);
	}

}
	}
```

Output:

```
array size: 5
array elements
pen@15db9742
pen@6d06d69c
pen@7852e922
pen@4e25154f
```

program 3:

```
class pen
{
	String color;
	double price;
	pen(String color,double price)
	{
		this.color=color;
		this.price=price;
	}
	void details()
	{
		System.out.println("ink color: "+color);
		System.out.println("ink price: "+price);
	}

}

class mainclass
{
	public static void main(String[] args) {
	pen[] p1=new pen[5];
	//store of objects of pens
	p1[0]=new pen("Black",17.24);
	p1[1]=new pen("Red",19.24);
	p1[2]=new pen("blue",17.24);
	p1[3]=new pen("yellow",14.24);
	p1[4]=new pen("orange",11.24);
	System.out.println("array size: "+p1.length);
	System.out.println("color\tprice");
System.out.println(".....");
```

JAVA PART II

```
for(int i=0;i<p1.length-1;i++)
		{
			System.out.println(p1[i].color+"\t"+p1[i].price);
		}

}
		}
```

Output:

```
array size: 5
color	price
.............
Black	17.24
Red	19.24
blue	17.24
yellow	14.24
```

program 4:

```
class pen
{
		String color;
		double price;
		pen(String color,double price)
		{
			this.color=color;
			this.price=price;
		}
		void details()
		{
			System.out.println("pen[color: \t"+this.color+"price\t"+this.price+"]");
			//System.out.println("ink price: "+price);
		}

}

class mainclass
{
		public static void main(String[] args) {
		pen[] p1=new pen[5];
		//store of objects of pens
		p1[0]=new pen("Black",17.24);
		p1[1]=new pen("Red",19.24);
		p1[2]=new pen("blue",17.24);
		p1[3]=new pen("yellow",14.24);
		p1[4]=new pen("orange",11.24);
		System.out.println("array size: "+p1.length);
		System.out.println("color\tprice");
System.out.println(".............");
for(int i=0;i<p1.length-1;i++)
		{
			//System.out.println(p1[i].color+"\t"+p1[i].price);
		p1[i].details();
		}

}
		}
```

OUTPUT:

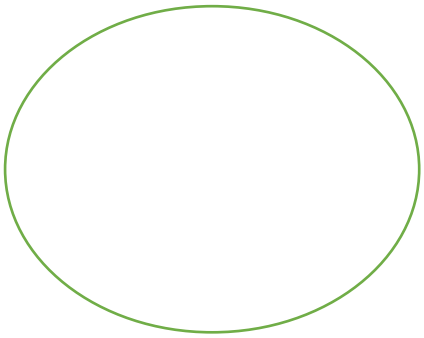
JAVA PART II

array size: 5
color price
.....
pen[color: Blackprice 17.24]
pen[color: Redprice 19.24]
pen[color: blueprice 17.24]
pen[color: yellowprice 14.24]

assignment: write a java program to store record of 5 employees and display the employee properties in a tabular format.

21/08/17

Packages:



Modules

Submodules

- A java package is a set of java program developed for a particular feature in a project
- A java package contains jva source code

JAVA PART II

- A java package can contain another package
- If we define a package inside another package it is called as sub-package
- The sub-package can java source code or its su-packages
- Whenever we create a java project with package structure each package represents a folder in them project

Package declaration:

Package packagename;

Package name-.lower class(industry standards)

Rules:

1. Package declaration must be first statement sof java source code.
2. A java source file can have only one package declaration

```
package login;
class demo1
{
        define member variables;
        define member functions;
}
```

- Class demo belongs to package login.

Ex 2:

Source code 1	Source code 2	Source code 3

package login;
class demo1
{
define member variables;
define member functions;
}
 	
class demo2
{
define member variables;
define member functions;
}
 	
class demo3
{
define member variables;
define member functions;
}
	package login;
	class run1
	{
	define member variables;
	define member functions;
	}
	
	class run2
	{
	define member variables;
	define member functions;
	}
		package sent;
		class sample
		{
		define member variables;
		define member functions;
		}
		
		class sample2
		{
		define member variables;
		define member functions;
		}

- A class belongs to one package can be accessed from another the class belonging to another package.

JAVA PART II

- The accessing is possible if class and its members have the right access.
- If a class belonging to a package wants to access the member of the class belonging to another package then the current class must import the other class by using import statements .
- Without import statements, a class can not access the members of the class from another package .

Import statement;

Import packagename.classname;

Rule :

1. Import statement must after package definition .
2. Multiple import statements are allowed

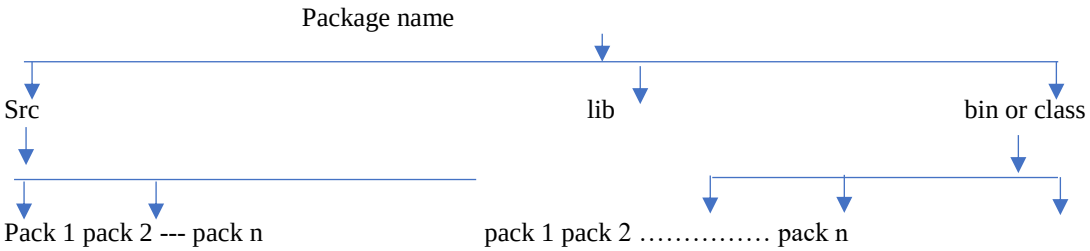
Types of import :

➔ There are 2 types of import

1. Import-> import all the members of the class.

   Ex. **import** java.util.Scanner;

2. Static import-> import only static members of class
   Ex. import java.util.*;



.java .java . javan .class .class .class

➔ .jar –

Command: java -d ...\bin pack1\Demo.java

To execute class file Goto

Bin folder …\bin>java pack2.sample

```
package pack1;
class demo1
{
        public static void main(String[] args) {
                System.out.println("Running demo1 class from pack1 package  ");
        }
}
```

```
package pack2;
```

JAVA PART II

```
class sample1
{
	public static void main(String[] args) {
		System.out.println("Running sample 1 class from pack2 package  ");
	}
}
```

Java provides 4 types of access specifiers

1. Private
2. Package
3. Protected
4. Public

i. Private: are restricted up to the class body. The private members must be used only inside the class body.
It cannot be accessed from another class even though it belongs to the same package.

ii. The package level access members are restricted up to the package. It can be accessed from another class belonging to the Sam class
It is not possible to access from the class which belongs to another package

iii. Protected members have same access as package members
The protected members are restricted up to the package level.
The only difference between package level and protected is the protected members can have accessed from outside the package by inheriting it.

iv. The public members of a class has access from any package
It can be used by the class belonging to the same package as well as different packages

```
package pack2;

public class demo1
{
	private int p=34;
	int q=45;
	protected int r=90;
	public int s=97;
}
class demo2
{
	void disp()
	{
		demo1 d1=new demo1();
		System.out.println("x value= "+p);
		System.out.println("q value= "+q);
		System.out.println("r value= "+r);
		System.out.println("s value= "+s);
	}
}
```

```
package pack2;
import pack1.demo1;
class sampl1
{
```

JAVA PART II

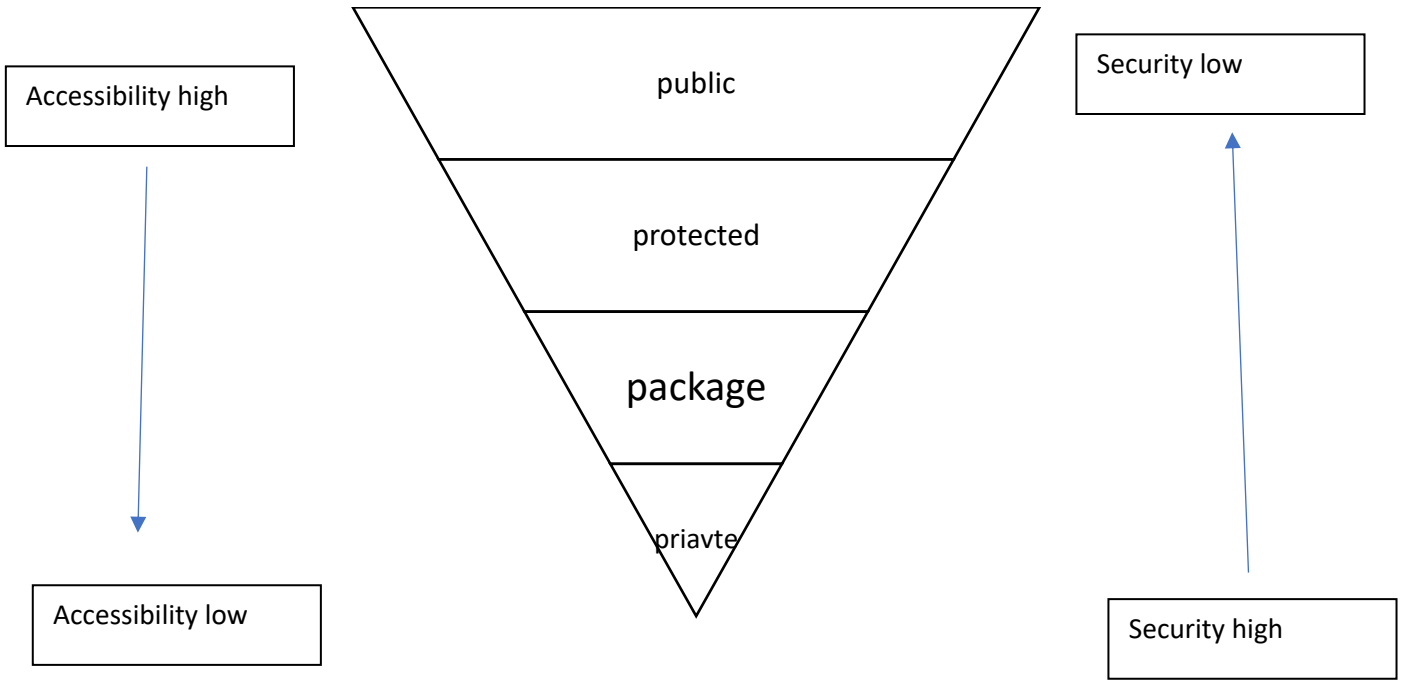
```
void view()
{
        demo1 d1=new demo1();
        System.out.println("s value ="+s);
}
    }
```

IMPORTANT POINTS :

- The default access in the class body is package level access.
- For a class, we can give only two access, either public or package level.
- A java source file can have only one public class. The source file name must be name of the class having public access.
- The constructor provided by the compiler will always have the same access of its class.
- The user defined constructor can provide any access of the 4 access to the constructor.

Q. what happens if the constructor declared as private?

- The object can be created only in the class body. It restricts the object creation in the class.
-



Access specifiers	Inside class	Inside same class	From outside class
Private	Yes	No	No
Package level	Yes	Yes	No
Protected	Yes	Yes	Yes(is a relationship)
Public	Yes	Yes	Yes

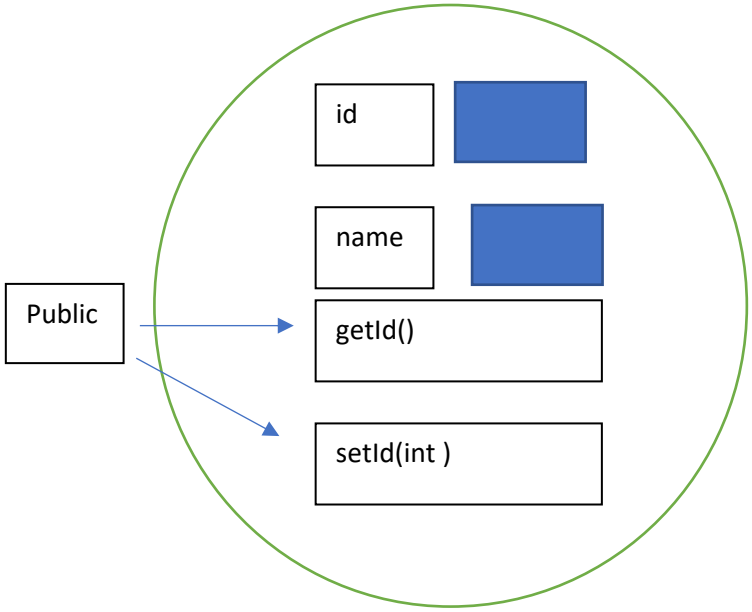
ENCAPSULATION:

JAVA PART II

- Encapsulation is one of the OOPS principles which specifies the data to bonded to the class, in other words binding the members to the class body and protecting the by using relevant access specifiers is known as encapsulation.
- In java language, we cannot declare and define variables outside the class because java by default supports encapsulation.

JAVA BEAN CLASS:

1. Class must be public
2. Constructor must be public
3. Members variables
4. must be private
5. Defining getter method
6. Defining the setter method with public




```
public class student
{
	private int id;
	private String name;
	public int getId()
	{
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}

	public String getName()
	{
		return name;
	}

	public void setName(String name) {
		this.name = name;
```

```
	}
	}
```

- Defining a public class with private members variables, public constructors and public getter setter method is known as java bean class.
- The getters method is used to provide read access whereas setters methods are used to provide write access.
- While developing project, the java bean class widely used in implementing Data access object(DAD) and Data Transfer object(DTO).

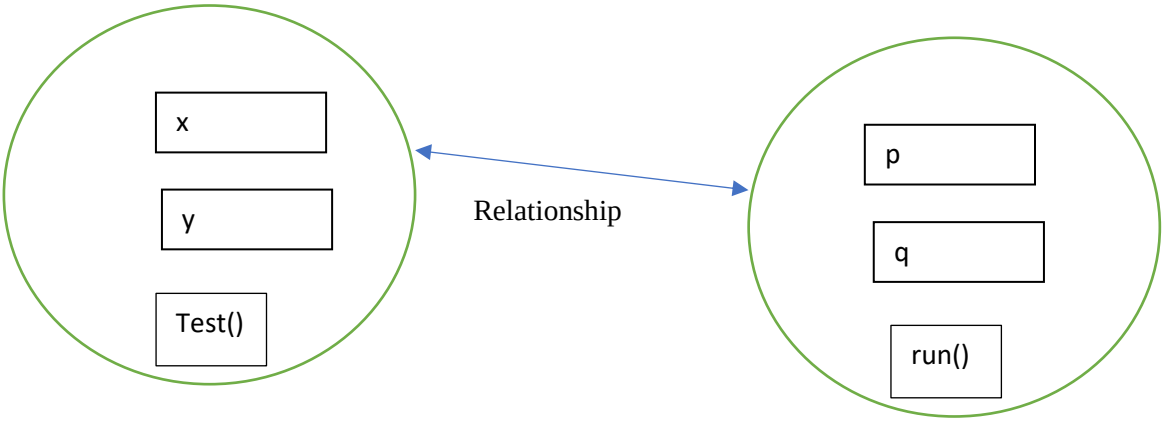
```
package pack1;

public class Employee {
		int id;
		String name;
		double salary;
		public Employee(int id, String name, double salary) {
			super();
			this.id = id;
			this.name = name;
			this.salary = salary;
		}
}
```

```
package pack1;
import java.util.Scanner;


public class mainclass1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner scan=new Scanner(System.in);

	Employee[] emparray=new Employee[5];

	for (int i = 0; i < emparray.length; i++) {
		System.out.println("Enter Employee ID");
		int id=scan.nextInt();
		System.out.println("enter Employee name");
		String name=scan.next();
		System.out.println("Enter employee salary");
		double salary=scan.nextDouble();

	 emparray[i]=new Employee(id,name,salary);

	}
	System.out.println("..............................");
	System.out.println("ID\tNAME|tSALARY");
	System.out.println("..............................");
	for (int i = 0; i < emparray.length; i++) {
```

```
                System.out.println(emparray[i].id+"\t"+emparray[i].name+"\t"+emparray[i].salary);
        }
        }

}
```

OUTPUT:

```
Enter Employee ID
1234
enter Employee name
hdb
Enter employee salary
3400
Enter Employee ID
6776
enter Employee name
kpl
Enter employee salary
8787
Enter Employee ID
8787
enter Employee name
koppal
Enter employee salary
877834783874
Enter Employee ID
77
enter Employee name
hdb1
Enter employee salary
90
Enter Employee ID
78
enter Employee name
hdb3
Enter employee salary
78
.....
ID      NAME|tSALARY
.....
1234    hdb     3400.0
6776    kpl     8787.0
8787    koppal  8.77834783874E11
77      hdb1    90.0
78      hdb3    78.0
```



JAVA PART II

Relationships:

1. Is A
2. Has A
3. Use A

- Class diagram is a pictorial representation of java class and its members .
- The class diagram is used in designing the properties of the object and the relationship b/w the object .
- It is a part of UML diagram .
- **CLASS DIAGRAM HAS 3 SECTIONS:**
  1. Name of the class
  2. Its for representing the member variables of the class
  3. Used for representing the constructor and the members fun of the class
- In industry, we first design the class diagram and based on the class diagram, we write the program,

Class diagram :

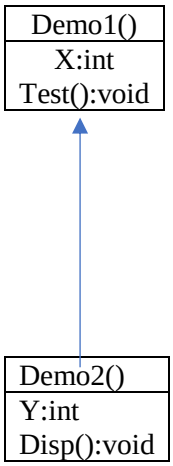
Class name
Variablename:type Variablename:type . . .. . .
Class name() Functionality (args):return type Functionality (args):return type . . . .. .

Employee
Id:int Name:String Salary:double

Employee(int,String,salary)
getId();
setId(int);
getName();
setName(String);

JAVA PART II

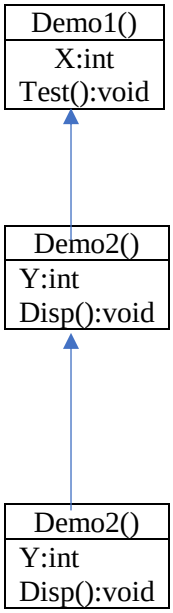
1. Single inheritance:



Extends

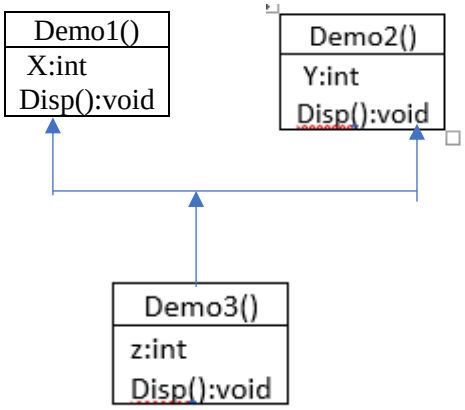
```
Public Demo2 extends Demo1()
{

}
```

2. Multilevel inheritance

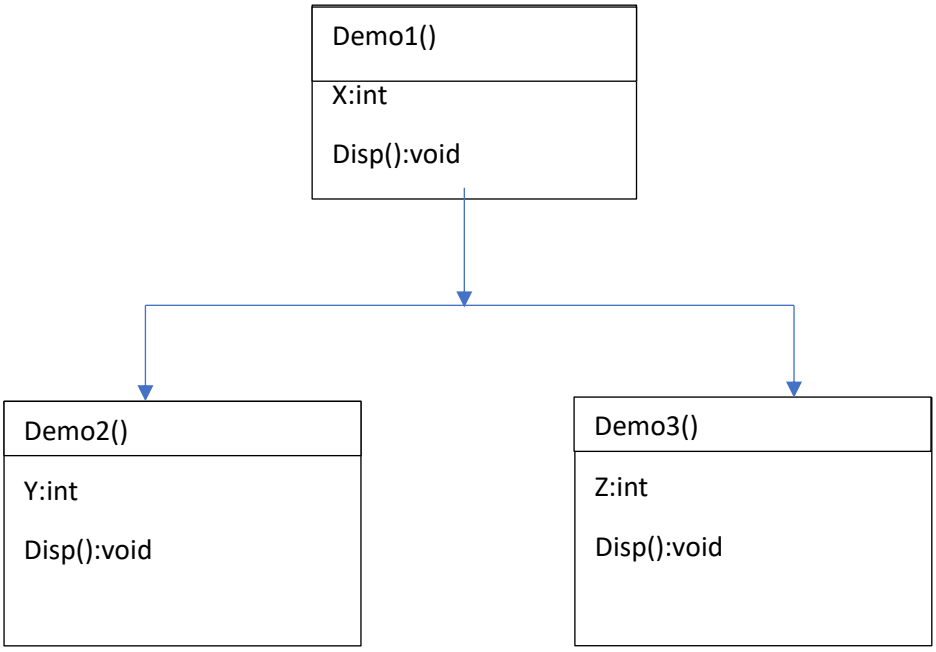


JAVA PART II

3. Multiple inheritance



4. Hierarchical inheritance:



Demo1.java

```
package pack1;

public class demo1 {
int x=12;

void test()
```

```
{
	System.out.println("Running test() method.....");
}
}
```

Demo2.java

```
package pack1;

public class demo2 extends demo1 {
int y=34;
void disp()
{
	System.out.println("Running disp()");
}
}
```

Mainclass1.java

```
package pack1;

public class mainclass1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
demo2 d1=new demo2();
System.out.println("x value= "+d1.x);
System.out.println("x value= "+d1.y);
d1.test();
d1.disp();
	}

}
```

Output:

```
x value= 12
x value= 34
Running test() method.....
Running disp()
```

Demo3.java

```
package pack1;

public class demo3 extends demo2{
char z='a';
```

JAVA PART II

```
void view(){
        System.out.println("running view() method");
}
}
```

Mainclass1.java

```
package pack1;

public class mainclass1 {

        /**
         * @param args
         */
        public static void main(String[] args) {
                // TODO Auto-generated method stub
demo3 d1=new demo3();
System.out.println("x value= "+d1.x);// demo1 property
System.out.println("x value= "+d1.y);//demo2 property
System.out.println("z value= "+d1.z);//demo 3 property

d1.test();
d1.disp();
d1.view();
        }

}
```

Output:

```
x value= 12
x value= 34
z value= a
Running test() method.....
Running disp()
running view() method
```

- A class inheriting members from another class is known as inheritance.
- The class from where the members are inherited is known as base class or super class.
- The class to which the members are inherited is known as derived class of sub class.
- The sub class always inherits the properties from the super class.
- Hence we can say sub class is a type of super class.
- The sub class can inherits only non-static members of the super class.
- The static members of the super class will never be inherited to the sub class.
- If the super class is having non-static members with private declaration, that members also will not inherited to sub class bcz the private members has a access restricted to the class .
- Whenever we create the object of sub class. It will have the property of super class.
- There are 4 types of inheritance
    1. Single inheritance
        i. In this type of inheritance, the sub class inherits the property from one super class
    2. Multilevel inheritance.

i. In this type of inheritance, the sub class, the sub class inherits the property from super class which inherits the property from another super class.

3. Multiple inheritance
   i. In this type of inheritance, a class inherits from the more than one super class.
   ii. Java doesn't support multiple inheritance.
4. Hierarchal inheritance:
   i. In this type of inheritance, the super class properties are inherited to the more than sub class

Notes :

- A class can be declared with a final keyword.
- Such class is known as final class.
- A class cannot inherit the members from the final class.
- In other words , the final class cannot have sub classes.

```
 final class A
{
        .......;
        .......;
        .......;
        .......;
        .......;
}
class X
{
Static int x;
        members;
}
final class Y extends X
{
        members;
}
```

- The protected member of a class can be accessed from a class outside the package by inheriting it.
- The inherited protected member must be used only inside the inherited class.

Sample.java

```
package pack1;

public class sample1 {
protected int i=12;
public int k=45;
void test()
{
        System.out.println("Running test().....");
}
}
```

Sampl2.java

```
package pack2;
```

JAVA PART II

```
import pack1.sample1;

public class sample2 extends sample1{
int j=35;
void disp()
{
        System.out.println("i value= "+i);
        System.out.println("j value= "+j);
        System.out.println("k value= "+k);
}
}
```

Mainclass2.java

```
package pack2;

public class mainclass2 {

        /**
         * @param args
         */
        public static void main(String[] args) {
                // TODO Auto-generated method stub
System.out.println();
sample2 s1=new sample2();
s1.disp();
        }

}
```

Output:

```
i value= 12
j value= 35
k value= 45
```

Advantages :

i. Code reusability
ii. S/w extensibility
iii. Modification

Person.java

```
package pack2;

public class person {
String name;
int age;
}
```

Student.java

```
package pack2;
```

JAVA PART II

```
public class student extends person{
int rollno;
double marks;
}
```

Employee.java

```
package pack2;

public class Employee extends student {
int id;
double salary;
}
```

Mainclass3.java

```
package pack2;

public class mainclass3 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
student st1=new student();
st1.name="hdb";
st1.age=23;
st1.marks=97.99;
st1.rollno=12334;

Employee emp=new Employee();
emp.name="hdb1";
emp.age=24;
emp.id=7676;
emp.salary=5667.99;
	}

}
```

Demo4.java

```
package pack1;

public class demo4 {
int x=12;

}
class  demo5 extends demo4
{
	int x=24;
	void display()
	{
	System.out.println("x value in current class :"+this.x);
	System.out.println("x value in super class class :"+super.x);

	}
```

```
}
```

Mainclass4.java

```
package pack1;

public class mainclass4 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
demo5 d1=new demo5();
d1.display();
	}

}
```

Output:

```
x value in current class :24
x value in super class class :12
```

- Java provides a special keyword super, which is used to refer the super property in the sub class.
- The super keyword must be always used in non-static method body or constructor body of the sub-class.

Note:

i. Every class defined in java language must inherit from super class.
ii. The super class can be defined by user or defined by compiler
iii. If user doesn't define any super class, the compiler defines a super class by name Object.
iv. The Object is a root class of java language.
v. Every class must inherit from Object class.

```
Class X extends Object
{

}
Class extends X
{

}
```

Constructor calling statements:

- Constructor is called by new operator.
- this and super :- keyword
- without using new operator, we can call constructor by this() and super()

a constructor can make a call to another class by using constructor calling statement.

JAVA PART II

There are 2 constructor calling statements

1. super()
2. this()

Rules:

i. both calling statement must be used inside the constructor body
ii. both must be the first statement of the constructor body.
iii. Either this() calling statement or super() calling statement must be used.
iv. Only one calling statement is allowed.

Ex. Let's define student constructor

//invalid

Student()

{

SOP("running student constrcuctor");

this(); // must be first statement

}

//invalid

Student()

{

this(); //only one calling statement is allowed

this();

SOP("Running student constrcuctor");

}

//invalid

Student()

{

this(); //either this() or super() calling statement is allowed

super();

SOP("Running student constructor");

}

Ex. 2:

```
package pack1;

public class sample2 {
int x;
double y;
```

```
public sample2(int x)
{
	this(1.0);
	System.out.println("Running sample2(int)..."+x);
	this.x=x;
}
public sample2(double y)
{
System.out.println("Running sample2(double)..."+y);
}
void display()
{
	System.out.println("x value = "+x);
	System.out.println("y value = "+y);
}
}
package pack1;

public class Mainclass5 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
sample2 d1=new sample2(13);
d1.display();
	}

}
```

Output:

```
Running sample2(double)...1.0
Running sample2(int)...13
x value = 13
y value = 0.0
```

Note: Java language doesn't supports recursive constructive calling whereas it supports recursive function calling.

JAVA PART II

METHOD OVERLOADING AND OVERRIDING

- Method or f unctions :- to perform specific operations
- Overloading:
  Same operation:- with different parameters /arg.
  (multiple operation)

S/W :

JAVA PART II

Demo2.java

```
package pack1;

public class demo2 {
void disp(int arg1)
{
	System.out.println("Running disp(int) method");
	System.out.println("arg1 value = "+arg1);
}
}
class Demo3 extends demo2
{
	void disp(double arg1)
	{
		System.out.println("Running disp(int) method");
		System.out.println("arg1 value = "+arg1);
	}
}
```

Mianclass.java

```
package pack1;

public class mainclass {

	/**
	 * @param args
	 */
```

```
        public static void main(String[] args) {
                // TODO Auto-generated method stub
Demo3 d1=new Demo3();
d1.disp(12.5);
d1.disp(15);
        }

}
```

OUTPUT:

```
Running disp(int) method
arg1 value = 12.5
Running disp(int) method
arg1 value = 15
```

- Defining multiples methods in a class with the same name and different arguments is known as method overloading.
- The arguments list should differ either in the type of argument or in the length of the arguments.
- In the class , we can overload both static method and non-static method.
- The overloaded methods are invoked on the basis of arguments.
- The subclass can overload the methods of super class.
- The method is used to achieve compile time polymorphism.
- When developing any application if we come across an operation. Which need to be performed with a different values then we go for the method overloading.

NOTE:

- The main method of a class can be overloaded
- The JVM begins the execution only by invoking the main method having string array as a arguments.

1. You have assigned to build an application for a client, the client specifies a requirement (the user must login to an app either by using email id or by using phone no.) demonstrate the sample code for the client.

```
Class Demo1
{
Void test()
{
……;
…….;
}
Class Demo2 extends Demo1
{

Void test()
{
……..;
……..;
}
```

```
        }
Demo2 d1=new Demo2();
d1.test(); // run new implementation


Demo1 d2=new Demo1();
d2.test(); // run old implementation
```

- Inheriting a method from a super class and changing its implementation in the sub class is known as method overriding.
- While overriding the method the sub class should rename the same method declaration of the super class.
- To perform method overriding Is a relationship is compulsory. \
- The sub class can not override the following methods of super class
  i. Static method :- because static methods are not inherited to sub classes
  ii. Private method:- because it is restricted only to the class where it is declared
  iii. Final non-static method:- the final keyword doesn't allow to perform the method overriding in the sub class.
- The method overriding is used to achieve run time polymorphism
- While overriding a method if we change the arguments list then it will have considered as method overloading.
- While developing an application if we want to change the functionality of old feature we go for method overriding.

Demo1.java

```
package pack1;

public class Demo1 {

        void test(int arg1)
        {
                System.out.println("test() method is defined in Demo1 class");
        }
}
class Demo2 extends Demo1
{
// Demo2 overrides Demo1 method
// we can provide protected or public.
        void test(int arg1)
        {
                super.test(12);// calling super class implementation
                System.out.println("test() method is defined in Demo2 class");
        }
}
```

Mainclass.java

```
package pack1;

public class mainclass {

        /**
```

```
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Demo2 d1=new Demo2();
d1.test(12);
//Demo1 d2=new Demo1();
//d2.test();
	}

}
```

Output:

```
test() method is defined in Demo1 class
test() method is defined in Demo2 class
```

- While overriding the methods the subclass has a permission to increase the access level of the inherited method but doesn't have permission to decrease or reduce the access level.

```
Class X
{
Private void disp()
{
….;
}
}
Class Y extends X
{
Public void disp()
{
…..;
}
}
```

```
package pack1;

public class sample1 {
static void disp()
{
	System.out.println("Running disp()...");
}
}
class sampl2 extends sample1
{
	//static hiding
	 static void disp()
	{
		System.out.println("Running overrided disp()...");
	}
}
```

JAVA PART II

DATA TYPE CASTING

int x=25;

x is a variable of type int
25 is a value of type int

Int y=4.5;

Y is a variable of type int
4.5 is a variable of type double

EX. Int x=(int)59.98; ///narrowing

x- int type

59.98- double type

JAVA PART II

SOP(x);

Double y=(double)25; // widening or

y- double type

25- int type

Mainclass.java

```
package datatypecasting;

public class mainclass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
int x=(int)59.9825;// narrowing
double y=(double)25;//Widening
System.out.println("x value = "+x);
System.out.println("y value ="+y);

int p=35;
double q=56.12;
int a=(int)q;
double b=(double)p;

System.out.println("a value = "+a);
System.out.println("b value = "+b);

	}

}
```

Output:

```
x value = 59
y value =25.0
a value = 56
b value = 35.0
```

mainclass.java

```
package datatypecasting;

public class mainclass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
int x=(int)59.9825;//explicit  narrowing
double y=25;//auto Widening or implicit widening
```

```
System.out.println("x value = "+x);
System.out.println("y value ="+y);

int p=35;
double q=56.12;
int a=(int)q;
double b=p;//auto Widening or implicit widening

System.out.println("a value = "+a);
System.out.println("b value = "+b);

        }

}
```

Output:

```
x value = 59
y value =25.0
a value = 56
b value = 35.0
```

Calculator.java

```
package datatypecasting;

public class calculator {
void square(int num)
{
        System.out.println("calculating square of "+num);
        int res=num*num;
        System.out.println("result is "+res);
}
}
```

MainClass.java

```
package datatypecasting;

public class MainClass2 {
public static void main(String[] args) {
        calculator cl=new calculator();
        cl.square(25);
        cl.square((int)6.7);
}
}
```

Output:

```
calculating square of 25
result is 625
calculating square of 6
result is 36
```

calculator.java

```
package datatypecasting;
```

JAVA PART II

```
public class calculator {
void square(int num)
{
	System.out.println("calculating square of "+num);
	int res=num*num;
	System.out.println("result is "+res);
}
void square(double num)
{
	System.out.println("calculating square of "+num);
	double res=num*num;
	System.out.println("result is "+res);
}
}
```

MainClass.java

```
package datatypecasting;

public class MainClass2 {
public static void main(String[] args) {
	calculator cl=new calculator();
	cl.square(25);
	//cl.square(6.7);
}
}
```

Output:

```
calculating square of 25
result is 625
```

MainClass.java

```
package datatypecasting;

public class MainClass3 {
public static void main(String[] args) {
	char c1='a';
	char c2='z';
	char c3='A';
	char c4='Z';

	int n1=(int)c1;
	int n2=(int)c2;
	int n3=(int)c3;
	int n4=(int)c4;

	System.out.println(" ascii value of "+c1+" is "+n1);
	System.out.println(" ascii value of "+c2+" is "+n2);
	System.out.println(" ascii value of "+c3+" is "+n3);
	System.out.println(" ascii value of "+c4+" is "+n4);
	}
}
```

JAVA PART II

Output:

```
ascii value of a is 97
 ascii value of z is 122
 ascii value of A is 65
 ascii value of Z is 90
```

MainClass.java

```
package datatypecasting;

public class MainClass4 {
public static void main(String[] args) {
		char[] src={'j','a','v','a'};
		char[] dest=new char[src.length];
		System.out.println("array elements in lower case");
		for (int i = 0; i < src.length; i++) {
		System.out.print(src[i]);
		}
		for (int i = 0; i < src.length; i++) {
			int n=(int)src[i];// char to ASCII
			char c=(char)(n-32);//ASCII to char
			dest[i]=c;//store in dest array
			//dest[i]=(char)(src[i]-32);
		}
		System.out.println("\narray elements in upper case");
		for (int i = 0; i < dest.length; i++) {
			System.out.print(dest[i]);
		}
}
}
```

Output:

```
array elements in lower case
java
array elements in upper case
JAVA
```

MainClass4.java

```
package datatypecasting;

import java.util.Scanner;

public class MainClass4 {
public static void main(String[] args) {
		Scanner scan=new Scanner(System.in);
		System.out.println("Enter string in lower case only");
		String s1=scan.next();
		char[] src=s1.toCharArray();
		char[] dest=new char[src.length];


		for (int i = 0; i < src.length; i++) {
			int n=(int)src[i];// char to ASCII
			char c=(char)(n-32);//ASCII to char
```

```
                dest[i]=c;//store in dest array
                //dest[i]=(char)(src[i]-32);
        }
        System.out.println("upper case: ");
        for (int i = 0; i < dest.length; i++) {
                System.out.print(dest[i]);
        }
}
}
```

Output:

```
Enter string in lower case only
java
upper case:
JAVA
```

Mainclass.java

```
package datatypecasting;

import java.util.Scanner;

public class MainClass4 {
public static void main(String[] args) {
        Scanner scan=new Scanner(System.in);
        System.out.println("Enter string in lower case & upper case only");
        String s1=scan.next();
        char[] src=s1.toCharArray();
        char[] dest=new char[src.length];


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

                if(src[i]>='a'&&src[i]<='z')
                {
                        dest[i]=(char)(src[i]-32);
                }
                else if(src[i]>='A'&&src[i]<='Z')
                {
                        dest[i]=(char)(src[i]+32);
                }
                //store in dest array

        }
        System.out.println("the resultant string is.... ");
        for (int i = 0; i < dest.length; i++) {
                System.out.print(dest[i]);
        }
}
}
```

Output:

```
Enter string in lower case & upper case only
hAnuManTa
the resultant string is....
HaNUmANtA
```

JAVA PART II

CLASS CASTING

1. Demo1 d1=new Demo1();--→ type matching
- D1 is a variable of object Demo1 type

Demo2 d2=(Demo1)new Demo2(); --→ type mismatching

Demo2 type casted to Demo1 type

Demo1 type

Demo2 d1=(Demo2)new Demo1();-→ type mismatching

- D1 is a variable of object of Demo1 type
  Demo2 type

  Rules:
  i. Classes must have Is a Relationship
  ii. Class must have properties of another class to which it must be casted.

- Casting one type of information to another type is known as type casting
- There are two type casting in java
  i. Data type casting or primitive casting
  ii. Class type casting or non-primitive casting

1. Data type casting
- Casting one data type to another data type is known as data type casting
- There are two type of data type casting\
  i. Widening
  ii. Narrowing
- Casting lower data type to any of the higher data type is known as widening. Widening can be performed either implicitly or explicitly
- The implicit widening is done by compiler
- Casting an higher data to any of the lower data type is known as narrowing.
- Whenever we perform narrow operation, there will be always some precision loss
- The narrowing operation should be performed explicitly in the code because the compiler will not perform implicitly

2. Class type casting
- Casting one class type to another class type is known as class type casting
- To perform class type casting, We have to satisfy the below rules
  i. Class must have Is a relationship
  ii. Class must have the properties of another class to which it must be casted
- If the first rule is not satisfied, the compiler throws error
- If 2nd rule is not satisfied, the JVM will throw class ClassCastException

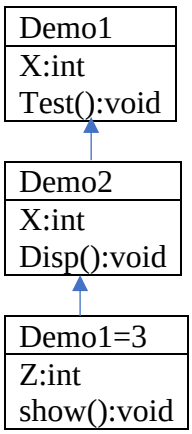
JAVA PART II

- There are 2 types of class type casting
  i. Upcasting
  ii. Down casting
- Casting sub class type to super class type is known as up casting
- Upcasting can be performed either implicitly or explicitly
- The implicit up casting is done by compiler
- The casting super class type to sub class type is known as down casting.
- The down casting should be performed explicitly in the code
- Down casting should be done only the object which is already upcasted.

Questions:

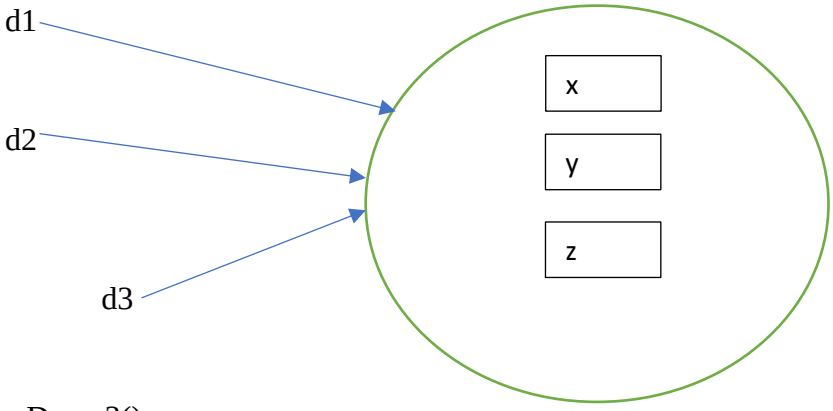
1. What is class ClassCastException. when it occurs?
2. Why compiler will not throw error for down casting

- Whenever we perform up casting the object must show only the properties of the class to which is casted.




```
package classtypecasting;

public class MainClass1 {
public static void main(String[] args) {
        Demo2 d1=(Demo2)new Demo3();// up casting
        //Demo2 is casted to Demo1
        //we can access only Demo1 properties

        System.out.println("x value = "+d1.x);
        d1.test();
        d1.disp();

        Demo1 d2=(Demo1)new Demo3();//down casting
        System.out.println("y value "+d2.x);
        d2.test();

}
}
```

JAVA PART II



Demo3 d1=new Demo3();
Demo2 d2=d1;

Downcasting :

- Demo1 d1=(Demo1)new Demo3();
- Using d1, we can access only Demo1 properties.
- Demo2 d2=(Demo2)d1;
- Demo1 is casted to Demo2 type.
- Using d2, we can access Demo2 and Demo3 properties.
- Demo3 d2=(Demo3)d1;
- Demo1 type is casted to Demo3 type.

Demo1 d1=new Demo2();
Using d1, we can access Demo1 properties.
Demo2 d2=(Demo2)d1;

JAVA PART II

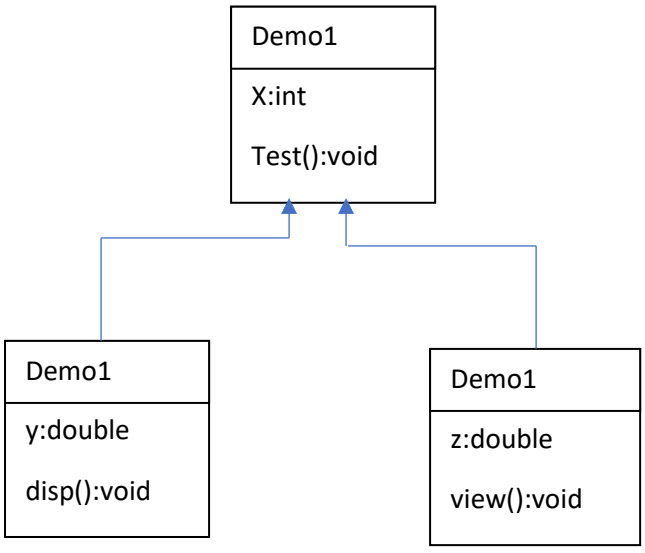
Using d2, we can access Demo1 and Demo2 properties.
Demo3 =(Demo3)d1;// ClassCastException.

```
//Demo1 d1=new Demo1();
//Demo2 d2=(Demo2)d1;
//Demo3 d3=(Demo3)d1;
```

```
Demo1 d1=new Demo3();
Demo2 d2=(Demo2)d1;
        Demo3 d3=(Demo3)d1;
```

Ex. Demo1 d1;
Case

- The sub class can be casted to any of the super class object
- The objects start showing the class to which it is casted.
- If the same object has to behave like subclass (original properties) then we have to go for down casting.
- If we have a reference variable of class type, for that ref variable we can assign its class object or we can assign the subclass object.



CLASS TYPE FUNCTION ARGUMENTS

JAVA PART II

Demo1.java

```
package pack1;

public class Demo1 {

	int x=12;
	void test()
	{
		System.out.println("Running test() method...");
	}
}
```

Sample1.java

```
package pack1;

public class Sample1 {
	int y=45;
	//function arguement is a class type

	void disp(Demo1 arg1)
	{
		System.out.println("Running disp(Demo1) method");
		System.out.println("x value = "+arg1.x);
		arg1.test();
	}
}
```

MianClass.java

```
package pack1;

public class MainClass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Sample1 s1=new Sample1();
s1.disp(new Demo1());//passing Demo1 class object to disp()
	}

}
```

OUTPUT:

```
Running disp(Demo1) method
x value = 12
Running test() method...
```

Mainclass.java

```
package pack1;

public class MainClass {
```

```
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Sample1 s1=new Sample1();
Demo1 d1=new Demo1();
s1.disp(d1);
	}

}
```

1. Assume that you are developing a module in a HRMS the company, which is going to use the s/w decided to give a bonus of 5% of emp salary. Updating the bonus is a responsible of finance. demonstrate a java program to implement the above program.

Employee.java

```
public class Employee {
int id;
String name;
double salary;
public Employee(int id, String name, double salary) {
	this.id = id;
	this.name = name;
	this.salary = salary;
}


}
```

Finance.java

```
public class Finance {


	void updateSalary(Employee arg1,int bonusRate)
	{
		System.out.println("Updating Bonus to...\n"+"Emp Name "+arg1.name+"\nEmp Id
"+arg1.id+"\nEmp Salary is "+arg1.salary);
		arg1.salary=arg1.salary+arg1.salary*bonusRate/100;

	}
}
```

MainClass.java

```
public class MainClass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Finance d1=new Finance();
Employee e1=new Employee(12, "hdb",2500.00);
d1.updateSalary(e1,5);
System.out.println("Emp New Salary is "+e1.salary);
```

```
	}

}
```

OUTPUT:
Updating Bonus to...
Emp Name hdb
Emp Id 12
Emp Salary is 2500.0
Emp New Salary is 2625.0

- The function can be defined with two types of arguments
  i. Primitive type arguments
  ii. Non-primitive arguments
- If the function is primitive, then we have to pass a primitive value to the functions.
- If the function argument is a non-primitive type then the caller function has pass the object of the class type declared in the function arguments.
- While passing the objects we can pass the object directly or we can pass the object reference.

Demo1.java

```
package pack1;

public class Demo1 {

	int id;
	int x;
	public Demo1(int id, int x) {
		this.id = id;
		this.x = x;
	}


}
```

Sample1.java

```
package pack1;

public class Sample1 {
	int y=45;
	//function argument is a class type


	void update(Demo1 arg1,int value)
	{
		arg1.x=value;
	}
}
```

ObjectDB.java

JAVA PART II

```
package pack1;

public class ObjectDB {

	Demo1[] objArr={
					new Demo1(105,25),
					new Demo1(104,58),
					new Demo1(106,65),
					new Demo1(107,85),
					new Demo1(108,74)
					};
Demo1 getObject(int id){
	System.out.println("searching for the object based on");
	Demo1 ref=null;
	for (int i = 0; i < objArr.length; i++) {
		 if (id==objArr[i].id) {
			System.out.println("object found");
			ref=objArr[i];
			break;

		}

	}
	return ref;
}
void display()
{
	for (int i = 0; i < objArr.length; i++) {
		System.out.println(objArr[i].id+"\t "+objArr[i].x);


	}
}
}
```

OUTPUT:

```
105	 25
104	 58
106	 65
107	 85
108	 74
Enter Id value to be searched..
104
searching for the object based on
object found
Enter the value to be updated
500
105	 25
104	 500
106	 65
107	 85
108	 74
```

Employee.java

```
package pack1;

public class Employee {
int id;
String name;
```

```
double salary;
public Employee(int id, String name, double salary) {
	this.id = id;
	this.name = name;
	this.salary = salary;
}

}
```

EmpDbupdate.java

```
package pack1;

public class EmpDBupdate {

	void Empupdate(Employee arg1, double sal)
	{
		arg1.salary=sal;
		System.out.println("sal is updating..");
	}
}
```

EmpDBupdate.java

```
package pack1;

public class EmpObjDB {

	Employee [] objArr={
			new Employee(1234,"hdb1",2000.00),
			new Employee(1235,"hdb2",800.00),
			new Employee(1236,"hdb3",200.00),
			new Employee(1237,"hdb4",3000.00),
			new Employee(1238,"hdb5",9000.00),
			new Employee(1239,"hdb6",1000.00)
	};

	Employee getObject(int id)
	{
		System.out.println("searching for the Employee Id");
		Employee ref=null;
		for (int i = 0; i < objArr.length; i++) {

			if(id==objArr[i].id)
			{
				System.out.println("Employee ID exits");
				//System.out.println("object found");
				 ref=objArr[i];
				break;
			}
		}
		return ref;

	}

	void disp()
	{
		System.out.println("Displaying Employee Records");
		System.out.println("id\tname\tsalary");
		for (int i = 0; i < objArr.length; i++) {


			System.out.println(objArr[i].id+"\t"+objArr[i].name+"\t"+objArr[i].salary);
		}
```

```
	}

}
```

EmpMainClass.java

```
package pack1;

import java.util.Scanner;

public class EmpMainClass {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner scan=new Scanner(System.in);

EmpDBupdate u=new EmpDBupdate();
EmpObjDB E=new EmpObjDB();
E.disp();

System.out.println("Enter Employee Id");
int id;
id=scan.nextInt();
Employee d1=E.getObject(id);
System.out.println("Enter Employee salary to be updated");
double salary = scan.nextDouble();
u.Empupdate(d1, salary);
E.disp();


System.out.println("program ended");
	}

}
```

OUTPUT:

```
Displaying Employee Records
id	name	salary
1234	hdb1	2000.0
1235	hdb2	800.0
1236	hdb3	200.0
1237	hdb4	3000.0
1238	hdb5	9000.0
1239	hdb6	1000.0
Enter Employee Id
1234
searching for the Employee Id
Employee ID exits
Enter Employee salary to be updated
5000
sal is updating..
Displaying Employee Records
id	name	salary
1234	hdb1	5000.0
1235	hdb2	800.0
1236	hdb3	200.0
1237	hdb4	3000.0
1238	hdb5	9000.0
1239	hdb6	1000.0
```

program ended

- The function can have 2 types of return types
  i. Primitive type
  ii. Non-primitive type
- If the return type is of primitive type then that function should return the primitive type.
- The caller function should store the return value.
- The function return type can be non-primitive type, the function has to return the object or the ref of the class type declared in the return type field.
- The caller function should copy/store the reference in another ref variable to access the object.

Demo1.java

```
package pack1;

public class Demo1 {

	int x=12;
	void test()
	{
		System.out.println("Running test() method");
	}
}
```

Demo2.java

```
package pack1;

public class Demo2 extends Demo1{
double y=12.34;
void disp(){
	System.out.println("running disp() method");
}
}
```

Demo3.java

```
package pack1;

public class Demo3 extends Demo1 {
char z='a';
void view()
{
	System.out.println("running test() method");
}
}
```

Sample1.java

```
package pack1;

public class Sample1 {


	void useObject(Demo2 arg1)
	{
		System.out.println("running useObject() method");
		System.out.println("x value = "+arg1.x);
		System.out.println("y value = "+arg1.y);
		arg1.test();
		arg1.disp();
	}
```

```
}
```

MainClass1.java

```
package pack1;

public class MainClass1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Sample1 s1=new Sample1();
Demo2 d1=new Demo2();
Demo1 d2=new Demo1();
Demo3 d3=new Demo3();
s1.useObject(d1);//passing Demo2 type reference
s1.useObject((Demo2)d2);//passing Demo1 type
//s1.useObject((Demo2)d3);//passing Demo3 type ref
//compile type error, No Is s relationship b/w Demo2 and Demo3
	}

}
```

OUTPUT:

```
running useObject() method
x value = 12
y value = 12.34
Running test() method
running disp() method
Exception in thread "main" java.lang.ClassCastException: pack1.Demo1 cannot be cast to pack1.Demo2
	at pack1.MainClass1.main(MainClass1.java:15)
```

GENERALIZATION:

Demo1.java

```
package pack1;

public class Demo1 {

	int x=12;
	void test()
	{
		System.out.println("Running test() method");
	}
}
```

Demo2.java

```
package pack1;

public class Demo2 extends Demo1{
double y=12.34;
void disp(){
	System.out.println("running disp() method");
}
}
```

Demo3.java

```
package pack1;

public class Demo3 extends Demo1 {
char z='a';
```

JAVA PART II

```
void view()
{
	System.out.println("running test() method");
}
}
```

Sample1.java

```
package pack1;

public class Sample1 {


	void useObject(Demo1 arg1)
	{
		System.out.println("running useObject() method");
		System.out.println("x value = "+arg1.x);
		//System.out.println("y value = "+arg1.y);
		//arg1.test();
	//	arg1.disp();
	}
}
```

MainClass.java

```
package pack1;

public class MainClass1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Sample1 s1=new Sample1();
Demo2 d1=new Demo2();
Demo1 d2=new Demo1();
Demo3 d3=new Demo3();
s1.useObject(d1);//passing Demo2 type reference
s1.useObject(d2);//passing Demo1 type ref
s1.useObject(d3);//passing Demo3 type ref

	}

}
```

OUTPUT:

```
running useObject() method
x value = 12
running useObject() method
x value = 12
running useObject() method
x value = 12
```

- Defining functions which run for different types of object is known as Generalization, for a generalized function we can pass the object which has the properties with the class declare with the object.
- Defining a functions which works only for on type of object is known as specialization.
- For the specialized function if we try to pass for diff object we might get compile time error or ClassCastException.

JAVA PART II

Down Casting to access specific properties:-

```
package pack1;

public class Sample1 {


	void useObject(Demo1 arg1)
	{
		System.out.println("running useObject() method");
		System.out.println("x value = "+arg1.x);
		//System.out.println("y value = "+arg1.y);
		//arg1.test();
	//	arg1.disp();

		Demo2 ref1=(Demo2)arg1;
		System.out.println("y value ="+ref1.y);
		ref1.disp();
	}
}
```

Mainclass.java

```
package pack1;

public class MainClass1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
Sample1 s1=new Sample1();
Demo2 d1=new Demo2();
Demo1 d2=new Demo1();
Demo3 d3=new Demo3();
s1.useObject(d1);//passing Demo2 type reference
	}

}
```

Sample1.java

```
package pack1;

public class Sample1 {


	void useObject(Demo1 arg1)
	{
		System.out.println("running useObject() method");
		System.out.println("x value = "+arg1.x);
	arg1.test();

	if(arg1 instanceof Demo2)
	{
		Demo2 ref1=(Demo2)arg1;
		System.out.println("y value ="+ref1.y);
		ref1.disp();
	}
```

```
        else if(arg1 instanceof Demo3)
        {
            Demo3 ref1=(Demo3)arg1;
            System.out.println("z value: "+ref1.z);
            ref1.view();
        }
            }
}
```

MainClass.java

```
package pack1;

public class MainClass1 {

            /**
             * @param args
             */
            public static void main(String[] args) {

                        // TODO Auto-generated method stub
Sample1 s1=new Sample1();
Demo1 d1=new Demo1();
Demo2 d2=new Demo2();
Demo3 d3=new Demo3();
s1.useObject(d1);//passing Demo2 type reference
System.out.println("---------------------------");
s1.useObject(d2);//passing Demo2 type reference
System.out.println("---------------------------");
s1.useObject(d3);//passing Demo2 type reference
            }

}
```

OUTPUT:

```
running useObject() method
x value = 12
Running test() method
---------------------------
running useObject() method
x value = 12
Running test() method
y value =12.34
running disp() method
---------------------------
running useObject() method
x value = 12
Running test() method
z value: a
running view() method
```

1. The govt body gives specialty or services to every citizen of country. If a citizen is a student with merit score the govt gives scholarship. If the citizen is a employee the govt apply tax of 30%. Every citizen should get enrolled for the Aadhar number. Design the application by using the oops principles, inheritance, generalization and specialization.

```
package pack1;

public class Citizen1 {

            String name;
```

```
	int age;
	public Citizen1(String name, int age) {
		this.name = name;
		this.age = age;
	}
}
package pack1;

public class Student1 extends Citizen1{
int rollNo;
double marks;
public Student1(String name,int age,int rollNo, double marks) {
	super(name,age);
	this.rollNo = rollNo;
	this.marks = marks;
}

}
package pack1;

public class Government {


	void Scholorship(Student arg1)
	{
		if(arg1.marks>70.00)
		{
			System.out.println(arg1.name+" is eligible for scholorship");
		}else
		{
			System.out.println(arg1.name+" is not eligible for scholrship");
		}
	}
	void tax(Employee1 arg1)
	{
		double taxamount=0.0;
		System.out.println("calculating tax for"+arg1.name);
		if(arg1.salary>3000.00)
		{
			taxamount=(arg1.salary*30)/100;
			System.out.println("tax amount is "+taxamount);

		}
		else
			System.out.println("not applicable for tax");
	}

void enrolAadhar(Citizen1 arg1)
{
	System.out.println(arg1.name+" enrolled for aadhar successfully");

}
}
package pack1;

public class MainClass3 {
public static void main(String[] args) {
	Government gov=new Government();
	Citizen1 c=new Citizen1("hdb",23);
	Student1 s=new Student1("hdb1", 20, 1234, 70.45);
```

```
        Employee1 e=new Employee1("hdb2",12,1045, 50000.00);
        gov.enrolAadhar(e);
        gov.enrolAadhar(c);
        gov.enrolAadhar(s);
        gov.enrolAadhar(s);
        gov.tax(e);

}
}
```

POLYMORPHISM

- An object showing diff behavior at stages of its life cycle is known as polymorphism.
- There are 2 types of polymorphism
  i. Compile time polymorphism
  ii. Runtime polymorphism
- In compile time polymorphism the method declaration is bonded to which method declared at the time of compilation by the compiler. Since binding is done by compiler is known as compile time polymorphism.
- Since the binding happen at the compilation time, we call its as early binding
- The binding done by the compiler cannot re-bonded hence it is known as static binding. \
- The overloaded are the examples of compile time polymorphism
- In run time polymorphism the method declaration is bonded to the method definition by the JVM at run time.
- The binding is done on the basis of object.
- Since binding is done at the run time we call it as run time polymorphism.
- It is also known as late binding because the binding happens at after compilation during execution
- The binding done by the JVM can be re-bonded hence it is known as dynamic binding.
- To achieve run time polymorphism, we have to have to satisfy the following concepts
  a. Is a Relationship
  b. Method overriding
  c. Class type casting
- The overridden methods are the example of run time polymorphism.

JVM uses 4 diff memory area to run the java program
i. Heap area- objects
ii. Class area- static member
iii. Method area- only method definition
iv. Stack area-execution

- The heap area is used to store the objects created in the program.
- The memory allocation is random.
- The object details(properties and functions) are loaded in heap memory in a hash table structure.

Identifier	Value
Id	1045
Name	Hdb
getId()	Address
getName()	Address

- The class area is used to store the static members of the class.
- In the class area, a pool of memory is created to load the static members of the class. The pool is known as static pool. The name of the pool will be name of the class.
- The method area is used to store the method definition part of the method.
- If the method declaration is static, then declaration is in class
- If the method declaration is non-static then the method declaration is stored in heap area in the object.
- The stack area is used for the execution purpose whenever we invoke a method in class.
- The method enters into the stack area for execution purpose and returns back to the method area after execution.

JAVA PART II

- If we have any local variables in the method is loaded in the stack area. It will be available in stack area as long as methods are running in stack. The stack follows last in first out. The last entered method into the stack should finish first and get out of the stack.

Class loader:

- A class loader is a program in the JVM which is responsible to load the members of the class into the memory. Thus class loader always first loads static members and runs static block.
- Class loader confirms the loading of static members first before loading non -static members.

```
package pack1;

public class animal {

	void noise()
	{
		System.out.println("animal makes noise...");
	}
}

package pack1;

public class cat extends animal {

	void noise()
	{
		System.out.println("meow...meow...");
	}
}
package pack1;

public class Dog  extends animal{
void noise()
{
	System.out.println("bow....bow...");
}
}
package pack1;

public class Snake extends animal{

	void noise()
	{
		System.out.println("hisss...hisss..");
	}
}

package pack1;

public class AnimalSimulator {

	void makenoise(animal arg)
	{
		arg.noise();
	}
}
package pack1;

public class MainClass {
```

JAVA PART II

```
public static void main(String[] args) {
        AnimalSimulator ani=new AnimalSimulator();
        cat c1=new cat();
        Dog d1=new Dog();
        Snake s1=new Snake();
        ani.makenoise(c1);
        ani.makenoise(d1);
        ani.makenoise(s1);

}
}
```

Output:
meow...meow...
bow....bow...
hisss...hisss..

Methods:
1. Method declaration
2. Methid definition

- Method having only declaration (no method body)

➔ Abstract returntype methodname(arguments);

Ex. Abstract int swap(int a,int b);

Concrete methods:

➔ Method having both declaration and body

```
Returntype methodname(arguments)
{
-----;
-----; // code to do operation
-----;
}
```

```
package pack1;

abstract public class Demo1 {
static int x=22;
int y;
Demo1()
{
        y=45;
}
void test()
{
        System.out.println("running test() method...");

}
static void view()
{
        System.out.println("running view method");
}
abstract void disp();
}
```

JAVA PART II

```
package pack1;

public class MainClass1 {

	public static void main(String[] args) {
		//referring static members of abstract class
		System.out.println(" x value : "+Demo1.x);
		Demo1.view();
		//referring non-static members of abstract class
		Demo1 d1;// reference variable of abstract class Demo1 type
		//d1=new Demo1(); // can not create object, if a class is abstract
	}
}
```

- Defining a method with only declaration and no definition is known as abstract methods
- The abstract method must be declared using abstract keyword, the abstract do not specify the body.
- The abstract method must be declared either in the abstract class or java interface.
- A class declared using abstract keyword is known as abstract class.
- In a abstract class we can declare and defin static member.
- We can define constructors.
- If class is abstract it is not compulsory to declare the abstract methods. But if the method is abstract the class must be abstract class.
- In other words, if a class is containing the abstract class then we have to declare the class as abstract class otherwise compiler throws error.
- In a abstract class we can define both concrete method and abstract methods.
- In a abstract class we can define only concrete method because defining abstract method is not compulsory.
- The abstract keyword can not combined with the following keywords
  i. Static
  ii. Final
  iii. Private
- We can not create the object of the abstract class. We can declare ref variables of abstract class.

```
package pack1;

abstract public class Demo2 {
int x=12;
void test()
{
	System.out.println("running test() method..");
}
}
class Demo3 extends Demo2
{

}

package pack1;

public class MainClass2 {

	public static void main(String[] args) {
		System.out.println("******************");
		Demo3 d1=new Demo3();
		System.out.println(" x value : "+d1.x);
		d1.test();

		Demo2 d2=new Demo3();
```

```
                System.out.println("----------");

                System.out.println(" x value : "+d2.x);
                d2.test();


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

Output:

```
******************
 x value : 12
running test() method..
----------
 x value : 12
running test() method..
******************
```

```
package pack1;

abstract public class Demo2 {
int x=12;
abstract void disp();
void test()
{
        System.out.println("running test() method..");
}
}
class Demo3 extends Demo2
{
        void disp(){
                System.out.println("running disp() method");
        }
}

package pack1;

public class MainClass2 {

        public static void main(String[] args) {
                System.out.println("******************");
                Demo3 d1=new Demo3();
                System.out.println(" x value : "+d1.x);
                d1.test();
                d1.disp();

                Demo2 d2=new Demo3();
                System.out.println("----------");

                System.out.println(" x value : "+d2.x);
                d2.test();


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

JAVA PART II

OUTPUT:

```
******************
x value : 12
running test() method..
running disp() method
---------------
x value : 12
running test() method..
*****************
```

```
package pack1;

public abstract class Demo4 {

	abstract  void test();
	abstract void disp();

}
abstract class Demo5 extends Demo4
{
	void test()
	{
		System.out.println("test() method is defined in Demo5 class");
	}
}
 class Demo6 extends Demo5
{
	void disp()
	{
		System.out.println("Disp() method is defined in Demo6 class");
	}
	}


package pack1;

public class MainClass3 {


	public static void main(String[] args) {
		Demo4 d1=new Demo6();
		d1.disp();
		d1.test();

	}

}
package pack1;

public abstract class Sample1 {
abstract void test();
}
abstract class Sample2 extends Sample1{
	abstract void disp();
}
class Sample3 extends Sample2
```

```
{
	void disp()
	{
		System.out.println("disp() methid is defined in Sample3 class");
	}
	void test()
	{
		System.out.println("test() method is defined in Sampple3 class");
	}
}
package pack1;

public class MainClass4 {
public static void main(String[] args) {
	Sample1 s1=new Sample3();
	s1.test();

	System.out.println("---------------");
	Sample2 s2=new Sample3();
	s2.test();
	s2.disp();

}
}
```

Output:

```
test() method is defined in Sampple3 class
---------------
test() method is defined in Sampple3 class
disp() methid is defined in Sample3 class
```

- If a class inherits from a class, the sub class has to fulfill the contract of the abstract class
- The contract specifies that the sub class must define all the inherited abstract methods otherwise the sub class must be declared as abstract class.
- The abstract class can inherit from another class
- An abstract class can inherit from concrete class.

```
 package pack1;

abstract public class Sample4 {

	abstract public int square(int num);
}
class Sample5 extends Sample4
{
	public int square(int arg)
	{
		int res=arg*arg;
		return res;
	}
}
```

JAVA INTERFACE

- An interface is a java type definition block used to define only the abstract method.
- The interface is declared with a keyword interface.

JAVA PART II

- Inside interface we ca only declare static variables. The static variable must be final.
- Inside a interface we can not define a constructor.
- We can not define block also.
- Inside interface, only abstract methods are allowed
- In interface only public access is allowed.
- By default, the interface body is abstract.
- By default, the interface variables are final and static.
- By default, the method is abstract.
- By default, the access specifiers is public.
- We can not create the object of interface type. Because it is a pure abstract.
- Since we can declare only abstract in the interface body. it is known as pure abstract body.

```
public interface Demo1 {
int var1=23;
void test();
}
class Sample implements Demo1
{
        public void test()
        {
                System.out.println("test() method is defined inside Sample classS");
        }
        }
```

- The methods of the interface can be implemented in the class by using implements keyword.
- Whenever a class implements an interface the class must provide implementation to the all abstract methods of the interface. Otherwise the class must be declared as abstract.
- The class which an implementation to the interface is known as implementation class.
- A class can give implementation to more than one interface.

```
public interface Demo1 {

void test();
void disp();
}
abstract class Sample implements Demo1
{
        public void test()
        {

                System.out.println("test() method is defined inside Sample classS");
        }
}
class Sample2 extends Sample
{
        public void disp()
        {
                System.out.println("disp() method defined in Sample 2 class");
        }
        }


public class MainClass {
public static void main(String[] args) {
        Sample2 s2=new Sample2();
```

```
		s2.test();
		s2.disp();

		System.out.println("-----------------");
		Demo1 d1=new Sample2();
		d1.test();
		d1.disp();

}
}
```

OUTPUT:

```
test() method is defined inside Sample classS
disp() method defined in Sample 2 class
-----------------
test() method is defined inside Sample classS
disp() method defined in Sample 2 class
```

```
public interface Demo2 {

		void test();
}
interface Demo3 extends Demo2
{

		void disp();
}
class Sample3 implements Demo3
{
		public void disp()
		{
			System.out.println("disp() method in Sample3");
		}
		public void test()
		{
			System.out.println("test() method in Sample3");
		}
}



public class MainClass {
public static void main(String[] args) {
		Demo2 d1=new Sample3();
		d1.test();
		System.out.println("-0----------0--------");
		Demo3 d2=new Sample3();
		d2.disp();
		d2.disp();
}
}
```

Output:

```
test() method in Sample3
-0----------0--------
disp() method in Sample3
```

JAVA PART II

disp() method in Sample3

```
public interface Demo4 {
void test();
}
interface Demo5
{
	void disp();
}
class Sample4 implements Demo4,Demo5
{
	public void disp()
	{
		System.out.println("disp() method is implemented in Sample4 class");
	}
	public void test()
	{
		System.out.println("test() method is implemented in Sample4 class");
	}
}
```

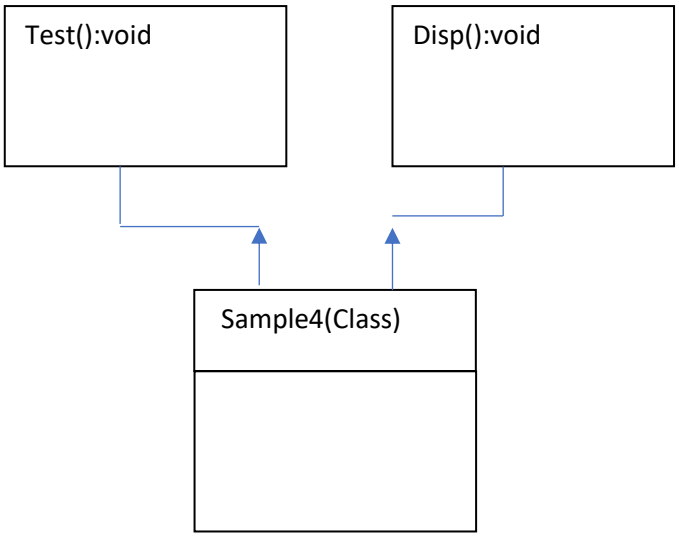
```
public class MainClass3 {

	public static void main(String[] args) {

		Demo4 d4=new Sample4();
		d4.test();
		System.out.println("-----------------");
		Demo5 d5=new Sample4();
		d5.disp();



	}
}
```

Output:

```
test() method is implemented in Sample4 class
-----------------
disp() method is implemented in Sample4 class
```

- A class can implement more than one interface.
- If the multiple interfaces are having same method declaration then the class must provide one implementation.

JAVA PART II




```
class Sample extends Object implements Demo1
{

}
```

```
public interface Demo4 {
void test();
}
class Run1
{
	public void test()
	{
		System.out.println("test() method is defined in Run1 class");
	}
}
class Sample5 extends Run1 implements Demo4{

}
```

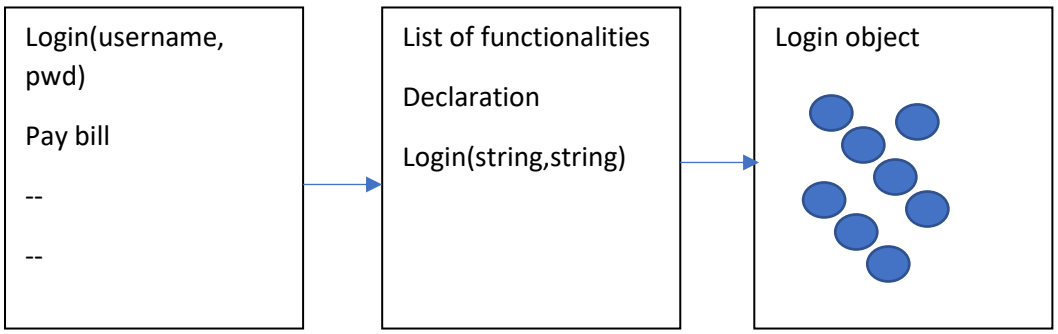
- In this example, the class sample5 need not provide implementation to the abstract method of the interface because the concrete method inherited from the Run1() because its acts as implementation method to the interface method.
- This works perfectly if the method signatures are same including the access specifiers.
- Defining an empty interface is known as marker interfaces.

```
interface Demo1
{

}
```

- Abstraction is one of the major oops principle used in developing the applications.
- The principle specifies that hide the implementation of the object functionality provide an interface to make use of the object functionality.

JAVA PART II

- In abstraction program, the code will be using the functionality of the object without knowing the exact implementation of the object.
- The abstraction code is written in the following steps
  - i. Declare all the essential functionality of the object in an interface. We can also
  - ii. use the abstract class for this step if we need concrete methods also.
    - In this step we specify only the essential functionality of the object.
  - iii. Define various implementation class for the functionality declared in the interface.
    - If the implementation are deferring we write multiple implementation class for one interface.
  - iv. Ref the implementation by using the interface type ref variables.
    - While implementing a project generally we write program in 3 layers
      1. Utilization layer
      2. Interface layer
      3. Implementation layer.
- The utilization layer uses the functionality which are present in the implementation layer
- The utilization layer can be another java program or can be client part of the application or it can be another s/w application.
- The utilization layer will not access directly the implementation layer functionary .
- It makes use of the functionality through the implementation layer
- The implementation layer provides set of functionality about the functionalities present in the implementation layer.
- The interface layer doesn't have the implementation.
- The implementation layer has all the functionality specified in the interface layer.
- The implementation might be single or multiple implementation for the same class.
- This is also known as programming to interface.
- The advantages are
  - Any changes made in the implementation layer doesn't affect the utilization layer.
  - But if we change the interface layer then we have to change utilization layer.




```
package bankApp;

public interface Account {

        void deposit(double amt);
        void withdraw(double amt);
        void balanceEnquiry();
}
```

JAVA PART II

```
package bankApp;

public class SavingAccount implements Account
{

	String custname;
	double accbal;
	public SavingAccount(String custname, double accbal) {

		this.custname = custname;
		this.accbal = accbal;
	}
	public void deposit(double amt) {
		System.out.println("Depositing Rs. "+amt);
		accbal=accbal+amt;
	}
	public void withdraw(double amt) {
		System.out.println("Withdrawing Rs. "+amt);
		accbal=accbal-amt;

	}
	public void balanceEnquiry() {
		System.out.println("Account balance is  Rs. "+accbal);

	}

}


package bankApp;

public class LoanAccount implements Account{
	String custname;
	double accbal;
	public LoanAccount(String custname, double accbal) {

		this.custname = custname;
		this.accbal = accbal;
	}
	public void deposit(double amt) {
		System.out.println("Depositing Rs. "+amt);
		accbal=accbal-amt;
	}
	public void withdraw(double amt) {
		System.out.println("Withdrawing Rs. "+amt);
		accbal=accbal+amt;

	}
	public void balanceEnquiry() {
		System.out.println("OutStanding balance is  Rs. "+accbal);

	}

}
package bankApp;

public class TestBankApp {


	public static void main(String[] args) {
```

```
		System.out.println("welcome to HDB Banking Application");

		Account a=new LoanAccount("hdb", 10000.00);
		a.balanceEnquiry();
		a.deposit(5000.00);
		a.balanceEnquiry();
		a.withdraw(7000.00);
		a.balanceEnquiry();


	}
}
```

OUTPUT:

```
welcome to HDB Banking Application
OutStanding balance is  Rs. 10000.0
Depositing Rs. 5000.0
OutStanding balance is  Rs. 5000.0
Withdrawing Rs. 7000.0
OutStanding balance is  Rs. 12000.0
```

```
package bankApp;

public class AccountDB {

	static Account getAccount(String custname,double initAmt,char accType)
	{


		Account acc=null;
		if(accType=='s')
		{
			acc=new SavingAccount(custname, initAmt);

		} else if(accType=='l')
		{
			acc=new LoanAccount(custname, accType);
		}




		return acc;

	}

}


package bankApp;

public class TestBankApp {


	public static void main(String[] args) {
```

```
                System.out.println("welcome to HDB Banking Application");

                Account a=AccountDB.getAccount("hdb", 10000.00, 's');
                a.balanceEnquiry();
                a.deposit(5000.00);
                a.balanceEnquiry();
                a.withdraw(7000.00);
                a.balanceEnquiry();


        }
}
```

Getting non-static member variables:

```
package bankApp;

public class Demo1 {
        int x=12;

        void test()
        {
                System.out.println("test method....");
        }

}
```

```
package bankApp;

public class Sample {

        double y=52;
        Demo1 d1=new Demo1();
        void disp() {
                System.out.println("disp method....");
        }

}
```

```
package bankApp;

public class MainSanple {



        public static void main(String[] args) {
                Sample s1=new Sample();
                System.out.println(s1.d1);
                s1.d1.test();
        }



}
```

JAVA PART II

Getting static member variables

```
package bankApp;

public class Sample {

	double y=52;
	static Demo1 d1=new Demo1();
	void disp() {
		System.out.println("disp method....");
	}

}
```

```
package bankApp;

public class MainSanple {



	public static void main(String[] args) {
		Sample s1=new Sample();
		System.out.println(s1.d1);
		System.out.println(Sample.d1.x);
		Sample.d1.test();
	}

}
```

Output:

```
bankApp.Demo1@15db9742
12
test method....
```

Sample.d1.test();
Where,
Sample – Classname
d1-static ref variable of Demo1 type
test()- non static function

System.out.println();

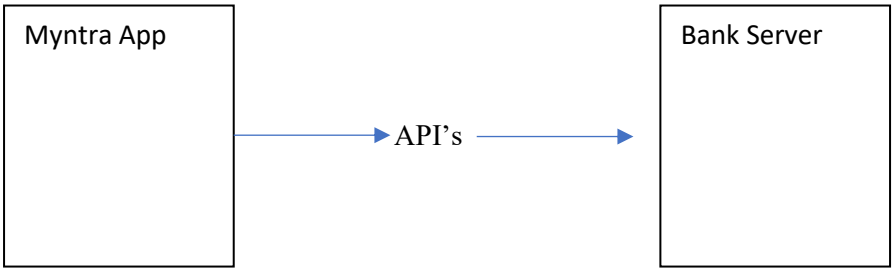
Where
System- classname(library)
Out- static ref variable
Println()- non-static function

JAVA PART II

- Defining a class having a ref to another class object is known as has a relationship.
- The class has ref variables has a member.
- The ref variable can be static or non-static member variable.
- If the class is having non-static ref variables, for each object creation of the class the contained object copy will be created.
- If the class is having static ref variable for each object creation of the class , only one copy of the contained object will be created.

APPLICATION PROGRAMMING INTERFACES(API'S)

- Set of java programs.
- Build on abstraction principles.
- Provide a set of specification about the object(class) functionalities.



JAVA PART II

Demo1 d1=new Demo1()

Demo2 d2=new Demo2()

If(d1==d2)

{

SOP(“YES”);

}else

{

SOP(“NO”);

}

OUTPUT:

NO (because the value of d1(address of d1) is compared with value of d2)

Int x=20;

Int y=20;

If(x==y)

SOP(“Values are same”);

else

SOP(“values are not same”);

JAVA PART II

Equals method(not overridden in the Demo2 class)

```
package pack1;

public class Demo2 {

	int x;

	public Demo2(int x) {

		this.x = x;
	}
	public String toString()
	{
		return "Demo2 :x "+x;

	}


}
```

```
package pack1;

public class MainClass3 {

	public static void main(String[] args) {
		Demo2 d1=new Demo2(25);
		Demo2 d2=new Demo2(25);
		if(d1.equals(d2))
		{
			System.out.println("Objects are equal in property");
		}else
		{
			System.out.println("Objects are not in eqaul in propery");
		}
	}
}
```

Output:

Objects are not in eqaul in propery

Equals overridden in the Demo2 class

```
package pack1;

public class Demo2 {

	int x;

	public Demo2(int x) {

		this.x = x;
	}
```

```
	public boolean equals(Object arg){
		Demo2 ref=(Demo2)arg;
		return this.x==ref.x;
	}
	public String toString()
	{
		return "Demo2 :x "+x;

	}


}
package pack1;

public class MainClass3 {

	public static void main(String[] args) {
		Demo2 d1=new Demo2(25);
		Demo2 d2=new Demo2(25);
		if(d1.equals(d2))
		{
			System.out.println("Objects are equal in property");
		}else
		{
			System.out.println("Objects are not in eqaul in propery");
		}
	}
}
```

Output:

Objects are equal in property

Syntax:
Public Boolean equals(Object o)

- The equal methods in the object is implemented to check the equality b/w the current and the given object based on the hashCode values.
- The fun returns if the current Object hashCode is equal to the given Object hashCode otherwise returns false
- If we want to check the equality of two objects on the basis of its property then we must override the equals methods in the class.

CHECKING EQUALITY OF TWO OBJECTS BASED ON TWO PROPERTIES:

```
package pack1;

public class Demo2 {

	int x;
	double y;

	public Demo2(int x) {

		this.x = x;
```

```
	}
	public Demo2(int x,double y) {

		this.x = x;
		this.y=y;
	}

	public boolean equals(Object arg){
		Demo2 ref=(Demo2)arg;
		return this.x==ref.x && this.y==ref.y;
	}
	public String toString()
	{
		return "Demo2 :x "+x;

	}


}
package pack1;

public class MainClass3 {

	public static void main(String[] args) {
		Demo2 d1=new Demo2(25,12.34);
		Demo2 d2=new Demo2(25,12.34);
		if(d1.equals(d2))
		{
			System.out.println("Objects are equal in property");
		}else
		{
			System.out.println("Objects are not in eqaul in propery");
		}
	}
}
```

OUTPUT:

Objects are equal in property

Note:

- Whenever we print object ref or objects itself the JVM internally calls toString() function defined in the instance, the print statements prints the return value of the toString() function.
- If toString() is overridden we get overridden results otherwise we get object class implementation results.
- The toString() method is overridden to represent the state of the object in a string format.
- hashCode() is overridden to generate unique has code number using our own algorithm, equals is overridden to check the equality b/w two objects on the objects properties.

Assignments :

1. write a Java program to check whether two wall clocks are equal or not in the type. The program should output both wall clocks showing the o/p of they are equal . both the wall clocks are not showing the o/p if they are not equal.

JAVA PART II

- String is a class defined in the java.lang package
- The string class is final
- The string class has overloaded constructor.
- The string class is immutable class.
- String class is thread safe.
- The string class implements comparable interface hence strings are comparable in nature.
- The array or collection of strings can be sorted in the string class.
- toString() is overridden to return the String value stored in the String type objects.
- HashCode() is overridden to generate hash code number on string value basis.
- Equals is overridden to compare toString objects based on the string value.

  - The string class objects can be created in two ways
    i. Using new operator
    ii. Without using new operator
  - The string objects created without operator is called as String literal.
  - The string objects are stored in the separate memory pool known as string pool.

  - The string pool area is divided into

    a. Constant pool area
    b. Non-constant pool area
  - The constant pool are doesn't duplicate string where as non- constant pool are allows duplicate string
  - The string objects created using new operator are always stored in the non constant pool area whereas string object created without new operator are stored in the constant Boolean.

String constructor ;

1. No arg constructor
   String s1=new String();
   //creates an empty string class object
2. String type arg constructor
   String s2=new String("jspiders");
   //creates a String class object with specified string value
3. Char array type arg constructor
   Char [] ch={'j','a','v','a'};
   String s3=new String(ch);
   // creates a String class object with specified char value

```
package pack1;

public class MainClass1 {
	public static void main(String[] args) {
		String s1=new String("jspiders");
		String s2=new String("jspiders");
//		String s3="hdb";
//		String s4="hdb";
```

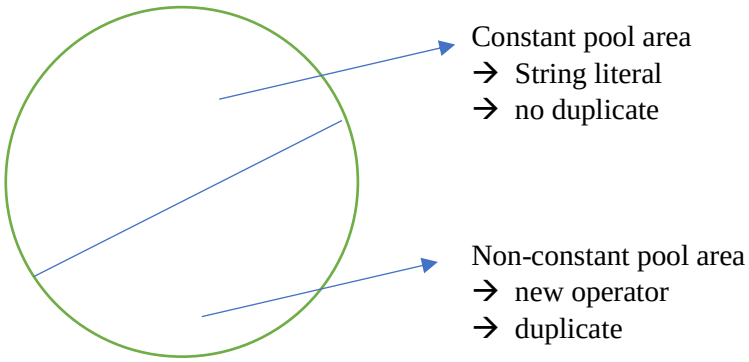
```
                System.out.println(s1);
                System.out.println(s2);
//              System.out.println(s3);


                int n1=s1.hashCode();
                int n2=s2.hashCode();
                System.out.println(n1);
                System.out.println(n2);
                if(s1==s2)
                {
                        System.out.println("same");
                }else
                {
                        System.out.println("not same");
                }

        }

}
```

Output:

```
jspiders
jspiders
-1950442876
-1950442876
not same
```

String class object:

i. New operator
ii. Without new operator

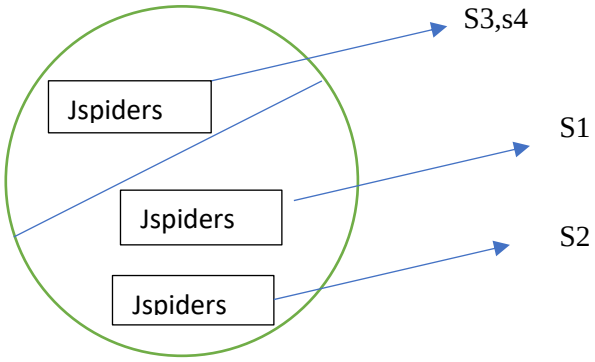



```
                String s1=new String("jspiders");
                String s2=new String("jspiders");
                String s3="jspiders";
                String s4="jspiders";
```

S1==s2 //false

JAVA PART II

S1==s3// false
S3==s4//true



X=34;

String s1=new String(“java”);
--
--
--
s1=new String(“android”); //re-initialization
.
.
.
.
s1=new String(“springs”); // re-initialization

---

e.g
String s1=”java”;
.
.
.
s1=”android”; // re-initialization

.
.
.
s1=”springs”; // re-initialization

- Any class which doesn’t allow to change the state of the object is known as immutable class
- Whenever we create an object of immutable class, we can not change the value of its states , if we try to change the value it creates new object rather than changing the existing object.
- In java language, string class and wrapper class are immutable.

JAVA PART II

- From JDK 1.5 onwards the java library has been updated with two classes, String builder and String buffer, Both are immutable in nature.

STRING FUNCTIONS

```
s1.charAt(index);
s1.startsWith(prefix);
s1.lastIndexOf(ch);
s1.indexOf(ch);
s1.indexOf(ch, fromIndex);
s1.concat(str);
s1.endsWith(suffix);
s1.substring(beginIndex, endIndex);
s1.toUpperCase();
        s1.toLowerCase();
```

```
String s1="javadeveloper";

System.out.println(s1.charAt(8));
System.out.println(s1.startsWith("java"));
System.out.println(s1.lastIndexOf('j'));;
System.out.println(          s1.indexOf('v'));
System.out.println(s1.indexOf('a', 4));
System.out.println(s1.concat("dev"));
System.out.println(s1.endsWith("ja"));
System.out.println(s1.substring(5, 10));
System.out.println(s1.toUpperCase());
System.out.println(s1.toLowerCase());
```

Output:

```
l
true
0
2
-1
javadeveloperdev
false
```

JAVA PART II

evelo
JAVADEVELOPER
javadeveloper

```
public int length();
public char charAt(int index);
public int indexOf(char c);
public int indexOf(int index, char c);
public int lastindexOf(char c);
public boolean contains(String charSeq);
public boolean startWith(String charSeq);
public boolean endsWith(String charSEq);
public String substring(int startWith);
public String substring(int startIndex,int endIdex);
public String toUpper(String ch);
        public String toUpper(String Ch);
```

```
package pack1;

public class Demo1 {

    public static void main(String[] args) {

        String str="virat kohli is captain of Indian Cricket team";
        String[] a=str.split(" ");
        System.out.println("Total Words "+a.length);
        for (int i = 0; i < a.length; i++) {
            int n=a[i].length();
            System.out.println(a[i]+"--->"+n);

        }



    }


}
```

Output:

JAVA PART II

Total Words 8
virat--->5
kohli--->5
is--->2
captain--->7
of--->2
Indian--->6
Cricket--->7
team--->4

String Builder and String buffer:

- Both the classes are defined in the java.lang.package
- Both classes are final classes.
- Both the classes are mutable in nature.
- The constructor in both classes are overloaded
- We can create the object only by using new operator.
- In both the classes, only toString() function is overloaded.
- hashCode() and equals() functions are not over loaded.
- Both classes doesn't implement comparable interfaces hence they are not comparable in nature.
- Array of object of both classes can not be sorted.
- Both classes functions wise same but the string buffer functions are synchronized.
- String buffer is a thread shape, String builder is not a thread shape.
- Both the classes are introduced in java language from JDK 1.5 onwards.

String	StringBuffer	StringBuilder
Defined in java.lang.package	Defined in java.lang.package	Defined in java.lang.package
String is final class	StringBuffer is final class	StringBuilder is a final class
String is immutable class	StringBuffer is a mutable class	StringBuilder is a mutable class
String is comparible interfaces	StringBuffer is not comparible interface	StringBuffer is not comparible interface
Array of String type Objects can be sorted	Can not be sorted	Can not be sorted
String class methods are not synchronized	Not synchronized	Are synchronized
String class is thread safe (because of immutable)	Not a thread safe	Is thread safe
In string class toString(), equals(), hashCode() are overloaded	In StringBuffer, only toString() is overloaded	In StringBuilder, only toString() is overloaded

//Program

JAVA PART II

// program

```
package pack1;

public class Demo3 {

	public static void main(String[] args) {
		String s1="developer";
		StringBuilder sb=new StringBuilder(s1);
		//string type is represented in StringBuilder format
		//manipulation

		sb.reverse();

		s1=sb.toString();
		System.out.println(s1);


	}
```

```
}
```

Output:

```
repoleved
```

Method Summary

char	charAt(int index)
	Returns the char value at the specified index.
int	codePointAt(int index)
	Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index)
	Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex)
	Returns the number of Unicode code points in the specified text range of this String.
int	compareTo(String anotherString)
	Compares two strings lexicographically.
int	compareToIgnoreCase(String str)
	Compares two strings lexicographically, ignoring case differences.
String	concat(String str)
	Concatenates the specified string to the end of this string.
boolean	contains(CharSequence s)
	Returns true if and only if this string contains the specified sequence of char values.
boolean	contentEquals(CharSequence cs)
	Compares this string to the specified CharSequence.
boolean	contentEquals(StringBuffer sb)
	Compares this string to the specified StringBuffer.
static String	copyValueOf(char[] data)
	Returns a String that represents the character sequence in the array specified.
static String	copyValueOf(char[] data, int offset, int count)
	Returns a String that represents the character sequence in the array specified.
boolean	endsWith(String suffix)
	Tests if this string ends with the specified suffix.
boolean	equals(Object anObject)
	Compares this string to the specified object.

boolean equalsIgnoreCase(String anotherString)

Compares this String to another String, ignoring case considerations.

static String format(Locale l, String format, Object... args)

Returns a formatted string using the specified locale, format string, and arguments.

static String format(String format, Object... args)

Returns a formatted string using the specified format string and arguments.

byte[] getBytes()

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

byte[] getBytes(Charset charset)

Encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.

void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)

Deprecated. This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the getBytes() method, which uses the platform's default charset.

byte[] getBytes(String charsetName)

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

Copies characters from this string into the destination character array.

int hashCode()

Returns a hash code for this string.

int indexOf(int ch)

Returns the index within this string of the first occurrence of the specified character.

int indexOf(int ch, int fromIndex)

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

int indexOf(String str)

Returns the index within this string of the first occurrence of the specified substring.

int indexOf(String str, int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

String intern()

Returns a canonical representation for the string object.

JAVA PART II

boolean isEmpty()

Returns true if, and only if, length() is 0.

int lastIndexOf(int ch)

Returns the index within this string of the last occurrence of the specified character.

int lastIndexOf(int ch, int fromIndex)

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

int lastIndexOf(String str)

Returns the index within this string of the rightmost occurrence of the specified substring.

int lastIndexOf(String str, int fromIndex)

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

int length()

Returns the length of this string.

boolean matches(String regex)

Tells whether or not this string matches the given regular expression.

int offsetByCodePoints(int index, int codePointOffset)

Returns the index within this String that is offset from the given index by codePointOffset code points.

boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)

Tests if two string regions are equal.

boolean regionMatches(int toffset, String other, int ooffset, int len)

Tests if two string regions are equal.

String replace(char oldChar, char newChar)

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

String replace(CharSequence target, CharSequence replacement)

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.

String replaceAll(String regex, String replacement)

Replaces each substring of this string that matches the given regular expression with the given replacement.

String replaceFirst(String regex, String replacement)

Replaces the first substring of this string that matches the given regular expression with the given replacement.

String[] split(String regex)

Splits this string around matches of the given regular expression.

String[] split(String regex, int limit)

Splits this string around matches of the given regular expression.

boolean startsWith(String prefix)

Tests if this string starts with the specified prefix.

boolean startsWith(String prefix, int toffset)

Tests if the substring of this string beginning at the specified index starts with the specified prefix.

CharSequence subSequence(int beginIndex, int endIndex)

Returns a new character sequence that is a subsequence of this sequence.

String substring(int beginIndex)

Returns a new string that is a substring of this string.

String substring(int beginIndex, int endIndex)

Returns a new string that is a substring of this string.

char[] toCharArray()

Converts this string to a new character array.

String toLowerCase()

Converts all of the characters in this String to lower case using the rules of the default locale.

String toLowerCase(Locale locale)

Converts all of the characters in this String to lower case using the rules of the given Locale.

String toString()

This object (which is already a string!) is itself returned.

String toUpperCase()

Converts all of the characters in this String to upper case using the rules of the default locale.

String toUpperCase(Locale locale)

Converts all of the characters in this String to upper case using the rules of the given Locale.

String trim()

Returns a copy of the string, with leading and trailing whitespace omitted.

static String valueOf(boolean b)

Returns the string representation of the boolean argument.

JAVA PART II

static String valueOf(char c)

Returns the string representation of the char argument.

static String valueOf(char[] data)

Returns the string representation of the char array argument.

static String valueOf(char[] data, int offset, int count)

Returns the string representation of a specific subarray of the char array argument.

static String valueOf(double d)

Returns the string representation of the double argument.

static String valueOf(float f)

Returns the string representation of the float argument.

static String valueOf(int i)

Returns the string representation of the int argument.

static String valueOf(long l)

Returns the string representation of the long argument.

static String valueOf(Object obj)

Returns the string representation of the Object argument.

String s1=”java”;

String s2=”developer”;

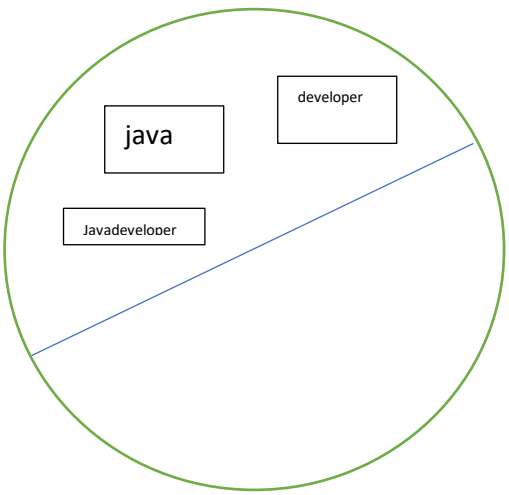
String s3=”javadeveloper”;

String s4=”java”+”developer”;

String s5=s1+”developer”;

String s6=”java”+s2;

String s7=s1+s2;



JAVA PART II

JAVA PART II

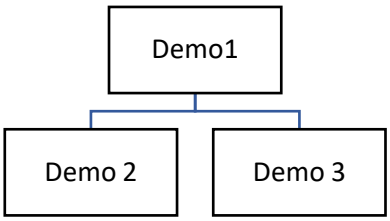
Arrays:

i. Array of primitive types
ii. Array of non-primitive types

Int[] a1=new int[10];

Float [] a2=new float[10];

Demo1 [] a3=new Demo1[10];
→ 10 objects of Demo1 type
a. Can store Demo1 type
b. Can store demo2 Object
c. Can store Demo3 Object




```
package arrays;

abstract public class Citizen {

	private int age;
	private String name;
	public Citizen(int age, String name) {
		super();
		this.age = age;
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
```

```
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}



}
```

```
package arrays;

public class Student extends Citizen {

	private int RollNo;
	private double marks;
	public Student(int age, String name, int rollNo, double marks) {
		super(age, name);
		RollNo = rollNo;
		this.marks = marks;
	}
	public int getRollNo() {
		return RollNo;
	}
	public void setRollNo(int rollNo) {
		RollNo = rollNo;
	}
	public double getMarks() {
		return marks;
	}
	public void setMarks(double marks) {
		this.marks = marks;
	}




}
```

```
package arrays;

public class Employee extends Citizen {
	private int id;
	private double salary;
	public Employee(int age, String name, int id, double salary) {
		super(age, name);
		this.id = id;
		this.salary = salary;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
```

```
	public double getSalary() {
		return salary;
	}
	public void setSalary(double salary) {
		this.salary = salary;
	}


}



package arrays;

public class Deko2 {


	public static void main(String[] args) {
		Citizen [] citArr=new Citizen[5];

		citArr[0]=new Student(19, "rakesh", 678, 78.99);
		citArr[1]=new Student(25, "suresh", 679,35.78);
		citArr[2]=new Employee(19, "ramesh",123, 25000.00);
		citArr[3]=new Student(90, "hdb",673, 15000.00);

		citArr[4]=new Employee(67, "jaggesh",988, 78000.00);


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

	System.out.println(citArr[i].getName()+"\t"+citArr[i].getAge());


		}
		System.out.println("----------------------------------------");
		for (int i = 0; i < citArr.length; i++) {
			if(citArr[i] instanceof Student) {
				Student st=(Student)citArr[i];

	System.out.println(st.getName()+"\t+"+st.getRollNo()+"\t"+st.getAge()+"\t"+
st.getMarks());
			}
		}
		System.out.println("----------------------------------------");
			System.out.println("----------------------------------------
");
			System.out.println("Updating grace marks fro all student");

			for (int j= 0; j < citArr.length; j++) {
				if(citArr[j] instanceof Student) {
					Student st=(Student)citArr[j];
					st.setMarks(st.getMarks()+5);
				}
			}
			System.out.println("----------------------------------------
");
			System.out.println("----------------------------------------
");
```

JAVA PART II

```
		for (int i = 0; i < citArr.length; i++) {
			if(citArr[i] instanceof Student) {
				Student st=(Student)citArr[i];

	System.out.println(st.getName()+"\t+"+st.getRollNo()+"\t"+st.getAge()+"\t"+
st.getMarks());
			}




	}

	}
}
```

Output:

```
rakesh 19
suresh 25
ramesh 19
hdb    90
jaggesh      67
----------------------------------------
rakesh +678   19     78.99
suresh +679   25     35.78
hdb    +673   90     15000.0
----------------------------------------
----------------------------------------
Updating grace marks fro all student
----------------------------------------
----------------------------------------
rakesh +678   19     83.99
suresh +679   25     40.78
hdb    +673   90     15005.0
```

Object [] objArr=new Object[10];

address	Address	Address	Address	address
0	1	2	3	4

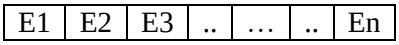
Drawbacks of Arrays:

JAVA PART II

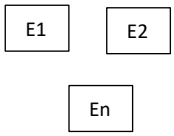
i. Limited in size
ii. Same type values.
iii. Linear DS
iv. No algorithm

Data Structure

i. Linear DS



ii. Non-linear DS



➔ Java Collection Frameworks(JCF) available in Java.util
➔ JDK 1.5 onwards,

- Lists
- Queues
- Tree
- Map

---------- Go to Java Part-II Doc----------