Unit 2

1 What are constructors ? explain different types of constructor with example

ANS:

Constructors:

- A constructor is a public method (a method that can be accessed anywhere in a program) with the same name as the class and it enables an object to initialize itself when it's created.
- Ones defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- Constructor have no return type, not even void.

- There are two types of constructor:-

1. **Default constructor:-** Default constructor fires automatically and initializes all instance variable to the default values. It will fire in the absence of even if a single parameterized constructor.

2. **Parameterized constructor:-** If the method defines parameters inside the method in parenthesis, it is called parameterized constructor

- A *constructor* initializes an object immediately upon creation. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors have no return type, not even **void**. This is because the implicit return type of a class. constructor is the class type itself.

Example:-

```
class Box
{
double  width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
 {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume()
 {
return width * height * depth;
```

	}
	}
	class BoxDemo7
	{
	public static void main(String args[])
	{
	// declare, allocate, and initialize Box objects
	Box mybox1 = new Box(10, 20, 15);
	Box mybox2 = new Box(3, 6, 9);
	double vol;
	// get volume of first box
	vol = mybox1.volume();
	System.out.println("Volume is " + vol);
	// get volume of second box
	vol = mybox2.volume();
	System.out.println("Volume is " + vol);
	}
	}
	The output from this program is shown here:
	Volume is 3000.0
	Volume is 162.0

2	With the help of suitable JAVA programs describe following : **ANS:** **Method (function) Overloading** Method overloading is the phenomenon using the same method name more than once in the sameclass and making it perform different functions. o Different number of parameter passed o Different order of passing parameter o Different data types of the parameter When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Method overloading is used when objects are required to perform similar tasks but using different input parameters. Method overloading is one of the ways that Java implements polymorphism. **Example:-** // Demonstrate method overloading.

```
class OverloadDemo
{
void test()
 {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a)
 {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b)
 {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a)
{
System.out.println("double a: " + a);
return a*a;
}
}
class Overload
 {
public static void main(String args[])
 {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
```

```
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Constructor Overloading:

in addition to overloading normal methods, you can also overload constructor methods. here is a program that contains as improved version of rectangle with constructor overloading

```
/* Program to demonstrate  Constructor overloading */
class Rectangle
{
 int length;
 int width;
 //  Constructor used when all values are specified
 Rectangle(int l,int w)
 {
        length = l;
        width = w;
 }
 //  Constructor used when no values are specified
 Rectangle()
 {
        length = 0;
        width = 0;
 }
```

```
//  Constructor used when one value is specified for both,length and width
 Rectangle(int l)
 {
        length = width = l;
 }
 // compute and return Area of a Rectangle
 int area()
 {
        return length*width;
 }
}
class RectangleDemo7
{
 public static void main(String args[])
 {
        // create Rectangles using the various constructors
        Rectangle rect1 = new Rectangle(20,15);
        Rectangle rect2 = new Rectangle();
        Rectangle rect3 = new Rectangle(20);
        int a;
        a = rect1.area();
 System.out.println("Area of First Rectangle(length:"+rect1.length+", width:"+rect1.width+") is:
 "+a);
        a= rect2.area();
 System.out.println("Area of Second Rectangle(length:"+rect2.length+", width:"+rect2.width+") is:
 "+a);
        a= rect3.area();
 System.out.println("Area of Third Rectangle(length:"+rect3.length+", width:"+rect3.width+") is:
 "+a);
 }
}
```

O/P:-

Area of First Rectangle(length:"20", width:15)is:300

Area of Second Rectangle(length:"0", width:"0")is: 0

Area of Third Rectangle(length:"20", width:"20")is:400

5 Write short note on :- **call by value** & **call by reference.**

ANS:

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

```
/* Program to demonstrates Call by Value  */
class Sample
{
void cal(int x,int y)
{
x=x+2;
y=y-2;
}
}
class CallByValue
{
public static void main(String args[])
{
Sample s=new Sample();
int p=20,q=10;
System.out.println(" p and q Before call :"+p+",
```

```
/* Program to demonstrates Call by reference
*/
class Sample
{
int p,q;
Sample(int x,int y)
{
p=x;
q=y;
}
// Pass an Object
void cal(Sample obj)
{
obj.p=obj.p+2;
obj.q=obj.q-2;
}
```

	"+q);
	s.cal(p,q);
	System.out.println(" p and q After call :"+p+",
	"+q);
	}
	}
	o/p:
	P and q Before call: 20, 10
	p and q After call: 20, 10
		}
		class CallByRef
		{
		public static void main(String args[])
		{
		Sample s=new Sample(20,10);
		System.out.println(" p and q Before call :
		"+s.p+", "+s.q);
		s.cal(s);
		System.out.println(" p and q After call :
		"+s.p+", "+s.q);
		}
		}
		o/p :
		P and q Before call: 20, 10
		p and q After call: 22, 8

6	Explain **static class members** with example. **Static field/ Members :** When a member is declared static, it can be accessed before any objects of its class declared, and without reference to any object. E such a member, precede its declaration with the keyword static. You can declare both methods and variables to be static. The common example of a static member is main(). main() is declared as static because it must be called before any objects exist. Rules for static members : They can only call other static methods. They must only access static data. They cannot refer to this or super in any way. (The keyword super is relates to inheritance and is described in Chapter 4) If you need to do computation in order to initialize your static variables, you can declare static block	

which gets executed exactly once, when the class is first loaded The following program shows a class that has a static method, some static variables, and a static initialization block.

```
/* Program to demonstrate static variables, methods, and blocks */
class StaticDemo1
{
        static int a = 5;
        static int b;
        static void display(int c)
        {
                System.out.println(" a = "+a);
                System.out.println(" b = "+b);
                System.out.println(" c = "+c);
        }
        static
        {
                System.out.println(" static block initialized ... ");
                b = a * 5;
        }
        public static void main(String args[])
        {
                display(40);
        }
}
/*  output
F:\java\static>javac StaticDemo1.java

F:\java\static>java  StaticDemo1
 static block initialized....
 a = 5
 b = 25
 c = 40        */
```

```
/* Program to demonstrate static variables, methods */
class StaticDemo
{
        static int i = 1;
        static int j = 5;
        static void display( )
        {       System.out.println(" i = "+i);
        }
}
class StaticDemo2
{       public static void main(String args[])
        {
                StaticDemo.display();
                System.out.println(" j = "+StaticDemo.j);
        }
}
/*  output
F:\java\static>javac StaticDemo2.java

F:\java\static>java  StaticDemo2
 i = 1
 j = 5          */
```

7 Write short note : Access Protection in java Or

Discuss different levels of Access Protection available in java ?

Ans:

- Java's four level of access protection:

Public

Private

Protected

Default

- Some aspects of access control are related mostly to inheritance or package.(A package is,

essentially, a grouping of classes)protected is applied only when inheritance is invoked. :

- Public Access:

  When a member of a class is modified by public specifier ,then that member can be accessed by any other code. Now you can understand why main() has always been preceded by the public specifier.it is called by code that is outside the program that is by java run time system
- Private Access: when a member of class is specified as private, then that member can only be accessed by other members of its class.
- Default Access: when no specifier is used then by default the member of class is public within its own package, but cannot be accessed outside of its package.in the classes development so far ,all members of a class have use the default access mode .which is essentially public. This access specifier is also known as friendly access. Difference between default access and public modifier is given below :

Public	default
The public modifier makes fields visible in all classes ,regardless of their packages	The default/friendly access makes fields visible only in the same package but not other packages.

- Access specifier precedes the rest of a members type specification. That is it must begin a members declaration statement.
- E.g

class member	syntax
instance variable	public int i; private float f;

instance method	private float method(int x,flat y)
	{
	}

Example of public & private

```
/* Program to demonstrate Access Protection */
class Demo
{
```

```
  int p;           // default access
  public int q;    // public access
  private int r;  // private access
  // methods to access r
  void setr(int i)
  {       // set r's value
                r = i;
  }
  int getr()
  {     // get r's value
        return r;
  }
}
class AccessProDemo
{
  public static void main(String args[])
  {
        Demo d= new Demo();
        // OK : 'p' and 'q' may be accessed directly
        d.p=300;
        d.q=400;

        // NOT OK : 'r' cannot be accessed outside the class, hence will cause an error
        // d.r=50;
        // You must access 'r' through its methods
        d.setr(500);       //OK
        System.out.println(" p, q, and r = "+d.p+", "+d.q+", and "+d.getr());
  }
}
```

o/p:-

P, q, and r =300,400, and 500

8 **Explain & compare member variable and local variable.**

Member variable :

The data or variable defined within a class are called member variables or instance variable

Local variable:

We often need to use temporary variables while completing a task in method.

Example :

```
void add()
{
int a;    //this is local variable
}
```

Difference between member variable and local variable.

	Member variable	**Local variable**
Accessibility	Member variable of a class are accessible from all instance method of the class	Local variable & parameters are accessible only from the method in which they are declared
Availability	Member variable of are available/can access till reference to an object exists .	Local variables are available only while the method is being executed
Creation of memory space	The new allocates memory for an object(that means for member variable) during run time	Memory space for local variable & parameters is allocated upon declaration and at the beginning of the method .
Destroy of memory space	Java used the concept of Garbage collector	Local variables & parameters are erased when the execution of a method is completed

9 **Write short note on - this keyword**

ANS:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the current object.That is, **this** is always a reference to the object on which the method was invoked.

You can use **this** anywhere a reference to an object of the current class type is permitted.

Method declaration with the keyword static(class method) cannot use this.

To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.
Box(double w, double h, double d)
{
this.width = w;
this.height = h;
this.depth = d;
}
```

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding:

As you know, it is illegal in Java to declare two **local variables** with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including **formal parameters** to methods, which overlap with the names of the class' **instance variables**. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.

This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth)
{
this.width = width;
this.height = height;
this.depth = depth;
```

	} A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt. Although **this** is of no significant value in the examples just shown, it is very useful in certain situations.