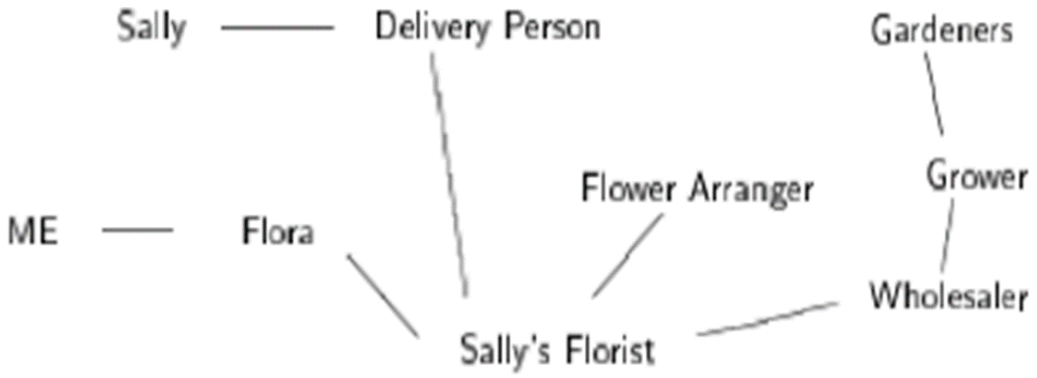
UNIT-1

Object-Oriented Thinking - A way of viewing the World

Agents and Communities:

- An object-oriented program is structured as a community of interacting agents, called objects
- Consider the scenario where I (stays at Hyderabad) wants to send a flower bouquet to my friend, who is at Chennai.



- Each object has a role to play
- Each object provides a service, or performs an action, that is used by other members of the community

Messages & Methods:

- Actions are initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action
- The message encodes the request for an action
- It is accompanied by any additional information (arguments) needed to carry out the request
- The receiver is the object to whom the message is sent
- In response to a message, the receiver will perform some method to satisfy the request

Responsibilities:

- A fundamental concept in object-oriented programming is to describe the behavior in terms of responsibilities.

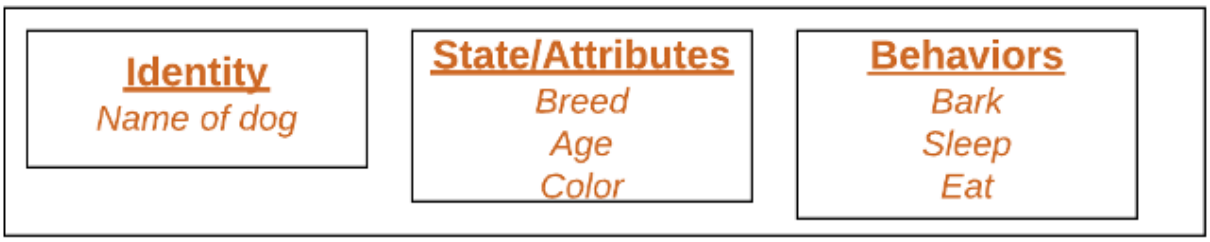
OOPS Concepts

Object

- It is the basic unit of Object Oriented Programming and it represents the real life entities.
- Examples: student, teacher, fan, dog, pen….etc.
- Objects have two characteristics: they all have state and behavior.

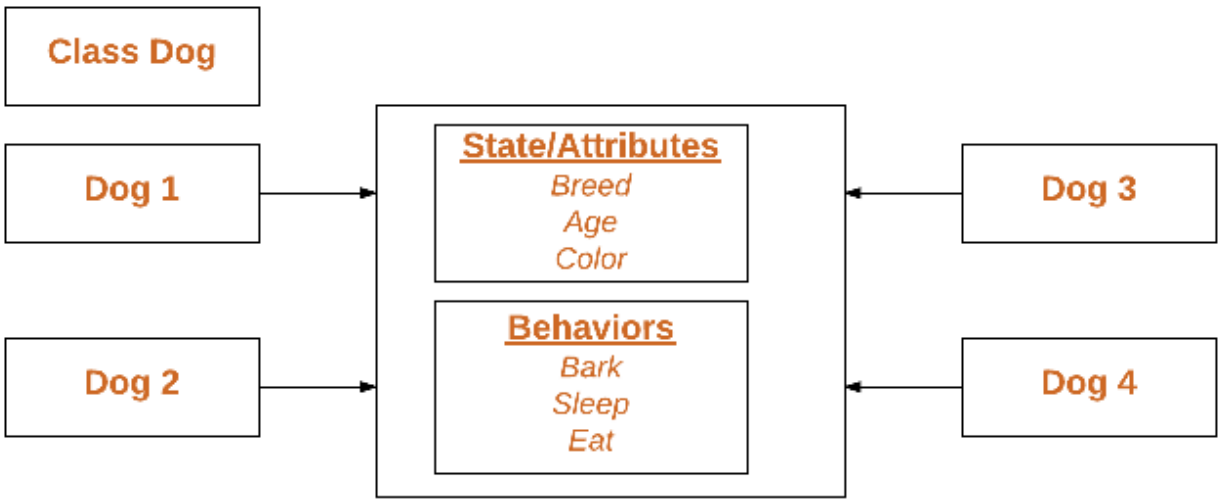
An Object Consists of

- **State:** It is represented by *attributes* of an object.
- **Behavior:** It is represented by *methods* of an object.
- **Identity:** It gives a unique name to an object.



Class:

- A class is a user defined blueprint or prototype (template) from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type.
- It has definitions of methods and data.



Data Hiding & Encapsulation

Using private access modifiers(access specifiers), we can hide data from other classes.

Binding data and its operations together into a single unit is known as encapsulation.

A java class is the example of encapsulation.

Abstraction

Hiding internal details (implementation details) and showing functionality is known as abstraction.

For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

Existing class: Super Class / Base Class / Parent Class

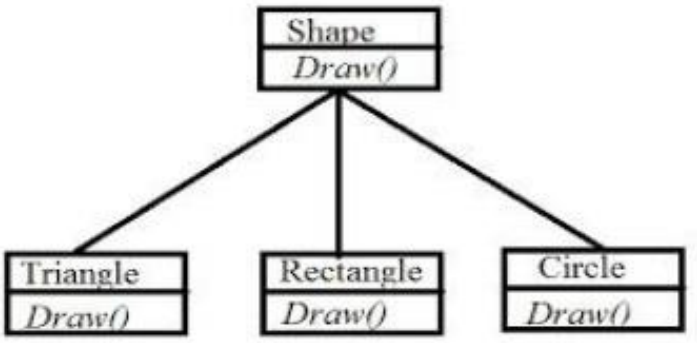
New class: Sub Class / Derived Class / Child Class

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.

When you inherit from an existing class, you can reuse methods and fields of the parent class.

Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.



We can say Triangle IS-A Shape, Rectangle IS-A Shape..etc.

Why use inheritance in java

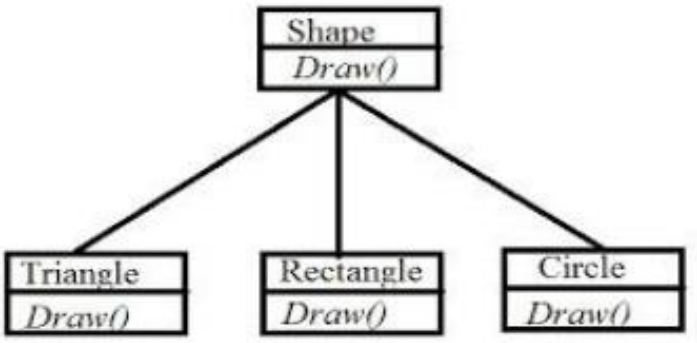
For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Polymorphism:

Polymorphism is the ability of an object (methods) to take on many forms.

Generally it occurs when we have many classes that are related to each other by inheritance.



In Java, we use method overloading and method overriding to achieve polymorphism.

JAVA BUZZ WORDS/ FEATURES

Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand.

Java inherits the C/C++ syntax and many of the object-oriented features of C++.

Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

Object Oriented

In java everything is an Object which has some data and behavior.

Un like C++, in java even the function main() has to be part of some class only.

We can't have any method outside of the class.

Java can be easily extended as it is based on Object Model.

Platform Independent:

C/CPP compiler produces ".exe" file which is interpreted by OS and the instructions in the ".exe" file are specific to OS(ie platform dependent).

Java Compiler produces ".class" file (byte code), which is interpreted on any platform with corresponding JVM.

The instructions in the ".class" file (byte code) is platform independent.

JVM(Java Virtual Machine) is a engine that enable the computer to run java programs.

JVM is the java interpreter, which converts java byte code into machine language.

It starts the java program execution by calling main() method.

It is part of JRE(Java Runtime Environment).

Compiled & Interpreted but High performance

Java is compiled and interpreted language.

Java Compiler generates '.class' file (Byte code).

Java Interpreter interprets this byte code, converts into an executable file (exe file).

Most previous attempts at cross-platform solutions have done so at the expense of performance.

Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Architecture-neutral

Java is "write once run anywhere anytime" language.

Operating system upgrades, processor upgrades, and changes in any core system resources does not make a program to malfunction.

Robust

Java uses strong memory management.

There is no pointers, avoids security problems.

Manages memory allocation and de allocation (In fact, de allocation is completely automatic, because Java provides garbage collection for unused objects.)

Java makes an effort to eliminate error-prone situations by emphasizing mainly on compile time error checking and runtime checking(exception handling & type checking).

Multithreaded

With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously.

Multithreading is a special type of Multitasking.

Multitasking:

Running multiple tasks simultaneously.

Each task is a separate program.

If we run programs sequentially one after the other, if the currently running program requires an I/O, the processor will be idle until the i/o operation is completed.

The aim of multi tasking is to reduce this idle time of the processor.

With the help of CPU Scheduling algorithms(Example- Round Robin), we can assign processor another job when the currently running program requires an i/o.

Multitasking- Advantages:

Throughput(no.of jobs that can be completed in a unit time) will be increased.

Quick response time for jobs(i.e no job needs to wait for a long time for the processor).

Thread:

Part of a program.

Sequence of instructions, which has independent path of execution.

Multithreading:

Running multiple threads of a program simultaneously.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java.

RMI (Remote Method Invocation) and EJB (Enterprise Java Bean) are used for creating distributed applications.

This feature of Java makes us able to access resources(objects/files) by calling the methods from any machine on the internet.

Secure

No explicit pointer

Java Programs run inside a virtual machine sandbox. Java Security Manager determines what resources a java class can access such as reading and writing to the local disk.

Primitive Data types:

Keyword	Type	Example
boolean	true or false	true
byte	8-bit integral value	123
short	16-bit integral value	22456
int	32-bit integral value	123
long	64-bit integral value	3123456789L

float	32-bit floating-point value	123.45f
double	64-bit floating-point value	123.456
char	16-bit Unicode value	'a'

variables:

Variables are containers for storing data values.

The value depends on the data type of the variable.

The value can be altered during program exam execution.

There are three types of variables in java: local, instance and static.

Syntax: datatype variable_name[=default_value];

Example: int x=50;

Expressions

Any unit of code that can be evaluated to a value is an expression.

Example-1: 10+15

Example-2: (x*y)/z;

Operators:

1) Unary Operators: expr++, expr--, ++expr, –expr

Example:

```
int counter = 0;
System.out.println(counter); // Outputs 0
System.out.println(++counter); // Outputs 1
```

2) Arithmetic: *, /, %, +, -

Example:

```
int a=10;
int b=5;
System.out.println(a+b);//15
```

3) Relational: <, >, <=, >=

Example:

```
int x = 10, y = 20, z = 10;
System.out.println(x < y); // Outputs true
```

4) Equality: ==, !=

Example:

```
int x = 10, y = 20;
if(x==y)
System.out.println("Both Same");
else
System.out.println("Both Not Same");
```

5) Bitwise Operators:

Bitwise OR (|):

This operator is a binary operator, denoted by '|'.

It returns bit by bit OR of input values, i.e, if either of the bits is 1, it gives 1, else it gives 0.

For example, a = 5 = 0101 (In Binary), b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

```
  0101
| 0111
  ————
  0111  = 7 (In decimal)
```

Bitwise AND (&):

This operator is a binary operator, denoted by '&'.

It returns bit by bit AND of input values, i.e, if both bits are 1, it gives 1, else it gives 0.

For example, a = 5 = 0101 (In Binary), b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

```
  0101
& 0111
  ————
```

0101 = 5 (In decimal)

Bitwise XOR (^):

This operator is a binary operator, denoted by '^'.

It returns bit by bit XOR of input values, i.e, if corresponding bits are different, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7

0101

^ 0111

————

0010 = 2 (In decimal)

Bitwise Complement (~) –

This operator is a unary operator, denoted by '~'.

It returns the one's complement representation of the input value, i.e, with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

For example,

a = 5 = 0101 (In Binary)

Bitwise Compliment Operation of 5

~ 0101

————

1010 = 10 (In decimal)

Right shift operator (>>):

Shifts the bits of the number to the right and fills the voids left with the sign bit (1 in case of negative number and 0 in case of positive number).

The leftmost bit and a depends on the sign of initial number.

Similar effect as of dividing the number with some power of two.

Example 1:

a = 10

a>>1 = 5

Example 2:

a = -10

a>>1 = -5

Left shift operator (<<):

Shifts the bits of the number to the left and fills 0 on voids left as a result.

Similar effect as of multiplying the number with some power of two.

Examples:

a = 5 = 0000 0101

b = -10 = 1111 0110

a << 1 = 0000 1010 = 10

a << 2 = 0001 0100 = 20

b << 1 = 1110 1100 = -20

b << 2 = 1101 1000 = -40

6) Logical Operators(Logical Short Circuit Operators): &&, ||

Example:

```
if(x != null && x.getValue() < 5)
{
 // Do something
}
```

The getValue() will be called only when x is not null.

7) Ternary : ? :

Example:

```
int a=2;
int b=5;
```

int min=(a<b)?a:b;

System.out.println(min);

8) Assignment: = += -= *= /= %= &= ^= |= <<= >>= >>>=

Example:

int a=10;

int b=20;

a+=4;//a=a+4 (a=10+4)

b-=4;//b=b-4 (b=20-4)

System.out.println(a);

System.out.println(b);

Access Modifiers / Access Specifiers

Access modifiers tells us the scope of the variable(ie where a variable can be accessed).

private: private members(variables & methods) can be accessed only with in the class.

default: these memebers can be accessed in any class with in the package.

protected: these members can be accessed in any class with in the package. Also if the class to which these members belong to has any derived classes in other packages, those classes also can access.

public: these member can be accessed in any class.

instance data memebers Vs static data member

Instance variables:

Each object has a separate copy of value for instance variables.

They can be accessed with object only.

Static variables:

For a static variable, there will be only one copy of value in the memory for a given class, irrespective of how many objects the class has.

All the object share this static variable.

We can access static variables with class name itself. No need to create object for the class.

Static method Vs Non-Static method

Non Static (instance) Method:

These methods operate on instance variables of the class.

We need an object to call instance methods.

Static methods:

These behavior of these methods is common to all objects.

Can call these methods directly with the class name no need of object.

Wrapper Classes:

Byte, Short, Integer, Long, Float, Double, Character & Boolean are the wrapper classes in java.

One of the main use of these classes is data type conversion.

Command Line Arguments - Example

```
class  Test
{
   public static void main(String args[])
   {
      int x,y,z;
      x=Integer.parseInt(args[0]);
      y=Integer.parseInt(args[1]);
      z=x+y;
      System.out.println("The sum="+z);
    }
}
```

Taking input from user at Runtime (Scanner Class) - Example

```
import java.util.Scanner;
```

```
class ex3
{
  public static void main(String args[])
  {
        int x,y,z;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the first no:");
        x=sc.nextInt();
        System.out.println("Enter the second no:");
        y=sc.nextInt();
        if(x>y)      System.out.println("First no is bigger");
        else
        System.out.println("Second no is bigger");
     }
}
```

Control Structures

while Loop:

to execute a statement or code block repeatedly as long as an expression is true

Syntax:

```
while (expression)
{
   Statement(s) to be executed if expression is true
}
```

do...while Loop

Syntax:

```
do
{
```

```
  Statement(s) to be executed;
}
while (expression);
```

for Loop:

```
for (initialization; test condition; iteration statement)
{
  Statement(s) to be executed if test condition is true
}
```

Arrays

Array is an object which contains elements of a similar data type.

Additionally, the elements of an array are stored in a contiguous memory location.

It is a data structure where we store similar elements.

We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

One Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[]   arr; (or)
dataType    []arr; (or)
dataType    arr[];
```

Instantiation of an Array in Java

```
arrayRefVar   =   new    datatype[size];
```

Example:

```
int   a[]=new   int[5];
int   a[]={};
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

int[][] arr=new int[3][3];

Example-1:

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

	for(int i=0;i<3;i++)
	{
		for(int j=0;j<3;j++)
		{
			System.out.print(arr[i][j]+" ");
		}
	 System.out.println();
	}
	}
}
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array.

In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class  TestJaggedArray
{
   public static void main(String[] args)
    {
        //declaring a 2D array with odd columns
        int    arr[][] = new    int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];

       //initializing a jagged array
      int count = 0;

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

      //printing the data of a jagged array
      for (int i=0; i<arr.length; i++)
     {
            for (int j=0; j<arr[i].length; j++)
            {
                System.out.print(arr[i][j]+" ");
```

```
            }
            System.out.println();//new line
        }
    }
}
```

Class Example (working with multiple objects)

```
class   Student
{
     private int sno;
     private String sname;
     public void init(int x,String y)
     {
         sno=x;
        sname=y;
      }
     public void display()
      {
          System.out.println(sno);
          System.out.println(sname);
       }
}
class  Test
{
  public static void main(String args[])
  {
        Student s1;
        s1=new Student();
```

```
        s1.init(1,"akbar");
        Student s2=new Student();
        s2.init(2,"vijay");
        s1.display();
        s2.display();
    }
}
```

Array of Objects:

```
class   Student
{
    private int sno;
    private String sname;
    public void init(int x,String y)
     {
         sno=x;
         sname=y;
     }
     public void display()
    {
         System.out.println(sno);
         System.out.println(sname);
   }
}
class   Test
{
  public static void main(String args[])
    {
```

```
    Student   s[]=new  Student[5];
    s[0]=new student();
    s[0].init(1,"xxx");
    s[0].display();
    s[1]=new student();
    s[1].init(2,"yyy");
    s[1].display();
    s[2]=new student();
    s[2].init(3,"zzz");
    s[2].display();
  }
}
```

Method Overloading / Adhoc Polymorphism / Compile time Polymorphism / Early Binding:

Defining multiple methods with the same method signature (method name along with parameters) is called method overloading.

We can overload two methods as follows

1. By varying no. of arguments.
2. By varying data type of the arguments.
3. By varying order of data types.

Note: we can't overload two methods by varying only return types.

Example:

```
class  Test
{
   static void sum(int x,int y)
   {
     System.out.println("sum:"+(x+y));
   }
```

```
    static void sum(float x,float y)
    {
       System.out.println("sum:"+(x+y));
     }
     static void sum(double x,double y)
    {
        System.out.println("sum:"+(x+y));
    }
    static void sum(int x,int y,int z)
   {
      System.out.println("sum:"+(x+y+z));
    }
}
class   Test2
{
   public static void main(String args[])
   {
        Test.sum(10,20);
        Test.sum(10,20,30);
       Test.sum(10.5,20.5);
       Test.sum(10.5f,20.5f);
  }
}
```

this key word

the keyword ‘this’ is a reference to the current object.

Example:

```
class   Student
```

```
{
     private int sno;
     private String sname;
     public void init(int sno,String sname)
    {
       this.sno=sno;
       this.sname=sname;
    }
    public void display()
    {
        System.out.println(sno);
        System.out.println(sname);
    }
}
class   Test
{
  public static void main(String args[])
   {
      Student   s1=new   Student();
      s1.init(1,"shiva");
      s1.display();
   }
}
```

Constructors

Constructor is a special method(member function) which is automatically called every time when a new object of the class is created.

The name of the method is same as the class name.

Constructor should not have return type.

Generally constructors are used to initialize object(data members).

Constructors can be overloaded.

If do not define any constructor, the compiler will provide as the default constructor for our class. The default no-argument constructor.

If we define at least one constructor (parameterized constructor for example), then the compiler will not provide as the default constructor.

Example – Constructor / Parameterized Constructor / Constructor overloading

```
class  sample
{
    int x,y,z;
    sample()
       {
         x=0;
         y=0;
        z=0;
      }
      sample(int x, int y, int z)
       {
            this.x=x;
            this.y=y;
            this.z=z;
        }
       sample(int a)
       {
            x=a;
            y=a;
            z=a;
        }
```

```
        void display()
        {
            System.out.println(x);
            System.out.println(y);
            System.out.println(z);
        }
}
class   Test
{
   public static void main(String args[])
     {
        sample  s1=new sample(10,20,30);
        s1.display();
        sample s2=new sample();
        s2.display();
        sample s3=new sample(100);
        s3.display();
     }
}
```

Strings

string basically represents sequence of char values.

In Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created.

Example:

1.String s1=”Shiva”; // string literal created on the string pool

2. String s2=new String(“Ravi”) // an exclusive string object created on the pool

Java String class methods

1. charAt(int index):

Returns char value for the given index.

Example:

```
String string = "animals";

System.out.println(string.charAt(0)); // a
```

2. length():

returns string length

Example:

```
String string = "animals";

System.out.println(string.length()); // 7
```

3. equals() & equalsIgnoreCase():

To check whether two strings contain same data or not.

Example:

```
System.out.println("abc".equals("ABC")); // false
```

4. substring():

looks for characters in a string. It returns parts of the string.

Example:

```
String string = "animals";

System.out.println(string.substring(3)); // mals
```

5. indexOf( ):

looks at the characters in the string and finds the first index that matches the desired value.

indexOf can work with an individual character or a whole String input.

It can also start from a requested position.

Example:

```
System.out.println(string.indexOf("al")); // 4

System.out.println(string.indexOf("al", 5)); // -1
```

6. contains():

returns true or false after matching the sequence of char value.

Example:

System.out.println("abc".contains("b")); // true

7. startsWith() and endsWith():

Example:

System.out.println("abc".startsWith("a")); // true

System.out.println("abc".startsWith("A")); // false

System.out.println("abc".endsWith("c")); // true

System.out.println("abc".endsWith("a")); // false

8. replace():

The replace() method does a simple search and replace on the string

Example:

System.out.println("abcabc".replace('a', 'A')); // AbcAbc

System.out.println("abcabc".replace("a", "A")); // AbcAbc

9. toLowerCase(): returns a string in lowercase.

Example: System.out.println("Abc123".toLowerCase()); // abc123

10. toUpperCase(): returns a string in uppercase.

Example: String string = "animals";

System.out.println(string.toUpperCase()); // ANIMALS

11. trim(): removes whitespace from the beginning and end of a String.

Example:

System.out.println("abc".trim()); // abc

System.out.println("\t a b c\n".trim()); // a b c

Inheritance:

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

Types of Inheritance:

1. Single: When a class inherits another class, it is known as a single inheritance.

2. Multi Level: When there is a chain of inheritance, it is known as multilevel inheritance.

3. Multiple: When one class inherits multiple classes, it is known as multiple inheritance. Multiple and hybrid inheritance is supported through interface only.

4. Hierarchical: One base class is used to create many sub classes.

5. Hybrid: uses more than on type of inheritance.

Member Access with Inheritance / Constructors with inheritance

```
class Student
{
    int sno;
    String sname;
    Student(int sno,String sname)
    {
        this.sno=sno;
        this.sname=sname;
     }
     void display_student()
     {
         System.out.println(sno);
         System.out.println(sname);
```

```
      }
}
class  ITStudent  extends  Student
{
    private int ds,java,co;
    ITStudent(int sno,String sname,int ds,int java,int co)
   {
        super(sno,sname);
        this.ds=ds;
        this.java=java;
        this.co=co;
   }
   void display()
   {
        display_student();
        System.out.println(ds);
        System.out.println(java);
        System.out.println(co);
   }
}
class  Test
{
  public static void main(String args[])
   {
     ITStudent   s1=new  ITStudent(1,"shiva",77,88,65);
     s1.display();
   }
}
```

Super keyword / Method Overriding:

Another (apart from calling base class constructor) use of super keyword is to base class version of the overridden method.

```
class Student
{
    int sno;
    String sname;
    Student(int sno,String sname)
   {
       this.sno=sno;
       this.sname=sname;
   }
  void display()
   {
       System.out.println(sno);
       System.out.println(sname);
   }
}
class  ITStudent   extends   Student
{
  int ds,co,java;
  ITStudent(int sno,String sname,int ds,int co, int java)
  {
    super(sno,sname);
    this.ds=ds;
    this.co=co;
    this.java=java;
```

```
  }
  // overriding display() method which is inherited from base class.
  void display()
  {
     // using super keyword to call base class version of disply().
      super.display();
      System.out.println(ds);
      System.out.println(java);
      System.out.println(co);
  }
}
class  Test
{
  public static void main(String args[])
   {
    ITStudent  s1=new  ITStudent(1,"shiva",77,88,99);
    s1.display();
   }
}
```

Multi Level Inheritance Example:

```
class Animal
{
        void eat()
        {
                System.out.println("eating...");
        }
}
```

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

Abstract Class Example/Abstract Method Example/ Runtime Polymorphism/ Full Polymorphism / Dynamic Method dispatch mechanism/ Late Binding

```
abstract  class   Figure
```

```
{
     double   dim1,dim2;
     public   Figure(int a, int b)
    {
        dim1=a;
        dim2=b;
    }
    abstract double area();
}
class   Rectangle   extends   Figure
{
      Rectangle(int x, int y)
      {
         super(x,y);
      }
      double area()
      {
          System.out.println("from rectangle class..");
           return dim1*dim2;
      }
}
Class  Triangle  extends  Figure
{
     Triangle(int x, int y)
     {
         super(x,y);
      }
     double  area()
```

```
    {
        System.out.println("from triangle class..");
        return (dim1*dim2)/2;
      }
}
class  Test
{
    public static void main(String args[])
    {
        Figure ref1;
        double a;
        ref1=new Rectangle(10,4);
        a=ref1.area();
        System.out.println("AREA="+a);
        ref1=new Triangle(10,4);
        a=ref1.area();
        System.out.println("AREA="+a);
     }
}
```

final keyword

final with a variable:

The value of the variable is fixed.

Example:

```
final  int  x=10; // x value is 10 fixed, cant change
```

final methods:

```
Class  Test
{
```

```
    final   float   sum(int x, int y)   {
        System.out.println(x+y);
   }
}
```

The method sum() cant be overridden in sub classes if any for the class ‘Test’.

final class:

```
final class Test {
// attributes
// methods
}
```

As the class ‘Test’ is final, we can’t create derived classes for this class.

Forms of Inheritance:

The inheritance concept used for the number of purposes in the java programming language. One of the main purposes is substitutability. The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object. For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.

The substitutability can achieve using inheritance, whether using extends or implements keywords.

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

Specification

This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.

Construction

This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

Eextension

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

Combination

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

Benefits of Inheritance

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

Object Class:

The Object class is the parent class of all the classes in java by default.

Methods of Object Class:

hashCode(): Returns the hashcode number for this object.

equals(Object obj): Compares the given object to this object.

clone(): creates and returns the exact copy (clone) of this object.

toString(): eturns the string representation of this object.

notify(): wakes up single thread, waiting on this object's monitor.

wait(): causes the current thread to wait until another thread notifies (invokes notify() or notifyAll() method).

finalize(): is invoked by the garbage collector before object is being garbage collected.