UNIT – 2

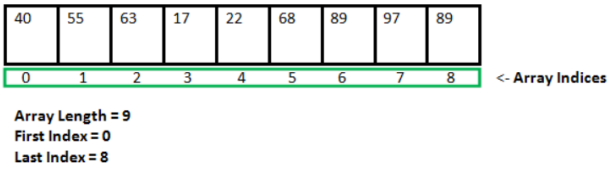
CLASSES & OBJECTS INHERITANCE

Arrays

Java 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. Following are some important points about Java arrays.

- ✓ In Java, all arrays are dynamically allocated.
- ✓ Arrays may be stored in contiguous memory [consecutive memory locations].
- ✓ Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using size of.
- ✓ A Java array variable can also be declared like other variables with [] after the data type.
- ✓ The values in the array are ordered, and each has an index beginning with 0.
- ✓ Java array can also be used as a static field, a local variable, or a method parameter.
- ✓ This storage of arrays helps us randomly access the elements of an array [Support Random Access]



Advantages

- ✓ Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
- ✓ Random access: We can get any data located at an index position.

Disadvantages

- ✓ Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.
- ✓ Java cannot store heterogeneous data. It can only store a single type of primitives

Types of Array in java

Single Dimensional Array

- An array ehich has only one dimension. Also known as a linear array, the elements are stored in a single row.

Creating, Initializing, and Accessing an Arrays

Syntax to Declare an Array in Java

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

Ex : int arr[];

Instantiation of an Array in Java

datatype array_name[]=new datatype[size];

Here, type specifies the type of data being allocated, size determines the number of elements in the array, and var-name is the name of the array variable that is linked to the

array. To use new to allocate an array, you must specify the type and number of elements to allocate.

Ex : int arr[]=new int[10];

Assigning values to array or Array literal in java

int arr[]={10,20,30,40,50}

int arr[] = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

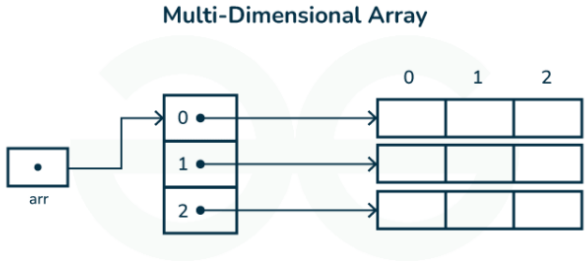
Note: The elements in the array allocated by new will automatically be initialized to zero.

```
class SingleDimension
{
        public static void main(String args[])
        {
                int a[]={10,20,30,40,50};
                int b[]=new int[]{60,70,80};
                for (int i = 0; i < a.length; i++)
                        System.out.print(a[i]+" ");
                System.out.println();
                for (int j = 0; j < b.length; j++)
                        System.out.print(b[j]+" ");
        }
}
```

```
E:\JAVA>javac SingleDimension.java

E:\JAVA>java SingleDimension
10 20 30 40 50
60 70 80
E:\JAVA>
```

Multidimensional Array

- In this 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[][];

Ex : int arr[][];

Instantiation of an Array in Java

datatype array_name[]=new datatype[row][col];

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

Assigning values to array or Array literal in java

int arr[][] = { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };

int arr[][] = new int[][]{{1,2,3},{4,5,6}};

```
class MultiDimension
{
	public static void main(String args[])
	{
		int a[][] = {{2,7,9}, {3,6,1}, {7,4,2}};
		int b[][] = new int[][]{{1,2},{4,5}};

		for (int i = 0; i < a.length; i++)
		{
			for (int j = 0; j < a.length; j++)
				System.out.print(a[i][j]+" ");
			System.out.println();
		}
		System.out.println();
		for (int i = 0; i < b.length; i++)
		{
			for (int j = 0; j < b.length; j++)
				System.out.print(b[i][j]+" ");
			System.out.println();
		}

	}
}
```

```
E:\JAVA>javac MultiDimension.java

E:\JAVA>java MultiDimension
2 7 9
3 6 1
7 4 2

1 2
4 5
```

Jagged Array

- A jagged array is an array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D array but with a variable number of columns in each row. These types of arrays are also known as Jagged arrays.

Syntax: data_type array_name[][] = new data_type[n][]; //n: no. of rows

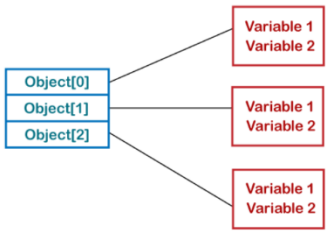
Ex : int a[][]=new int[][]={{2,7,9},{2,5,1,3,2},{4,9,2}}

```
class Jagged
{
	public static void main(String args[])
	{
		int a[][]=new int[][]{{2,7,9}, {3,6,1}, {7,4,2}, {2,5,1,3,2}, {4,9,2}};
		for (int i = 0; i < 5; i++)
		{
			for (int j = 0; j < a[i].length; j++)
				System.out.print(a[i][j]+" ");
			System.out.println();
		}
		System.out.println();
	}
}
```

```
E:\JAVA>javac Jagged.java

E:\JAVA>java Jagged
2 7 9
3 6 1
7 4 2
2 5 1 3 2
4 9 2
```

Array of Objects:

Java is an object-oriented programming language. Most of the work done with the help of objects. We know that an array is a collection of the same data type that dynamically creates objects and can have elements of primitive types. Java allows us to store objects in an array. In Java, the class is also a user-defined data type. An array that conations class type elements are known as an array of objects.



Creating an Array of Objects

Before creating an array of objects, we must create an instance of the class by using the new keyword. We can use any of the following statements to create an array of objects.

Syntax :

ClassName[] objarr;

ClassName objarr[];

Ex:

Student[] arrd;

Student arrd[];

declare and instantiate an array of objects:

ClassName objarr[]=new ClassName[size];

- For example, if you have a class Student, and we want to declare and instantiate an array of Demo objects with five objects/object references then it will be written as:

Ex: Student arrd[]=new Student[5];

- And once an array of objects is instantiated like this, then the individual elements of the array of objects needs to be created using the new keyword.
- Once the array of objects is instantiated, we need to initialize it with values. We cannot initialize the array in the way we initialize with primitive types as it is different from an array of primitive types. In an array of objects, we have to initialize each element of array i.e. each object/object reference needs to be initialized.

  Example program:

```
class Student
{
        public int id;
        public String name;
        Student(int pin, String names)
        {
                id = pin;
                name = names;
        }
        public static void main(String args[])
        {
                Student[] a = new Student[3];
                a[0] = new Student(1, "aaa");
                a[1] = new Student(2, "bbb");
                a[2] = new Student(3, "ccc");
                for(int i=0;i<a.length;i++)
                {
                        System.out.printf("student at %d is %d %s\n",i,a[i].id,a[i].name);
                }
        }
}
```

```
E:\JAVA>javac Student.java

E:\JAVA>java Student
student at 0 is 1 aaa
student at 1 is 2 bbb
student at 2 is 3 ccc
```

Class and Objects

Classes

In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities. The class represents a group of objects having similar properties and behavior. For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

- Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- Class does not occupy memory.
- Class is a group of variables of different data types and a group of methods.
- A Class in Java can contain:
  - ✓ Data member
  - ✓ Method
  - ✓ Constructor
  - ✓ Nested Class
  - ✓ Interface

Components of Java Classes

In general, class declarations can include these components, in order:

- ✓ Modifiers: A class can be public or has default access (Refer this for details).
- ✓ Class keyword: class keyword is used to create a class.
- ✓ Class name: The name should begin with an initial letter (capitalized by convention).
- ✓ Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- ✓ Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- ✓ Body: The class body is surrounded by braces, { }.

Syntax for class declaration

```
access_modifier class <class_name>
{
    data member;
    method;
    constructor;
    nested class;
    interface;
}
```

program

```
class Stu
{
    int id;
    String name;
    public static void main(String args[])
    {
        Stu s = new Stu();
        System.out.println(s.id);
        System.out.println(s.name);
    }
}
```

```
E:\JAVA>javac Stu.java

E:\JAVA>java Stu
0
null
```

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

There are various types of classes that are used in real-time applications such as nested classes, anonymous classes, and lambda expressions.

Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

- ✓ State: It is represented by attributes of an object. It also reflects the properties of an object.
- ✓ Behavior: It is represented by the methods of an object. It also reflects the response of an object with other objects.
- ✓ Identity: It gives a unique name to an object and enables one object to interact with other objects.

Syntax: ClassName obj_name = new ClassName()

Ex: Student s=new Student()

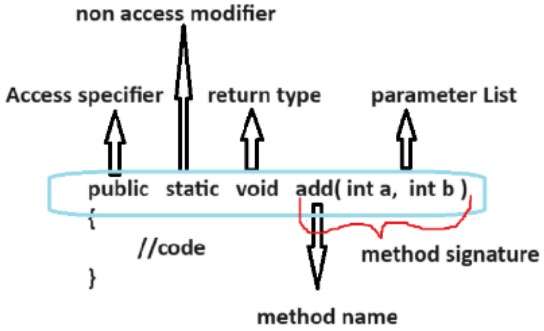
```
class Stu
{
    void msg()
    {
        System.out.println("this is method");
    }
    public static void main(String args[])
    {
        Stu s = new Stu();
        s.msg();
    }
}
```

```
E:\JAVA>javac Stu.java

E:\JAVA>java Stu
this is method
```

Methods in java

- A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code.

Method declaration



- Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.
- Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:
    - Public: The method is accessible by all classes when we use public specifier in our application.
    - Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.
    - Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
    - Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.
- Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.
- Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a verb and start with a lowercase letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in uppercase except the first word.

Ex : sum(), area(), areaOfCircle(), stringSlicing()….

Types of Method

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), nextInt(), sqrt(), etc.

```
class PreMethods
{
        public static void main(String[] args)
        {
                System.out.print("maximum number : " + Math.max(10,7));
        }
}
```

```
E:\JAVA\ANU>javac PreMethods.java

E:\JAVA\ANU>java PreMethods
maximum number : 10
```

User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

There are two ways to create a method in Java:

- **Instance method**

These methods are created inside the class outside the main method. The method of the class is known as an instance method. It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Access the instance data using the object name.

Syntax:

```
// Instance Method
void method_name(){
  body // instance area
}
```

- **Static Method**

These methods are created inside the class outside the main method by using static keyword. A method with static is known as static method. The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is invoked by using the class name or by direct method call. The best example of a static method is the main() method.

Program on methods

```
class UserMethods
{
        static void sum(int a,int b)
        {
                System.out.println("Static method Addition "+(a+b));
        }
        void add(int a,int b)
        {
                System.out.println("Instance method Addition "+(a+b));
        }
        public static void main(String[] args)
        {
                System.out.println("**** USER DEFINED METHODS ****");
                UserMethods.sum(10,5);
                UserMethods um=new UserMethods();
                um.add(20,30);
        }
}
```

```
E:\JAVA\ANU>javac UserMethods.java

E:\JAVA\ANU>java UserMethods
**** USER DEFINED METHODS ****
Static method Addition 15
Instance method Addition 50
```

Java program on static fields and Methods

- **Static** variables and methods belong to a class and are called with the Class Name rather than using object variables, like ClassName.methodName();
- There is only one copy of a static variable or method for the whole class. For example, the main method is static because there should only be 1 main method.
- Static methods only have access to other static variables and static methods. Static methods cannot access or change the values of instance variables or the this reference (since there is no calling object for them), and static methods cannot call non-static methods. However, non-static methods have access to all variables (instance or static) and methods (static or non-static) in the class.

```
class StaticFields
{
        static int a=100;
        static void sum(int a,int b)
        {
                System.out.println("Static method Addition "+(a+b));
        }
        public static void main(String[] args)
        {
                StaticFields.sum(10,5);
                System.out.println("Static variable "+StaticFields.a);
        }
}
```

```
E:\JAVA\ANU>javac StaticFields.java

E:\JAVA\ANU>java StaticFields
Static method Addition 15
Static variable 100
```

Constructors in Java

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

- ✓ Constructor name must be the same as its class name
- ✓ A Constructor must have no return type
- ✓ A Java constructor cannot be abstract, static and final.

Types of Java constructors

There are Three types of constructors in Java:

- ✓ Default constructor
- ✓ No-arg constructor
- ✓ Parameterized constructor

Default constructor

If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code. This constructor is known as default constructor. We would not find it in your source code (the java file) as it would be inserted into the code during compilation and exists in .class file.

Lets understand this using an example program.

```
class DefaultCon
{
        public static void main(String[] args)
        {
                DefaultCon dc = new DefaultCon();
        }
}
```

After compilation

```
class DefaultCon
{
        DefaultCon()
        {
        }
        public static void main(String[] args)
        {
                DefaultCon dc = new DefaultCon();
        }
}
```

no-arg constructor

Constructor with no arguments is known as no-arg constructor. The signature is same as default constructor; however, body can have any code unlike default constructor where the body of the constructor is empty.

```
class NoArgCon
{
        NoArgCon()
        {
                System.out.println("No-Arg constructor");
        }
        public static void main(String[] args)
        {
                NoArgCon nac = new NoArgCon();
        }
}
```

```
E:\JAVA\ANU>javac NoArgCon.java

E:\JAVA\ANU>java NoArgCon
No-Arg constructor
```

Parameterized constructor

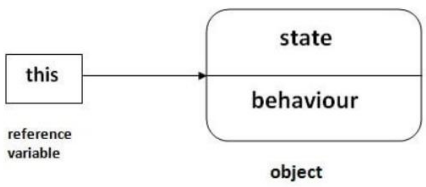
A constructor with arguments (or you can say parameters) is known as a Parameterized constructor. During initialization of an object, which constructor should get invoked depends upon the parameters one passes.

```
class ParaCon
{
        ParaCon(float a,float b)
        {
                System.out.println("Multiplication "+(a+b));
        }
        public static void main(String[] args)
        {
                System.out.println("Parameterized constructor");
                ParaCon pc = new ParaCon(4.2f,4.2f);
        }
}
```

```
E:\JAVA\ANU>javac ParaCon.java

E:\JAVA\ANU>java ParaCon
Parameterized constructor
Multiplication 8.4
```

this keyword

In Java, this is a reference variable that refers to the current class object.



Usage of this keyword

Here is given the 6 usage of java this keyword.

- ✓ this can be used to refer current class instance variable.
- ✓ this can be used to invoke current class method (implicitly)
- ✓ this() can be used to invoke current class constructor.
- ✓ this can be passed as an argument in the method call.
- ✓ this can be passed as argument in the constructor call.
- ✓ this can be used to return the current class instance from the method.

this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
class VarThis
{
        int a;
        VarThis(int a)
        {
                this.a=a+1;
                System.out.println("instance a ="+this.a);
                System.out.println("local a ="+a);
        }
        public static void main(String[] args)
        {
                VarThis t=new VarThis(10);


        }
}
```

```
E:\JAVA\ANU>javac VarThis.java

E:\JAVA\ANU>java VarThis
instance a =11
local a =10
```

this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example.

```
class MethodThis
{
        void msg()
        {
                System.out.println("msg method");
        }
        void call()
        {
                this.msg();
                System.out.println("call method");
        }
        public static void main(String[] args)
        {
                MethodThis t=new MethodThis();
                t.call();
        }
}
```

```
E:\JAVA\ANU>javac MethodThis.java

E:\JAVA\ANU>java MethodThis
msg method
call method
```

this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
class ConThis
{
        ConThis()
        {
                this(10,20);
                System.out.println("no-arg Constructor");
        }
        ConThis(int a,int b)
        {
                System.out.println("parameterized Constructor");
        }
        public static void main(String[] args)
        {
                ConThis t=new ConThis();
        }
}
```

```
E:\JAVA\ANU>javac ConThis.java

E:\JAVA\ANU>java ConThis
parameterized Constructor
no-arg Constructor
```

Constructor overloading in Java

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

```
class ConOverload
{
        ConOverload()
        {
                int a=1,b=2;
                System.out.println("no parameter Addition "+(a+b));
        }
        ConOverload(int x)
        {
                System.out.println("one parameter Integer Addition "+(x+x));
        }
        ConOverload(float x)
        {
                System.out.println("one parameter Float Addition "+(x+x));
        }

        ConOverload(int x, int y)
        {
                System.out.println("two parameters Addition "+(x+y));
        }
        public static void main(String args[])
        {
                ConOverload c1=new ConOverload();
                ConOverload c2=new ConOverload(5);
                ConOverload c3=new ConOverload(3.4f);
                ConOverload c4=new ConOverload(10,20);
        }
}
```

output:

```
E:\JAVA\ANU>javac ConOverload.java

E:\JAVA\ANU>java ConOverload
no parameter Addition 3
one parameter Integer Addition 10
one parameter Float Addition 6.8
two parameters Addition 30
```

INHERITANCE

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Need of Inheritance

- Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.

- Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.

Terminology

- **Class**: Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class/Base Class**: The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class/Child Class/Derived Class**: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

extends keyword

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Syntax :

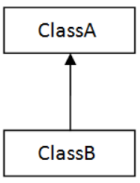
```
class DerivedClass extends BaseClass
{
   //methods and fields
}
```

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.



1) Single

Program:

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

        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Barking");

        }
        public static void main(String[] args)
        {
                Dog d = new Dog();
                d.eat();
                d.bark();
        }
}
```

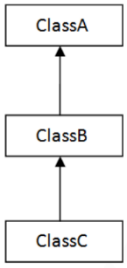
```
E:\JAVA\ANU>javac Dog.java

E:\JAVA\ANU>java Dog
Eating
Barking
```

Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

Program:

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

        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Barking");

        }
}
class BabyDog extends Dog
{
        void sleep()
        {
                System.out.println("Sleeping");

        }
        public static void main(String[] args)
        {
                BabyDog b = new BabyDog();
                b.eat();
                b.bark();
                b.sleep();
        }
}
```



2) Multilevel

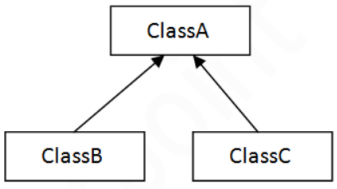
```
E:\JAVA\ANU>javac BabyDog.java

E:\JAVA\ANU>java BabyDog
Eating
Barking
Sleeping
```

Hierarchical Inheritance

When two or more classes inherits a single class, or one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

Program:

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

        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Barking");

        }
}
class Cat extends Animal
{
        void meow()
        {
                System.out.println("Meowing");

        }
        public static void main(String[] args)
        {
                Cat c = new Cat();
                c.eat();
                //c.bark();  ERROR
                c.meow();
        }
}
```



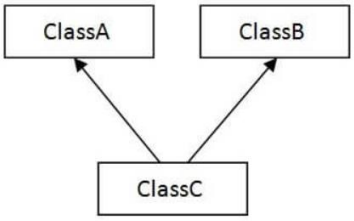
3) Hierarchical

```
E:\JAVA\ANU>javac Cat.java

E:\JAVA\ANU>java Cat
Eating
Meowing
```

Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces.

```
class A
{
        void msg1()
        {
                System.out.println("Hello");
        }
}
class B
{
        void msg2()
        {
                System.out.println("Welcome");
        }
}
class C extends A,B //suppose if it were
{

        public static void main(String args[])
        {
                C obj=new C();
                obj.msg1();
                obj.msg2();
        }
}
```

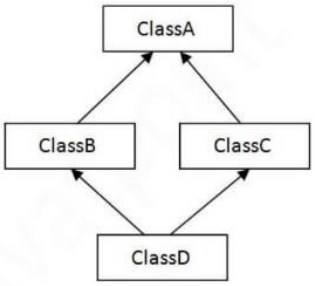


4) Multiple

```
Compile Time Error
```

Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces.



5) Hybrid

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Method overloading in Java is also known as Compile-time Polymorphism, or Static Polymorphism or Early binding.
- It is similar to constructor ovaloading in java (that allows a class having More than one constructor having different parameter List.

Different ways to overload the method

There are two ways to overload the method in java

- ✓ By changing number of arguments
  - int add(int a,int b)
  - int add(int a,int b,int c)
- ✓ By changing the data type
  - void add(int a, float b)

Program:

```
class OverLoad
{
        void add(int a,int b)
        {
                System.out.println(a+b);
        }
        void add(float a,float b)
        {
                System.out.println(a+b);
        }
        void add(int a,int b,int c)
        {
                System.out.println(a+b+c);
        }
        public static void main(String args[])
        {
                OverLoad ol=new OverLoad();
                ol.add(10,20);
                ol.add(2.3f,4.4f);
                ol.add(4,2,6);
        }
}
```

```
E:\JAVA\ANU>javac OverLoad.java

E:\JAVA\ANU>java OverLoad
30
6.7
12
```

Method Overriding

- Method Overriding is achieved through Inheritance.
- In Java, Overriding is a feature that allows a subclass is having a method which is already present in its super-classes or parent classes.
- Declaring a method in subclass which is already present in parent class is known as Method overiding .
- Overriding is done so that A Child class can give its own implementation to a method which is already provided by parent class. In this case the method in parent class is called Overridden method and the method in child class is called overriding Method.
- Method Overriding in java is also known as Runtime Polymorphism or Dynamic Polymorphism or Late Binding.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Rules:-

- Method must have same name in the Parent class.
- Method must have same parameters as in the parent class method.
- The return type must be same as the Return type declared in original Overridden method in the super class
- We cannot override a method marked with final and you cannot override a method marked with Static.

Program:

```
class Parent
{
        void msg()
        {
                System.out.println("parent Overridden method");
        }
}
class OverRide extends Parent
{
        void msg()
        {
                System.out.println("child Overriding method ");
        }
        public static void main(String args[])
        {
                OverRide or=new OverRide();
                or.msg();
        }
}
```

```
E:\JAVA\ANU>javac OverRide.java

E:\JAVA\ANU>java OverRide
child Overriding method
```

Difference Between Compile-time polymorphism and Runtime polymorphism

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading helps to increase the readability of the program.	Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
The argument list should be different while doing method overloading.	The argument list should be the same in method overriding.

Super Keyword in Java

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

- **super can be used to refer immediate parent class instance variable**

```
class Parent
{
        int a=10;
}
class Child extends Parent
{
        int b=20;
        void msg()
        {
                System.out.println("Child variable "+b);
                System.out.println("Parent variable "+super.a);
        }
        public static void main(String args[])
        {
                Child c=new Child();
                c.msg();
        }
}
```

```
E:\JAVA\ANU>javac Child.java

E:\JAVA\ANU>java Child
Child variable 20
Parent variable 10
```

- **super can be used to invoke immediate parent class method**

When you override a method in a subclass, you may still want to call the original method from the superclass to maintain some of its behavior. You can achieve this using the super keyword. This is particularly useful if the superclass method has some useful functionality that you want to extend or modify in the subclass.

```
class Parent
{
        void msg()
        {
                System.out.println("Parent Method");
        }
}
class SuperMeth extends Parent
{
        void msg()
        {
                super.msg();
                System.out.println("Child Method");
        }
        public static void main(String args[])
        {
                SuperMeth sm=new SuperMeth();
                sm.msg();
        }
}
```

```
E:\JAVA\ANU>javac SuperMeth.java

E:\JAVA\ANU>java SuperMeth
Parent Method
Child Method
```

- **super() can be used to invoke immediate parent class constructor**

```
class A
{
        A()
        {
                System.out.println("Parent Constructor");
        }
}
class B extends A
{
        B()
        {
                super();
                System.out.println("Child Constructor");
        }
        public static void main(String args[])
        {
                B b=new B();
        }
}
```

```
E:\JAVA\ANU>javac B.java

E:\JAVA\ANU>java B
Parent Constructor
Child Constructor
```

Final keyword

The final keyword in java is used to restrict the user. It is used to indicate that a variable, method, or class cannot be modified or extended. The java final keyword can be used in many context. Final can be used with:

- ✓ final variables – to create constant variables
- ✓ final methods – to prevent Method Overriding
- ✓ final classes – to prevent Inheritance

➢ **Final with variables**: When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.

```
class FinalVar
{
        public static void main(String args[])
        {
                final float pi=3.14f;
                //pi=3.146798f; ERROR
                System.out.println("final variable "+pi);
        }
}
```

```
E:\JAVA\ANU>javac FinalVar.java

E:\JAVA\ANU>java FinalVar
final variable 3.14
```

➢ **Final with methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class's public API and should not be modified by subclasses.

```
class Parent
{
       final void msg()
       {
              System.out.println("parent");
       }
}
class FinalMeth extends Parent
{
       void msg()
       {
              System.out.println("child");
       }
       public static void main(String args[])
       {
              FinalMeth fm=new FinalMeth();
              fm.msg();
       }
}
```

```
E:\JAVA\ANU>javac FinalMeth.java
FinalMeth.java:10: error: msg() in FinalMeth cannot override msg()
 in Parent
       void msg()
            ^
  overridden method is final
1 error
```

➢ **Final with classes**: When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.

```
final class Parent
{
       int a=10;
}
class FinalClass extends Parent
{
       public static void main(String args[])
       {
              FinalClass c=new FinalClass();
              System.out.println(c.a);
       }
}
```

```
E:\JAVA\ANU>javac FinalClass.java
FinalClass.java:5: error: cannot inherit from final Parent
class FinalClass extends Parent
                         ^
1 error
```

Object class

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.

Methods of Object class

- Clone()

Object cloning refers to the creation of an exact copy of an object. It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

- finalize()
  This method is called just before an object is garbage collected. It is called the Garbage Collector on an object when the garbage collector determines that there are no more references to the object.
- toString()
  The toString() provides a String representation of an object and is used to convert an object to a String.
- getClass() - It returns the class object of “this” object
- Equals() - compares the given object to this object.
- notify() - wakes up single thread, waiting on this object's monitor.
- notifyAll() - wakes up all the threads, waiting on this object's monitor.
- wait() - causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies

Polymorphism

- Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java
  - ✓ **compile-time polymorphism (or) Static Binding (or) early binding (or) method overloading**
  - ✓ **runtime polymorphism (or) Dynamic Binding (or) late binding (or) method overriding**
- We can perform polymorphism in java by method overloading and method overriding.
  (**for method overloading and overriding - refer previous topics**)

Abstract clases

Before learning the Java abstract class, let's understand the abstraction in Java first.

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

- ✓ Abstract class (0 to 100%)
- ✓ Interface (100%)

abstract class

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- Points to Remember
  1. An abstract class must be declared with an abstract keyword.
  2. It can have abstract and non-abstract methods.
  3. We can have an abstract class without any abstract method.

4. If a class contains at least one abstract method then compulsory should declare a class as abstract
5. It cannot be instantiated (or) An instance of an abstract class can not be created.
6. An abstract class can have data members, abstract methods, non-abstract methods, constructors and main() method.
7. There can be a final method in abstract class but any abstract method in abstract class can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final".
8. If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method

Ex:

```
abstract class A
{
        //abstract methods
        //Non abstract methods
}
```

abstract Method

- A method declared using the abstract keyword within an abstract class and does not have a definition (implementation) is called an abstract method.
- When we need just the method declaration in a super class, it can be achieved by declaring the methods as abstracts.
- Abstract method is also called subclass responsibility as it doesn't have the implementation in the super class. Therefore a subclass must override it to provide the method definition.

Syntax

```
abstract return_type method_name( [ argument-list ] );
```

- Points to Remember
    1. An abstract method do not have a body (implementation), they just have a method signature (declaration). The class which extends the abstract class implements the abstract methods.
    2. If a non-abstract (concrete) class extends an abstract class, then the class must implement all the abstract methods of that abstract class. If not the concrete class has to be declared as abstract as well.
    3. As the abstract methods just have the signature, it needs to have semicolon (;) at the end.

Program

```
abstract class Animal
{
        abstract void sound();
}
class Dogs extends Animal
{
        void sound()
        {
                System.out.println("Barking");
        }
        public static void main(String args[])
        {
                Dogs d=new Dogs();
                d.sound();
        }
}
```

```
E:\JAVA\ANU>javac Dogs.java

E:\JAVA\ANU>java Dogs
Barking
```

String class

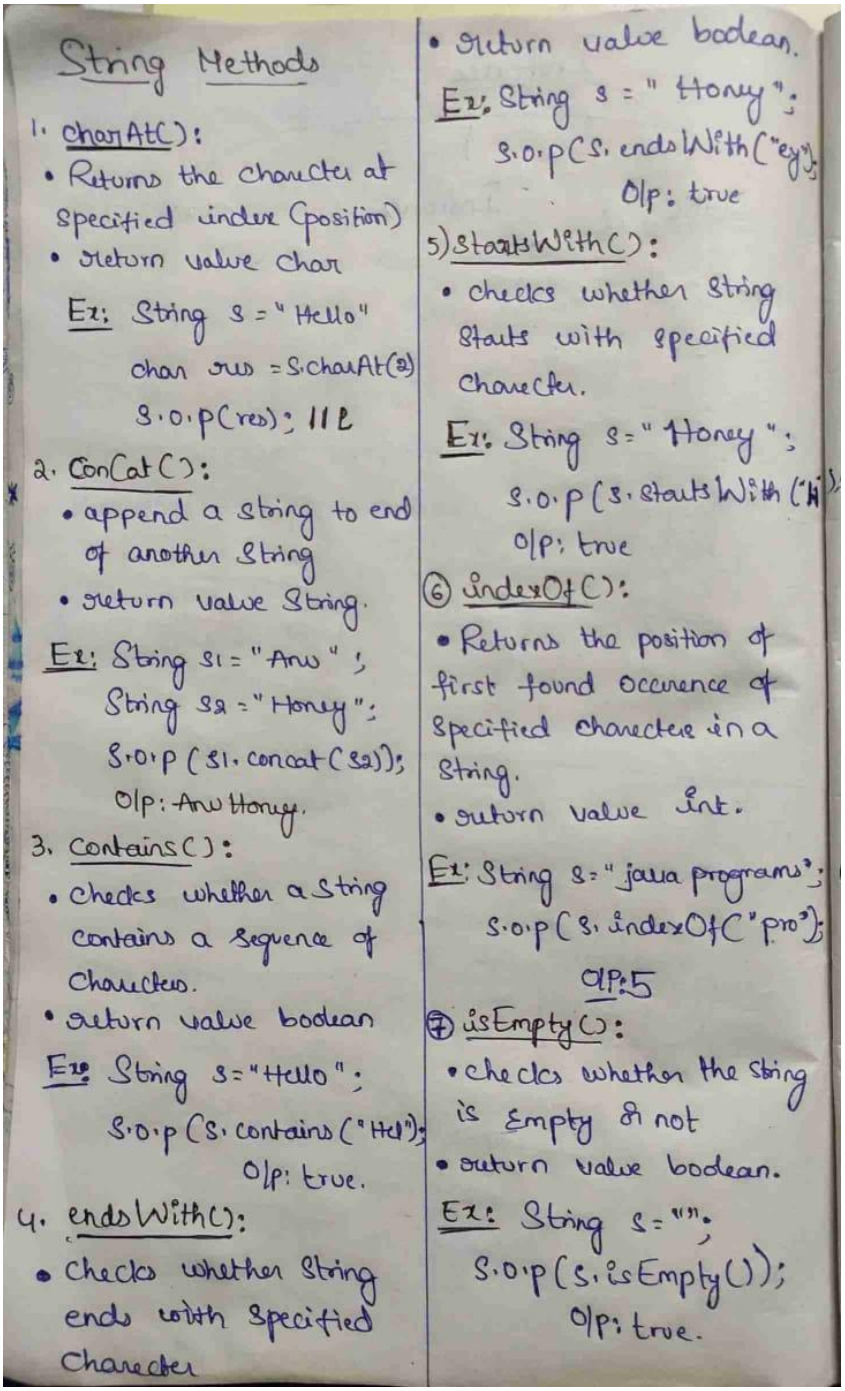
- In the Java programming language, strings are objects.
- The Java platform provides the String class to create and manipulate strings.
- String is basically an object that represents sequence of characters. An array of characters works same as Java string.
- We can create Strings in 2 ways:

```
String s = "Hello";
String str = new String("Hello");
```

- We can also create String by using character Array

```
char[] arr = { 'h', 'e', 'l', 'l', 'o', '.' };
String str = new String(arr);
System.out.println(str);
```

String class methods



String Methods

1. charAt():
- Returns the character at specified index (position)
- return value char

Ex: String s = "Hello"
char res = s.charAt(2)
s.o.p(res); llL

2. ConCat():
- append a string to end of another String
- return value String.

Ex: String s1 = "Anu";
String s2 = "Honey";
s.o.p(s1.concat(s2));
o/p: Anu Honey.

3. contains():
- Checks whether a string contains a sequence of characters.
- return value boolean

Ex: String s = "Hello";
s.o.p(s.contains("Hel"));
o/p: true.

4. endsWith():
- Checks whether String ends with specified character
- return value boolean.

Ex: String s = "Honey";
s.o.p(s.endsWith("ey"));
o/p: true

5) startsWith():
- checks whether String starts with specified character.

Ex: String s = "Honey";
s.o.p(s.startsWith("H"));
o/p: true

6) indexOf():
- Returns the position of first found occurence of specified characters in a String.
- return value int.

Ex: String s = "java programs";
s.o.p(s.indexOf("pro"));
o/p: 5

7) isEmpty():
- checks whether the string is Empty or not
- return value boolean.

Ex: String s = "";
s.o.p(s.isEmpty());
o/p: true.

⑧ join():

- joins one & more Strings with a specified separator.
- returns String.

Ex:

```
String str = String.join("-",
  "Apple", "Ball", "cat");
S.O.P(str);
```

O/P: Apple-Ball-cat.

⑨ length():

- Returns length of specified String.
- return value int

Eg: String s = "programmer";

```
S.O.P(S.length());
```

O/P: 10

⑩ split():

- splits a String into array of subStrings
- returns String.

Eg:

```
String s = "this is split fun";
String arr[] = s.split(" ");
for (String i: arr)
    S.O.P(s)
```

O/P: this
is
split
fun

String COMPARISION

- ==
- equals()
- compareTo()

• (==)

```
String s = "Hel";
S.O.P(s == "Hel") //true
```

• equals(): (case sensitive)

- compares 2 strings returns
  - → true if equal
  - → false if not-equal

Eg: String s = "Hi";

```
S.O.P(S.equals("Hi");
                //true
```

• equalsIgnoreCase():
case Insensitive.

• CompareTo():

- compares 2 strings lexicographically.

It returns:

- 0 → if Both S1 == S2
- +ve → if S1 > S2
- -ve → if S2 < S2

Eg:

```
String s1 = "Hella";
String s2 = "Hello";
S.O.P(S1.compareTo(s2));
```

O/P: -14

```
S.O.P(S2.compareTo(S1));
```

O/P: 14

• CompareToIgnoreCase():
Same but case Insensitive

Java program on searching a string in an Array of Strings

```
class StringSearch
{
        public static void main(String args[])
        {
                String s[]={"e","u","a","o","i"};
                String key="o";
                boolean found=false;
                for(int i=0;i<s.length;i++)
                {
                        if(key.equals(s[i]))
                        {
                                found=true;
                                break;
                        }
                }
                if(found)
                        System.out.println("FOUND");
                else
                        System.out.println("NOT FOUND");
        }
}
```

```
E:\JAVA\ANU>javac StringSearch.java

E:\JAVA\ANU>java StringSearch
FOUND
```

Java Program on String Sorting

```
import java.util.Arrays;
class StringSort
{
        public static void main(String args[])
        {
                String s[]={"e","u","a","o","i"};
                System.out.println("Before Sorting "+Arrays.toString(s));
                Arrays.sort(s);
                System.out.print("After Sorting   ");
                boolean found=false;
                for(String i: s)
                {
                        System.out.print(i+" ");
                }

        }
}
```

```
E:\JAVA\ANU>javac StringSort.java

E:\JAVA\ANU>java StringSort
Before Sorting [e, u, a, o, i]
After Sorting   a e i o u
```