**MODULE-2**

**Chapter – 1**

**LECTURE-1**

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts −

* Polymorphism
* Inheritance
* Encapsulation
* Abstraction
* Classes
* Objects
* Instance
* Method
* Message Parsing

**Classes and Objects:**

* Object: Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
* Class: A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

**i. Objects in Java:**

* Software objects have a state and a behaviour. A software object's state is stored in fields and behaviour is shown via methods. So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.
* An entity that has state and behaviour is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
* An object has three characteristics:
  + **State:** represents the data (value) of an object.
  + **Behaviour:** represents the behaviour (functionality) of an object such as deposit, withdraw, etc.
  + **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
* For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behaviour.
* **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
* **Object Definitions:**
  + An object is a real-world entity.
  + An object is a runtime entity.
  + The object is an entity which has state and behaviour.
  + The object is an instance of a class.

**ii. Classes in Java:**

* A class is a blueprint from which individual objects are created.
* A class can contain any of the following variable types.
  + Local variables: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
  + Instance variables: Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
  + Class variables: Class variables are variables declared within a class, outside any method, with the static keyword.
* A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
* A class in Java can contain:
  + **Fields (attributes)**
  + **Methods**
  + **Constructors**
  + **Blocks**
  + **Nested class and interface**
* **Syntax to declare a class:**

**class** <class\_name>{

    field;

    method;

}

**Instance variable in Java**

* A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

**Method in Java**

* In Java, a method is like a function which is used to expose the behavior of an object.
* Advantage of Method
  + Code Reusability
  + Code Optimization

**“new” keyword in Java**

* The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

**i. Object and Class Example: main within the class**

* In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.
* Here, we are creating a main() method inside the class.

***File: Student.java***

//Java Program to illustrate how to define a class and fields

//Defining a Student class.

class Student{

 int id; //field or data member or instance variable

 String name;   //creating main method inside the Student class

  public static void main(String args[]){

  Student s1=new Student();//creating an object of Student

  System.out.println(s1.id);//accessing member through reference variable

  System.out.println(s1.name);

 }

}

**Output:**

0

null

**ii. Object and Class Example: main outside the class**

* We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is ideal to save the file name with the class name which has main() method.

**File: TestStudent.java**

/\*Java Program to demonstrate having the main method in another class\*/

**class** Student

{

**int** id;

 String name;

}

//Creating another class TestStudent1 which contains the main method

**class** TestStudent{

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

  Student s1=**new** Student();

  System.out.println(s1.id);

  System.out.println(s1.name);

 }

}

**Output:**

0

Null

**LECTURE-2**

**Initialization of object**

There are 3 ways to initialize object in Java.

* By reference variable
* By method
* By constructor

**1) Object and Class Example: Initialization through reference**

* Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

**File: TestStudent.java**

class Student{

 int id;

 String name;

}

class TestStudent{

 public static void main(String args[]){

  Student s1=new Student();

  s1.id=101;

  s1.name="ABC";

  System.out.println(s1.id+" "+s1.name); //printing members with a white space

 }

}

**Output:** 101 ABC

We can also create multiple objects and store information in it through reference variable.

**File: TestStudent.java**

class Student

{

 int id;

 String name;

}

class TestStudent{

 public static void main(String args[]){

  Student s1=new Student();

  Student s2=new Student();

  s1.id=101;

 s1.name="ABC";

  s2.id=102;

  s2.name="XYZ";

  System.out.println(s1.id+" "+s1.name);

  System.out.println(s2.id+" "+s2.name);

 }

}

**Output:**

101 ABC

102 XYZ

**2) Object and Class Example: Initialization through method**

* Here we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

**File: TestStudent.java**

class Student{

 int rollno;

 String name;

 void insertRecord(int r, String n){

  rollno=r;

  name=n;

 }

 void displayInformation(){System.out.println(rollno+" "+name);}

}

class TestStudent{

 public static void main(String args[]){

  Student s1=new Student();

  Student s2=new Student();

  s1.insertRecord(111,"ABC");

  s2.insertRecord(222,"XYZ");

  s1.displayInformation();

  s2.displayInformation();

 }

}

**Output:**

111 ABC

222 XYZ

**EXAMPLE – 2: Object and Class Example: Rectangle**

**File: TestRectangle.java**

class Rectangle{

 int length;

 int width;

 void insert(int l, int w){    length=l;    width=w;   }

 void calculateArea()

{ System.out.println(“Area of rectangle is:” + length\*width); }

}

class TestRectangle{

 public static void main(String args[]){

  Rectangle r1=new Rectangle();

  Rectangle r2=new Rectangle();

  r1.insert(11,5);

  r2.insert(3,15);

  r1.calculateArea();

  r2.calculateArea();

}

}

**Output:**

Area of rectangle is: 55

Area of rectangle is: 45

**Anonymous object**

* Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.
* If you have to use an object only once, an anonymous object is a good approach.
* For example: **new** Calculation();//anonymous object
* Calling method through a reference:

Calculation c=**new** Calculation();

c.fact(5);

* Calling method through an anonymous object

**new** Calculation().fact(5);

Let's see the full example of an anonymous object in Java.

**class** Calculation{

**void** fact(**int**  n)

{

**int** fact=1;

**for**(**int** i=1;i<=n;i++){     fact=fact\*i;    }

 System.out.println("factorial is "+fact);

}

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

**new** Calculation().fact(5);//calling method with anonymous object

}

}

**Output:** Factorial is 120

**LECTURE-3**

[**Encapsulation in Java**](https://www.geeksforgeeks.org/encapsulation-in-java/)

* Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
* As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.  
  Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

**Example:** // Java program to demonstrate encapsulation

public class Encapsulate {

    // private variables declared and can only be accessed by public methods of class

    private String Name;

    private int Roll;

    private int Age;

     // get method for age to access private variable Age

    public int getAge()

    {         return Age;     }

     // get method for name to access private variable Name

    public String getName()

    {         return Name;     }

      // get method for roll to access private variable Roll

    public int getRoll()

    {         return Roll;     }

      // set method for age to access private variable age

    public void setAge(int newAge)

    {         Age = newAge;     }

      // set method for name to access private variable Name

    public void setName(String newName)

    {         Name = newName;     }

      // set method for roll to access private variable Roll

    public void setRoll(int newRoll)

    {         Roll = newRoll;     }

} // end of class Encapsulate

class TestEncapsulation {

    public static void main(String[] args)

    {

        Encapsulate obj = new Encapsulate();

        obj.setName("ABC");

        obj.setAge(20);

        obj.setRoll(111);

        System.out.println("'name: " + obj.getName());

        System.out.println("'age: " + obj.getAge());

        System.out.println("'roll: " + obj.getRoll());

         // Direct access of Roll is not possible due to encapsulation

        // System.out.println("'s roll: " + obj.Name); // will generate error

    }

}

**Output:**

name: ABC

age: 20

roll: 111

[**Abstraction in Java**](https://www.geeksforgeeks.org/abstraction-in-java-2/)

* Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
* Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

**Example:** // Java program to illustrate the concept of Abstraction

  abstract class Shape {

    String color;

      // these are abstract methods

    abstract double area();

    public abstract String toString();

    // abstract class can have a constructor

    public Shape(String color)

    {

        System.out.println("Shape constructor called");

        this.color = color;

    }

    // this is a concrete method

    public String getColor()

    {        return color;     }

}

class Circle extends Shape {

    double radius;

    public Circle(String color, double radius)

    {

       // calling Shape constructor

        super(color);

        System.out.println("Circle constructor called");

        this.radius = radius;

    }

  // override

    double area()

    {      return Math.PI \* Math.pow(radius, 2);

}

    public String toString()

    {

        return "Circle color is " + super.color  + "and area is : "   + area();

    }

}

 class Rectangle extends Shape {

     double length, width;

     public Rectangle(String color, double length, double width)

    {

         // calling Shape constructor

        super(color);

        System.out.println("Rectangle constructor called");

        this.length = length;

        this.width = width;

    }

  double area()

    {         return length \* width;     }

 public String toString()

    {

        return "Rectangle color is "   + super.color   + "and area is : "  + area();

    }

}

public class Test {

    public static void main(String[] args)

    {

        Shape s1 = new Circle("Red", 2.2);

        Shape s2 = new Rectangle("Yellow", 2, 4);

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

        System.out.println(s2.toString());

    }

}

**Output:**

Shape constructor called

Circle constructor called

Shape constructor called

Rectangle constructor called

Circle color is Redand area is : 15.205308443374602

Rectangle color is Yellowand area is : 8.0

**Difference between Abstraction and Encapsulation:**

| **ABSTRACTION** | **ENCAPSULATION** |
| --- | --- |
| Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |

**LECTURE- 4**

**Method Overloading in Java**

* If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**. Here methods have different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.
* Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int,int,int) for three parameters then it may be difficult to understand the behavior of the method because its name differs. Hence we can use a method having same name with different parameter.

**Advantage of method overloading**

* Method overloading increases the readability of the program.
* Different ways to overload the method

**There are two ways to overload the method in java**

* By changing number of arguments
* By changing the data type

In java, Method Overloading is not possible by changing the return type of the method only.

**1) Method Overloading: changing no. of arguments**

* In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

**class** Adder{

**static** **int** add(**int** a,**int** b){**return** a+b;}

**static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}

}

**class** TestOverloading1{

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

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}

}

**Output:**

22

33

**2) Method Overloading: changing data type of arguments**

// Java program to demonstrate working of method overloading in Java.

  // Overloaded sum(). This sum takes two int parameters

    public int sum(int x, int y)

    {        return (x + y);     }

      // Overloaded sum(). This sum takes three int parameters

    public int sum(int x, int y, int z)

    {        return (x + y + z);     }

      // Overloaded sum(). This sum takes two double parameters

    public double sum(double x, double y)

    {         return (x + y);     }

    public static void main(String args[])

    {

        Sum s = new Sum();

        System.out.println(“addition of two integers: ” + s.sum(10, 20));

        System.out.println(“addition of three integers: ” +s.sum(10, 20, 30));

        System.out.println(“addition of two double values: ” +s.sum(10.5, 20.5));

    }

}

**Output :**

addition of two integers: 30

addition of three integers: 60

addition of two double values: 31.0

**QUESTIONS:**

**Q) Why Method Overloading is not possible by changing the return type of method only?**

**ANS:** In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

class Adder{

static int add(int a,int b){return a+b;}

static double add(int a,int b){return a+b;}

}

class TestOverloading3{

public static void main(String[] args){

System.out.println(Adder.add(11,11));//ambiguity

}

}

**Output:**

Compile Time Error: method add(int,int) is already defined in class Adder

**NOTE:** System.out.println(Adder.add(11,11)); Here, java compiler can not determine which sum() method should be called.

**LECTURE- 5**

**Constructors in Java**

* In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) 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**

The rules defined for the constructor are:

* Constructor name must be the same as its class name
* A Constructor must have no explicit return type
* A Java constructor cannot be abstract, static, final, and synchronized

**Note:** We can use [**access modifiers**](https://www.javatpoint.com/access-modifiers) while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

**Types of Java constructors**

There are two types of constructors in Java:

* Default constructor (no-arg constructor)
* Parameterized constructor

**i. Java Default Constructor:**

* A constructor is called "Default Constructor" when it doesn't have any parameter.

**Syntax**: <class\_name>(){}

**Example:** //Java Program to create and call a default constructor

class defaultcon{

//creating a default constructor

defaultcon (){System.out.println("Default Constructor is created");}

public static void main(String args[]){

defaultcon b=new defaultcon ();  //calling a default constructor

}

}

**Output:** Default Constructor is created

**NOTE:** If there is no constructor in a class, compiler automatically creates a default constructor.

**QUESTION:**

**Q) What is the purpose of a default constructor?**

**ANSWER:** The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

**Example:** Default constructor that displays the default values

class Student {

int id;

String name;

void display (){

System.out.println(id+" "+name);

}

public static void main (String args[]){

Student s1=new Student ();

Student s2=new Student ();

s1.display();

s2.display();

}

}

**Output:**

0 null

0 null

**Explanation:** In the above class, we are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

**ii. Java Parameterized Constructor**

* A constructor which has a specific number of parameters is called a parameterized constructor.
* The parameterized constructor is used to provide different values to distinct objects. However, we can provide the same values also.

**Example:** //Java Program to demonstrate the use of the parameterized constructor.

class Student{

    int id;

    String name;

    //creating a parameterized constructor

    Student (int i, String n){

    id = i;

    name = n;

    }

    //method to display the values

    void display (){ System.out.println (id+" "+name);}

   public static void main (String args[]){

    //creating objects and passing values

    Student s1 = new Student (111,"ABC");

    Student s2 = new Student (222,"MNO");

    //calling method to display the values of object

    s1.display();

    s2.display();

   }

}

**Output:**

111 ABC

222 MNO

**Constructor Overloading in Java**

* In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
* Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.
* Overloaded constructor is called based upon the parameters specified when **“new”** is executed.

**Example of Constructor Overloading**

class Student {

    int id;

    String name;

    int age;

    //creating two arg constructor

    Student (int i, String n){

    id = i;

    name = n;

    }

    //creating three arg constructor

    Student (int i, String n, int a){

    id = i;

    name = n;

    age=a;

    }

    void display(){System.out.println(id+" "+name+" "+age);}

   public static void main(String args[]){

    Student s1 = new Student (111,"ABC");

    Student s2 = new Student (222,"XYZ",25);

    s1.display();

    s2.display();

   }

}

**Output:**

111 ABC 0

222 XYZ 25

**Q) When do we need Constructor Overloading?**

**ANSWER:** Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading.

Example:

class Box

{

    double width, height, depth;

      // constructor used when all dimensions specified

    Box(double w, double h, double d) {

        width = w;

        height = h;

        depth = d;

    }

    // constructor used when no dimensions specified

    Box()

    {         width = height = depth = 0;     }

    // constructor used when cube is created

    Box(double len)

    {        width = height = depth = len;     }

    // compute and return volume

    double volume()

    {        return width \* height \* depth;     }

}

public class Test

{

    public static void main(String args[])

    {

        // create boxes using the various constructors

        Box mybox1 = new Box(10, 20, 15);

        Box mybox2 = new Box();

        Box mycube = new Box(7);

        double vol;

       // get volume of first box

        vol = mybox1.volume();

        System.out.println(" Volume of mybox1 is " + vol);

       // get volume of second box

        vol = mybox2.volume();

        System.out.println(" Volume of mybox2 is " + vol);

       // get volume of cube

        vol = mycube.volume();

        System.out.println(" Volume of mycube is " + vol);

    }

}

**Output:**

Volume of mybox1 is 3000.0

Volume of mybox2 is 0.0

Volume of mycube is 343.0

**Important points to be taken care while doing Constructor Overloading :**

* Constructor calling must be the **first** statement of constructor in Java.
* If we have defined any parameterized constructor, then compiler will not create default constructor. and vice versa if we don’t define any constructor, the compiler creates the default constructor(also known as no-arg constructor) by default during compilation
* Recursive constructor calling is invalid in java.

**Difference between constructor and method in Java**

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| * A constructor is used to initialize the state of an object. | * A method is used to expose the behavior of an object. |
| * A constructor must not have a return type. | * A method must have a return type. |
| * The constructor is invoked implicitly. | * The method is invoked explicitly. |
| * The Java compiler provides a default constructor if you don't have any constructor in a class. | * The method is not provided by the compiler in any case. |
| * The constructor name must be same as the class name. | * The method name may or may not be same as the class name. |

**Java Copy Constructor**

* We can copy the values from one object to another using copy constructor.

**Example:** //Java program to initialize the values from one object to another object.

class Student {

    int id;

    String name;

    //constructor to initialize integer and string

    Student (int i,String n){

    id = i;

    name = n;

     }

    //constructor to initialize another object

    Student (Student s){

    id = s.id;

    name =s.name;

    }

    void display (){System.out.println(id+" "+name);}

    public static void main (String args[ ]){

    Student s1 = new Student (111,"ABC");

    Student s2 = new Student (s1);

    s1.display();

    s2.display();

   }

}

**Output:**

111 ABC

111 ABC

**Q) Does constructor return any value?**

**ANSWER:** Yes, it is the current class instance (You cannot use return type yet it returns a value).

**Q) Can constructor perform other tasks instead of initialization?**

**ANSWER:** Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

**Q) What is the purpose of Constructor class?**

**ANSWER:** Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

**LECTURE- 6**

**JAVA ARRAY OF OBJECT**

* As defined by its name, it stores an **array of objects**. Unlike a traditional array that store values like string, integer, Boolean, etc an array of objects stores objects. The array elements store the location of the reference variables of the object.
* **Syntax:** Class obj[ ]= new Class[array\_length]

**Example: To create Array Of Objects**  
**Step 1)** Copy the following code into an editor

class ObjectArray{

public static void main(String args[]){

Account obj[] = new Account[2] ;

//obj[0] = new Account();

//obj[1] = new Account();

obj[0].setData(1,2);

obj[1].setData(3,4);

System.out.println("For Array Element 0");

obj[0].showData();

System.out.println("For Array Element 1");

obj[1].showData();

}

}

class Account{

int a;

int b;

public void setData(int c, int d){

a=c;

b=d;

}

public void showData(){

System.out.println("Value of a ="+a);

System.out.println("Value of b ="+b);

}

}

**Step 2)** Save , Compile & Run the Code.  
**Step 3)** Error=? Try and debug before proceeding to step 4.  
**Step 4)** The line of code, Account obj[] = new Account[2]; exactly creates an array of two reference variables as shown below

**Step 5)** Uncomment Line # 4 & 5. This step creates objects and assigns them to the reference variable array as shown below. Your code must run now.

**Output:**

For Array Element 0

Value of a =1

Value of b =2

For Array Element 1

Value of a =3

Value of b =4

**Access Modifiers in Java**

* The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

* **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
* **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
* **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
* **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

**NOTE:** There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

**Understanding Java Access Modifiers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| Private | Y | N | N | N |
| Default | Y | Y | N | N |
| Protected | Y | Y | Y | N |
| Public | Y | Y | Y | Y |

**1) Private:** The private access modifier is accessible only within the class.

**Example:** In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

class A{

private int data=40;

private void msg(){System.out.println("Hello java");}

}

public class Simple{

 public static void main(String args[]){

   A obj=new A();

   System.out.println(obj.data);//Compile Time Error

   obj.msg();//Compile Time Error

   }

}

**2) Default:** If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example:** In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

package pack;

class A{

  void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B{

  public static void main(String args[]){

   A obj = new A();//Compile Time Error

   obj.msg();//Compile Time Error

  }

}

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

**3) Protected:**

* The protected access modifier is accessible within package and outside the package but through inheritance only.
* The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
* It provides more accessibility than the default modifier.

**Example:** In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

package pack;

public class A{

protected void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B extends A{

  public static void main(String args[]){

   B obj = new B();

   obj.msg();

  }

}

**Output:** Hello

**4) Public:**

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example:**

//save by A.java

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();

  }

}

**Output:** Hello

**LECTURE- 7**

**“this” keyword in java**

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.



**Usage of java 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.

**1) 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.

**Example: Understanding the problem without this keyword**

class Student{

int rollno;

String name;

float fee;

Student(int rollno, String name, float fee){

rollno=rollno;

name=name;

fee=fee;

}

void display(){

System.out.println(rollno+" "+name+" "+fee);

}

}

class TestThis1{

public static void main(String args[]){

Student s1=new Student(111,"ABC",5000f);

Student s2=new Student(112,"XYZ",6000f);

s1.display();

s2.display();

}}

**Output:**

0 null 0.0

0 null 0.0

* In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

**Example: Solution of the above problem by this keyword**

class Student{

int rollno;

String name;

float fee;

Student(int rollno,String name,float fee){

this.rollno=rollno;

this.name=name;

this.fee=fee;

}

void display(){System.out.println(rollno+" "+name+" "+fee);}

}

class TestThis2{

public static void main(String args[]){

Student s1=new Student(111,"ankit",5000f);

Student s2=new Student(112,"sumit",6000f);

s1.display();

s2.display();

}}

**Output:**

111 ABC 5000

112 XYZ 6000

* If local variables (formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

**2) this: to invoke current class method**

* We may invoke the method of the current class by using the this keyword. If we don't use the “this” keyword, compiler automatically adds this keyword while invoking the method.

Let's see the example



**Example:** class A{

void m(){System.out.println("hello m");}

void n(){

System.out.println("hello n");

//m();//same as this.m()

this.m();

}

}

class TestThis4{

public static void main(String args[]){

A a=new A();

a.n();

}}

**Output:**

hello n

hello m

**3) 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.

**Example: Calling default constructor from parameterized constructor:**

class A{

A(){System.out.println("hello a");}

A(int x){

this();

System.out.println(x);

}

}

class TestThis5{

public static void main(String args[]){

A a=new A(10);

}}

**Output:**

hello a

10

**Using this() in constructor overloading**

* this() reference can be used during constructor overloading to call default constructor implicitly from parameterized constructor. “this()” should be the first statement inside a constructor.

**Example:** // Java program to illustrate role of this() in Constructor Overloading

class Box

{

    double width, height, depth;

    int boxNo;

    // constructor used when all dimensions and boxNo specified

    Box(double w, double h, double d, int num) {

        width = w;

        height = h;

        depth = d;

        boxNo = num;

    }

    // constructor used when no dimensions specified

    Box()

    {           width = height = depth = 0;     }

    // constructor used when only boxNo specified

    Box(int num)

    {

       // this() is used for calling the default constructor from parameterized constructor

        this();

        boxNo = num;

    }

    public static void main(String[] args)

    {

        // create box using only boxNo

        Box box1 = new Box(1);

        // getting initial width of box1

        System.out.println(box1.width);

    }

}

**Output:** 0.0

**4) this: to pass as an argument in the method**

* The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

class S2{

  void m(S2 obj){

  System.out.println("method is invoked");

  }

  void p(){

  m(this);

  }

  public static void main(String args[]){

  S2 s1 = new S2();

  s1.p();

  }

}

**Output:** method is invoked

Application of this that can be passed as an argument: In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

**5) this: to pass as argument in the constructor call**

* We can pass the “this” keyword in the constructor also. It is useful if we have to use one object in multiple classes.
* Example:

class B{

  A obj;

  B(A obj){      this.obj=obj;    }

  void display(){

    System.out.println(obj.data); //using data member of A class

  }

}

class A{

  int data=10;

  A(){

   B b=new B(this);

   b.display();

  }

  public static void main(String args[])

{     A a=new A();    }

}

**Output:**10

**6) this keyword can be used to return current class instance**

* We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).
* Syntax of this that can be returned as a statement

return\_type method\_name(){  return this;  }

**Example of this keyword that you return as a statement from the method**

class A{

A getA(){  return this;  }

void msg(){System.out.println("Hello java");}

}

class Test1{

public static void main(String args[]){

new A().getA().msg();

}

}

**Output:**

Hello java

**LECTURE-8**

**Java static keyword**

* The static keyword in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables), methods, blocks and [nested classes](https://www.javatpoint.com/java-inner-class). The static keyword belongs to the class than an instance of the class.
* The static can be:
  + Variable (also known as a class variable)
  + Method (also known as a class method)
  + Block
  + Nested class

**1) Java static variable**

* If you declare any variable as static, it is known as a static variable.
* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.
* Advantages of static variable: It makes the program memory efficient (i.e., it saves memory).

**Understanding the problem without static variable**

class Student{

     int rollno;

     String name;

     String college="ABIT";

}

* Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all [objects](https://www.javatpoint.com/object-and-class-in-java). If we make it static, this field will get the memory only once.
* Java static property is shared to all objects.

**Example of static variable**

//Java Program to demonstrate the use of static variable

class Student{

   int rollno;//instance variable

   String name;

   static String college ="ABIT";//static variable

   //constructor

   Student(int r, String n)

{     rollno = r;     name = n;     }

   //method to display the values

   void display (){System.out.println(rollno+" "+name+" "+college);}

}

//Test class to show the values of objects

public class TestStaticVariable{

 public static void main(String args[]){

 Student s1 = new Student(111,"ABC");

 Student s2 = new Student(222,"XYZ");

 //we can change the college of all objects by the single line of code Student.college="IT"

 s1.display();

 s2.display();

 }

}

**Output:**

111 ABC ABIT

222 XYZ ABIT

**Program of the counter without static variable**

* In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

/\*Java Program to demonstrate the use of an instance variable which get memory each time

when we create an object of the class.  \*/

class Counter{

int count=0;//will get memory each time when the instance is created

Counter(){

count++;//incrementing value

System.out.println(count);

}

public static void main(String args[]){

//Creating objects

Counter c1=new Counter();

Counter c2=new Counter();

Counter c3=new Counter();

}

}

**Output:**

1

1

1

**Program of counter by static variable**

* We know that static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

//Java Program to illustrate the use of static variable which is shared with all objects.

class Counter2{

static int count=0;//will get memory only once and retain its value

Counter2(){

count++;//incrementing the value of static variable

System.out.println(count);

}

public static void main(String args[]){

//creating objects

Counter2 c1=new Counter2();

Counter2 c2=new Counter2();

Counter2 c3=new Counter2();

}

}

**Output:**

1

2

3

**2) Java static method**

* If you apply static keyword with any method, it is known as static method.
* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

**Example:** //Java Program to get the cube of a given number using the static method

  class Calculate{

  static int cube(int x){

  return x\*x\*x;

  }

  public static void main(String args[]){

  int result=Calculate.cube(5);

  System.out.println(result);

  }

}

**Output:**125

**Restrictions for the static method**

There are two main restrictions for the static method. They are:

* The static method can not use non static data member or call non-static method directly.
* “this” and “super” cannot be used in static context.

class A{

 int a=40;//non static

    public static void main(String args[]){

  System.out.println(a);

 }

}

**Output:** Compile Time Error

**Q) Why is the Java main method static?**

**Ans)** It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

**3) Java static block**

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

**Example of static block**

class A2{

  static{System.out.println("static block is invoked");}

  public static void main(String args[]){

   System.out.println("Hello main");

  }

}

**Output:** static block is invoked

Hello main

**CHAPTER-2**

**LECTURE - 9**

**Inheritance in Java**

* **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.

**Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

**class** Subclass-name **extends** Superclass-name

{

   //methods and fields

}

* 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.
* In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

**Types of inheritance in java**

* On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
* In java programming, multiple and hybrid inheritance is supported through interface only. 

**Note: Multiple inheritance is not supported in Java through class.**

* When one class inherits multiple classes, it is known as multiple inheritance. For Example:



**1. Single Inheritance Example**

* When a class inherits another class, it is known as a single inheritance.

Class A

{

public void methodA()

{ System.out.println("Base class method"); }

}

Class B extends A

{

public void methodB()

{ System.out.println("Child class method"); }

public static void main(String args[])

{

B obj = new B();

obj.methodA(); //calling super class method

obj.methodB(); //calling local method

}

}

**OUTPUT:**

Base class method

Child class method

**2. Multilevel Inheritance Example**

When there is a chain of inheritance, it is known as multilevel inheritance.

Class X

{

public void methodX()

{ System.out.println("Class X method"); }

}

Class Y extends X

{

public void methodY()

{ System.out.println("class Y method"); }

}

Class Z extends Y

{

public void methodZ()

{ System.out.println("class Z method"); }

public static void main(String args[])

{

Z obj = new Z();

obj.methodX(); //calling grand parent class method

obj.methodY(); //calling parent class method

obj.methodZ(); //calling local method

}

}

**OUTPUT:**

Class X method

Class Y method

Class Z method

**3. Hierarchical Inheritance Example**

* When two or more classes inherits a single class, it is known as hierarchical inheritance.

class A

{

public void methodA()

{ System.out.println("method of Class A"); }

}

class B extends A

{

public void methodB()

{ System.out.println("method of Class B"); }

}

class C extends A

{

public void methodC()

{ System.out.println("method of Class C"); }

}

class D extends A

{

public void methodD()

{ System.out.println("method of Class D"); }

}

class JavaExample

{

public static void main(String args[])

{

B obj1 = new B();

C obj2 = new C();

D obj3 = new D();

//All classes can access the method of class A

obj1.methodA();

obj2.methodA();

obj3.methodA();

}

}

**Output:**

method of Class A

method of Class A

method of Class A

**Q) Why multiple inheritance is not supported in java?**

* To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
* Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
* Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

class A{

void msg(){System.out.println("Hello");}

}

class B{

void msg(){System.out.println("Welcome");}

}

class C extends A, B{

 public static void main(String args[]){

   C obj=new C();

   obj.msg();//Now which msg() method would be invoked?

}

}

**OUTPUT:** Compile Time Error

**LECTURE-9**

**Inheritance and constructors in Java**

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

**Example:**

class Base {

  Base() {

    System.out.println("Base Class Constructor Called ");

  }

}

class Derived extends Base {

  Derived() {

    System.out.println("Derived Class Constructor Called ");

  }

}

public class Main {

  public static void main(String[] args) {

    Derived d = new Derived();

  }

}

**Output:**

Base Class Constructor CalledDerived Class Constructor Called

**Super Keyword in Java**

* The super keyword in java is a reference variable that is used to refer parent class objects.  The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

**1. Use of super with variables:**

This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

/\* Base class vehicle \*/

class Vehicle

{     int maxSpeed = 120; }

 /\* sub class Car extending vehicle \*/

class Car extends Vehicle

{

    int maxSpeed = 180;

    void display()

    {

        /\* print maxSpeed of base class (vehicle) \*/

        System.out.println("Maximum Speed: " + super.maxSpeed);

    }

}

/\* Driver program to test \*/

class Test

{

    public static void main(String[] args)

    {

        Car small = new Car();

        small.display();

    }

}

**Output:** Maximum Speed: 120

* In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in subclass using super keyword.

**2. Use of super with methods:**

This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

/\* Base class Person \*/

class Person

{

    void message()

    {         System.out.println("This is person class");     }

}

/\* Subclass Student \*/

class Student extends Person

{

    void message()

    {        System.out.println("This is student class");     }

    // Note that display() is only in Student class

    void display()

    {

        // will invoke or call current class message() method

        message();

        // will invoke or call parent class message() method

        super.message();

    }

}

/\* Driver program to test \*/

class Test

{

    public static void main(String args[])

    {

        Student s = new Student();

       // calling display() of Student

        s.display();

    }

}

**Output:**

This is student class

This is person class

* In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

**3. Use of super with constructors:**

super keyword can also be used to access the parent class constructor. One more important thing is that, ‘super’ can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

/\* superclass Person \*/

class Person

{

    Person()

    {

        System.out.println("Person class Constructor");

    }

}

/\* subclass Student extending the Person class \*/

class Student extends Person

{

    Student()

    {

        // invoke or call parent class constructor

        super();

        System.out.println("Student class Constructor");

    }

}

/\* Driver program to test\*/

class Test

{

    public static void main(String[] args)

    {

        Student s = new Student();

    }

}

**Output:**

Person class Constructor

Student class Constructor

In the above example we have called the superclass constructor using keyword ‘super’ via subclass constructor.

**Other Important points:**

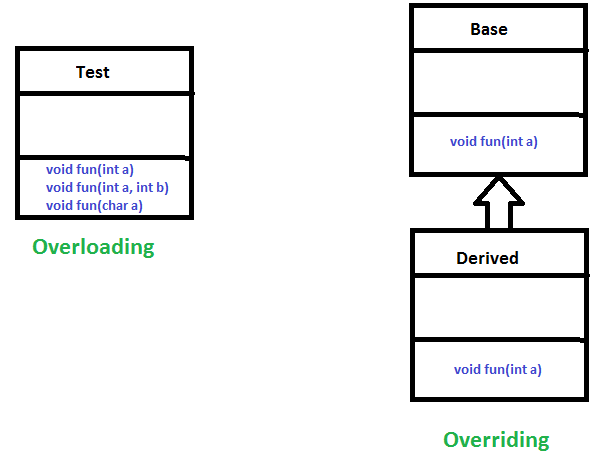
* Call to super() must be first statement in Derived(Student) Class constructor.
* If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
* If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*.
* 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.

**LECTURE-10**

**Polymorphism in Java**

* **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.
* Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person possess different behaviour in different situations. This is called polymorphism.
* Polymorphism is considered as one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
* **In Java polymorphism is mainly divided into two types:**
  + Compile time Polymorph0ism (method overloading and operator overloading)
  + Runtime Polymorphism (method overriding)

1. **Compile time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.
2. [**Runtime polymorphism**](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/): It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

**EXAMPLE:**  
[](https://www.geeksforgeeks.org/overloading-in-java/)

**i. Method Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

**Example:** By using different types of arguments

// Java program for Method overloading

class MultiplyFun {

    // Method with 2 parameter

    static int Multiply(int a, int b)

    {         return a \* b;     }

   // Method with the same name but 2 double parameter

    static double Multiply(double a, double b)

    {        return a \* b;     }

}

 class Main {

    public static void main(String[ ] args)

    {

         System.out.println(MultiplyFun.Multiply(2, 4));

         System.out.println(MultiplyFun.Multiply(5.5, 6.3));

    }

}

**Output:**

8

34.65

**ii. Operator Overloading**: Java also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

* In java, Only “+” operator can be overloaded:
* To add integers
* To concatenate strings

**Example**:

|  |
| --- |
| // Java program for Operator overloading  class OperatorOVERDDN {      void operator(String str1, String str2)      {          String s = str1 + str2;          System.out.println("Concatinated String - " + s);      }      void operator(int a, int b)      {          int c = a + b;          System.out.println("Sum = " + c);      }  }  class Main {      public static void main(String[] args)      {          OperatorOVERDDN obj = new OperatorOVERDDN();          obj.operator(2, 3);          obj.operator("java ", "program");      }  } |

**Output:**

Sum = 5

Concatinated String – java program

**Runtime Polymorphism in Java**

* **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
* In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

**Upcasting:** If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



class A{}

class B extends A{}

A a=new B();//upcasting

For upcasting, we can use the reference variable of class type or an interface type. For Example:

interface I{}

class A{}

class B extends A implements I{}

Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

**Example of Java Runtime Polymorphism**

* In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.
* Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

class Bike{

  void run(){System.out.println("running");}

}

class Splendor extends Bike{

  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){

    Bike b = new Splendor();//upcasting

    b.run();

  }

}

**Output:** running safely with 60km.

* [**Method overriding**](https://www.geeksforgeeks.org/overriding-in-java/)occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Example:**

|  |
| --- |
| // Java program for Method overridding  class Parent {        void Print()      {        System.out.println("parent class");     }  }  class subclass1 extends Parent {      void Print()      {          System.out.println("subclass1");      }  }  class subclass2 extends Parent {      void Print()      {         System.out.println("subclass2");     }  }  class TestPolymorphism3 {      public static void main(String[] args)      {         Parent a;          a = new subclass1();          a.Print();          a = new subclass2();          a.Print();      }  } |

**Output:**

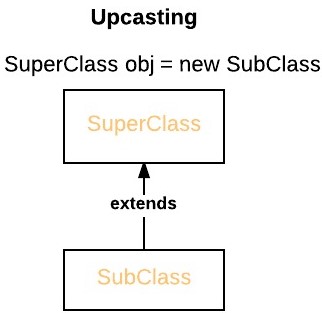
subclass1

subclass2

**LECTURE-11**

**Dynamic Method Dispatch or Runtime Polymorphism in Java**

* Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
* When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
* At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
* A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



* Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

Example: // A Java program to illustrate Dynamic Method Dispatch using hierarchical inheritance

class A

{

    void m1()

    {         System.out.println("Inside A's m1 method");     }

}

class B extends A

{

    // overriding m1()

    void m1()

    {         System.out.println("Inside B's m1 method");     }

}

class C extends A

{

    // overriding m1()

    void m1()

    {        System.out.println("Inside C's m1 method");     }

}

 // Driver class

class Dispatch

{

    public static void main(String args[])

    {

        // object of type A

        A a = new A();

       // object of type B

        B b = new B();

       // object of type C

        C c = new C();

       // obtain a reference of type A

        A ref;

       // ref refers to an A object

        ref = a;

       // calling A's version of m1()

        ref.m1();

       // now ref refers to a B object

        ref = b;

        // calling B's version of m1()

        ref.m1();

        // now ref refers to a C object

        ref = c;

        // calling C's version of m1()

        ref.m1();

    }

}

**Output:**

Inside A's m1 method

Inside B's m1 method

Inside C's m1 method

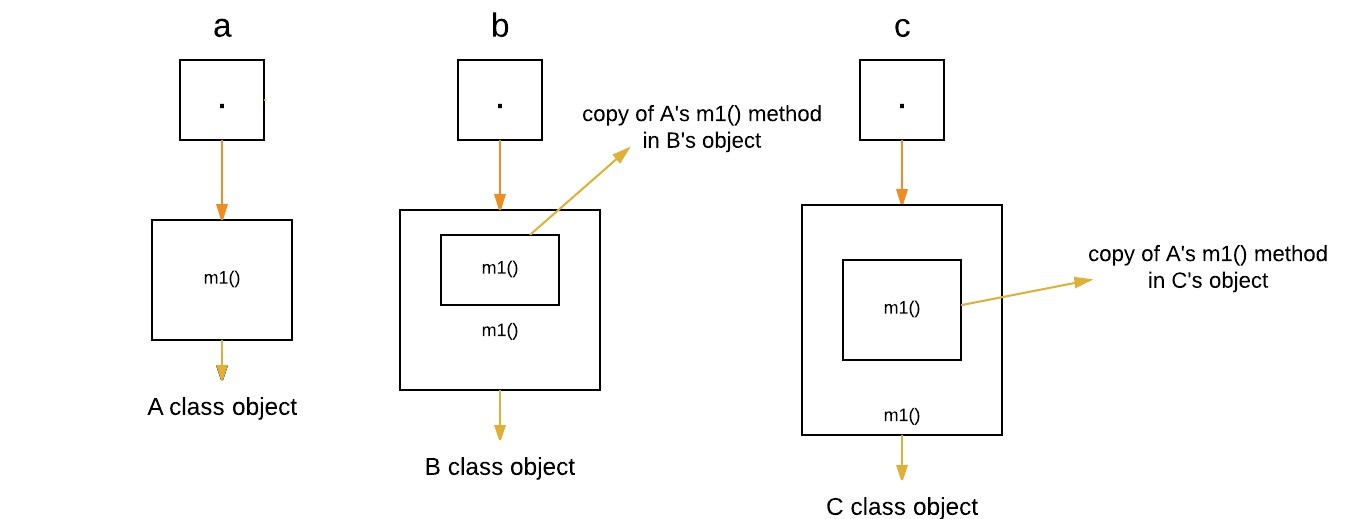
**Explanation:**

* The above program creates one superclass called A and it’s two subclasses B and C. These subclasses overrides m1( ) method.
* Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

A a = new A(); // object of type A

B b = new B(); // object of type B

C c = new C(); // object of type C



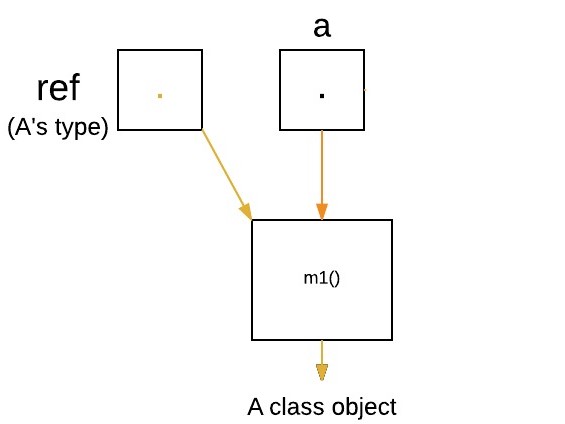
Now a reference of type A, called ref, is also declared, initially it will point to null.

A ref; // obtain a reference of type A

Now we are assigning a reference to each **type of object** (either A’s or B’s or C’s) to ref, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined **by the type of object being referred to at the time of the call.**

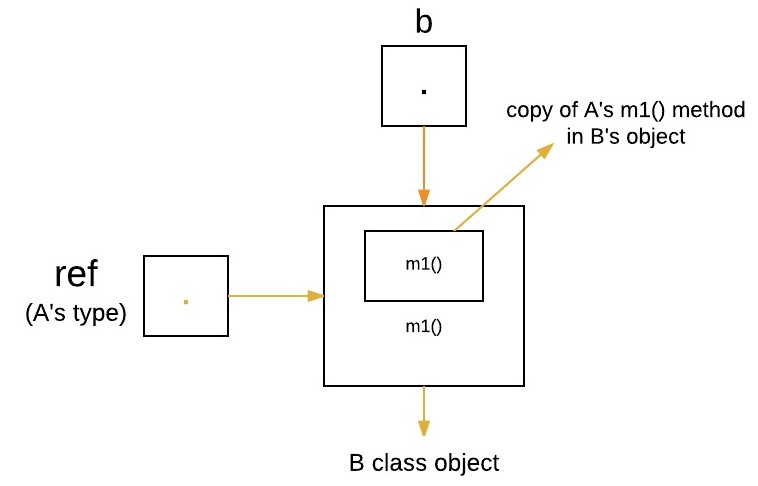
ref = a; // r refers to an A object

ref.m1(); // calling A's version of m1()

[](https://media.geeksforgeeks.org/wp-content/uploads/q1.jpeg)

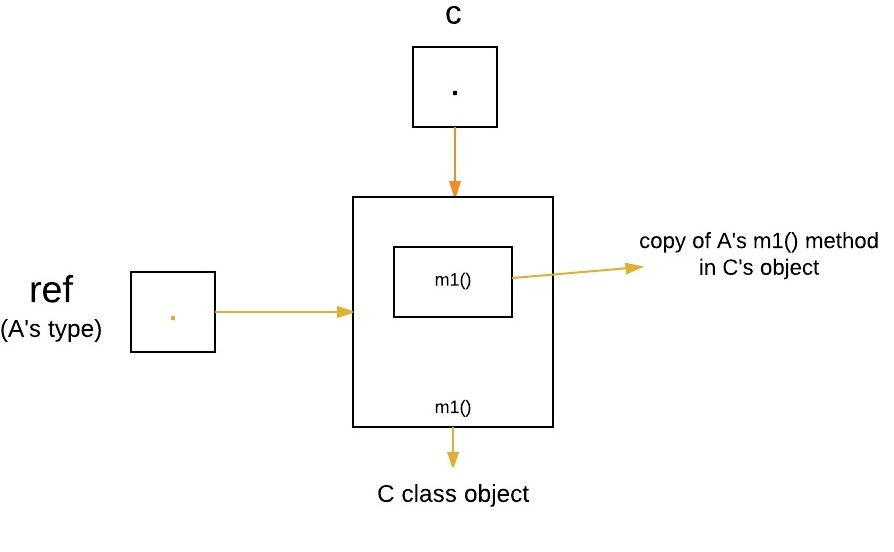
ref = b; // now r refers to a B object

ref.m1(); // calling B's version of m1()

[](https://media.geeksforgeeks.org/wp-content/uploads/q2.jpeg)

ref = c; // now r refers to a C object

ref.m1(); // calling C's version of m1()



**Runtime Polymorphism with Data Members**

* In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

**Example:** /\* Java program to illustrate the fact that runtime polymorphism cannot be achieved by data members \*/

// class A

class A

{     int x = 10; }

// class B

class B extends A

{     int x = 20; }

// Driver class

public class Test

{

    public static void main(String args[])

    {

        A a = new B(); // object of type B

        // Data member of class A will be accessed

        System.out.println(a.x);

    }

}

**Output:** 10

**Explanation :**

* In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will **always** refer to data member of super class.

**Advantages of Dynamic Method Dispatch**

* Dynamic method dispatch allow Java to support [overriding of methods](https://www.geeksforgeeks.org/overriding-in-java/) which is central for run-time polymorphism.
* It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
* It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

**CHAPTER-3**

**STRING MANIPULATION**

**LECTURE-12**

**Java - Strings Class**

* Strings, which are widely used in Java programming, are a sequence of characters.
* In Java programming language, strings are treated as objects.
* The Java platform provides the String class to create and manipulate strings.

**Creating Strings**

* The most direct way to create a string is to write −

String greeting = "Hello world!";

* Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.
* As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

**Example**

public class StringDemo {

public static void main(String args[]) {

char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };

String helloString = new String(helloArray);

System.out.println( helloString );

}

}

**Output:** hello.

**Note** − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then we should use String Buffer & String Builder Classes.

**String Length:**

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the **length()** method, which returns the number of characters contained in the string object.

**Example:** public class StringDemo {

public static void main(String args[]) {

String palindrome = "Dot saw I was Tod";

int len = palindrome.length();

System.out.println( "String Length is : " + len );

}

}

**Output:** String Length is : 17

**Concatenating Strings**

* The String class includes a method for concatenating two strings −

string1.concat(string2);

* This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in − "My name is ".concat("ABC");
* Strings are more commonly concatenated with the + operator, as in −

"Hello," + " world" + "!"

* which results in − "Hello, world!"

**Example:** public class StringDemo {

public static void main(String args[]) {

String string1 = "JAVA";

String string2 = "PROGRAMMING";

System.out.println("string1 + string2");

}

}

**Output:** JAVA PROGRAMMING

**Creating Format Strings**

* You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.
* Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement.
* For example, instead of −

System.out.printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

* We can write −

String fs;

fs = String.format("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

System.out.println(fs);

**String Methods:**

**Java - String charAt() Method**

* This method returns the character located at the String's specified index. The string indexes start from zero.
* **Syntax:** public char charAt(int index)
* **Parameters:** Here is the detail of parameters −

index − Index of the character to be returned.

* **Return Value:** This method returns a char at the specified index.

**Example**

public class Test {

public static void main(String args[]) {

String s = "Strings are immutable";

char result = s.charAt(8);

System.out.println(result);

}

}

**Output:** a

**Java - String compareTo() Method**

* This method compares this String to another Object.
* **Syntax:** int compareTo(Object o)
* **Parameters:** O − the Object to be compared.
* **Return Value:** The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

**Example**

public class Test {

public static void main(String args[]) {

String str1 = "Strings are immutable";

String str2 = new String("Strings are immutable");

String str3 = new String("Integers are not immutable");

int result = str1.compareTo( str2 );

System.out.println(result);

result = str2.compareTo( str3 );

System.out.println(result);

}

}

**Output:** 0

10

**String compareTo(String anotherString)**

* This method compares two strings lexicographically.
* **Syntax:** int compareTo(String anotherString)
* **Parameters:** anotherString − the String to be compared.
* **Return Value:** The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

**Example:** public class Test {

public static void main(String args[]) {

String str1 = "Strings are immutable";

String str2 = "Strings are immutable";

String str3 = "Integers are not immutable";

int result = str1.compareTo( str2 );

System.out.println(result);

result = str2.compareTo( str3 );

System.out.println(result);

result = str3.compareTo( str1 );

System.out.println(result);

}

}

**Output:** 0

10

-10

**Java - String compareToIgnoreCase() Method**

* This method compares two strings lexicographically, ignoring case differences.
* **Syntax:** int compareToIgnoreCase(String str)
* **Parameters:** str − the String to be compared.
* **Return Value:** This method returns a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations.

**Example**

public class Test {

public static void main(String args[]) {

String str1 = "Strings are immutable";

String str2 = "Strings are immutable";

String str3 = "Integers are not immutable";

int result = str1.compareToIgnoreCase( str2 );

System.out.println(result);

result = str2.compareToIgnoreCase( str3 );

System.out.println(result);

result = str3.compareToIgnoreCase( str1 );

System.out.println(result);

}

}

**Output:** 0

10

-10

**Java - String concat() Method**

* This method appends one String to the end of another. The method returns a String with the value of the String passed into the method, appended to the end of the String, used to invoke this method.
* **Syntax:** public String concat(String s)
* **Parameters:** s − the String that is concatenated to the end of this String.
* **Return Value:** This method returns a string that represents the concatenation of this object's characters followed by the string argument's characters.

**Example**

public class Test {

public static void main(String args[]) {

String s = "Strings are immutable";

s = s.concat(" all the time");

System.out.println(s);

}

}

**Output:** Strings are immutable all the time

**Java - String contentEquals() Method**

* This method returns true if and only if this String represents the same sequence of characters as specified in StringBuffer.
* **Syntax:** public boolean contentEquals(StringBuffer sb)
* **Parameters:** sb − the StringBuffer to compare.
* **Return Value:** This method returns true if and only if this String represents the same sequence of characters as the specified in StringBuffer, otherwise false.

**Example:**

public class Test {

public static void main(String args[]) {

String str1 = "Not immutable";

String str2 = "Strings are immutable";

StringBuffer str3 = new StringBuffer( "Not immutable");

boolean result = str1.contentEquals( str3 );

System.out.println(result);

result = str2.contentEquals( str3 );

System.out.println(result);

}

}

**Output**

true

false

**Java - String copyValueOf() Method**

* This method returns a String that represents the character sequence in the array specified.
* **Syntax:** public static String copyValueOf(char[] data)
* **Parameters:** data − the character array.
* **Return Value:** This method returns a String that contains the characters of the character array.

**Example:** public class Test {

public static void main(String args[]) {

char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};

String Str2 = "";

Str2 = Str2.copyValueOf( Str1 );

System.out.println("Returned String: " + Str2);

}

}

**Output:** Returned String: hello world

**Java - String copyValueOf(data, offset, count)**

* This returns a String that represents the character sequence in the array specified.
* **Syntax:** public static String copyValueOf(char[] data, int offset, int count)
* **Parameters:**

data − the character array.

offset − initial offset of the subarray.

count − length of the subarray.

* **Return Value:** This method returns a String that contains the characters of the character array.

**Example:** public class Test {

public static void main(String args[]) {

char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};

String Str2 = "";

Str2 = Str2.copyValueOf( Str1, 2, 6 );

System.out.println("Returned String: " + Str2);

}

}

**Output:** Returned String: llo wo

**Java - String endsWith() Method**

* This method tests if this string ends with the specified suffix.
* **Syntax:** public boolean endsWith(String suffix)
* **Parameters:** suffix − the suffix.
* **Return Value:** This method returns true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise. Note that the result will be true if the argument is the empty string or is equal to this String object as determined by the equals (Object) method.

**Example**

public class Test {

public static void main(String args[]) {

String Str = new String("This is really not immutable!!");

boolean retVal;

retVal = Str.endsWith( "immutable!!" );

System.out.println("Returned Value = " + retVal );

retVal = Str.endsWith( "immu" );

System.out.println("Returned Value = " + retVal );

}

}

**Output:** Returned Value = true

Returned Value = false

**Java - String equals() Method**

* This method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
* **Syntax:** public boolean equals(Object anObject)
* **Parameters:** anObject − the object to compare this String against.
* **Return Value:** This method returns true if the String are equal; false otherwise.

**Example**

public class Test {

public static void main(String args[]) {

String Str1 = new String("This is really not immutable!!");

String Str2 = Str1;

String Str3 = new String("This is really not immutable!!");

boolean retVal;

retVal = Str1.equals( Str2 );

System.out.println("Returned Value = " + retVal );

retVal = Str1.equals( Str3 );

System.out.println("Returned Value = " + retVal );

}

}

**Output:** Returned Value = true

Returned Value = true

**Java - String equalsIgnoreCase() Method**

* This method compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case, if they are of the same length, and corresponding characters in the two strings are equal ignoring case.
* **Syntax:** public boolean equalsIgnoreCase(String anotherString)
* **Parameters:** anotherString − the String to compare this String against.
* **Return Value:** This method returns true if the argument is not null and the Strings are equal, ignoring case; false otherwise.

**Example:** public class Test {

public static void main(String args[]) {

String Str1 = new String("This is really not immutable!!");

String Str2 = Str1;

String Str3 = new String("This is really not immutable!!");

String Str4 = new String("This IS REALLY NOT IMMUTABLE!!");

boolean retVal;

retVal = Str1.equals( Str2 );

System.out.println("Returned Value = " + retVal );

retVal = Str1.equals( Str3 );

System.out.println("Returned Value = " + retVal );

retVal = Str1.equalsIgnoreCase( Str4 );

System.out.println("Returned Value = " + retVal );

}

}

**Output:** Returned Value = true

Returned Value = true

Returned Value = true

**Java – String indexOf() Method**

* **Description:** This method returns the index within this string of the first occurrence of the specified character or -1, if the character does not occur.
* **Syntax:** public int indexOf(char ch)
* **Parameter:** ch − a character.

**Example:** public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Found Index :" );

System.out.println(Str.indexOf( 'o' ));

}

}

**Output:** Found Index :4

**Java - String indexOf(int ch, int fromIndex)**

* This method returns the index within this string of the first occurrence of the specified character, starting the search at the specified index or -1, if the character does not occur.
* **Syntax:** public in indexOf(char ch, int fromIndex)
* **Parameters:** Here is the detail of parameters −

ch − a character.

fromIndex − the index to start the search from.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Found Index :" );

System.out.println(Str.indexOf( 'o', 5 ));

}

}

**Output:** Found Index :9

**Java – String indexOf(String str) Method**

* **Description:** This method returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.
* **Syntax:** int indexOf(String str)
* **Parameters:** str − a string.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

String SubStr1 = new String("to");

System.out.println("Found Index :" + Str.indexOf( SubStr1 ));

}

}

**Output:** Found Index :8

**Java – String lastIndexOf() Method**

* **Description:** This method returns the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.
* **Syntax**: int lastIndexOf(int ch)
* **Parameters:** ch − a character.
* **Return Value:** This method returns the index.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Found Last Index :" );

System.out.println(Str.lastIndexOf( 'o' ));

}

}

**Output:** Found Last Index :18

**Java - String length() Method**

* **Description:** This method returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.
* **Syntax:** public int length()
* **Return Value:** This method returns the the length of the sequence of characters represented by this object.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str1 = new String("Welcome to Java programming");

String Str2 = new String("Tutorials" );

System.out.print("String Length :" );

System.out.println(Str1.length());

System.out.print("String Length :" );

System.out.println(Str2.length());

}

}

**Output:**

String Length :27

String Length :9

**Java - String startsWith() Method**

* **Description:** This method has two variants and tests if a string starts with the specified prefix beginning a specified index or by default at the beginning.
* **Syntax:** public boolean startsWith (String prefix)
* **Parameters**: prefix − the prefix to be matched.
* **Return Value:** It returns true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise.

**Example:** public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Return Value :" );

System.out.println(Str.startsWith("Welcome") );

System.out.print("Return Value :" );

System.out.println(Str.startsWith("Tutorials") );

}

}

**Output:** Return Value :true

Return Value :false

**Java - String substring() Method**

* **Description:** This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex – 1, if the second argument is given.
* **Syntax:** public String substring(int beginIndex)
* **Parameters:** beginIndex − the begin index, inclusive.
* **Return Value:** The specified substring.

**Example:** public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Return Value :" );

System.out.println(Str.substring(10) );

}

}

**Output:** Return Value : Java programming

**Java - String toLowerCase() Method**

* **Description:** This method has two variants. The first variant converts all of the characters in this String to lower case using the rules of the given Locale. This is equivalent to calling toLowerCase(Locale.getDefault()). The second variant takes locale as an argument to be used while converting into lower case.
* **Syntax:** public String toLowerCase()
* **Return Value:** It returns the String, converted to lowercase.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Return Value :");

System.out.println(Str.toLowerCase());

}

}

**Output:** Return Value :welcome to java programming

**Java - String toUpperCase() Method**

* **Description:** This method has two variants. The first variant converts all of the characters in this String to upper case using the rules of the given Locale. This is equivalent to calling toUpperCase(Locale.getDefault()). The second variant takes locale as an argument to be used while converting into upper case.
* **Syntax:** public String toUpperCase()
* **Return Value:** It returns the String, converted to uppercase.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str = new String("Welcome to Java programming");

System.out.print("Return Value :" );

System.out.println(Str.toUpperCase() );

}

}

**Output:** Return Value : WELCOME TO JAVA PROGRAMMING

**Java - String trim() Method**

* This method returns a copy of the string, with leading and trailing whitespace omitted.
* **Syntax:** public String trim()
* **Return Value:** It returns a copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.

**Example:** import java.io.\*;

public class Test {

public static void main(String args[]) {

String Str = new String(" Welcome to Java programming ");

System.out.print("Return Value :" );

System.out.println(Str.trim() );

}

}

**Output:** Return Value :Welcome to Java programming

**LECTURE - 13**

**String Buffer and String Builder Classes**

* The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.
* Unlike Strings, objects of type StringBuffer and String builder can be modified over and over again without leaving behind a lot of new unused objects.
* The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe (not synchronised).
* It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However, if the thread safety is necessary, the best option is StringBuffer objects.

**Example:** public class Test {

public static void main(String args[]) {

StringBuffer sBuffer = new StringBuffer("test");

sBuffer.append(" String Buffer");

System.out.println(sBuffer);

}

}

**Output:** test String Buffer

**Java StringBuffer class**

* Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
* Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

**Important methods of StringBuffer class**

**1) StringBuffer append() method**

* The append() method concatenates the given argument with this string.

class StringBufferExample{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

**2) StringBuffer insert() method**

* The insert() method inserts the given string with this string at the given position.

class StringBufferExample2{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

**3) StringBuffer replace() method**

* The replace() method replaces the given string from the specified beginIndex and endIndex.

class StringBufferExample3{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

**4) StringBuffer delete() method**

* This method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

class StringBufferExample4{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

**5) StringBuffer reverse() method**

* The reverse() method of StringBuilder class reverses the current string.

class StringBufferExample5{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

**6) StringBuffer capacity() method**

* The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2.
* For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBufferExample6{

public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

}

}

**7) StringBuffer ensureCapacity() method**

* The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBufferExample7{

public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

}

**Java StringBuilder class**

* Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

**Java StringBuilder Examples**

**1) StringBuilder append() method**

* The StringBuilder append() method concatenates the given argument with this string.

class StringBuilderExample{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

**2) StringBuilder insert() method**

* The StringBuilder insert() method inserts the given string with this string at the given position.

class StringBuilderExample2{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

**3) StringBuilder replace() method**

* The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

class StringBuilderExample3{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

**4) StringBuilder delete() method**

* This method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

class StringBuilderExample4{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

**5) StringBuilder reverse() method**

* The reverse() method of StringBuilder class reverses the current string.

class StringBuilderExample5{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

**6) StringBuilder capacity() method**

* The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2.
* For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBuilderExample6{

public static void main(String args[]){

StringBuilder sb=new StringBuilder();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

}

}

**7) StringBuilder ensureCapacity() method**

* The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2.
* For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBuilderExample7{

public static void main(String args[]){

StringBuilder sb=new StringBuilder();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

}

**Q. What is mutable string?**

**ANS:** A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

**Difference between String and StringBuffer**

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

|  |  |
| --- | --- |
| **String** | **StringBuffer** |
| String class is immutable. | StringBuffer class is mutable. |
| String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you cancat strings. |
| String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

**Difference between StringBuffer and StringBuilder**

* Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.
* A list of differences between StringBuffer and StringBuilder are given below:

|  |  |
| --- | --- |
| **StringBuffer** | **StringBuilder** |
| StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

# Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

**Need of Wrapper Classes**

* They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
* The classes in java.util package handles only objects and hence wrapper classes help in this case also. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.

**Primitive Data types and their Corresponding Wrapper class** [[](https://media.geeksforgeeks.org/wp-content/uploads/Wrapper-Class.png)](https://media.geeksforgeeks.org/wp-content/uploads/Wrapper-Class.png)

**Autoboxing and Unboxing**

1. **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example: conversion of int to Integer, long to Long, double to Double etc.
2. **Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example: conversion of Integer to int, Long to long, Double to double etc.

**Example:** //Java program to demonstrate Wrapping and UnWrapping in Java Classes

class WrappingUnwrapping

{

    public static void main(String args[])

    {

         byte a = 1; //  byte data type

          Byte byteobj = new Byte(a);         // wrapping around Byte object

        int b = 10;         // int data type

        Integer intobj = new Integer(b);         //wrapping around Integer object

        float c = 18.6f;         // float data type

        Float floatobj = new Float(c);         // wrapping around Float object

        double d = 250.5;         // double data type

        Double doubleobj = new Double(d);         // Wrapping around Double object

        char e='a';         // char data type

        Character charobj=e;         // wrapping around Character object

        //  printing the values from objects

        System.out.println("Values of Wrapper objects (printing as objects)");

        System.out.println("Byte object byteobj:  " + byteobj);

        System.out.println("Integer object intobj:  " + intobj);

        System.out.println("Float object floatobj:  " + floatobj);

        System.out.println("Double object doubleobj:  " + doubleobj);

        System.out.println("Character object charobj:  " + charobj);

          // objects to data types (retrieving data types from objects)

        // unwrapping objects to primitive data types

        byte bv = byteobj;

        int iv = intobj;

        float fv = floatobj;

        double dv = doubleobj;

        char cv = charobj;

        // printing the values from data types

        System.out.println("Unwrapped values (printing as data types)");

        System.out.println("byte value, bv: " + bv);

        System.out.println("int value, iv: " + iv);

        System.out.println("float value, fv: " + fv);

        System.out.println("double value, dv: " + dv);

        System.out.println("char value, cv: " + cv);

    }

}

**Output:**

Values of Wrapper objects (printing as objects)

Byte object byteobj: 1

Integer object intobj: 10

Float object floatobj: 18.6

Double object doubleobj: 250.5

Character object charobj: a

Unwrapped values (printing as data types)

byte value, bv: 1

int value, iv: 10

float value, fv: 18.6

double value, dv: 250.5

char value, cv: a