UNIT-II CLASSES AND OBJECTS

Java is an object oriented language and underlying structure of all java programs is a class.

Anything we wish to represent in a java program must be encapsulated in to a class that defines the state and behavior of basic programming components known as objects.

Using object-oriented programming, your overall program is made up of lots of different self-contained components (objects), each of which has a specific role in the program and all of which can talk to each other in predefined ways.

Object:

An object is anything that really exists in the world and can be distinguished from others. Every object has properties(or attributes) and can perform certain actions(or behaviour)

Ex:

A MotorCycle object can have make, color, enginestate etc. and can perform actions like startengine, accelerate, stop etc.

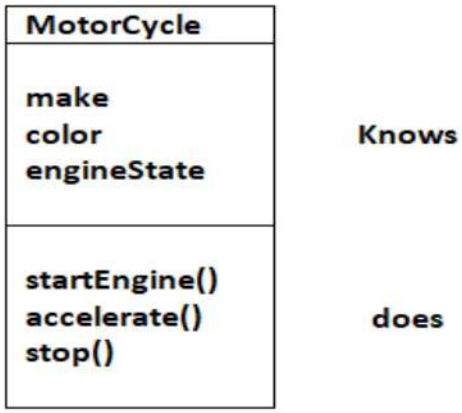
Class:

It is possible that some objects may have similar properties and actions. Such objects belong to same category called a 'class'.

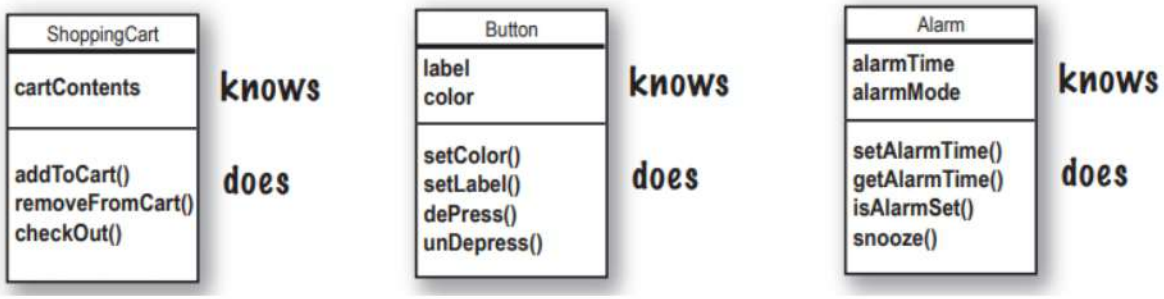
While writing a program in an object-oriented language, define classes of objects. Then instances of these classes called objects were created. **A class is a blueprint for an object** i.e., a class is a template for creating multiple objects with similar features. It tells the virtual machine how to make an object of that particular type .**An object is an instance of a class that has some state and behavior.**

When you design a class, think about the objects that will be created from that class type.

- Think about:
  - things the object knows
  - things the object does

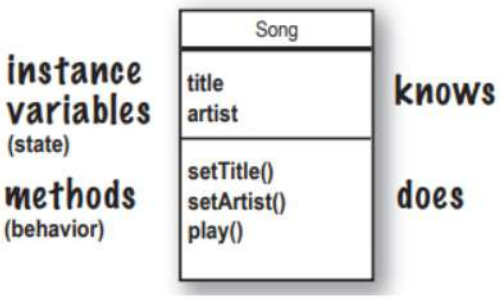


Ex:



Things an object knows (attributes) represent an object's state (the data) and Things an object can do are called methods represent object's behavior

Ex:



Declaration of a class:

The syntax to declare the class is given as follows

```
[Access_Modifiers] class Class_Name [extends Super_Class_Name] [implements interface_List]
{
 --- variables---
 --- methods---
 --- constructors---
 --- blocks----
 --- classes---
 --- abstract classes---
 --- interfaces---
 --- enums--------
}
```

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

- Access modifiers: A class can be public or has default access(for inner classes protected and private are also applicable)
- class keyword: class keyword is used to create a class.
- Class name: Any valid Identifier
- Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class (subclass) can only extend one parent.
- Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- Body: The class body surrounded by braces, { }.
- Variable declaration represents the type of data members which we use as a part of the class.
- Method definitions represents the type of the methods which we use as a part of the class to perform meaningful operations by making use of data members of the class

Ex:

```
class MotorCycle
{
      String make;
      String color;
      boolean engineState;
      void startEngine()
      {
          if (engineState == true)
              System.out.println("The engine is already on.");
          else
          {
              engineState = true;
              System.out.println("The engine is now on.");
          }
      }
}
```

Ex:

```
class Circle {
   double x,y;
   double r;
   double circumference(){
      return 2*3.14159*r;
   }
   double area(){
      return (22/7)*r*r;
   }
}
```

Creating Objects:

To create an object, the following syntax is used

- classname objectname;
  objectname = new classname();
  ex: MotorCycle m;
  m = new MotorCycle();
- classname objectname = new classname();
  ex: MotorCycle m = new MotorCycle();

Here 'new' is an operator that creates the object to MotorCycle class, hence the right hand side part of the statement is responsible for creating the object,

The left hand side part of the statement i.e.,

MotorCycle m

tells that m is a referrence variable of class MotorCycle class. This variable stores the reference number of the object returned by JVM, after creating the object.

Note:

When an object is created, the memory is allocated on 'heap'. After creation of an object, JVM produces a 'unique reference number for the object from the memory address of the object. This reference number is also called hash code number

Ex2: consider the Dog class as shown below.

```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

instance variables

a method

The 3 steps of object declaration, creation and assignment

```
1        3  2
Dog myDog = new Dog();
```

1 Declare a reference variable

```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a reference variable, and names that variable myDog. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

2 Create an object

```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a new Dog object on the heap (we'll learn a lot more about that process, especially in chapter 9.)

3 Link the object and the reference

```
Dog myDog = new Dog();
```

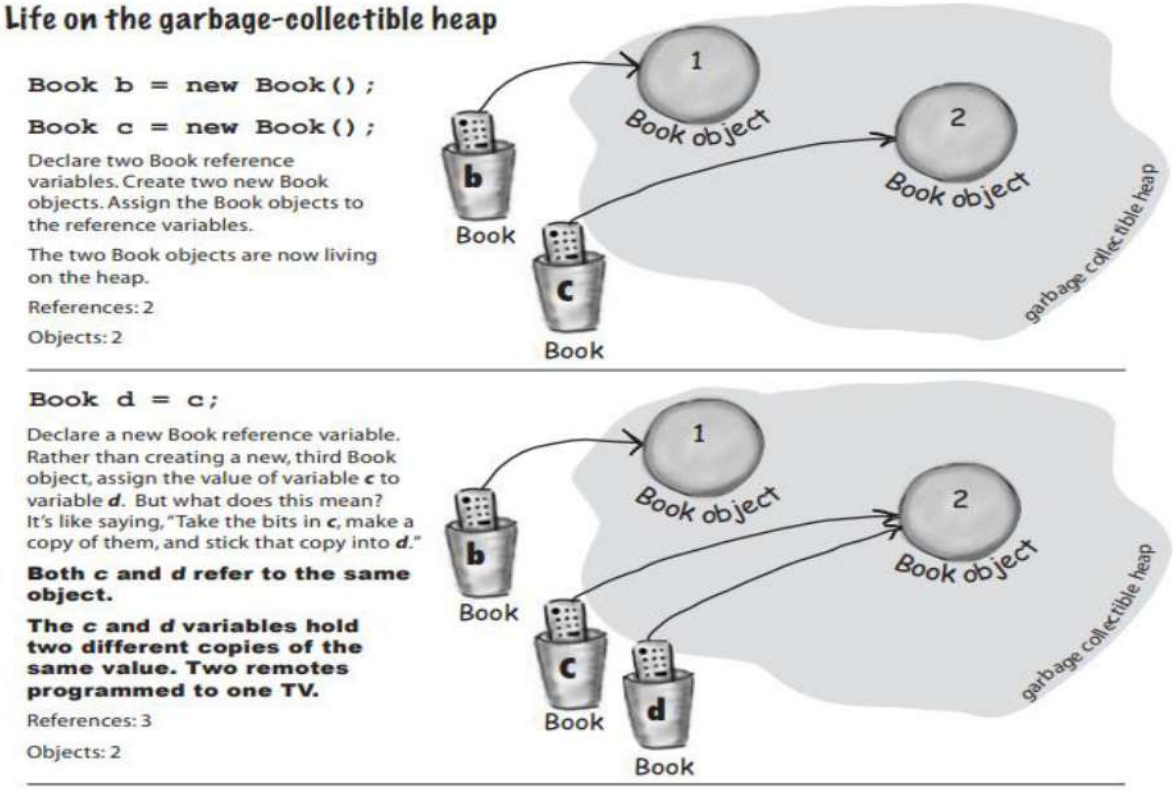
Assigns the new Dog to the reference variable myDog. In other words, ***programs the remote control.***

Assigning One Object to Another:

In Java, assigning one object to another involves assigning the reference of one object to another reference variable.

When you assign one object reference to another, both reference variables will point to the same object in memory. It does not create a new copy of the object.

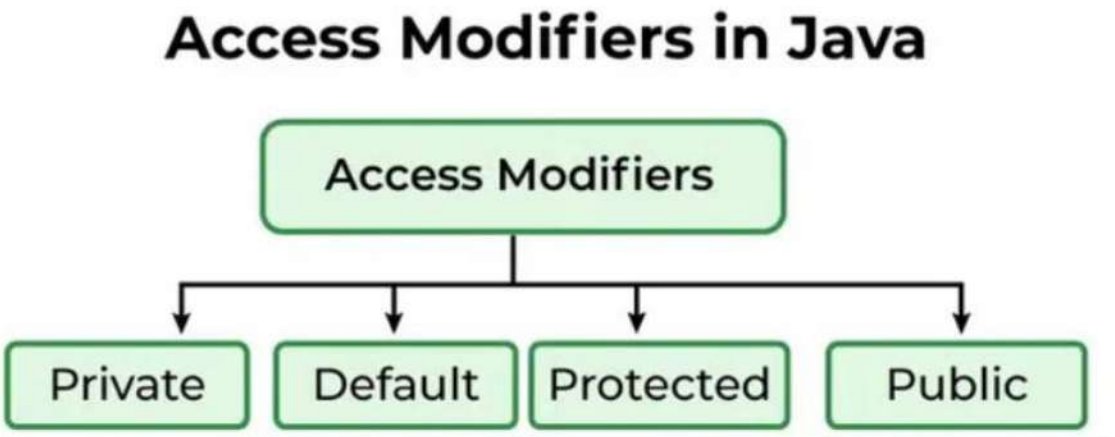
Any changes made through one reference will be reflected when accessing the object through the other reference.



Access Control for Class Members:

Access control in Java determines the visibility of class members (fields and methods) to other classes. In Java, you can use access modifiers to protect class members(fields and methods) when you declare them. It's a fundamental concept for object-oriented programming, ensuring data encapsulation and code modularity.

There are four types of access modifiers available in Java



1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **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.
3. **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.
4. **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.

This is illustrated in the following figure.

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

Methods:

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. We write a method once and use it many times. The method is executed only when we call or invoke it.

Defining a method:

A method-has two parts:

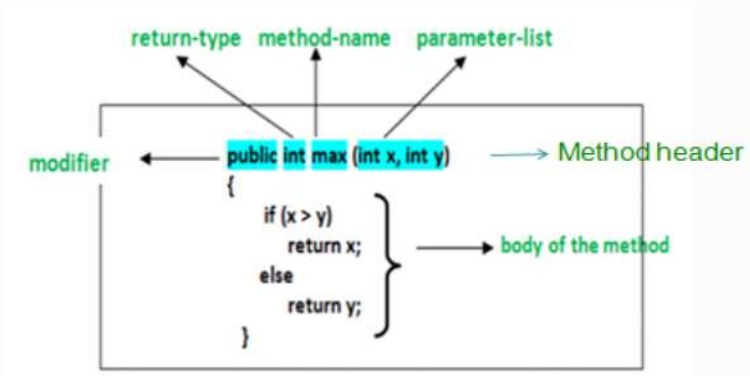
- method header or method prototype
- method body

syntax:

```
[access modifier] returntype  methodname ([parameters]) [throws ExceptionType1, .....ExceptionTypen]
{
   // method body
}
```

In general, method declarations has six components.

1. Access Modifier
2. return type
3. Method Name
4. Parameter-list
5. Exception list
6. Method body



- Acess Modifier:

  It defines the access type of the method i.e. from where it can be accessed in your application like public, private, protected and default.

- The return type:

  The data type of the value returned by the method or void if does not return a value.

- Method Name:

  A unique identifier for the method

- Parameter list:

Comma-separated list of the input parameters is defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().

- Exception list:

A comma-separated list of exception types that the method might throw.

- Method body:

It is enclosed between braces. The code you need to be executed to perform your intended operations.

Ex:

```
import java.io.IOException;

public class FileHandler {
    public String readFile(String fileName) throws IOException {
        // Code to read the file
        // ...
        return fileContent;
    }
}
```

Calling the method or method invocation:

To use the method's functionality, you need to "call" a method. This is as simple as **writing the name of the method by passing its parameters.**

static methods are any methods with the keyword static before the method name. To call a static method, object of the class is not needed. It can be accessed directly using a class name as follows

ClassName.methodName(arguments);

To call a non static method(or instance method), object of the class is needed. It can be accessed through the object reference as follows.

objectReferenceName.methodName(arguments);

Ex:

```
public class Test
{
   public static void main(String[] args)
   {
      int result = addNumbers(2, 3);  //calling a static method
      System.out.println("The sum is " + result);
   }

   public static int addNumbers(int a, int b)
   {
      int sum = a + b;
      return sum;
   }
}
```

```
Output:
The sum is 5
```

mutator methods and accessor methods

There are two types of methods in Java w.r.t the Object state manipulations.

a) Mutator Methods: (or setters or setter methods)

methods which are used to set/modify data in Objects are called mutator methods

b) Accessor Methods: (or getters or getter methods)

methods which are used to get/access data from Objects are called accessor methods

```
class Employee
{
   private String eid;  //instance variable
   private String ename;
   private float esal;
   private String eaddr;
   static String Companyname= "bytexl";

   //setters --mutator methods
   public void setEid(String eid)
   {
      this.eid=eid;
   }
   public void setEname(String ename)
   {
      this.ename=ename;
   }
   public void setEsal(float esal)
   {
   this.esal=esal;
```

```
    }
    public void setEaddr(String eaddr)
    {
       this.eaddr=eaddr;
    }


    //getters ---accessor methods
    public String getEid()
    {
       return eid;
    }
    public String getEname()
    {
       return ename;
    }

    public float getEsal()
    {
       return esal;
    }
    public String getEaddr()
    {
       return eaddr;
    }
}

public class Test
{
    public static void main(String[] args)
    {
       Employee emp=new Employee();
       emp. setEid("E-1111");
       emp.setEname("Pavan Kumar");
       emp.setEsal(5000.0f);
       emp.setEaddr("Hyd");
```

```
        System.out.println("Employee Details");
        System.out.println("--------------------");
        System.out.println("Employee Id :"+emp.getEid());
        System.out.println("Employee Name :"+emp.getEname());
        System.out.println("Employee Salary :"+emp.getEsal());
        System.out.println("Employee Address :"+emp.getEaddr());

    }
}
```

Output:

```
Employee Details
--------------------
Employee Id :E-1111
Employee Name :Pavan Kumar
Employee Salary :5000.0
Employee Address :Hyd
```

Passing Objects to methods:

In java, a method can receive an object as a parameter i.e., an object can be passed as a parameter to the method. The method can then use this object to access its properties and methods.

When passing an object to a method, you're actually passing a copy of the reference to that object, not the actual object itself. This means both the original reference and the copied reference point to the same object in memory.

```
class Employee
{
    private String eid;  //instance variable
    private String ename;
    private float esal;
    private String eaddr;
    static String CompanyName= "bytexl";
```

```
//setters --mutator methods
public void setEid(String eid)
{
   this.eid=eid;
}
public void setEname(String ename)
{
   this.ename=ename;
}
public void setEsal(float esal)
{
   this.esal=esal;
}
public void setEaddr(String eaddr)
{
   this.eaddr=eaddr;
}


//getters ---accessor methods
public String getEid()
{
   return eid;
}
public String getEname()
{
   return ename;
}

public float getEsal()
{
   return esal;
}
public String getEaddr()
{
   return eaddr;
```

```
    }
}

public class Test
{
   public static void main(String[] args)
    {
       Employee emp=new Employee();
       emp. setEid("E-1111");
       emp.setEname("Pavan Kumar");
       emp.setEsal(5000.0f);
       emp.setEaddr("Hyd");

       // Call the method, passing the Employee object as a parameter
       displayEmployeeDetails(emp);
    }
     public static void displayEmployeeDetails(Employee emp) {
       System.out.println("Employee Details");
       System.out.println("--------------------");
       System.out.println("Employee Id :"+emp.getEid());
       System.out.println("Employee Name :"+emp.getEname());
       System.out.println("Employee Salary :"+emp.getEsal());
       System.out.println("Employee Address :"+emp.getEaddr());
       System.out.println("Employee company name:" +
Employee.CompanyName);

    }
}
```

Output:

```
Employee Details
--------------------
Employee Id :E-1111
Employee Name :Pavan Kumar
Employee Salary :5000.0
Employee Address :Hyd
Employee company name:bytexl
```

Ex2:

```
class Car {
    private String make;
    private String model;

    public Car(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void displayDetails() {
        System.out.println("Make: " + make + ", Model: " + model);
    }
}

public class Main {
    // Method that changes the state of the Car object
    public static void updateCarModel(Car car, String newModel) {
        car.setModel(newModel); // Modifies the state of the passed
object
    }
```

```
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla");
        System.out.println("Before update:");
        myCar.displayDetails();

        // Passing the Car object to the method
        updateCarModel(myCar, "Camry");

        System.out.println("After update:");
        myCar.displayDetails(); // The state of the original object is
changed
    }
}
```

Output:

```
Before update:
Make: Toyota, Model: Corolla
After update:
Make: Toyota, Model: Camry
```

Note:

In Java, **everything is passed by value**, which means the method receives a copy of the variable's value. When passing an object to a method, you're actually passing a copy of the reference to that object, not the actual object itself. This means both the original and the copy of the reference point to the same object.

In Java, **if we pass reference variable as parameter to the methods then the parameter passing mechanism is "Call By Value"** only, because, in JAVA, reference variables are not storing address locations, reference variables are able to store Object reference value, where Object reference value is hexa decimal form of Hashcode, where Hashcode is an integer value provided by Heap manager as an unique identity for each and every object.

Method Overloading:

In Java, two or more methods may have the **same name** if they **differ in parameters** (different number of parameters, different types of parameters, or both).

These methods are called overloaded methods and this feature is called method overloading.

```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

Method overloading is achieved by either changing the number of arguments or changing the data type of arguments. It is not method overloading if we only change the return type of methods. There must be differences in the parameters(either in number or data types)

```
// Example program to illustrate method overloading
//method with same number of parameters but different datatypes

class A
{
    void add(int i,int j)
    {
        System.out.println(i+j);
    }
    void add(float i,float j)
    {
        System.out.println(i+j);
    }
    void add(String str1,String str2)
    {
```

```
        System.out.println(str1+str2);
    }
}
public class Test
{
   public static void main(String[] args)
   {
      A a=new A();
      a.add(10,20);
      a.add(22.22f,33.33f);
      a.add("abc","def");
   }
}
```

Output:

```
30
55.550003
abcdef
```

Ex2:

```
//program to illustrate static polymorphism-metod overloading

class Employee
{
    void gen_Salary(int basic, float hk, float pf, int ta)
    {
        float salary = basic+((basic*hk)/100)-((basic*pf)/100)+ta;
        System.out.println("Salary :"+salary);
    }

    void gen_Salary(int basic, float hk, float pf, int ta, int bonus)
    {
        float salary = basic+((basic*hk)/100)-((basic*pf)/100)+ta+bonus;
        System.out.println("Salary :"+salary);
    }
}
class Test2
{
    public static void main(String[] args)
    {
        Employee e = new Employee();
        e.gen_Salary(20000,25.0f,12.0f,2000);
        e.gen_Salary(20000,25.0f,12.0f,2000,5000);
    }
}
```

Output:

Salary :24600.0
Salary :29600.0

Recursive methods:

A method that calls itself repeatedly until it reaches a base case is called a recursive method and this phenomenon is called recursion.

Ex1: Write a java program to find factorial of an integer

```
public class RecursionExample {
    public static int factorial(int n) {
        if (n == 0 || n == 1) { // Base case
            return 1;
        } else { // Recursive case
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + " is " + result);
    }
}
```

Output:

Factorial of 5 is 120

Ex2: Write a java program to print Fibonacci sequence

```
import java.util.Scanner;

public class FibonacciSequence {

    // Function to print Fibonacci sequence
    public static void printFibonacci(int n) {
        int a = 0;
        int b = 1;

        System.out.print("Fibonacci Sequence: " + a + ", " + b);

        for (int i = 3; i <= n; i++) {
            int c = a + b;
            System.out.print(", " + c);
            a = b;
            b = c;
        }
    }

    // Main function to test the Fibonacci sequence generation
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of terms: ");
        int n = scanner.nextInt();

        // Print the Fibonacci sequence
        printFibonacci(n);
    }
}
```

Output:

```
Enter the number of terms: 10
Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Ex3: write a java program to solve towers of hanoi problem

```
public class TowersOfHanoi {

    public static void solveTowersOfHanoi(int n, char source, char auxiliary, char destination) {
        if (n == 1) {
            System.out.println("Move disk 1 from " + source + " to " + destination);
            return;
        }

        // Move n-1 disks from source to auxiliary
        solveTowersOfHanoi(n - 1, source, destination, auxiliary);

        // Move the nth disk from source to destination
        System.out.println("Move disk " + n + " from " + source + " to " + destination);

        // Move the n-1 disks from auxiliary to destination
        solveTowersOfHanoi(n - 1, auxiliary, source, destination);
    }


    public static void main(String[] args) {
        int n = 3; // Number of disks
        solveTowersOfHanoi(n, 'A', 'B', 'C');
                // A, B, and C are names of rods
    }
}
```

Output:

Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Constructors:

A constructor is similar to a method that is used to set initial values for object attributes. It is called automatically when an object of a class is created.

A constructor has the same name as that of the class and does not have any return type.

```
class Test
{
      Test()
      {
            // constructor body
       }
}
```

Here, Test() is a constructor. It has the same name as that of the class and doesn't have a return type.

Ex:

```
//program to illustrate constructors
public class Cons1
{
  String name;

 // constructor
 Cons1()
 {

  System.out.println("Constructor Called:");
  name = "pavan";
 }

 public static void main(String[] args) {

  // constructor is invoked while
  // creating an object of the Main class
  Cons1 obj = new Cons1();
  System.out.println("The name is " + obj.name);


  Cons1 obj2 = new Cons1();
  System.out.println("The name is " + obj2.name);
 }
}
```

```
Output:

Constructor Called:
The name is pavan
Constructor Called:
The name is pavan
```

Note1:

If we provide constructor name other than class name then compiler will rise an error like "Invalid Method declaration, return type required", because, compiler has treated the provided constructor as normal java method without the return type, but, for methods return is mandatory

Note 2:

If we provide return type to the constructors then Compiler will not rise any error and JVM will not rise any exception and JVM will not provide any output, because, the provided constructor is converted as normal java method. In this context, if we access the provided constructor as normal java method then it will be executed as like normal java method

Note3:

In Java, the following modifiers are NOT allowed for constructors:

static, final, abstract, synchronized, native, strictfp

If these modifiers are provided to the constructor, compiler will raise an error.

Note4:

If we declare constructor as "private" then that constructor is available upto the respective class only, not available to out side of the respective class. In this context, If we want to create object for the respective class then it is possible inside the same class only

Types of constructors:

There are 3 types of constructors.

- no-arg constructors
- Parameterized constructors
- Default constructors

No-arg constructors:

A No-Argument Constructor is a constructor that takes no arguments. It is a special type of constructor that is used to initialize objects when no arguments are passed to it.

```
class Cons2 {
        int num;
        String name;

        // this would be invoked while an object
        // of that class is created.
        Cons2()
        {
            System.out.println("Constructor called");
        }
}

public class Demo {
        public static void main(String[] args)
        {
                // this would invoke default constructor.
                Cons2 c2 = new Cons2();

                //  constructor provides the default
                // values to the object like 0, null
                System.out.println(c2.name);
                System.out.println(c2.num);
```

```
    }
}
```

Output:

```
Constructor called
null
0
```

Note:

A constructor that is explicitly defined by the programmer with no parameters is a no-arg constructor. It's defined by the programmer, not generated by the compiler.

Parameterized constructor:

A Constructor with arguments(or parameters) is known as parameterized constructors

```
//program to illustrate parameterized constructor

public class Employee
{

  int empId;
  String empName;

  //parameterized constructor with two parameters
  Employee(int id, String name)
  {
     empId=id;
     empName = name;
  }
  void info()
  {
```

```
        System.out.println("Id: "+empId+" Name: "+empName);
    }

    public static void main(String args[])
    {
            Employee obj1 = new Employee(10245,"pavan");
            Employee obj2 = new Employee(92232,"kumar");
            obj1.info();
            obj2.info();
    }
}
```

Output:

```
Id: 10245 Name: pavan
Id: 92232 Name: kumar
```

Default Constructor:

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.

The default constructor initializes any uninitialized instance variables with default values

```
public class Main {

  int a;  //instance variable
  boolean b;  //instance variable



  public static void main(String[] args) {

    // A default constructor is called
    Main obj = new Main();

    System.out.println("Default Value:");
```

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

    Main obj2= new Main();
  }
}
```

Output:

Default Value:
a = 0
b = false

Note:

By default, all the default constructors are no-arg constructors, but, all no-arg constructors are not default constructors. some no-arg constructors are provided by the compiler are called as Default Constructors.

Constructor overloading:

we can create two or more constructors with different parameters. This is called constructor overloading.(remember same name)

The Java platform differentiates constructors on the basis of the number of arguments in the list and their types.

```
//program to illustrate constructor overloading
public class Demo2 {

  String language;

  // constructor with no parameter
  Demo2() {
    this.language = "Java";
  }
```

```
  // constructor with a single parameter
  Demo2(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    // call constructor with no parameter
    Demo2 obj1 = new Demo2();

    // call constructor with a single parameter
    Demo2 obj2 = new Demo2("Python");

    obj1.getName();
    obj2.getName();
  }
}
```

Output:

```
Programming Langauage: Java
Programming Langauage: Python
```

Constructors vs methods:

Constructors	Methods
A constructor in Java is similar to a method that **can be used to set initial values for object attributes**	A method is a block of code or collection of statements that **can be used to perform a certain task or operation**
A constructor name and class name must be same	A method name and class name can be same or different
A constructor is called at the time of creating the object and it will be executed automatically	A method can be called after creating the object and is executed only when we call it.
A constructor is called only once per object	A method can be called several time on the object depending on our requirement.

this keyword:

In Java, this keyword is used to refer to the current object inside a method or a constructor.

There are various situations where this keyword is commonly used.

- Using this for Ambiguity Variable Names
- Using this in setters and getters methods of a class
- Using this in Constructor Overloading
- Passing this as an Argument

Ex:

```
//program to illustrate this keyword
//ambiguity in names between instance variable and the parameter
//Using this for Ambiguity Variable Names
public class Test {
  int age;
```

```
  Test(int age) {
    this.age = age;
    // age = age; // try with ;
  }

  public static void main(String[] args) {
    Test obj = new Test(8);
    System.out.println("obj.age = " + obj.age);
  }
}

/*
 * if we use age = age
 * In the above example, we have passed 8 as a value to the constructor.
 * However, we are getting 0 as an output. This is because the Java compiler gets confused because of the ambiguity in names between instance  variable and the parameter.
 */
```

Output:

```
obj.age = 8
```

Ex:

```
//program to illustrate usage of this keyword in constructor overloading
class Test
{
	Test()
	{
		this(10,20);
		System.out.println("I am from Default Constructor");
	}
	Test(int x)
	{
		this();
		System.out.println("I am from Single Parameter Constructor");
	}
	Test(int x, int y)
	{
		System.out.println("I am from Double Parameter Constructor");
	}
}
public class ThisConDemo
{
	public static void main(String args[])
	{
		Test t1 = new Test(10);
	}
}
```

Output:

```
I am from Double Parameter Constructor
I am from Default Constructor
I am from Single Parameter Constructor
```

Ex3:

```
//program to illustrate this keyword
//program to illustrate Passing this as an Argument

class ThisExample {
   // declare variables
   int x;
   int y;

   ThisExample(int x, int y) {
     // assign values of variables inside constructor
      this.x = x;
      this.y = y;

      // value of x and y before calling add()
      System.out.println("Before passing this to addTwo() method:");
      System.out.println("x = " + this.x + ", y = " + this.y);

      // call the add() method passing this as argument
      add(this);

      // value of x and y after calling add()
      System.out.println("After passing this to addTwo() method:");
      System.out.println("x = " + this.x + ", y = " + this.y);
   }

   void add(ThisExample o){
      o.x += 2;
      o.y += 2;
   }
}

public class Test4 {
   public static void main( String[] args ) {
      ThisExample obj = new ThisExample(1, -2);
   }
}
```

Output:

Before passing this to addTwo() method:
x = 1, y = -2
After passing this to addTwo() method:
x = 3, y = 0

Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Nested classes are divided into two categories:

non-static and

static.

Non-static nested classes are called *inner classes*. Nested classes that are declared static are called *static nested classes*.

Inner Classes

an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
   ...
   class InnerClass {
      ...
   }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

```
//program to illustrate inner classes

public class InnerClassDemo
  {
   private int x= 1;

   public static void main(String args[]){
      InnerClassDemo obj = new InnerClassDemo();
      InnerClassDemo.MyInnerClassDemo inner = obj.new
MyInnerClassDemo();

    // InnerClassDemo.MyInnerClassDemo inner = new
InnerClassDemo().new MyInnerClassDemo();
      inner. seeOuter();
   }
```

```
   // inner class definition
   class MyInnerClassDemo {
      public void seeOuter () {
        System.out.println("Outer Value of x is :" + x);
      }
   } // close inner class definition
} // close Top level class definition
```

Output:

Outer Value of x is :1

Static Nested Classes:

- As with static members, these belong to their enclosing class(outer class), and not to an instance of the class
- They can have all types of access modifiers in their declaration
- They only have access to static members in the enclosing class
- They cannot refer directly to instance variables or methods defined in its enclosing class. They can use them only through an object reference
- They can define both static and non-static members

```
public class Enclosing {

   private static int x = 1;

   public static class StaticNested {

      private void run() {
         // method implementation
         System.out.println("Static Nested class run method");
      }
```

```
    }


    public static void main(String[] args){
        Enclosing.StaticNested nested = new Enclosing.StaticNested();
        nested.run();
    }
}
```

Output:

Static Nested class run method