

JAVA

Classes and Objects- Classes, Objects, Creating objects, methods, constructors- constructor overloading, cleaning up unused objects- Garbage collector, class variable and methods, method overloading, static keyword, this keyword, arrays, Command line arguments, String and StringTokenizer.

Class and Object

- ✓ Classes and objects are the fundamental components of OOP's. **CLASS** are a blueprint or a set of instructions to build a specific type of object. It is a basic concept of Object-Oriented Programming which revolve around the real-life entities. Class in Java determines how an object will behave and what the object will contain.
- ✓ Object-oriented programming System(OOPs) is a programming paradigm based on the concept of "objects" that contain data and methods.
- ✓ The primary purpose of object-oriented programming is to increase the flexibility and maintainability of programs. Object oriented programming brings together data(variables) and its behaviour(methods) in a single location(object) makes it easier to understand how a program works.
- ✓ Everything in Java is associated with classes and objects, along with its attributes and methods. **For example:** in real life, a car is an object. The car has **attributes(variables)**, such as weight and color, and **methods**, such as drive and brake.

To create a class, use the keyword `class`

It means we can create a class with name car and it contain weight and colour as variables and drive (),break() as methods.

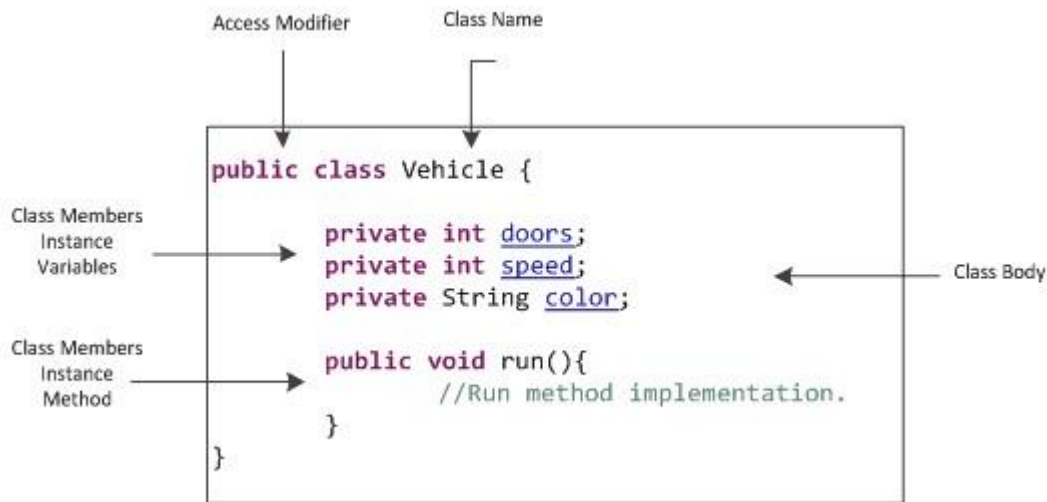
Syntax:

```
class <class_name>{    field;    method;
}
```

Declaration of Class:

A class is declared by use of the class keyword. The class body is enclosed between curly braces { and }. The data or variables, defined within a class are called instance variables.

The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.



Example:

```
public class Main {  int x
= 5;--->variables  void
disp()--->methods
{
    System.out.print(x);
}
    public static void main(String[] args) ---->main method where the execution
starts
    {
        Main k=new Main();--->object creation
k.disp();
    }
}
```

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.

- ✓ An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful.
- ✓ For example color name, table, bag, barking. When you send a message to an object, you are asking the object to invoke or execute one of its methods as defined in the 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 behavior*.
- ✓ The object is *an instance of a class*.

Syntax

```
ClassName Object_name = new ClassName();
```

Object: is a bundle of data(variable) and its behaviour(methods) derived from class. Objects have two characteristics: They have **states** and **behaviors**.

Examples of states and behaviors

Example 1:

Class : House Object:

tejaHouse **State:** city,
Color.

Behavior: Open door, close door, getdata, display

So if I had to write a class based on states and behaviours of House. I can do it like this: States can be represented as instance variables and behaviours as methods of the class.

```
import java.util.Scanner;

class House{
    String city;
    String color;
    Scanner input = new Scanner(System.in);

    void getdata()
    {
        System.out.print("Enter the city name and color of the house \n");
        city = input.next();          color = input.next();
    }

    void display()
    {
        System.out.print("The city name and color name of the house is \n "
+city+"\n"+color+"\n");
    }

    void openDoor()
    {
        System.out.println("The door is opened \n");
    }
}
```

```

    }

    void closeDoor()
    {
        System.out.println("The door is closed \n");
    }

    public static void main(String arh[])
    {
        House tejaHouse=new House();
        House raviHouse=new House();

        tejaHouse.getdata();
        raviHouse.getdata();

        System.out.println("The tejaHome details
are:");    tejaHouse.display();
tejaHouse.openDoor();
        tejaHouse.closeDoor();

        System.out.println("The   raviHome   details
are:");           raviHouse.display();
raviHouse.openDoor();
raviHouse.closeDoor(); }
    }

```

Output:

Enter the city name and color of the
house bhimavaram black

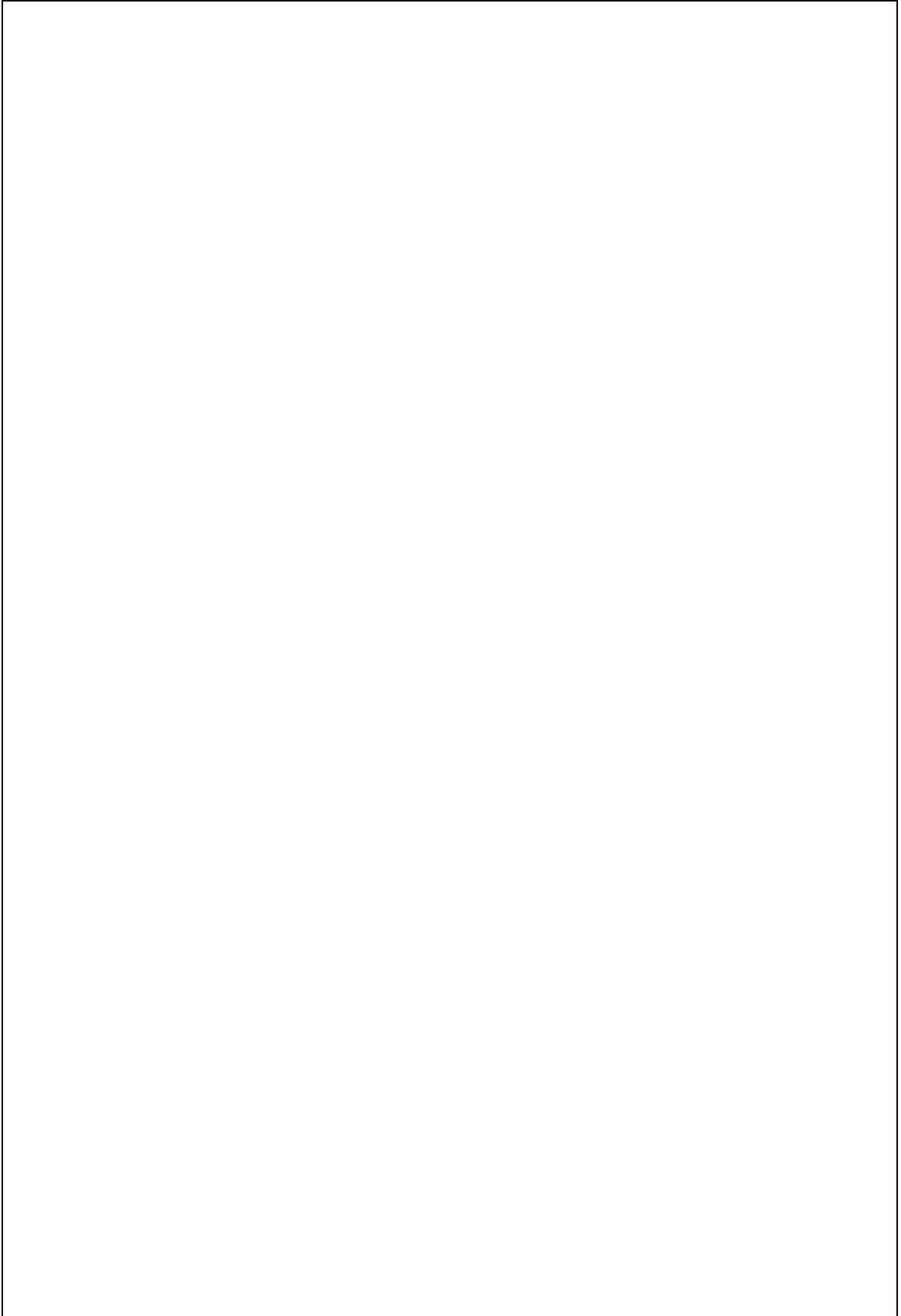
Enter the city name and color of the house
elur
blue

The tejaHome details are:

The city name and color name of the house is
bhimavaram
black
The door is opened
The door is closed

The raviHome details are:

The city name and color name of the
house is elur blue
The door is opened
The door is closed



In the above program I have created two objects(**tejaHome** and **raviHome**) for the class Home. Every object is having their own properties.i.e **tejaHome** is having its city.color,getData(),display(),openDoor(),closeDoor() and the object **raviHome** is also having its own members/properties city.color, getData(), display(), openDoor(), closeDoor().

Hence we conclude that every object is having its own members which are declared in the class.

What is the Difference between Object & Class?

- ✓ A **class** is a **blueprint or prototype** that defines the variables and the methods (functions) common to all objects of a certain kind.
- ✓ An **object** is a specimen of a class. Software objects are often used to model real-world objects you find in everyday life.
- ✓ If we have a blueprint of a house it means we can create any number of houses by using that blue prints. Every individual house is having its own properties linke door,color,kitchn etc.
- ✓ In similar way if we have a single class then we can create any number of objects by using given class. Every objects are having its own independent properties.
- ✓ In the above example one class House can create two objects **tejaHome, raviHome**. These two objects are having its own properties i.e variables and methods.

Java- method

In mathematics, we might have studied about functions. For example, $f(x) = x^2$ is a function that returns a squared value of x .

If $x = 2$, then $f(2) = 4$
If $x = 3$, $f(3) = 9$
and so on.

Similarly, in computer programming, a function is a block of code that performs a specific task.

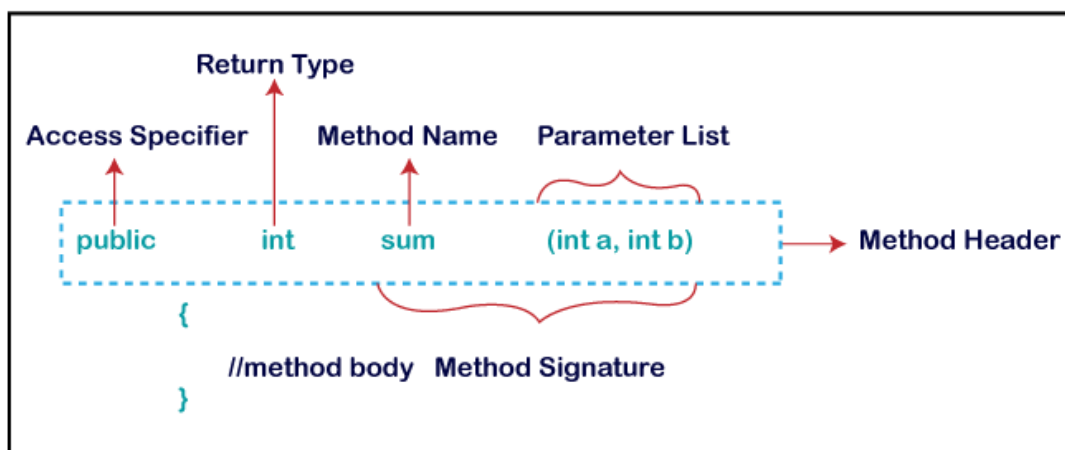
Methods in java

- ✓ In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **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. It is used to achieve the **reusability** of code.
- ✓ We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.

Method Declaration



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier: ○

Public: The method is accessible by all classes when we use public specifier in our application.

- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

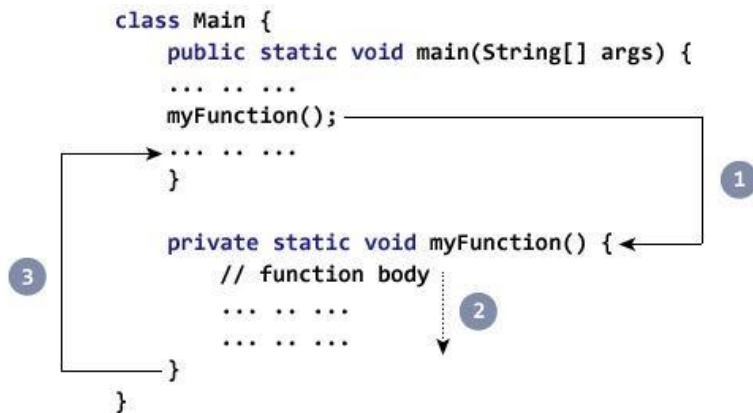
Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

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

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()



Types of methods: there are two types of methods in java. Those are

- 1. Predefined methods**
- 2. User defined methods**

Predefined Method

- ✓ In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**.
- ✓ We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length()**, **equals()**, **compareTo()**, **sqrt()**, etc.
- ✓ When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.
- ✓ Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

```

public class Demo
{
    public static void main(String[] args)
    {
        // using the max() method of Math class
        System.out.print("The maximum number is: " + Math.max(9,7));
    }
}

```

```
}  
}
```

Output:

The maximum number is: 9

In the above example, we have used three predefined methods **main()**, **print()**, and **max()**. We have used these methods directly without declaration because they are predefined. The **print()** method is a method of **PrintStream** class that prints the result on the console. The **max()** method is a method of the **Math** class that returns the greater of two numbers.

If see the **max()** method signature, we find the following:

```
max  
  
public static int max(int a,  
                      int b)  
  
Returns the greater of two int values.  
same value.  
  
Parameters:  
a - an argument.  
b - another argument.  
  
Returns:  
the larger of a and b.
```

In the above method signature, we see that the method signature has access specifier **public**, non-access modifier **static**, return type **int**, method name **max()**, parameter list **(int a, int b)**.

In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.

User-defined Method

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

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
public static void findEvenOdd(int num)
{
//method body  if(num%2==0)
System.out.println(num+" is even");  else
System.out.println(num+" is odd");  }
```

We have defined the above method named `findevenodd()`. It has a parameter **num** of type `int`. The method does not return any value that's why we have used `void`. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

How to Call or Invoke a User-defined Method

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from the user  int
num=scan.nextInt();
//method calling
findEvenOdd(num);
}
```

In the above code snippet, as soon as the compiler reaches at line **findEvenOdd(num)**, the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

Example-1:EvenOdd.java

```

import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from user int
        num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }

    //user defined method
    public static void findEvenOdd(int num)
    {
        //method body if(num%2==0)
        System.out.println(num+" is even");    else
        System.out.println(num+" is odd");
    }
}

```

Output 1:

Enter the number: 12 12
is even

Output 2:

Enter the number: 99 99
is odd

Let's see another program that return a value to the calling method.

In the following program, we have defined a method named **add()** that sum up the two numbers. It has two parameters n1 and n2 of integer type. The values of n1 and n2 correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

Example-2:Addition.java

```

public class Addition
{
    public static void main(String[] args)
    {
        int a = 19;
        int b = 5;
        //method calling
        int c = add(a, b); //a and b are actual parameters
        System.out.println("The sum of a and b is= " + c);
    }
    //user defined method
    public static int add(int n1, int n2) //n1 and n2 are formal parameters
    {
        int s;
        s=n1+n2;
        return s; //returning the sum
    }
}

```

Output:

The sum of a and b is= 24

Example3: Java Method

Let's see how we can use methods in a Java program.

```

class Main {

    public static void main(String[] args) {
        System.out.println("About to encounter a method.");

        // method call
        myMethod();

        System.out.println("Method was executed successfully!");
    }

    // method definition
    private static void myMethod(){
        System.out.println("Printing from inside myMethod()!");
    }
}

```

Output:

About to encounter a method.
Printing from inside myMethod().
Method was executed successfully!

Constructor in Java

- ✓ **CONSTRUCTOR** is a special method that is used to initialize a newly created object and is called just after the memory is allocated for the object.
- ✓ It can be used to initialize the objects to desired values or default values at the time of object creation. It is not mandatory for the coder to write a constructor for a class.
- ✓ There is no need to call the constructor by the user. Whenever object is created by new operator then the corresponding constructor will be executed.
- ✓ If no user-defined constructor is provided for a class, compiler will create default constructor and initializes member variables to its default values.
 - numeric data types are set to 0
 - char data types are set to null character („\0“)
 - reference variables are set to null

(or)

- ✓ In Java, a constructor is a block of codes similar to the method. It is called when an instance(object) of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- ✓ It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

Rules for creating a Java Constructor

1. It has the **same name** as the class
2. It should not return a value not even **void**
3. A Java constructor cannot be abstract, static, final, and synchronized

Syntax:

```
public class MyClass{  
    //This is the constructor  
    MyClass(){  
    }  
    ...  
}
```

Note that the constructor name matches with the class name and it doesn't have a return type.

How does a constructor work

we create the object of `MyClass` like this:

```
MyClass obj = new MyClass();
```

The **new keyword** here creates the object of class `MyClass` and invokes the constructor to initialize this newly created object.

Example1:

```
public class MyClass{  
    // Constructor  
    MyClass(){  
        System.out.println("BeginnersBook.com");  
    }  
    public static void main(String args[]){  
        MyClass obj = new MyClass();  
        ...  
    }  
}
```

Beginnersbook.com

←

...

New keyword creates the object of `MyClass` & invokes the constructor to initialize the created object.

Example2:

```
class Demo{
    int value1:
    int value2:
    Demo(){
        value1 = 10:
        value2 = 20:
        Svstem.out.println("Inside Constructor"):
    }

    public void displav(){
        Svstem.out.println("Value1 === "+value1):
        Svstem.out.println("Value2 === "+value2):
    }

    public static void main(String aras[]){
        Demo d1 = new Demo():
        d1.displav():
    }
}
```

Output:

```
Inside Constructor
Value1 === 10
Value2 === 20
```

Example3:

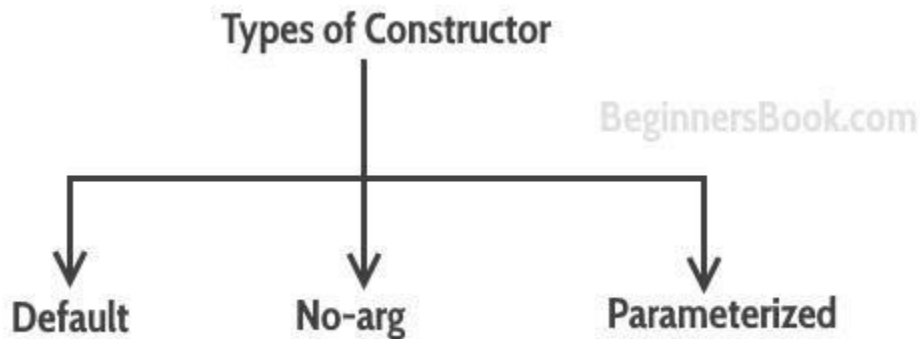
```
public class Hello {
    String name:
    //Constructor
    Hello(){
        this.name = "java":
    }
    public static void main(String[] aras) {
        Hello obi = new Hello():
        Svstem.out.println(obi.name):
    }
}
```

Output:

```
java
```

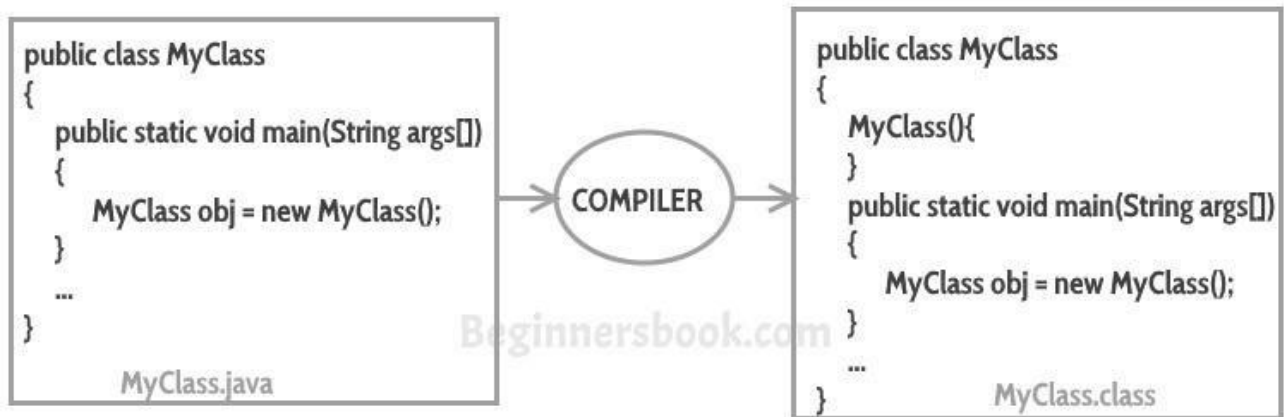
Types of Constructors

There are three types of constructors: Default, No-arg constructor and Parameterized.



Default constructor

- ✓ If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf. This constructor is known as default constructor.
- ✓ You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file. This process is shown in the diagram below:



The following program doesn't have any constructor.

```

class Student3{
int id;
String name; void
display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student3 s1=new
Student3(); Student3
s2=new Student3();
s1.display(); s2.display();
}
}

```

Output: 0 null

0 null **no-arg constructor:**

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

Example: no-arg constructor

```

class Demo
{
    public Demo()
    {
        System.out.println("This is a no argument constructor");
    }
    public static void main(String args[]) {
        new Demo();
    }
}

```

Output:

This is a no argument constructor

Parameterized constructor

Constructor with arguments(or you can say parameters) is known as Parameterized constructor.

Example: parameterized constructor

In this example we have a parameterized constructor with two parameters `id` and `name`. While creating the objects `obj1` and `obj2` I have passed two arguments so that this constructor gets invoked after creation of `obj1` and `obj2`.

```

public class Employee {

    int empId;
    String empName;

    //parameterized constructor with two parameters
    Employee(int id, String name){
        this.empId = id;
    }
    this.empName = name;
    void info(){
        System.out.println("Id: "+empId+" Name: "+empName);
    }

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

```

Output:

```

Id: 10245 Name: Chaitanva
Id: 92232 Name: Negan

```

Example2: parameterized constructor

In this example, we have two constructors, a default constructor and a parameterized constructor. When we do not pass any parameter while creating the object using new keyword then default constructor is invoked, however when you pass a parameter then parameterized constructor that matches with the passed parameters list gets invoked.

```
class Example2
{
    private int var;
    //default constructor
    public Example2()
    {
        this.var = 10;
    }
    //parameterized constructor
    public Example2(int num)
    {
        this.var = num;
    }
    public int getValue()
    {
        return var;
    }
    public static void main(String args[])
    {
        Example2 obj = new Example2();
        Example2 obj2 = new Example2(100);
        System.out.println("var is: "+obj.getValue());
        System.out.println("var is: "+obj2.getValue());
    }
}
```

Output:

```
var is: 10
var is: 100
```


Constructor Overloading

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter list. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Syntax:

Beginnersbook.com

```
public class Demo {  
    Demo() {  
        ..  
    }  
    Demo(String s) {  
        ...  
    }  
    Demo(int i) {  
        ...  
    }  
    .....  
}
```



Three overloaded constructors -
They must have
different
Parameters list

Examples of valid constructors for class Account are

```
Account(int a);  
Account (int a,int b);  
Account (String a,int b);
```

Example:

```
class Main{    int  
value1;    int value2;  
Main(){    value1 =  
10;    value2 = 20;  
    System.out.println("Inside 1st Constructor default Constructor");  
    }  
    Main(int a){    value1  
= a;  
    System.out.println("Inside 2nd Constructor-one parameter constructure");  
    }  
    Main(int a,int b){  
value1 = a;    value2 =  
b;  
    System.out.println("Inside 3rd Constructor-two parameter constructor");  
    }  
    public void display(){  
        System.out.println("Value1 === "+value1);  
        System.out.println("Value2 === "+value2+"\n");  
    }  
    public static void main(String args[]){
```

```
    Main d1 = new Main();  
d1.display();  
    Main d2 = new Main(30);  
d2.display();  
    Main d3 = new Main(30,40);  
    d3.display();  
}  
}
```

Output:

Inside 1st Constructor default Constructor

Value1 === 10

Value2 === 20

Inside 2nd Constructor-one parameter constructure

Value1 === 30

Value2 === 0

Inside 3rd Constructor-two parameter constructor

Value1 === 30

Value2 === 40

Java Copy Constructor

A copy constructor is used for copying the values of one object to another object.

```

class JavaExample{
    String web;
    JavaExample(String w){
        web = w;
    }

    /* This is the Copy Constructor, it
    *   copies the values of one object
    *   to the another object (the object
    *   that invokes this constructor)   */
    JavaExample(JavaExample je){
        web = je.web;
    }
    void disp(){
        System.out.println("Website: "+web);
    }

    public static void main(String args[]){
        JavaExample obj1 = new JavaExample("I love java programming");

        /* Passing the object as an argument to the constructor
    *   This will invoke the copy constructor
    */
        JavaExample obj2 = new JavaExample(obj1);
        obj1.disp():
        obj2.disp():
    }
}

```

Output:

Website: I love java programming
Website: I love java programming

Quick Recap about constructor:

1. Every class has a constructor whether it's a normal class or a abstract class.
2. Constructors are not methods and they don't have any return type.
3. Constructor name should match with class name .
4. Constructor can use any access specifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.
5. **Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.**
6. If you don't implement any constructor within the class, compiler will do it for.

7. Constructor overloading is possible but overriding is not possible. Which means we can have overloaded constructor in our class but we can't override a constructor.
8. Constructors can not be inherited.
9. A constructor can also invoke another constructor of the same class – By using `this()`. If you want to invoke a parameterized constructor then do it like this: **this(parameter list)**.

Garbage Collection in Java

- ✓ When JVM starts up, it creates a heap area which is known as runtime data area. This is where all the objects (instances of class) are stored. Since this area is limited, it is required to manage this area efficiently by removing the objects that are no longer in use.
- ✓ The process of removing unused objects from heap memory is known as **Garbage collection** and this is a part of memory management in Java.

(Or)

- ✓ **Garbage Collection in Java** is a process by which the programs perform memory management automatically. The Garbage Collector(GC) finds the unused objects and deletes them to reclaim the memory.
- ✓ In Java, dynamic memory allocation of objects is achieved using the `new` operator that uses some memory and the memory remains allocated until there are references for the use of the object.
- ✓ When there are no references to an object, it is assumed to be no longer needed, and the memory, occupied by the object can be reclaimed. There is no explicit need to destroy an object as Java handles the de-allocation automatically.
- ✓ The technique that accomplishes this is known as **Garbage Collection**. Programs that do not de-allocate memory can eventually crash when there is no memory left in the system to allocate. These programs are said to have *memory leaks*.
- ✓ **Garbage collection in Java happens automatically** during the lifetime of the program, eliminating the need to de-allocate memory and thereby avoiding memory leaks.
- ✓ In C language, it is the programmer's responsibility to de-allocate memory allocated dynamically using `free()` function. This is where Java memory management leads.
- ✓ Note: All objects are created in **Heap** Section of memory.

- ✓ Languages like C/C++ **don't** support automatic garbage collection, however in java, the garbage collection is automatic.

(OR)

- ✓ In java, garbage means unreferenced objects.
- ✓ Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- ✓ To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- ✓ It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- ✓ It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

When does java perform garbage collection?

There are many ways:

- By nulling the reference
- By assigning a reference to another

1. When the object is no longer reachable:

Example-1:

```
Book obj = new Book();  
obj = null;
```

Here the reference obj was pointing to the object of class Book but since we have assigned a null value to it, this is no longer pointing to that object, which makes the Book object unreachable and thus unusable. Such objects are automatically available for garbage collection in Java.

Example-2:

```
char[] savhello = { 'h', 'e', 'l', 'l', 'o'}:  
String str = new String(savhello);  
str = null;
```

Here the reference str of String class was pointing to a string "hello" in the heap memory but since we have assigned the null value to str, the object "hello" present in the heap memory is unusable.

Example-3:

```
Employee e=new Employee();  
e=null;
```

2) By assianina a reference to another:

Example-1:

```
Employee e1=new Employee();  
Employee e2=new Employee();  
e1=e2;//now the first object referred by e1 is available for garbage collection
```

Example-2:

```
Book obj1 = new Book();  
Book obj2 = new Book();  
obj2 = obj1;
```

Here we have assigned the reference obj1 to obj2, which means the instance (object) pointed by (referenced by) obj2 is not reachable and available for garbage collection.

How to request JVM for garbage collection(if user wants to initiate garbage collection)

- ✓ We now know that the unreachable and unusable objects are available for garbage collection but the garbage collection process doesn't happen instantly. Which means once the objects are ready for garbage collection they must have to wait for JVM to run the memory cleanup program that performs garbage collection.
- ✓ However you can request to JVM for garbage collection by calling **System.gc()** method (see the example below).

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}  

```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}  

```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Garbage Collection by user Example in Java

In this example we are demonstrating the garbage collection by calling System.gc(). In this code we have overridden a finalize() method. This method is invoked just before a object is destroyed by java garbage collection process. This is the reason you would see in the output that this method has been invoked twice.

```

public class JavaExample{    public
static void main(String args[]){
    /* Here we are intentionally assigning a null
    * value to a reference so that the object becomes
    * non reachable
    */
    JavaExample obj=new JavaExample();
    obj=null;

    /* Here we are intentionally assigning reference a
    * to the another reference b to make the object referenced      * by b
    unusable.
    */
    JavaExample a = new JavaExample();
    JavaExample b = new JavaExample();
    b = a;
    System.gc();
}    protected void finalize() throws
Throwable
{
    System.out.println("Garbage collection is performed by JVM");
}
}

```

Output:

```

Garbage collection is performed by JVM
Garbage collection is performed by JVM

```

Example2:

```

public class TestGarbage1{    public void
finalize(){System.out.println("object is garbage collected");}    public
static void main(String args[]){    TestGarbage1 s1=new    public
TestGarbage1();    TestGarbage1 s2=new TestGarbage1();    s1=null;
s2=null;
    System.gc();
}
}

```

Output:

```

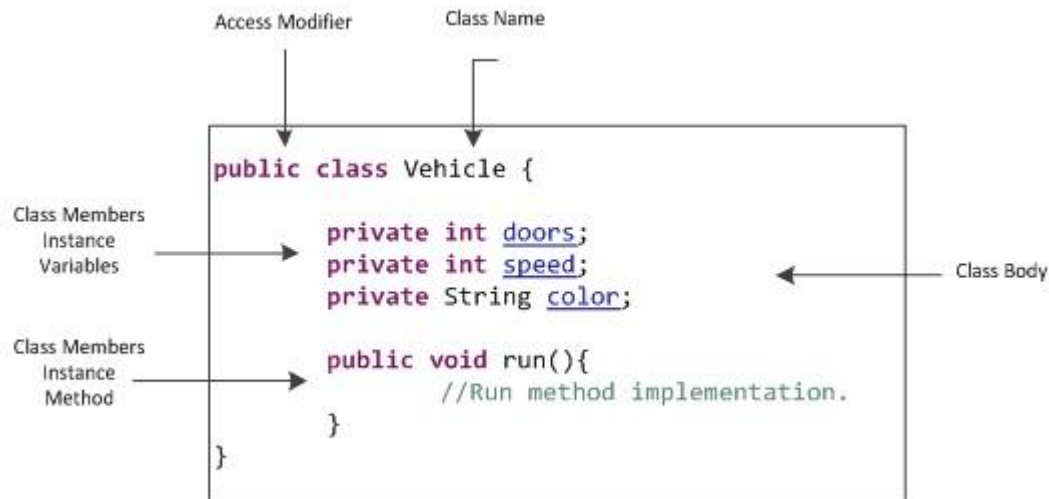
object is garbage collected object
is garbage collected

```

class variable and methods: to understand the class variable and methods first we have to learn declaration of class

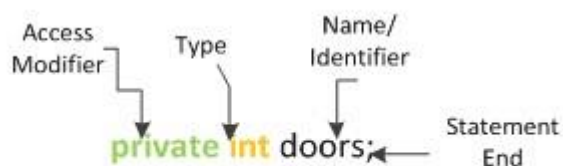
Declaration of Class:

A class is declared by use of the class keyword. The class body is enclosed between curly braces { and }. The data or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.



Declaration of Instance Variables :

- ✓ Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- ✓ Thus, the data for one object is separate and unique from the data for another object. An instance variable can be declared public or private or default (no modifier).
- ✓ When we do not want our variable's value to be changed out-side our class we should declare them private. public variables can be accessed and changed from outside of the class. We will have more information in OOP concept tutorial. The syntax is shown below.



What are class variables, instance variables and local variables in Java?

A variable provides us with named storage that our programs can manipulate. Java provides three types of variables.

- **Class variables** – Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block. There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- **Instance variables** – Instance variables are declared in a class, but outside a method. When space is allocated for an object in the heap, a slot for each instance variable value is created. Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- **Local variables** – Local variables are declared in methods, constructors, or blocks. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.

Example

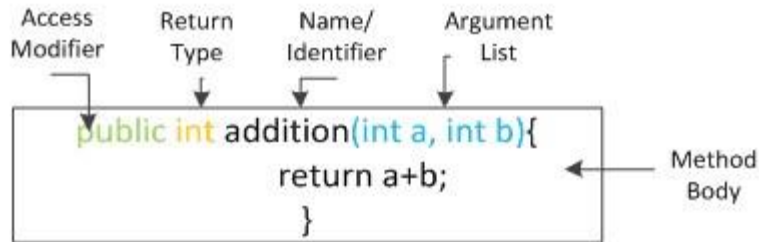
```
public class VariableExample{    int myVariable;  
→instance variable    static int data = 30;→class  
variable or static variable  
  
    public static void main(String args[]){  
        int a = 100; →Local variable  
        VariableExample obj = new VariableExample();  
  
        System.out.println("Value of instance variable myVariable: "+obj.myVariable);  
        System.out.println("Value of static variable data: "+VariableExample.data);  
        System.out.println("Value of local variable a: "+a);  
    }  
}
```

Output

```
Value of instance variable myVariable: 0  
Value of static variable data: 30 Value of local variable a: 100
```

Declaration of Methods :

- ✓ A method is a program module that contains a series of statements that carry out a task. To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method.
- ✓ Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times. The syntax to declare method is given below.



Java - Methods

- ✓ A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println()` method, for example, the system actually executes several statements in order to display a message on the console.
- ✓ Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating Method

Syntax

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

The syntax shown above includes –

- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.
- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

Considering the following example to explain the syntax of a method –

Example:

```
public static int methodName(int a, int b) {  
    // body  
}
```

Here,

- **public static** – modifier
- **int** – return type
- **methodName** – name of the method
- **a, b** – formal parameters
- **int a, int b** – list of parameters

Method definition consists of a method header and a method body.

Example for crating method:

Here is the source code of the above defined method called **min()**. This method takes two parameters num1 and num2 and returns the maximum between the two

```
/** the snippet returns the minimum between
two numbers */ public static int minFunction(int
n1, int n2) {   int min;   if (n1 > n2)       min =
n2;   else
    min = n1;

    return min; }
```

Example java program for methods:

```
public class ExampleMinNumber {

    public static void main(String[] args) {
int a = 11;    int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }
}
```

```
/** returns the minimum of two numbers */ public static int minFunction(int n1, int
n2) {   int min;   if (n1 > n2)       min = n2;   else
    min = n1;

    return min;
}
}
```

This will produce the following result –

Output

Minimum value = 6

Method Overloading in Java

- ✓ Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.
- ✓ It is similar to [constructor overloading](#) in Java, that allows a class to have more than one constructor having different argument lists.
- ✓ For example the argument list of a method **add(int a, int b)** having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

Or

- ✓ If a **class** has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- ✓ If we have to perform only one operation, having same name of the methods increases the readability of the **program**.
- ✓ 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 for you as well as other programmers to understand the behavior of the method because its name differs.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters. For example:

```
add(int, int) add(int,
float)
```

3. Sequence of Data type of parameters. For example:

```
add(int, float) add(float,  
int)
```

Invalid case of method overloading:

When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)  
float add(int, int)
```

Method overloading is an example of [Static Polymorphism](#)..

Points to Note:

1. Static Polymorphism is also known as compile time binding or early binding.
2. [Static binding](#) happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading examples method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```

class DisplayOverloading
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(char c, int num)
    {
        System.out.println(c + " "+num);
    }
}
class Sample
{
    public static void main(String args[])
    {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
    }
}

```

Output:

```

a a
10

```

In the above example – method `disp()` is overloaded based on the number of parameters – We have two methods with the name `disp` but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading – Difference in data type of parameters

In this example, method `disp()` is overloaded based on the data type of parameters – We have two methods with the name `disp()`, one with parameter of `char` type and another method with the parameter of `int` type.

```

class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
        System.out.println(c );
    }
}

```

```

}

class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();    obj.disp('a');
obj.disp(5);
    }
}

```

Output:

```

a
5

```

Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```

class DisplayOverloading3
{
    public void disp(char c, int num)
    {
        System.out.println("I'm the first definition of method disp");
    }
    public void disp(int num, char c)
    {
        System.out.println("I'm the second definition of method disp" );
    }
}
class Sample3
{
    public static void main(String args[])
    {
        DisplayOverloading3 obj = new DisplayOverloading3();
obj.disp('x', 51 );    obj.disp(52, 'y');
    }
}

```

Output:

```

I'm the first definition of method disp
I'm the second definition of method disp

```

Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

What it has to do with method overloading?

Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.

Lets take an example to see what I am talking here:

```
class Demo{
    void disp(int a, double b){
        System.out.println("Method A");
    }
    void disp(int a, double b, double c){
        System.out.println("Method B");
    }
    public static void main(String args[]){
        Demo obi = new Demo();
        /* I am passing float value as a second argument but
        * it got promoted to the type double. because there
        * wasn't any method having arg list as (int, float)
        */
        obi.disp(100, 20.67f);
    }
}
```

Output:

Method A

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn't any method with argument list as (int, float)

But this type promotion doesn't always happen, lets see another example:

```
class Demo{
    void disp(int a, double b){
        System.out.println("Method A");
    }
}
```

```

void disp(int a, double b, double c){
    System.out.println("Method B");
}
void disp(int a, float b){
    System.out.println("Method C");
}
public static void main(String args[]){
    Demo obi = new Demo();
    /* This time promotion won't happen as there is
     * a method with arg list as (int, float)
     */
    obi.disp(100, 20.67f);
}
}

```

Output:

Method C

As you see that this time type promotion didn't happen because there was a method with matching argument type.

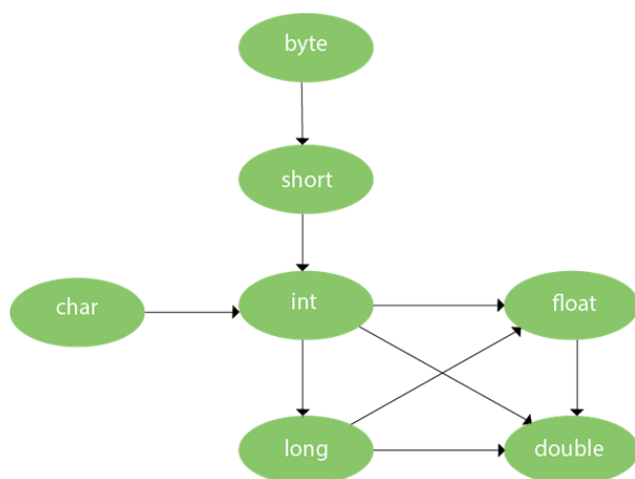
Type Promotion table:

The data type on the left side can be promoted to the any of the data type present in the right side of it.

```

byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double

```



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on.

Example of Method Overloading with TypePromotion

```
class OverloadingCalculation1{
    void sum(int a,long b){System.out.println(a+b);}
    void sum(int a,int b,int c){System.out.println(a+b+c);}

    public static void main(String args[]){
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20);//now second int literal will be promoted to long
        obj.sum(20,20,20);
    }
}
```

Output: 40
60

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
class OverloadingCalculation2{    void sum(int a,int
b){System.out.println("int arg method invoked");}    void sum(long a,long
b){System.out.println("long arg method invoked");}

    public static void main(String args[]){
        OverloadingCalculation2    obj=new    OverloadingCalculation2();
        obj.sum(20,20);//now int arg sum() method gets invoked
    }
}
```

Output: int arg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OverloadingCalculation3{    void sum(int a,long
b){System.out.println("a method invoked");}    void sum(long
a,int b){System.out.println("b method invoked");}

    public static void main(String args[]){
        OverloadingCalculation3 obj=new OverloadingCalculation3();
        obj.sum(20,20);//now ambiguity
    }
}
```

Output: Compile Time Error

Lets see few Valid/invalid cases of method overloading Case

1:

```
int mymethod(int a, int b, float c) int mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

```
int mymethod(int a, int b) int  
mymethod(float var1, float var2)
```

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different. **Case 3:**

```
int mymethod(int a, int b) int  
mymethod(int num)
```

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

```
float mymethod(int a, float b) float  
mymethod(float var1, int var2)
```

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

```
int mymethod(int a, int b) float  
mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn't matter while overloading a method.

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

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

Static keyword

Static keyword can be used with class, variable, method and block. Static members belong to the class instead of a specific instance(object), this means if you make a member static, you can access it without object. The static can be:

1. Static Variables
2. Static Methods
3. Static Blocks Of Code.

Static Variable

- ✓ **Static variable in Java** is variable which belongs to the class and initialized only once at the start of the execution. It is a variable which belongs to the class and not to object(instance).
- ✓ Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
 - A single copy to be shared by all instances(objects) of the class
 - A static variable can be accessed directly by the class name and doesn't need any object

Syntax :

`Static data type variable-name` → static variable declaration

`<class-name>.<variable-name>` → access static variable

Or

- ✓ A static variable is common to all the instances (or objects) of the class because it is a class level variable.
- ✓ In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

Few Important Points:

- Static variables are also known as Class Variables.
 - Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.
- or**
- 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 your program **memory efficient** (i.e., it saves memory).
- ✓ As I mentioned above that the static variables are shared among all the instances of the class, they are useful when we need to do memory management.
- ✓ In some cases we want to have a common value for all the instances like global variable then it is much better to declare them static as this can save memory (because only single copy is created for static variables).

Static variable example -1:

```
class VariableDemo
{
    static int count=0;
    public void increment()
    {
        count++;
    }
    public static void main(String args[])
    {
        VariableDemo obj1=new VariableDemo();
        VariableDemo obj2=new VariableDemo();
        obj1.increment();
        obj1.increment();
        System.out.println("Obj1: count is="+obj1.count);
        System.out.println("Obj2: count is="+obj2.count);
    }
}
```

Output:

```
Obj1: count is=2
Obj2: count is=2
```

As you can see in the above example that both the objects are sharing a same copy of static variable that's why they displayed the same value of count. Here only obj1 is gets incremented eventhough it effects obj2 also. Hence we can say static variables are common to all objects.

Static variable example -2:

```

import java.util.Scanner; public class
Main{ public static void main(String
args[]){

    Student s1 = new Student();
    s1.getdata();
    System.out.print("The value of S1 are\n:");
    s1.showData();

    Student s2 = new Student();
    s2.getdata();
    System.out.print("\n\nThe value of S2 are:\n");
    s2.showData();

    System.out.print("\n\nThe value of S1 again are:\n");
    s1.showData();

}
}

class Student
{ int a; static
int b;
Scanner scan=new Scanner(System.in);
public void getdata(){
    System.out.print("\n Enter the number: \n");
a=scan.nextInt(); b=scan.nextInt();
}

public void showData(){
    System.out.println("Value of a = "+a);
    System.out.println("Value of b = "+b);
}

//public static void increment(){
//a++;
//}

```

Output

Enter the number: 1

2
The value of S1 are
:Value of a = 1
Value of b = 2

Enter the number:

3

4

The value of S2 are:

Value of a = 3

Value of b = 4

The value of S1 again are:

Value of a = 1

Value of b = 4

Since b is a static variable if we change it by using s2 object then it reflected to s1 also.

Static variable example -3:

```
class Main
{
    static int a=0;
    int b=0;    public void
increment()
    {
a++;
b++;
    }

    void display()
    {
        System.out.println("a="+a+"\n"+"b="+b);
    }

    public static void main(String args[])
    {
        Main obj1=new Main();
        Main obj2=new Main();

        obj1.increment();
obj1.increment();    obj1.increment();

        System.out.println("The values of obj1 are:");
obj1.display();

        System.out.println("The values of obj2 are:");
obj2.display();

    }
}
```

The values of obj1 are:

a=3 b=3

The values of obj2 are:

a=3 b=0

Understanding the problem without static variable

```
class Student{
int rollno;
    String name;
    String college="ITS";
}
```

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](#). If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Static variable example -3:

```
//Java Program to demonstrate the use of static variable
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//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 TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        //we can change the college of all objects by the single line of code
        //Student.college="BBDIT";
        s1.display();
        s2.display();
    }
}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

Static variable example -4:: Static variables are shared among all the instances of class

In this example, String variable is non-static and integer variable is Static. As you can see in the output that the non-static variable is different for both the objects

but the static variable is shared among them, that's the reason the changes made to the static variable by object ob2 reflects in both the objects.

```
class JavaExample{
    //Static integer variable
    static int var1=77;
    //non-static string variable
    String var2;
    public static void main(String args[])
    {
        JavaExample ob1 = new JavaExample();
        JavaExample ob2 = new JavaExample();
        /* static variables can be accessed directly without
        * any instances. Just to demonstrate that static variables
        * are shared, I am accessing them using objects so that
        * we can check that the changes made to static variables
        * by one object, reflects when we access them using other
        * objects
        */
        //Assigning the value to static variable using object ob1
        ob1.var1=88;
        ob1.var2="I'm Object1";
        /* This will overwrite the value of var1 because var1 has a single
        * copy shared among both the objects.
        */
        ob2.var1=99;
        ob2.var2="I'm Object2";
        System.out.println("ob1 integer:"+ob1.var1);
        System.out.println("ob1 String:"+ob1.var2);
        System.out.println("ob2 integer:"+ob2.var1);
        System.out.println("ob2 String:"+ob2.var2);
    }
}
```

Output:

```
ob1 integer:99
ob1 String:I'm Object1
ob2 integer:99
ob2 String:I'm Object2
```

Static variable example -5: Java static and non-static Variables

```
class Test {
    // static variable
    static int max = 10;
    // non-static variable
    int min = 5;
}
```



```

public class Main {
    public static void main(String[] args) {
        Test obj = new Test();
        // access the non-static variable
        System.out.println("min + 1 = " + (obj.min + 1));
        // access the static variable
        System.out.println("max + 1 = " + (Test.max + 1));
    }
}

```

Output:

```

min + 1 = 6
max + 1 = 11

```

In the above program, we have declared a non-static variable named `min` and a static variable named `max` inside the class `Test`.

Inside the `Main` class, we can see that we are calling the non-static variable using the object of the class (`obj.min + 1`). However, we are calling the static variable by using the class name (`Test.max + 1`).

Note: Static variables are rarely used in Java. Instead, the static constants are used. These static constants are defined by `static final` keyword and represented in uppercase. This is why some people prefer to use uppercase for static variables as well.

Static variable initialization

1. Static variables are initialized when class is loaded.
2. Static variables are initialized before any object of that class is created.
3. Static variables are initialized before any static method of the class executes.

Default values for static and non-static variables are same.

primitive integers(long, short etc): **0**

primitive floating points(float, double):

0.0 boolean: **false** object references: **null**

Static final variables

The static final variables are constants. Lets have a look at the code below:

```

public class MyClass{
    public static final int MY_VAR=27;
}

```

Note: Constant variable name should be in Caps! you can use underscore(_) between.

1) The above code will execute as soon as the class `MyClass` is loaded, before static method is called and even before any static variable can be accessed. 2) The variable `MY_VAR` is **public** which means any class can use it. It is a **static** variable so you won't need any object of class in order to access it. It's **final** so the value of this variable can never be changed in the current or in any class.

Key points:

final variable always needs initialization, if you don't initialize it would throw a compilation error. have a look at below example-

```
public class MyClass{
    public static final int MY_VAR;
}
```

Error: variable MY_VAR might not have been initialized

```
class Main{    final int
speedlimit=90;//final variable    void
run(){
    System.out.println(speedlimit);
}
    public static void main(String args[]){
Main obj=new Main();    obj.run();
}
} //end of class    o/p:90
```

Java Static Methods

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.

Static methods can be accessed directly in static and non-static methods.

Syntax:

Static keyword followed by return type, followed by method name.

```
static return_type method_name();
```

Example 1: static method main is accessing static variables without object

```
class JavaExample{    static int i =
10;    static String s =
"Beginnersbook";
```

```
//This is a static method
public static void main(String args[])
{
    System.out.println("i:"+i);
    System.out.println("s:"+s);
}
}
```

Output:

```
i:10
s:Beginnersbook
```

Example 2: Static method accessed directly in static and non-static method

```
class Main{    static
int i = 100;
    static String s = "Beginnersbook";
int a=200;
    static void display()
    {
        System.out.println("\n this is static method");
        System.out.println("i:"+i);
        System.out.println("i:"+s);
    }

    void display1()
    {
        System.out.println("\n this is non-static method");
        System.out.println("i:"+i);
        System.out.println("i:"+s);
        System.out.println("a:"+a);
    }

    public static void main(String args[])
    {
        Main obj = new Main();

        obj.display1();
        display();
    }
}
```

Output:

```
this is non-static  
method i:100  
i:Beginnersbook
```

a:200

this is static method

i:100

i:Beginnersbook

Example 3: Static variables can be accessed directly in Static method

Here we have a static method disp() and two static variables var1 and var2. Both the variables are accessed directly in the static method.

```
class JavaExample3{
    static int var1;
    static String var2;
    //This is a Static Method
    static void disp(){
        System.out.println("Var1 is: "+var1);
        System.out.println("Var2 is: "+var2);
    }
    public static void main(String args[])
    {
        disp();
    }
}
```

Output:

Var1 is: 0

Var2 is: null

Example 4: Static Variable can be accessed directly in a static method

```
class JavaExample{
    static int age;
    static String name;
    //This is a Static Method
    static void disp(){
        System.out.println("Age is: "+age);
        System.out.println("Name is: "+name);
    }
    // This is also a static method
    public static void main(String args[])
    {
        age = 30;
        name = "Steve";
        disp();
    }
}
```

Output:

Age is: 30
Name is: Steve

The difference between regular (non-static) and static methods

Java is a Object Oriented Programming(OOP) language, which means we need objects to access methods and variables inside of a **class**. However this is not always true. While discussing static keyword in java, we learned that static members are class level and can be accessed directly without any instance. In this article we will see the difference between static and non-static methods.

Static Method Example

```
class StaticDemo
{
    public static void copyArg(String str1, String str2)
    {
        //copies argument 2 to arg1
        str2 = str1;
        System.out.println("First String arg is: "+str1);
        System.out.println("Second String arg is: "+str2);
    }
    public static void main(String args[])
    {
        /* This statement can also be written like this:
        * StaticDemo.copyArg("XYZ", "ABC");
        */
        copyArg("XYZ", "ABC");
    }
}
```

Output:

First String arg is: XYZ
Second String arg is: XYZ

As you can see in the above example that for calling static method, I didn't use any object. It can be accessed directly or by using class name as mentioned in the comments.

Non-static method example

```
class JavaExample
```

```

{
    public void display()
    {
        System.out.println("non-static method");
    }
    public static void main(String args[])
    {
        JavaExample obj=new JavaExample();
        /* If you try to access it directly like this:
        * display() then you will get compilation error
        */
        obj.display();
    }
}

```

Output:

non-static method

A non-static method is always accessed using the object of class as shown in the above example.

Notes:

How to call static methods: direct or using class name:

OR copyArg(s1, s2):

StaticDemo.copyArg(s1, s2);

How to call a non-static method: using object of the class: JavaExample

obj = new JavaExample();

Important Points:

1. Static Methods can access static variables without any objects, however nonstatic methods and non-static variables can only be accessed using objects.
2. Static methods can be accessed directly in static and non-static methods. For example the static public static void main() method can access the other static methods directly. Also a non-static regular method can access static methods directly.

Static Class

A class can be made **static** only if it is a nested class.

1. Nested static class doesn't need reference of Outer class
2. A static class cannot access non-static members of the Outer class We will see these two points with the help of an example:

Static class Example

```
class JavaExample{
```

```
private static String str = "BeginnersBook";

//Static class
static class MyNestedClass{
    //non-static method
    public void disp() {

        /* If you make the str variable of outer class
        * non-static then you will get compilation error      *
        because: a nested static class cannot access non-    * static
        members of the outer class.
        */
    }
}
```



```
        System.out.println(str);
    }
}
```

```
    }
    public static void main(String args[])
    {
        /* To create instance of nested class we didn't need the outer
        * class instance but for a regular nested class you would need
        * to create an instance of outer class first
        */
        JavaExample.MyNestedClass obj = new JavaExample.MyNestedClass();
        obj.disp();
    }
}
```

Output:

BeginnersBook

```
class Main{
    private static String str = "BeginnersBook";

    static class MyNestedClass{
        public void disp() {
            System.out.println(str);
        }
    }
    public static void main(String args[])
    {
        Main.MyNestedClass obj = new Main.MyNestedClass();
        obj.disp();
    }
}
```

Output:
BeginnersBook

Static Block

Static block is used for initializing the static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Example 1: Single static block

As you can see that both the static variables were initialized before we accessed them in the main method.

```
class JavaExample{
    static int num;    static
    String mystr;    static{
        num = 97;
        mvstr = "Static keyword in Java";
    }
    public static void main(String args[])
    {
        System.out.println("Value of num: "+num);
        System.out.println("Value of mvstr: "+mvstr);
    }
}
```

Output:

```
Value of num: 97
Value of mystr: Static keyword in Java
```

Example 2: Multiple Static blocks

Lets see how multiple static blocks work in Java. They execute in the given order which means the first static block executes before second static block. That's the reason, values initialized by first block are overwritten by second block.

```

class JavaExample2{
    static int num;
    static String mystr;
    //First Static block
    static{
        System.out.println("Static Block 1");
        num = 68;
        mystr = "Block1";
    }
    //Second static block
    static{
        System.out.println("Static Block 2");
        num = 98;
        mystr = "Block2";
    }
    public static void main(String args[])
    {
        System.out.println("Value of num: "+num);
        System.out.println("Value of mystr: "+mystr);
    }
}

```

Output:

```

Static Block 1
Static Block 2
Value of num: 98
Value of mystr: Block2

```

What is THIS Keyword in Java?

Keyword THIS is a reference variable in Java that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

this: to refer current class instance variable

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

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
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,"ankit",5000f);
Student s2=new Student(112,"sumit",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.

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 ankit 5000
112 sumit 6000
```

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

Program where this keyword is not required

```
class Student{ int
rollno;
String name; float
fee;
Student(int r,String n,float f){ rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
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 ankit 5000
112 sumit 6000
```

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

```
class Student{
int rollno;
String name; float fee; void getdata(int
rollno,String name,float fee){
this.rollno=rollno; this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);} }

class Main{ public static void
main(String args[]){ Student
s1=new Student();
s1.getdata(111,"ankit",5000f);
Student s2=new Student();
s2.getdata(112,"sumit",6000f);
s1.display();
s2.display(); }}
Output:
111 ankit 5000.0
112 sumit 6000.0
```

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.

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
```

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.


```

class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}

```

Output:

```

111 ankit iava null
112 sumit iava 6000

```

Rule: Call to this() must be the first statement in constructor.

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

```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, reference variable and this, output of both variables are same.

```
class A5{
void m(){
System.out.println(this);//prints same reference ID
}
public static void main(String args[]){
A5 obj=new A5();
System.out.println(obj);//prints the reference ID
obj.m();
}
}
```

Output:

```
A5@22b3ea59
A5@22b3ea59
```

Java - Arrays

- ✓ Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- ✓ Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.
- ✓ Normally, an array is a collection of similar type of elements which has contiguous memory location.
- ✓ **Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- ✓ Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

Syntax `dataType[] arrayRefVar;` // preferred way.

or `dataType arrayRefVar[];` // works but not preferred way.

Note – The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax –

```
double[] myList; // preferred way.  
or double myList[]; // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax –

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things –

- It creates an array using `new dataType[arraySize]`.
- It assigns the reference of the newly created array to the variable `arrayRefVar`.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays –

```
public class TestArray {  
  
    public static void main(String[] args) {        double[] myList = {1.9, 2.9, 3.4, 3.5};
```

```

// Print all the array elements    for (int i = 0; i < myList.length; i++) {
    System.out.println(myList[i] + " ");
}

// Summing all elements    double total = 0;    for (int i = 0; i < myList.length; i++) {
total += myList[i];
}
System.out.println("Total is " + total);

// Finding the largest element    double max = myList[0];    for (int i = 1; i <
myList.length; i++) {        if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
}
}

```

Output

1.9 2.9
 3.4
 3.5
 Total is 11.7
 Max is 3.5

Types of Array in java

There are two types of array. ○

Single Dimensional Array ○

Multidimensional Array

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

//Java

Program to illustrate how to declare, instantiate, initialize

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{ public static void
main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array System.out.println(a[i]);
}}
```

Output:

```
33
3
4
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
//body of the loop
}
```

Let us see the example of print the elements of Java array using the for-each loop.

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Output:

```
33
3
4
5
```

```
for(data_type variable:array){
```

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array. Let's see the simple example to get the minimum number of an array using a method.

```
//Java Program to demonstrate the way of passing an array
//to method.
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){ int min=arr[0];
```

```

for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];

System.out.println(min);
}

public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}

```

Output:

3

How to read data from scanner to an array in java?

The Scanner class of the java.util package gives you methods like nextInt(), nextByte(), nextFloat() etc. to read data from keyboard. To read an element of an array uses these methods in a for loop:

```

import java.util.Arrays; import
java.util.Scanner;

public class ReadingWithScanner {
public static void main(String args[]) {
    Scanner s = new Scanner(System.in);
    System.out.println("Enter the length of the array:");
int length = s.nextInt();    int [] myArray = new
int[length];
    System.out.println("Enter the elements of the array:");

    for(int i=0; i<length; i++ ) {
myArray[i] = s.nextInt();
    }

    System.out.println(Arrays.toString(myArray));
}
}

```

Output

Enter the length of the array:

5

Enter the elements of the array:

25

56

48

45

44

[25, 56, 48, 45, 44]

Java Multidimensional Arrays

Before we learn about the multidimensional array, make sure you know about [Java array](#).

A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named `a`. It is a 2-dimensional array, that can hold a maximum of 12 elements,

	Column 1	Column 2	Column 3	Column 4
Row 1	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 2	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 3	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Remember, Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1.

Let's take another example of the multidimensional array. This time we will be creating a 3-dimensional array. For example,

```
String[][][] data = new String[3][4][2];
```


Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
System.out.print(arr[i][j]+" ");
}
System.out.println();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Java command line arguments

A command-line argument is an information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main().

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}
```

compile by > javac CommandLineExample.java **run**

by > java CommandLineExample sonoo Output:

Your first argument is: sonoo

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A{ public static void main(String args[]){  
  
for(int i=0;i<args.length;i++)  
System.out.println(args[i]);  
  
}  
}
```

compile by > javac A.java **run by** >

java A sonoo jaiswal 1 3 abc

Output: sonoo

jaiswal

1

3

abc

Example

The following program displays all of the command-line arguments that it is called with -

```
public class CommandLine {  
    public static void main(String args[]) {  
for(int i = 0; i<args.length; i++) {  
        System.out.println("args[" + i + "]: " + args[i]);  
    }  
    }  
}
```

Try executing this program as shown here -

```
$java CommandLine this is a command line 200 -100
```

Output

This will produce the following result -

```
args[0]: this  
args[1]: is  
args[2]: a args[3]:  
command args[4]:
```

```
line args[5]: 200  
args[6]: -100
```

Java String

In [Java](#), string is basically an object that represents sequence of char values. An [array](#) of characters works same as Java string. For example:

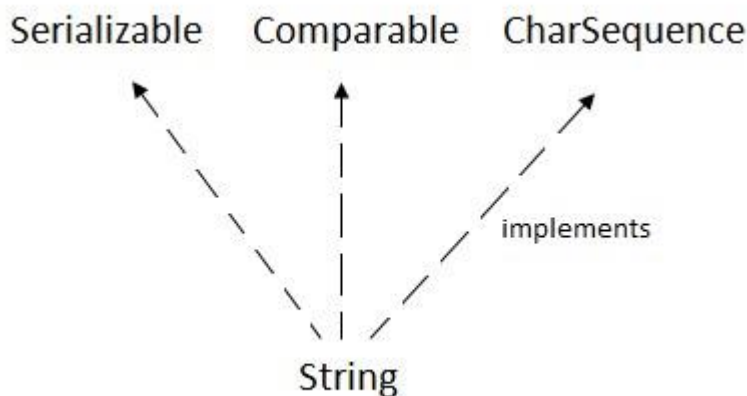
1. `char[] ch={'j','a','v','a','t','p','o','i','n','t'};`
2. `String s=new String(ch);`

is same as:

1. `String s="javatpoint";`

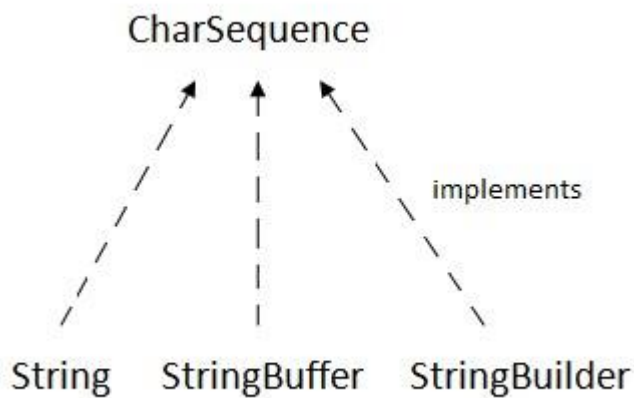
Java String class provides a lot of methods to perform operations on strings such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

The `java.lang.String` class implements *Serializable*, *Comparable* and *CharSequence* [interfaces](#).



CharSequence Interface

The `CharSequence` interface is used to represent the sequence of characters. `String`, [StringBuffer](#) and [StringBuilder](#) classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

What is String in java

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

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
1. String s1="Welcome";  
2. String s2="Welcome";//It doesn't create a new instance
```

2) By new keyword

```
1. String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, **JVM** will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by java string literal  char
ch[]={ 's','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

java strings
example

Java String Methods

Java **String** provides various methods that allow us to perform different string operations. Here are some of the commonly used string methods.

Methods	Description
concat()	joins the two strings together
equals()	compares the value of two strings
charAt()	returns the character present in the specified location
getBytes()	converts the string to an array of bytes
indexOf()	returns the position of the specified character in the string
length()	returns the size of the specified string
replace()	replaces the specified old character with the specified new character
substring()	returns the substring of the string
split()	breaks the string into an array of strings
toLowerCase()	converts the string to lowercase
toUpperCase()	converts the string to uppercase
valueOf()	returns the string representation of the specified data

Example 1: Java find string's length

```
class Main {  
    public static void main(String[] args) {  
  
        // create a string  
        String greet = "Hello! World";  
        System.out.println("The string is: " + greet);  
  
        //checks the string length  
        System.out.println("The length of the string: " + greet.length());  
    }  
}
```

Output

The string is: Hello! World
The length of the string: 12

In the above example, we have created a `length()` string named the method to get the size of the string. `greet`. Here we have used

Example 2: Java join two strings using concat()

```
class Main {    public static void  
main(String[] args) {  
  
    // create string  
    String greet = "Hello! ";  
    System.out.println("First String: " + greet);  
  
    String name = "World";  
    System.out.println("Second String: " + name);  
  
    // join two strings  
    String joinedString = greet.concat(name);  
    System.out.println("Joined String: " + joinedString);  
}  
}
```

Output

First String: Hello!
Second String: World

Joined String: Hello! World

In the above example, we have created 2 strings named `greet` and `name`.
Here, we have used the `concat()` string named `joinedString`.
gs. H

In Java, we can also join two strings using the `+` operator.

Example 3: Java join strings using + operator

```
class Main {  
    public static void main(String[] args) {  
  
        // create string  
        String greet = "Hello! ";  
        System.out.println("First String: " + greet);  
  
        String name = "World";  
        System.out.println("Second String: " + name);  
  
        // join two strings  
        String joinedString = greet + name;  
        System.out.println("Joined String: " + joinedString);  
    }  
}
```

Output

First String: Hello!
Second String: World
Joined String: Hello! World

Here, we have used the `+` operator to join the two strings.

Example 4: Java compare two strings

```
class Main { public static void
main(String[] args) {

    // create strings
    String first = "java programming";
    String second = "java programming";
    String third = "python programming";

    // compare first and second strings
    boolean result1 = first.equals(second);
    System.out.println("Strings first and second are equal: " + result1);
```



```
//compare first and third strings
boolean result2 = first.equals(third);
System.out.println("Strings first and third are equal: " + result2);
}
}
```

Output

```
Strings first and second are equal: true
Strings first and third are equal: false
```

In the above example, we have used the `equals()` method to compare the value of two strings.

The method returns `true` if both strings are the same otherwise it returns `false`.

Note: We can also use the `==` operator and `compareTo()` method to make a comparison between 2 strings.

Example 5: Java get characters from a string

```
class Main {
    public static void main(String[] args) {

        // create string using the string literal
        String greet = "Hello! World";
        System.out.println("The string is: " + greet);

        // returns the character at 3
        System.out.println("The character at 3: " + greet.charAt(3));

        // returns the character at 7
        System.out.println("The character at 7: " + greet.charAt(7));
    }
}
```

Output

```
The string is: Hello! World
The character at 3: l
The character at 7: W
```

In the above example, we have used the `charAt()` method to access the character from the specified position.

Example 6: Java Strings other methods

```
class Main {
```

```

public static void main(String[] args) {

    // create string using the new keyword
    String example = new String("Hello! World");

    // returns the substring World
    System.out.println("Using the subString(): " + example.substring(7));

    // converts the string to lowercase
    System.out.println("Using the toLowerCase(): " + example.toLowerCase());

    // converts the string to uppercase
    System.out.println("Using the toUpperCase(): " + example.toUpperCase());

    // replaces the character '!' with 'o'
    System.out.println("Using the replace(): " + example.replace('!', 'o'));
}
}

```

Output

```

Using the subString(): World
Using the toLowerCase(): hello! world
Using the toUpperCase(): HELLO! WORLD
Using the replace(): Helloo World

```

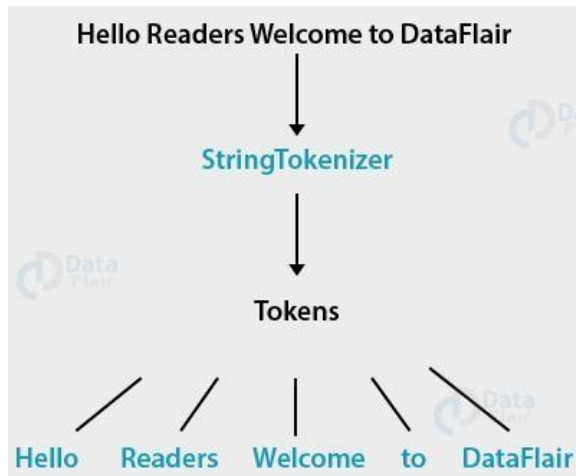
In the above example, we have created a string named `example` using the `new` keyword.

Here,

- the `substring()` method returns the string `World`
- the `toLowerCase()` method converts the string to the lower case
- the `toUpperCase()` method converts the string to the upper case
- the `replace()` method replaces the character `!` with `'o'`.

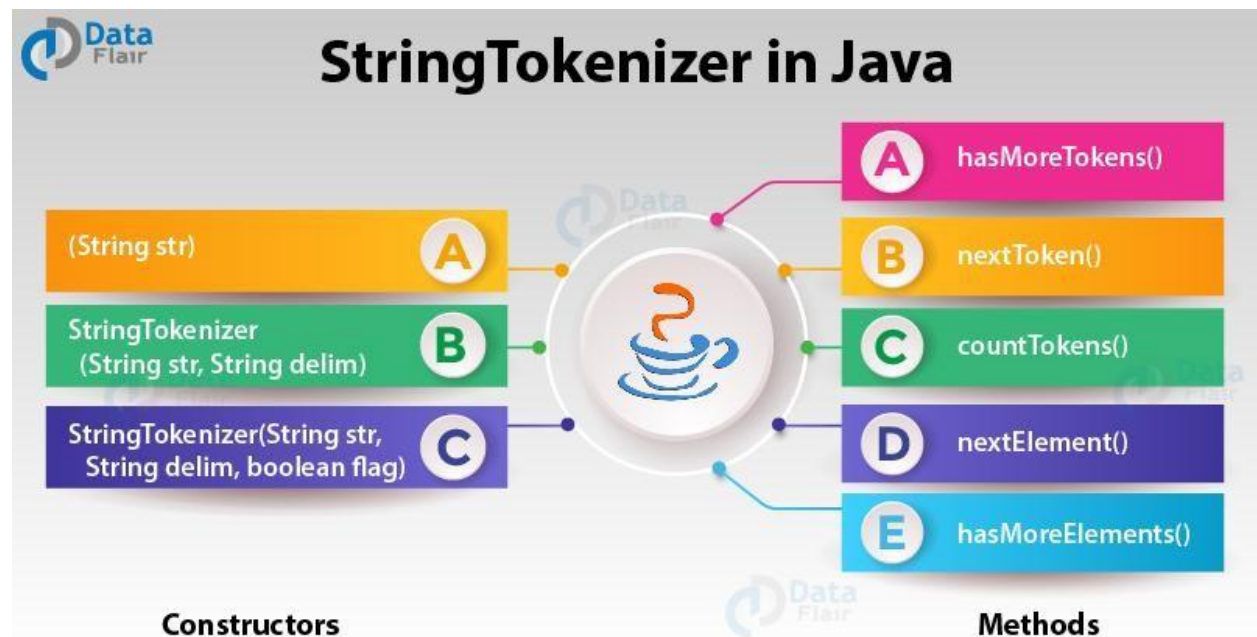
StringTokenizer in Java

`StringTokenizer` class is used for creating tokens in Java. It allows an application to break or split into small parts. Each split string part is called *Token*.



StringTokenizer Constructors

There are 3 [types of Constructors available in Java](#) StringTokenizer,



Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
-------------	-------------

StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim) delimiter.	creates StringTokenizer with specified string and
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimiter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

□ (String str)

str is a string to be tokenized and it considers default delimiters like newline, space, tab, carriage return and form feed which can be further tokenized.

□ **StringTokenizer(String str, String delim)** delim is set of delimiters that are used to tokenize the given string. □

StringTokenizer(String str, String delim, boolean flag)

Since the first two parameters have the same meaning. The flag serves the following purpose. If the flag is false, delimiter characters serve to separate tokens and if the flag is true, delimiter characters are considered to be tokens.

StringTokenizer(String str, String delim, boolean flag):

The first two parameters have same meaning. The flag serves following purpose.

If the **flag** is **false**, delimiter characters serve to separate tokens. For example, if string is "hello geeks" and delimiter is " ", then tokens are "hello" and "geeks".

If the **flag** is **true**, delimiter characters are considered to be tokens. For example, if string is "hello geeks" and delimiter is " ", then tokens are "hello", " " and "geeks".

Simple example of StringTokenizer class

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
import java.util.StringTokenizer; public class
Simple{
    public static void main(String args[]){
        StringTokenizer st = new StringTokenizer("my name is khan"," "); while
        (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

O
u
t
p
u
t
:
m
y

n
a

```
m  
e  
  
i  
s  
  
k  
h  
a  
n
```

Example

```
import java.util.*;    public class  
Test {  
    public static void main(String[] args) {  
        StringTokenizer st = new StringTokenizer("my,name,is,khan");  
        // printing next token  
        System.out.println("Next token is : " + st.nextToken(","));  
    }  
}
```

Output:

Test