Semester III
BCA311 JAVA Programming
(Unit-3)

Harshita Mathur

Department of Computer Science

Email : hmathur85@gmail.com

Array

- **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.
- we can get the length of the array using the length member.

Array continue...

- In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces.
- We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.
- Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

Advantages

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

Types of Array in java

There are two types of array:

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

- **Syntax to Declare an Array in Java:**

  dataType[] arr; (or)

  dataType []arr; (or)

  dataType arr[];

- **Instantiation of an Array in Java**
  - Although the first declaration above establishes the fact that arr is an array variable, **no actual array exists**. It merely tells the compiler that this variable (arr) will hold an array of the integer type. To link arr with an actual, physical array of integers, you must allocate one using **new** and assign it to arr.
  - When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array like this:

    arrayRefVar=new datatype[size];

Example

- int arr[]; //declaring array
  arr = new int[20]; // allocating memory to array

OR

- int[] arr= new int[20]; // combining both statements in one

Example

- //Java Program to illustrate how to declare, instantiate, initialize
- //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]);
}}
```

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
```

Array Literal

- In a situation, where the size of the array and variables of array are already known, array literals can be used.

- int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
  // Declaring array literal
- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java

```
//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]);
}}
```

Passing Array to a Method in Java

```
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
}}
```

Anonymous Array in Java

```
//Java Program to demonstrate the way of passing an anonymous array
//to method.

public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}

public static void main(String args[]){
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}}
```

Returning Array from the Method

```
//Java Program to return an array from the method
class TestReturnArray{
//creating method which returns an array
static int[] get(){
return new int[]{10,30,50,90,60};
}

public static void main(String args[]){
//calling method which returns an array
int arr[]=get();
//printing the values of an array
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}}
```

Arrays of Objects

- An array of objects is created just like an array of primitive type data items in the following way.

  Student[] arr = new Student[7]; //student is a user-defined class
- The studentArray contains seven memory spaces each of size of student class in which the address of seven Student objects can be stored.
- The Student objects have to be instantiated using the constructor of the Student class and their references should be assigned to the array elements.

Example

```
class Main{
   public static void main(String args[]){
     //create array of employee object
    Employee[] obj = new Employee[2] ;

     //create & initialize actual employee objects using constructor
     obj[0] = new Employee(100,"ABC");
     obj[1] = new Employee(200,"XYZ");

     //display the employee object data
     System.out.println("Employee Object 1:");
     obj[0].showData();
     System.out.println("Employee Object 2:");
     obj[1].showData();
  }
}
```

Example Continue...

```
//Employee class with empId and name as attributes
class Employee{
  int empId;
  String name;
  //Employee class constructor
  Employee(int eid, String n){
     empId = eid;
     name = n;
  }
public void showData(){
   System.out.print("EmpId = "+empId + "  " + " Employee Name = "+name);
   System.out.println();
 }
}
```

Output

```
Employee Object 1:

EmpId = 100    Employee Name = ABC

Employee Object 2:

EmpId = 200    Employee Name = XYZ
```

Multidimensional Array in Java

- Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other array.

- A multidimensional array is created by appending one set of square brackets ([]) per dimension.

- Examples:

  int[][] arr = new int[10][20]; //a 2D array or matrix

  int[][][] arr = new int[10][20][10]; //a 3D array

Syntax to Declare two dimensional Array in Java

- dataType[][] arrayRefVar; (or)
- dataType [][]arrayRefVar; (or)
- dataType arrayRefVar[][]; (or)
- dataType []arrayRefVar[];

Example to instantiate two dimensional Array in Java

- **int**[][] arr=**new int**[3][3];//3 row and 3 column

Example to initialize Multidimensional Array in Java

- arr[0][0]=1;
- arr[0][1]=2;
- arr[0][2]=3;
- arr[1][0]=4;
- arr[1][1]=5;
- arr[1][2]=6;
- arr[2][0]=7;
- arr[2][1]=8;
- arr[2][2]=9;

Example of two dimensional Java Array

```
class my
{
	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

Addition of 2 Matrices in Java

```
class my{
  public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};  $

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```

- Output:

  2 6 8

  6 8 10

Object Lifetime

- There is some memory allocated to JVM which it uses to store your program runtime data. Some areas are created by JVM, while some are created by other threads which might be used in the program. The area created by JVM is destroyed only when JVM exits while others might be destroyed as their contexts are destroyed.
- Objects are created from the **new** keyword in Java. Every time you do something like
- **ClassName objectName** = ***new*** **ClassName();**JVM allocates the necessary memory in the **Heap Area** to create this object, once this memory is allocated it holds the reference to this memory through some variable which is **objectName** here.

- All **object instances** and **arrays** are stored in a heap. Depending upon your system configuration the heap size can be made fixed or dynamic. There is only one heap during the execution of the program.
- While the reference variables of objects in heap are stored in the **JVM Stack**. These are stored in a stack along with method calls which you may already know so that it's convenient to keep track of data which will be needed to return value in the future.
- After the above operation, our object comes to life and is **Alive**!

Garbage Collection in Java

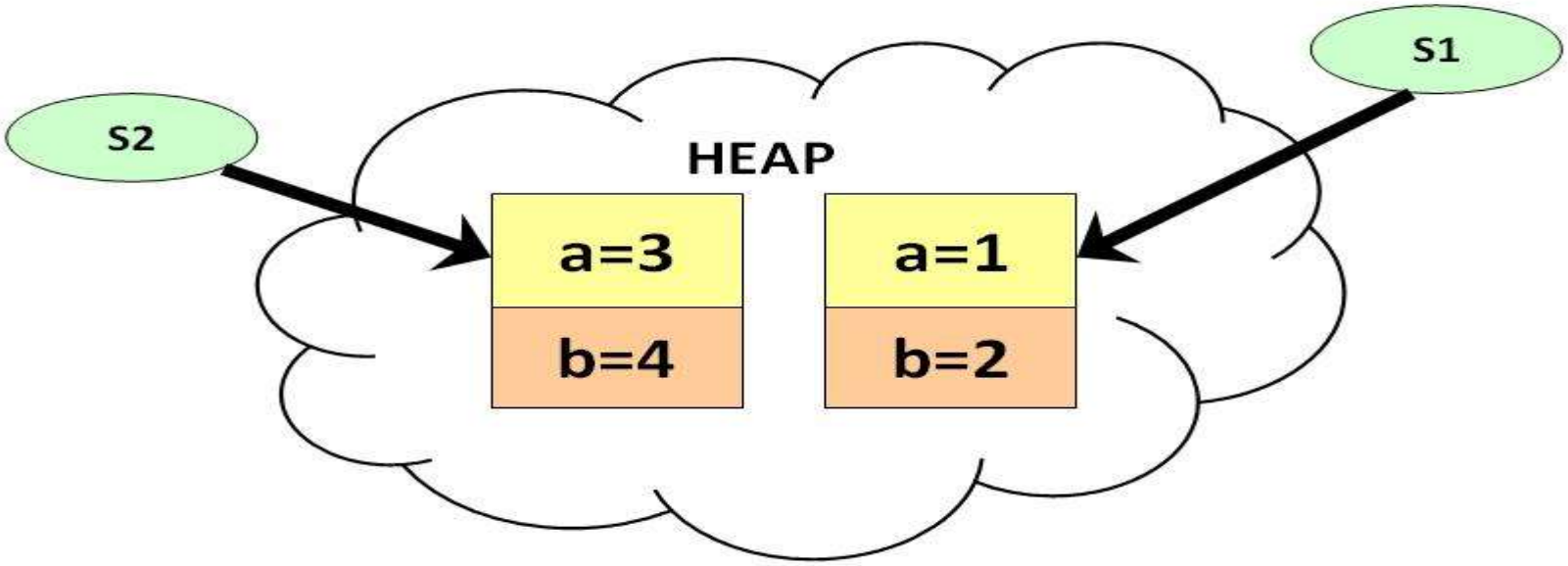
- **Garbage Collection in Java** is a process by which the programs perform memory management automatically.
- The Garbage Collector(GC) finds the unused objects and deletes them to reclaim the memory.
- In Java, dynamic memory allocation of objects is achieved using the new operator that uses some memory 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**.

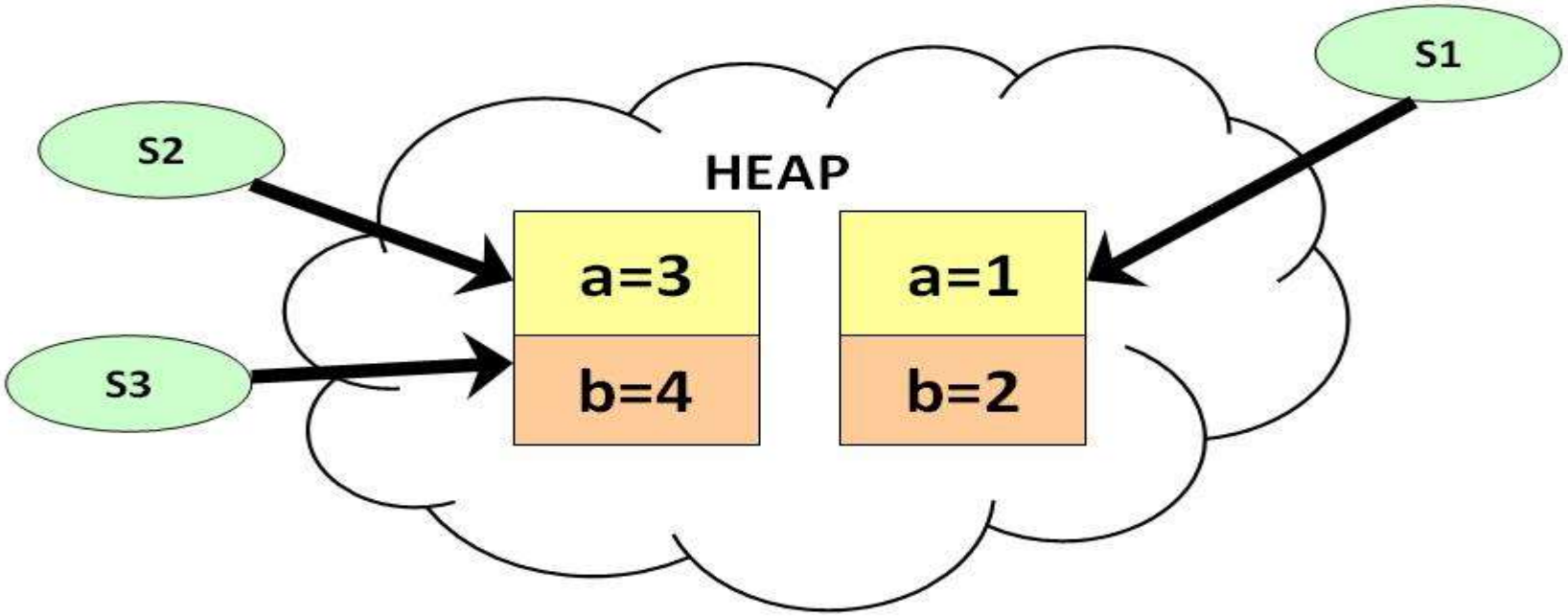
```
1. class Student{
2. int a;
3. int b;
4.
5. public void setData(int c,int d){
6. a=c;
7. b=d;
8. }
9. public void showData(){
10. System.out.println("Value of a = "+a);
11. System.out.println("Value of b = "+b);
12. }
13. public static void main(String args[]){
14. Student s1 = new Student();
15. Student s2 = new Student();
```

```
16. s1.setData(1,2);
17. s2.setData(3,4);
18. s1.showData();
19. s2.showData();
20. //Student s3;
21. //s3=s2;
22. //s3.showData();
23. //s2=null;
24. //s3.showData();
25. //s3=null;
26. //s3.showData();
27. }
28. }
```

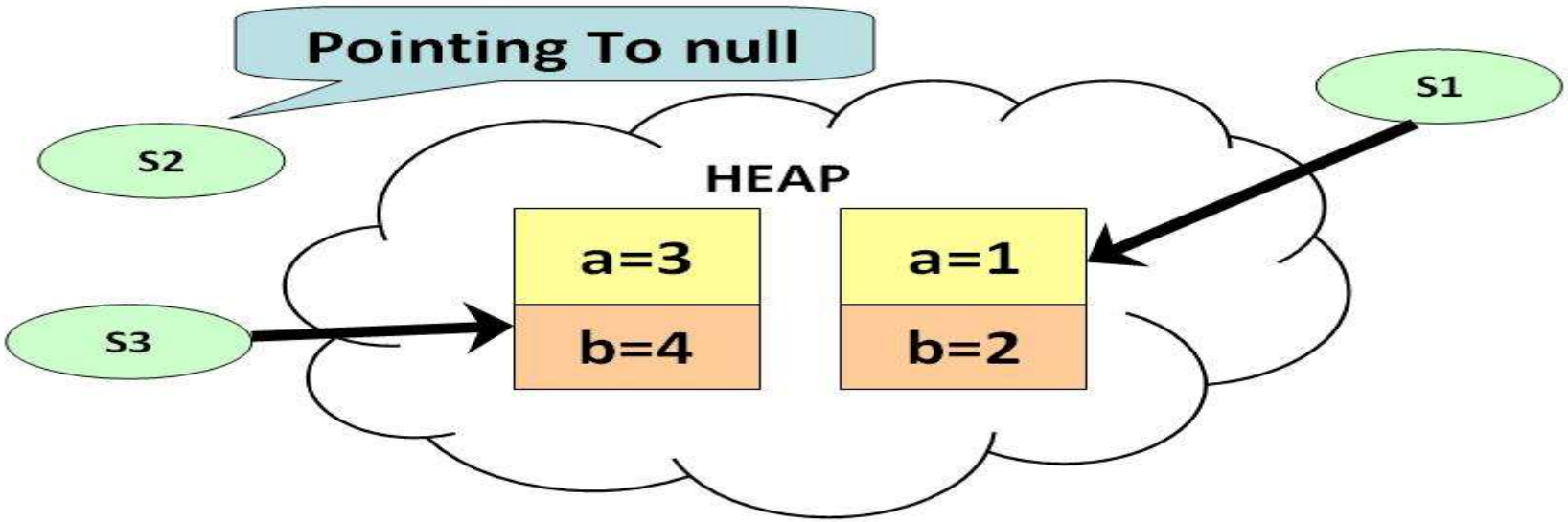
- As shown in the diagram, two objects and two reference variables are created.



- Uncomment line # 20,21,22. Save, compile & run the code
- As shown in the diagram below, two reference variables are pointing to the same object.

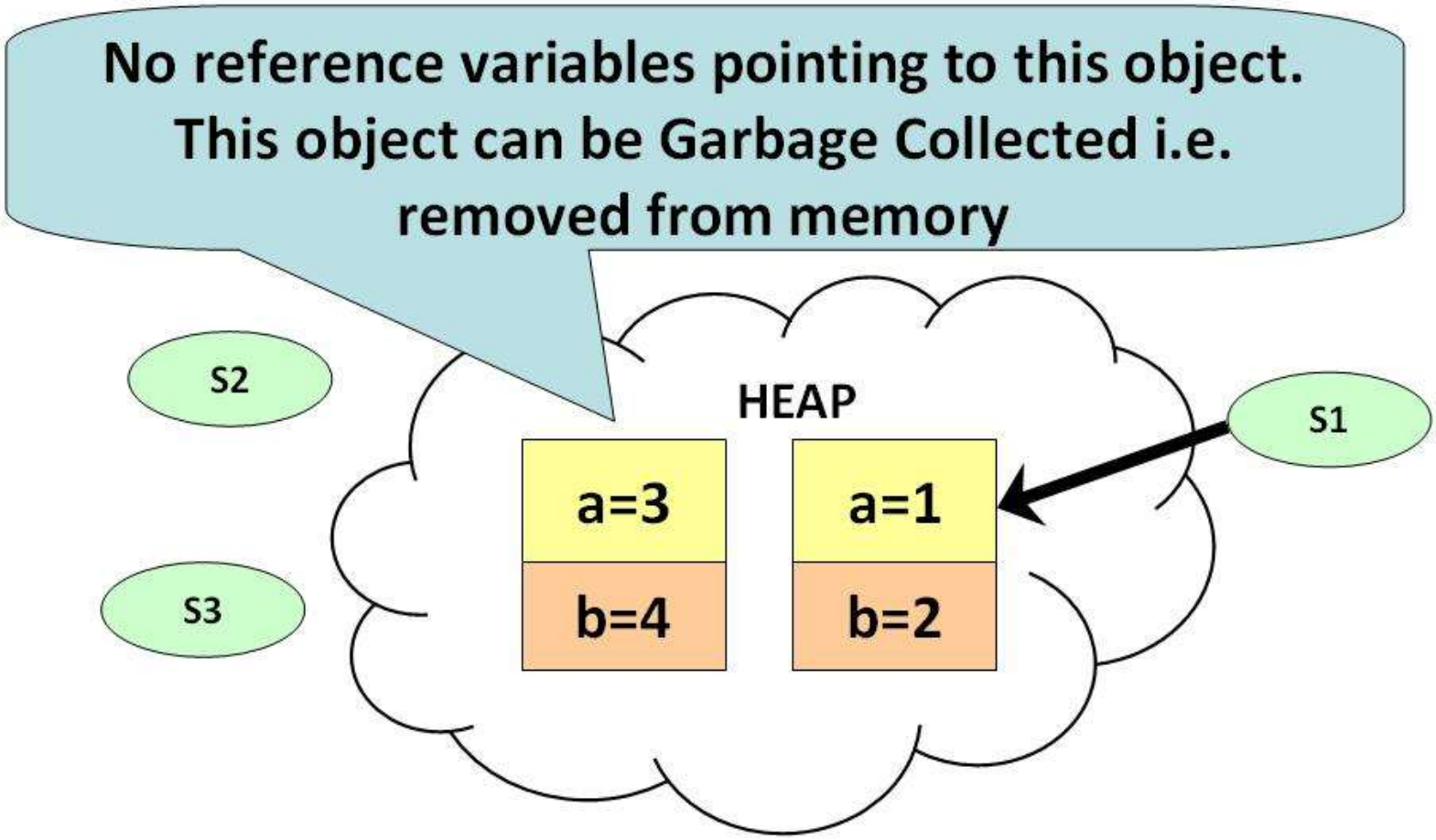


- Uncomment line # 23 & 24. Compile, Save & Run the code
- As show in diagram below, s2 becomes null, but s3 is still pointing to the object and is not eligible for java garbage collection.



- Uncomment line # 25 & 26. Save, Compile & Run the Code

- At this point there are no references pointing to the object and becomes eligible for garbage collection. It will be removed from memory, and there is no way of retrieving it back.


No reference variables pointing to this object.
This object can be Garbage Collected i.e.
removed from memory
S2
HEAP
S1
a=3
a=1
S3
b=4
b=2

Semester III
BCA311 JAVA Programming
(Unit-3)

Harshita Mathur

Department of Computer Science

Email : hmathur85@gmail.com

Access Modifiers

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

  There are four types of Java access modifiers:
- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

- Let's understand the access modifiers in Java by a simple table.

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

Simple example of **private** access modifier

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

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Example of **default** access modifier

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

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

Example of **protected** access modifier

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

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```
Output:Hello

Example of **public** access modifier

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
 package mypack;
import pack.*;
  class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:Hello

Constructor

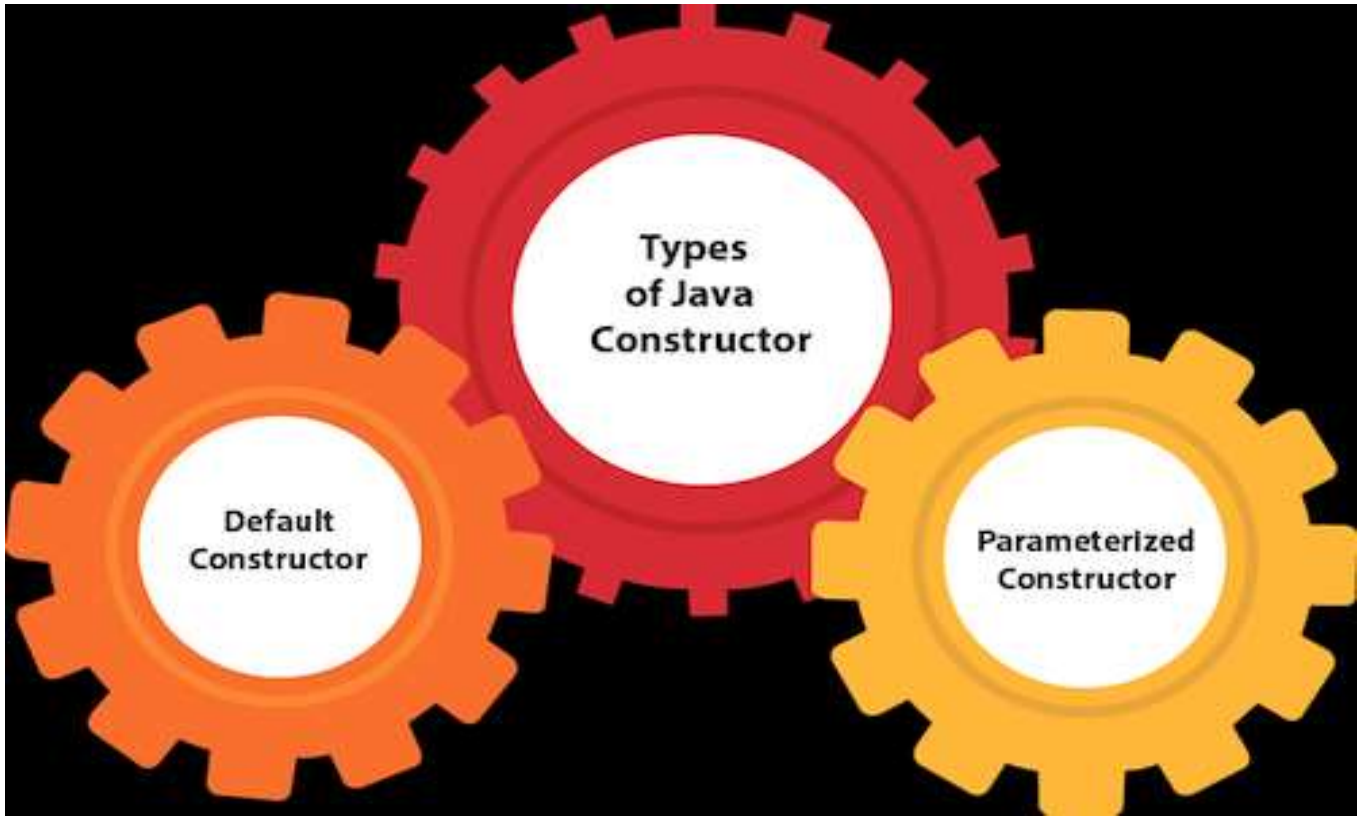
- In Java, a constructor is a block of codes similar to the method.
- It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an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

- It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.
- Constructor name must be the same as its class name.
- A Constructor must have no explicit return type.
- A Java constructor cannot be abstract, static, final, and synchronized.
- We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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



Java Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

- Syntax of default constructor:

  <class_name>(){}

Example of default constructor

- In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

- **Output:**

  Bike is created

- **If there is no constructor in a class, compiler automatically creates a default constructor**

Example of default constructor that displays the default values

```
class Student3{
int id;
String name;
//method to display the value of id and name
void display(){System.out.println(id+" "+name);}
public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}
```

- **Output:**

  0 null

  0 null

- **Here 0 and null values are provided by default constructor.**

Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.

- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

```
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

- **Output:**

  111 Karan

  222 Aryan

Constructor Overloading in Java

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
```

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

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

- **Output**:

  111 Karan 0

  222 Aryan 25

Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type

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

Method Overloading: changing no. of arguments

```
class MethodOverloading
{
    private static void display(int a)
    {
    System.out.println("Arguments: " + a);
     }
    private static void display(int a, int b)
    {
    System.out.println("Arguments: " + a + " and " + b);
     }
    public static void main(String[] args)
    {
    display(1);
    display(1, 4);
} }
```

- **Output**:

  Arguments: 1

  Arguments: 1 and 4

Method Overloading: changing data type of arguments

```
class MethodOverloading
{
 // this method accepts int
 private static void display(int a)
{
System.out.println("Got Integer data.");
}
// this method accepts String object
 private static void display(String a)
{
 System.out.println("Got String object.");
 }
 public static void main(String[] args)
 {
 display(1);
 display("Hello");
 } }
```

- **Output**:

  Got Integer data.

  Got String object.

Recursion

- Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.
- Syntax:

```
returntype methodname()
{
//code to be executed
methodname();//calling same method
}
```

Example

```
public class my {
static int count=0;
static void p(){
count++;
if(count<=5){
System.out.println("hello "+count);
p();
}
}
public static void main(String[] args) {
p();
}
}
```

- **Output:**

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

Example

```
public class my {
   static int factorial(int n){
        if (n == 1)
          return 1;
        else
          return(n * factorial(n-1));
   }

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

- **Output:**
  Factorial of 5 is: 120

finalize() method in Java

- finalize( ) method is a method of Object class which is called just before the destruction of an object by the garbage collector.
- After finalize( ) method gets executed completely, the object automatically gets destroyed.
- The purpose of calling finalize method is to perform activities related to clean up, resource deallocation etc.
- finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.
- Prototype:

  protected void finalize () throws Throwable

finalize() method

- Now, the finalize method which is present in the Object class, has an **empty implementation**, in our class clean-up activities are there, then we have to **override this method** to define our own clean-up activities.

- In order to Override this method, we have to explicitly define and call finalize within our code.

```
// Java code to show the
// overriding of finalize() method

import java.lang.*;

// Defining a class demo since every java class
// is a subclass of predefined Object class
// Therefore demo is a subclass of Object class
public class demo {

   protected void finalize() throws Throwable
   {
      try {

         System.out.println("inside demo's finalize()");
      }
      catch (Throwable e) {

         throw e;
      }
```

```
finally {

        System.out.println("Calling finalize method"
                        + " of the Object class");

        // Calling finalize() of Object class
        super.finalize();
    }
 }
```

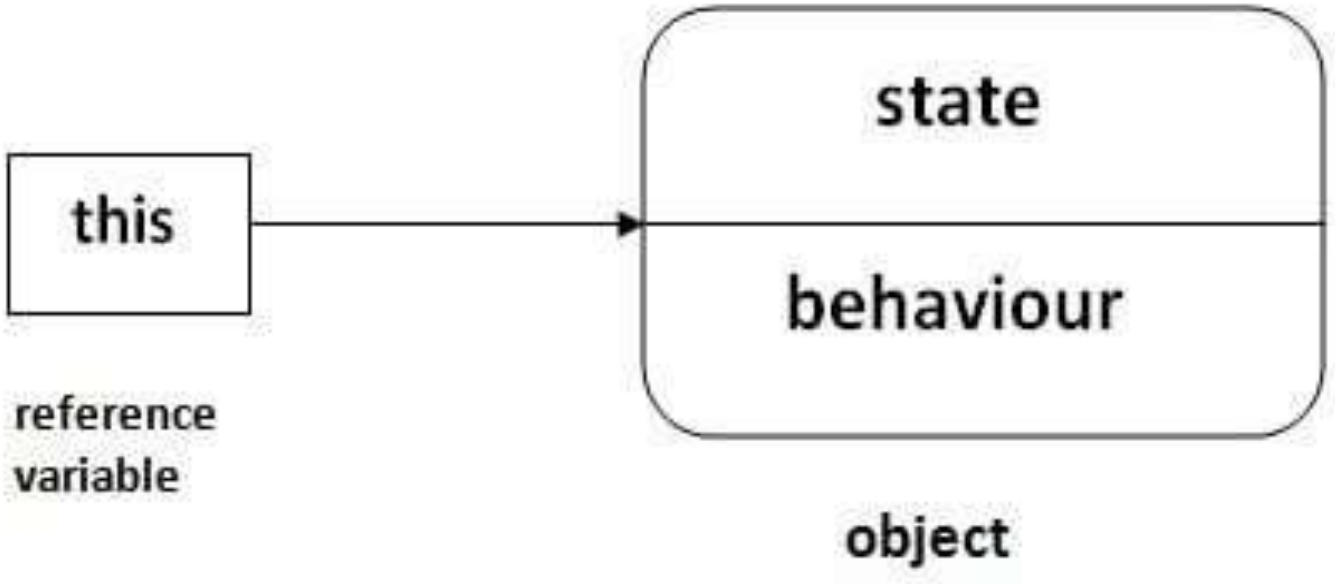
```
 // Driver code
   public static void main(String[] args) throws Throwable
   {

      // Creating demo's object
      demo d = new demo();

      // Calling finalize of demo
      d.finalize();
   }
}
```

Output:

```
        inside demo's finalize()
       Calling finalize method of the Object class
```

Use of this keyword in java

- In java, this is a **reference variable** that refers to the current object.



this: to refer current class instance variable

```
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 Test{
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

this: to invoke current class method

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class Test{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

this() : to invoke current class constructor

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class Test{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

```
hello a
10
```