K J's EDUCATIONAL INSTITUTES
TRINITY COLLEGE OF ENGINEERING & RESEARCH
Approved by AICTE, Government of Maharashtra & Affiliated to Savitribai Phule Pune University

Accredited NAAC Grade 'A+'

Department of Computer Engineering

Course Name :Principles of Programming Languages

Course Code: 210255

Unit : 3 - Java as Object Oriented Programming Language-Overview

Course Coordinator: Mrs. Shaikh J. N.

A.Y : 2024-25

Semester: II

Course Objectives:

To learn Object Oriented Programming (OOP) principles using Java Programming Language.

Course Outcomes:

On completion of the course, learner will be able to
CO3: Develop programs using Object Oriented Programming language : Java.

Contents :

- Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements ,String Handling: String class methods
- Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method,
- overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length arguments.

Topic	Book To Refer
Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements ,String Handling: String class methods Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method, overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length arguments.	**Herbert Schildt, "The Complete Reference Java", 9th Ed, TMH,ISBN: 978-0-07-180856-9.** **Programming With Java, 3rd Edition, E. Balaguruswamy**

SPPU Questions

UNIT -3		
Write short notes on Java Virtual Machine(JVM) with diagram.	CO3	U
State the uses of the final keyword in Java?	CO3	R, U
Define String in Java. Explain following operations of class strings in Java with example. i) To find length of the string ii) To compare two strings iii) To extract a character from a string iv) To concatenate two strings	CO3	R, U
Explain Java's role in Internet. Justify the following features of Java. i) Secure ii) Architectural Neutral iii) Distributed.	CO3	U
Summarize different access controls in Java. Explain the situation if you remove static modifier from the main method.	CO3	U
Explain following features of java in detail i) Security ii) Platform Independence iii) Object - oriented	CO3	U

Write short note on i) Garbage collector ii) this	CO3	U
Define constructor. Which are the types of Constructor used in Java? Explain with example.	CO3	R, U
Explain following features of java in detail i) Portability ii) Architecture Neutral iii) Distributed	CO3	U
Write short note on i) final ii) finalize ()	CO3	U
Explain one dimensional and multi - dimensional array used in Java with suitable examples	CO3	U
Explain why Java is secured, portable and Dynamic? Which of the concept in Java ensures these?	CO3	U
What are strings in java? Explain following operations of class strings in java with example. i) To find length of string ii) To compare strings iii) Extraction of characters from string iv) To search a substring	CO3	U, A

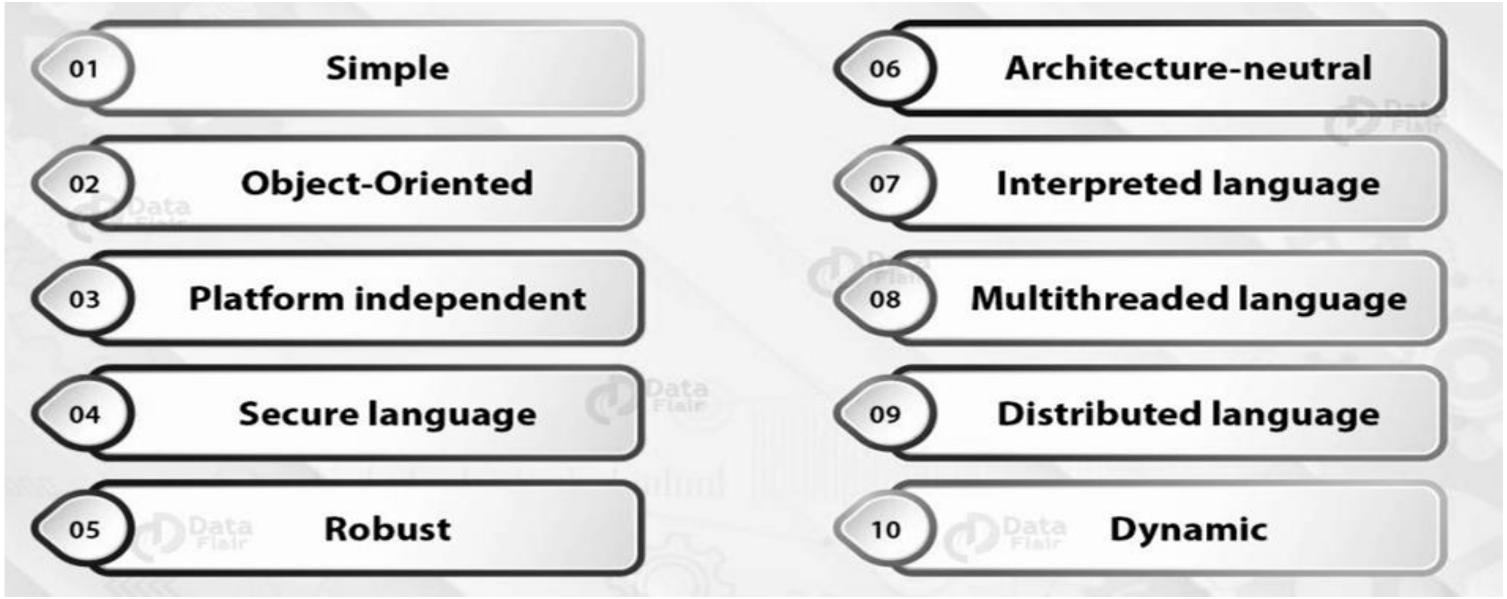
Write a program in Java using switch-case statement to perform addition, subtraction, Multiplication and Division of given two numbers and print the result.	CO3	A
Explain in brief below keywords with example in Java. i) Final ii) Static iii) This	CO3	U
Define Constructor. List its different types. Demonstrate with suitable example the different types of constructors used in Java.	CO3	R, U
Write a program which receives n integers. Store the integers in an array. Program outputs the number of odd and even numbers present in this array	CO3	A
Justify the meaning of each characteristic of java in the statement "java is simple, architecture -neutral ,portable, interpreted and robust and secured programming language."	CO3	U
Define String in Java programming. Explain the following operations of class string in Java with example. 1. To find the length of a string. 2. To compare two strings 3. Extraction of character from a string	CO3	U, A

What is constructor? Show with example the use and overloading default, parameterized and copy constructor?	CO3	U
What do you mean by method overloading? Demonstrate through a program in Java how method overloading is used to add two integers and three integers respectively.	CO3	U, A
Justify the meaning of each characteristic of Java in the statement “Java is simple, architecture neutral portable, interpreted and robust and secured programming language”.	CO3	U
Write a program in Java to perform the addition of two matrices (multidimensional arrays) and set the diagonal elements of resultant matrix to 0.	CO3	A
Explain the Garbage Collection concept in Java Programming with code example.	CO3	U
Explain command line arguments and variable length arguments in Java with an example.	CO3	U

History of Java

- Java was developed by **Sun Microsystems (which is now the subsidiary of Oracle**) in the year 1995. **James Gosling is known as the father of Ja**va. Before Java, its name was **Oak**. Since Oak was already a registered company, **so James Gosling and his team changed the Oak name to Java.**
- **Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.**

Features of Java



Java is Simple

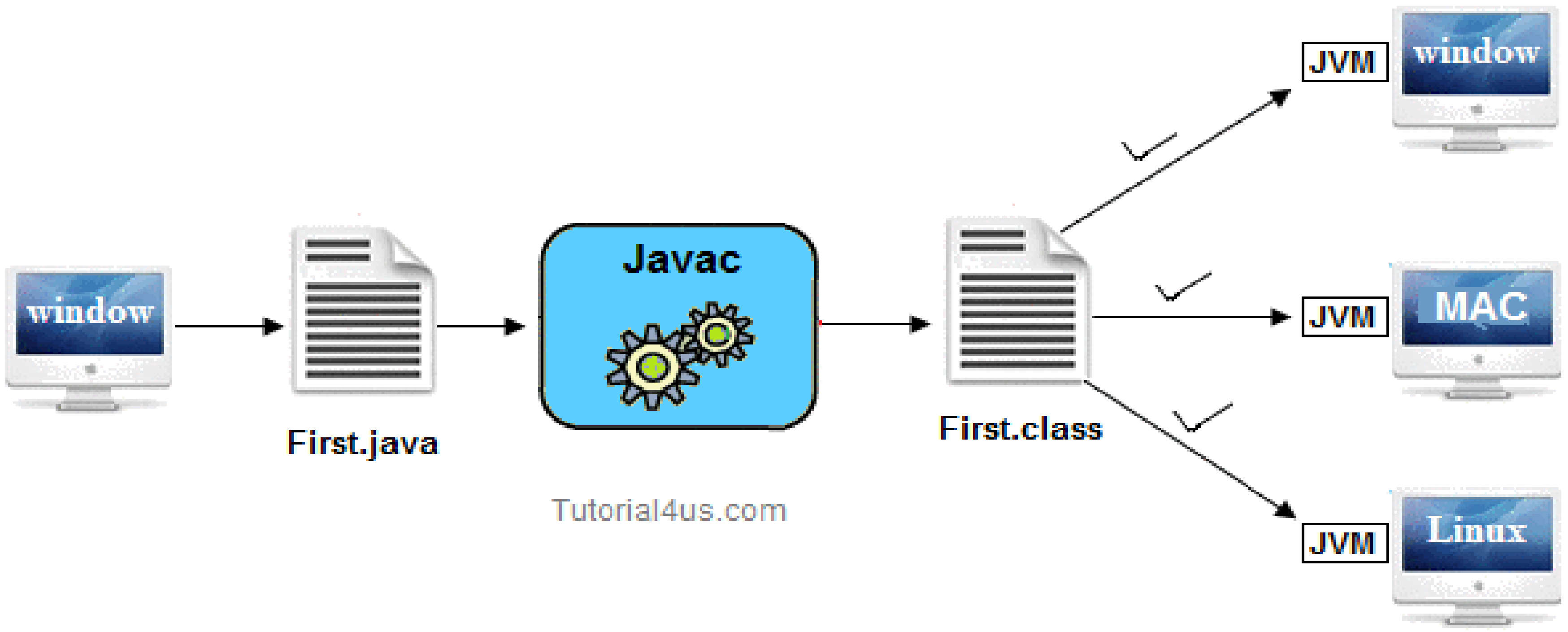
- It is f**ree from pointer due t**o this execution time of application is improved. [Whenever we write a Java program without pointers then internally it is converted into the equivalent pointer program].
- It has **Rich set of API** (application protocol interface).
- It has **Automatic Garbage Collector** which is always used to collect un-Referenced (unused) Memory location for improving performance of a Java program.
- It contains **user friendly syntax for developing any applications.**

Java is Object Oriented

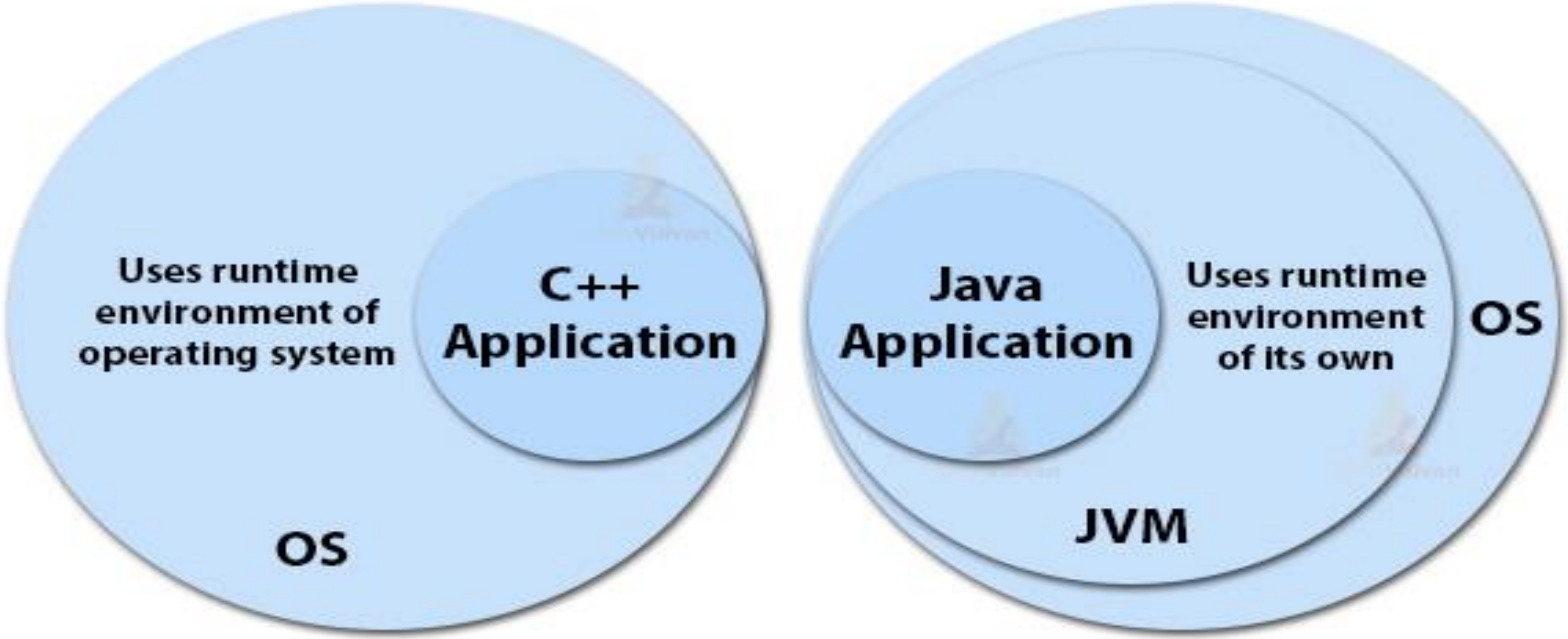
Since it is an object-oriented language, it will support the following features:

- Class
- Object
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Java is Platform Independent



Java is Secure

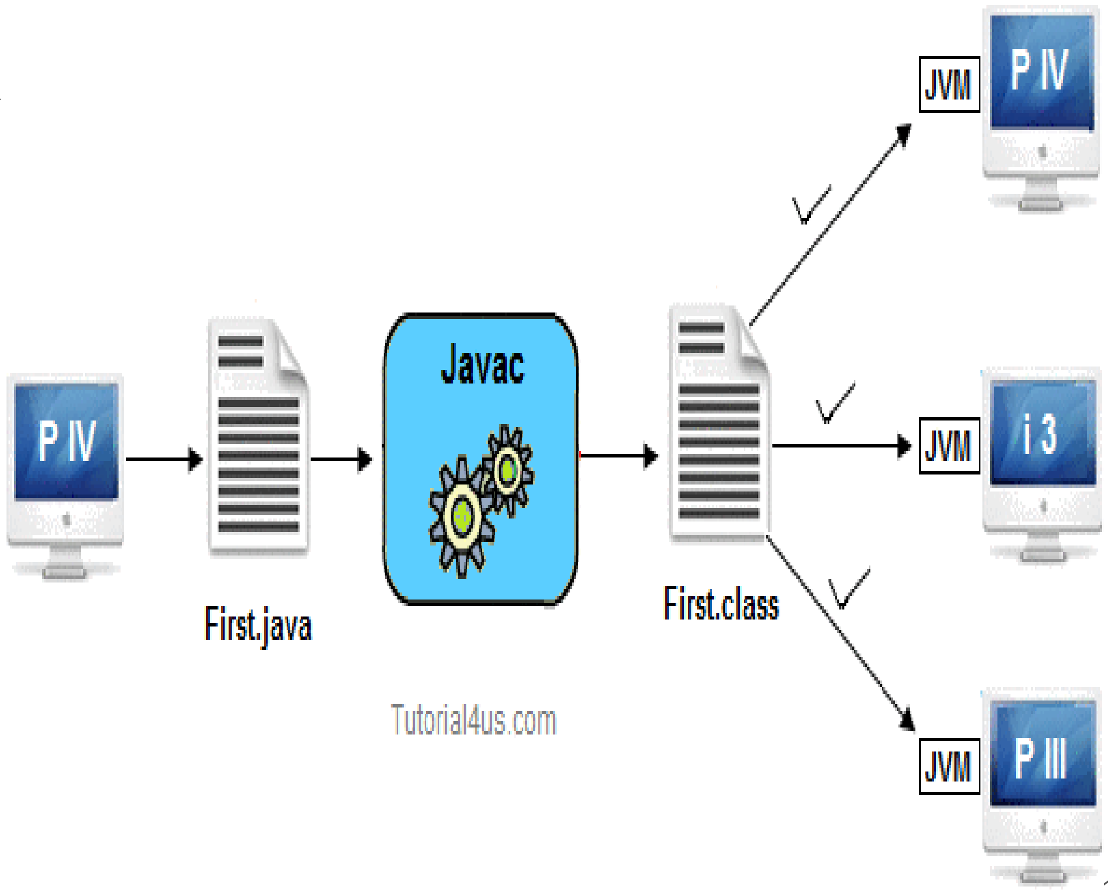

Uses runtime environment of operating system
C++ Application
OS
Java Application
Uses runtime environment of its own
OS
JVM

Security in Applications

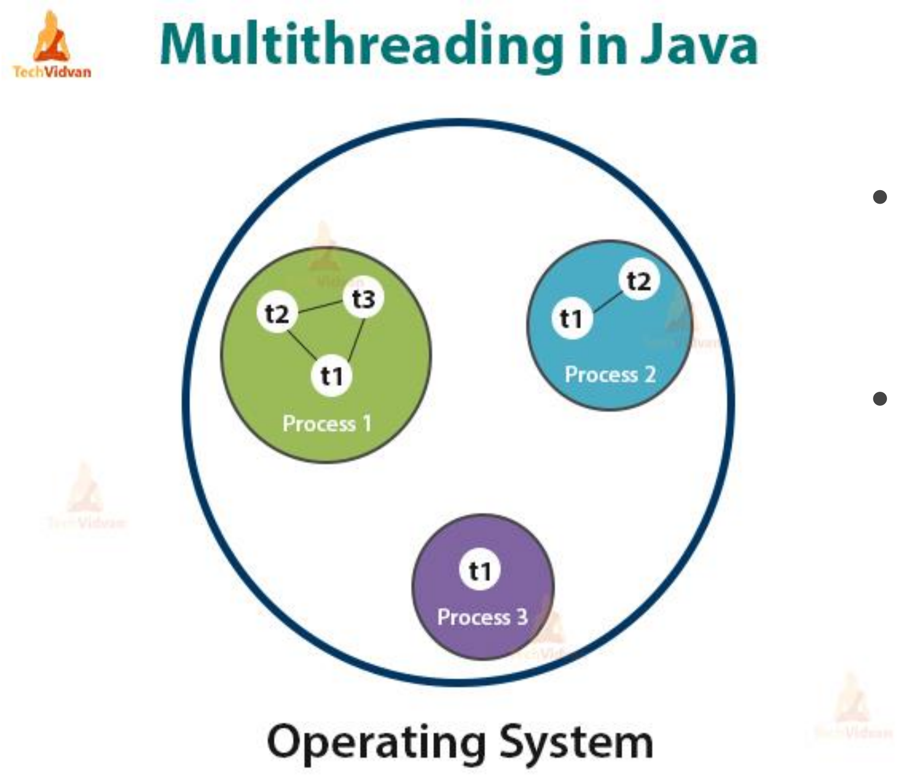
Java is Robust

- Capable of handling run-time errors,
- Supports automatic garbage collection
- Exception handling, and
- Avoids explicit pointer concept.

Java is Architecture Neutral



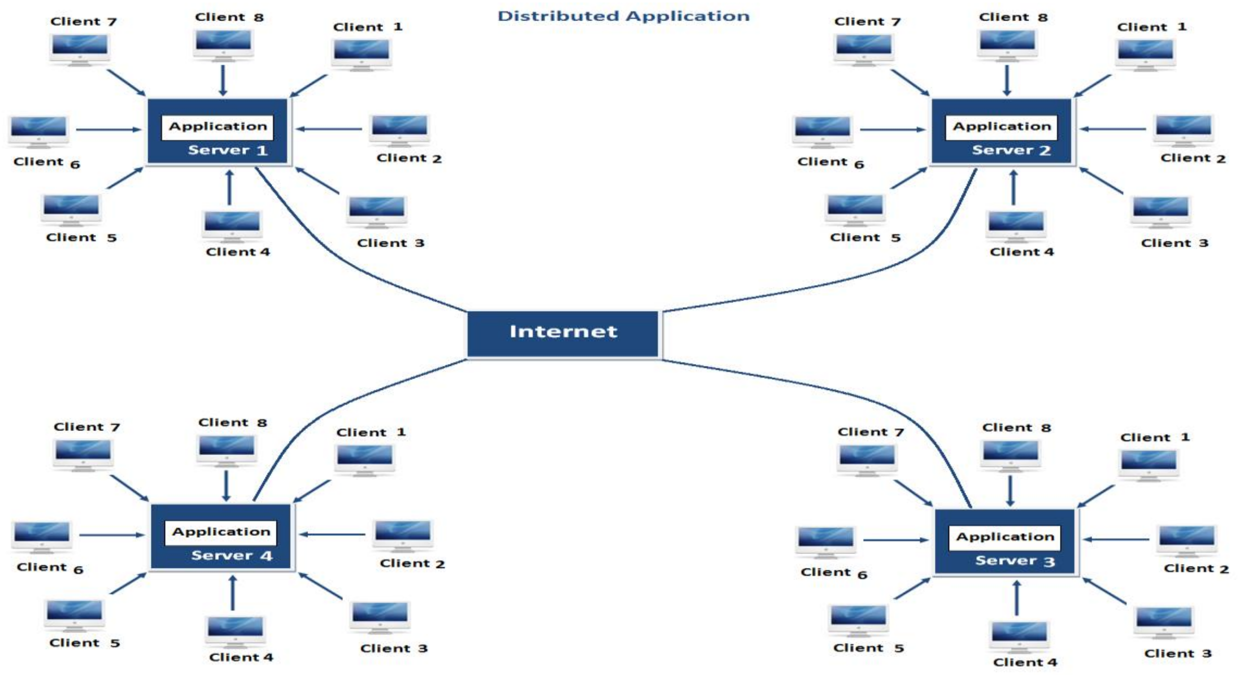
Java is Multithreading



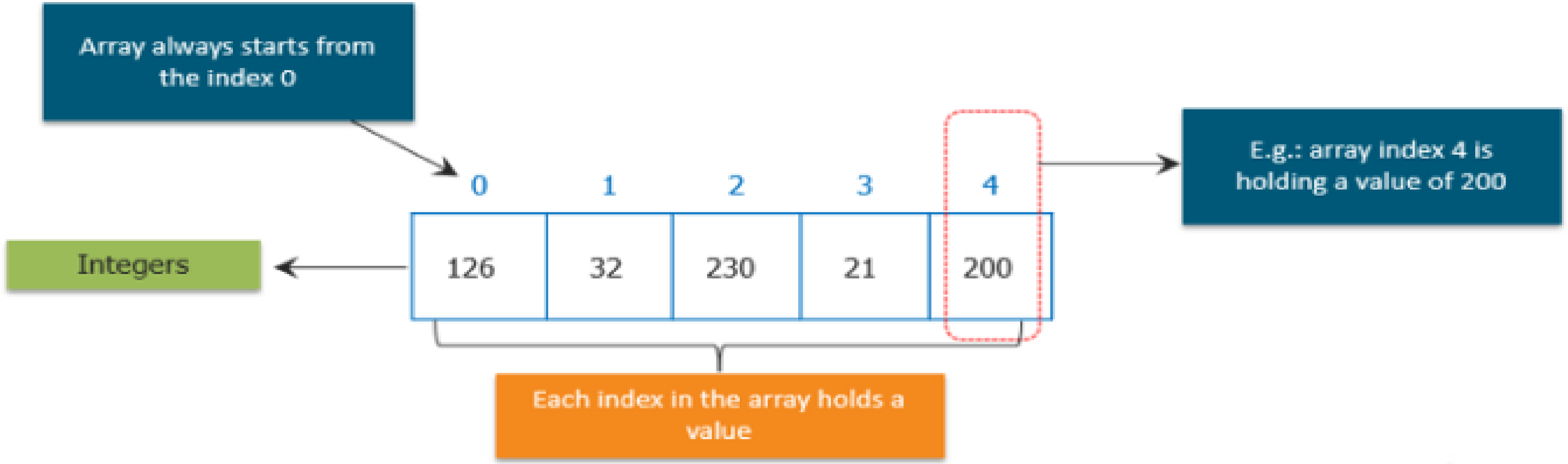
- Multithreaded means **handling multiple tasks simultaneously** or executing multiple portions (functions) of the same program in parallel.
- The code of **java is divided into smaller parts and Java executes them in a sequential and timely manner.**

Java is Distributed

- Multiple programmers **at many locations to work together on a single project.**
- Support **RMI (Remote Method Invocation) and EJB (Enterprise JavaBeans).**
- Extensive library of classes for interacting, **using TCP/IP protocols such as HTTP and FTP**, which makes creating network connections much easier than in C/C++.



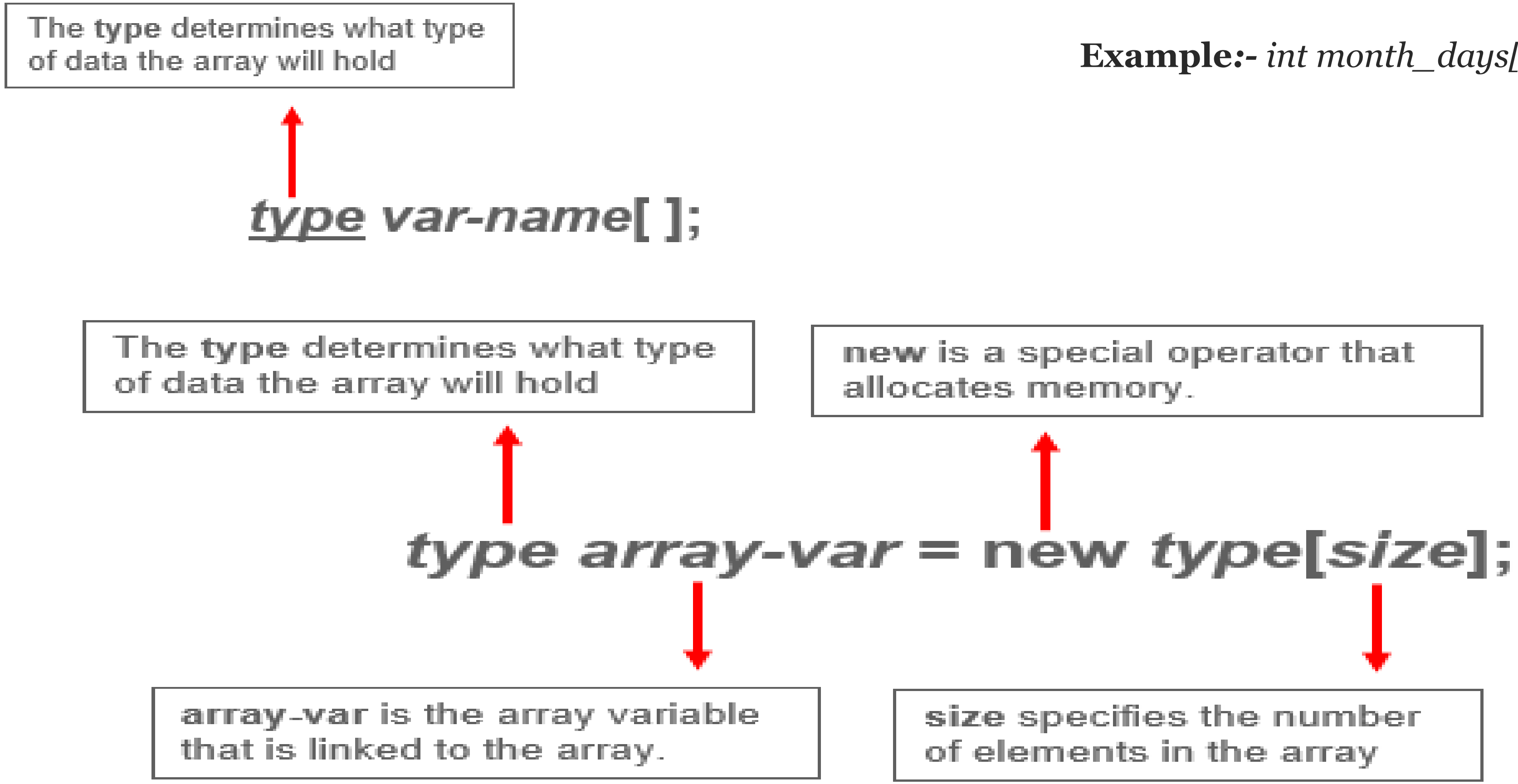
JAVA: Introduction to Array Data Type


Array always starts from the index 0
0
1
2
3
4
E.g.: array index 4 is holding a value of 200
Integers
126
32
230
21
200
Each index in the array holds a value

- Arrays in Java are **homogeneous data structures** implemented in Java as objects.
- Arrays **store one or more values** of a specific data type and provide indexed access to store the same.
- A **specific element in an array is accessed by its index.**
- Arrays offer a **convenient means of grouping related information.**

General Form of Java Array Initialization

Example:- *int month_days[];*



The type determines what type of data the array will hold

```
type var-name[ ];
```

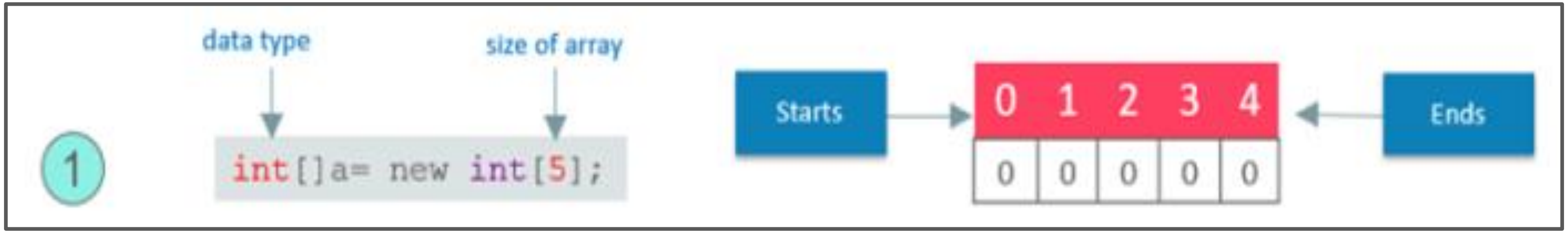
The type determines what type of data the array will hold

new is a special operator that allocates memory.

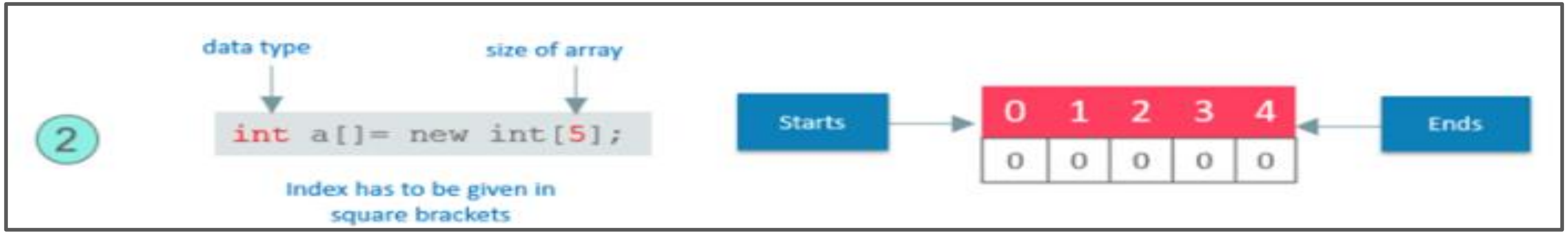
```
type array-var = new type[size];
```

array-var is the array variable that is linked to the array.

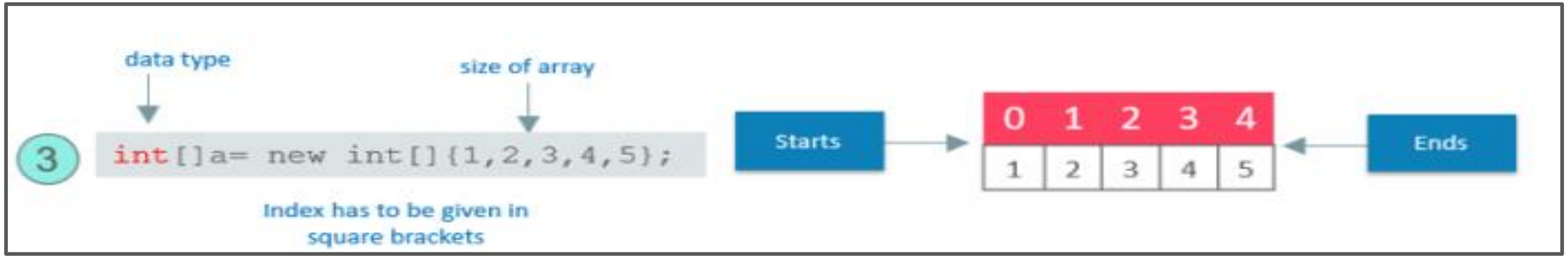
size specifies the number of elements in the array



```
int[]a= new int[5];
```



```
int a[]= new int[5];
```

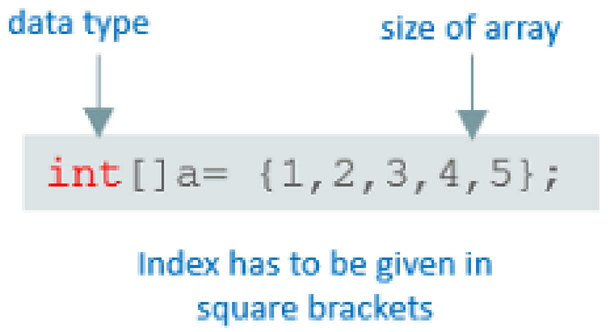


```
int[]a= new int[]{1,2,3,4,5};
```

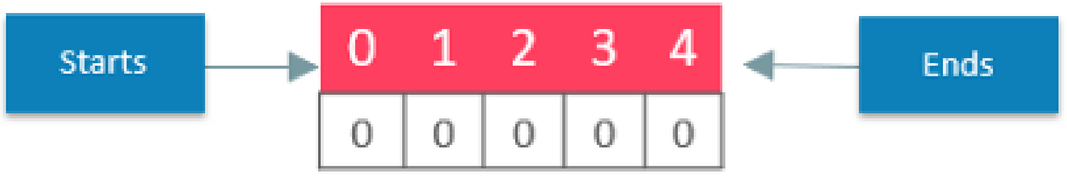
The type determines what type of data the array will hold

type var-name[] = {value1, value2, value3, value4,…};

An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements



```
int[]a= {1,2,3,4,5};
```



Implementing an Array

```
class MyArray{

public static void main(String args[]){

int month_days[ ] = {31,28,31,30,31,30,31,30,31,30,31};

                        System.out.println("April has " +
                month_days[3] + days.");

}

}
```

OUTPUT
```
April has 30days
```

Accessing a Specific Element in a Java Array

This statement assigns the value **90** to the second element of **month_days**

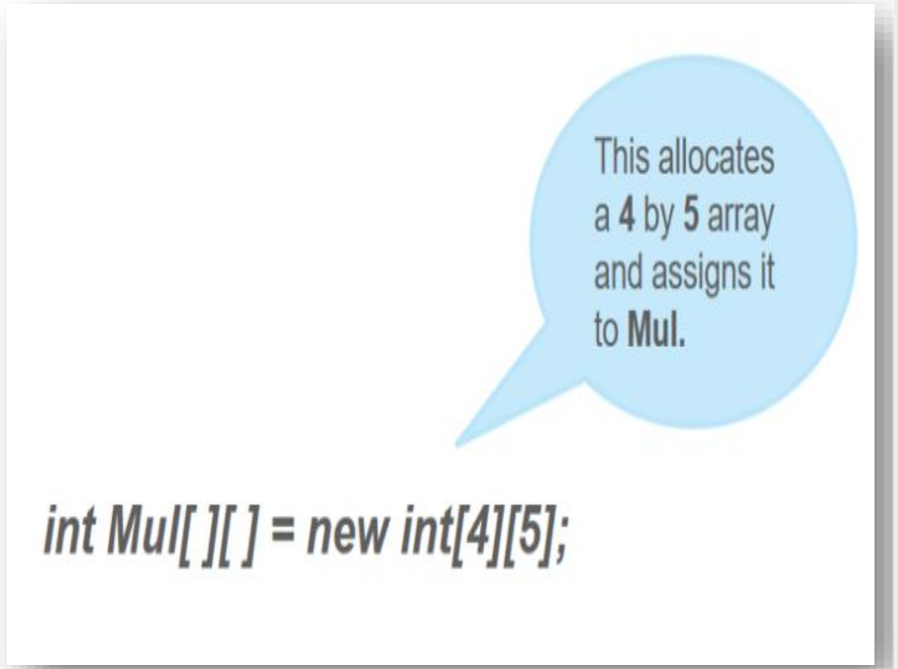
```
month_days[1] = 90;
```

```
public static void main(String args[]) {
   int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
     System.out.println("April has " + month_days[3] + "
days.");
     }
}
```

OUTPUT

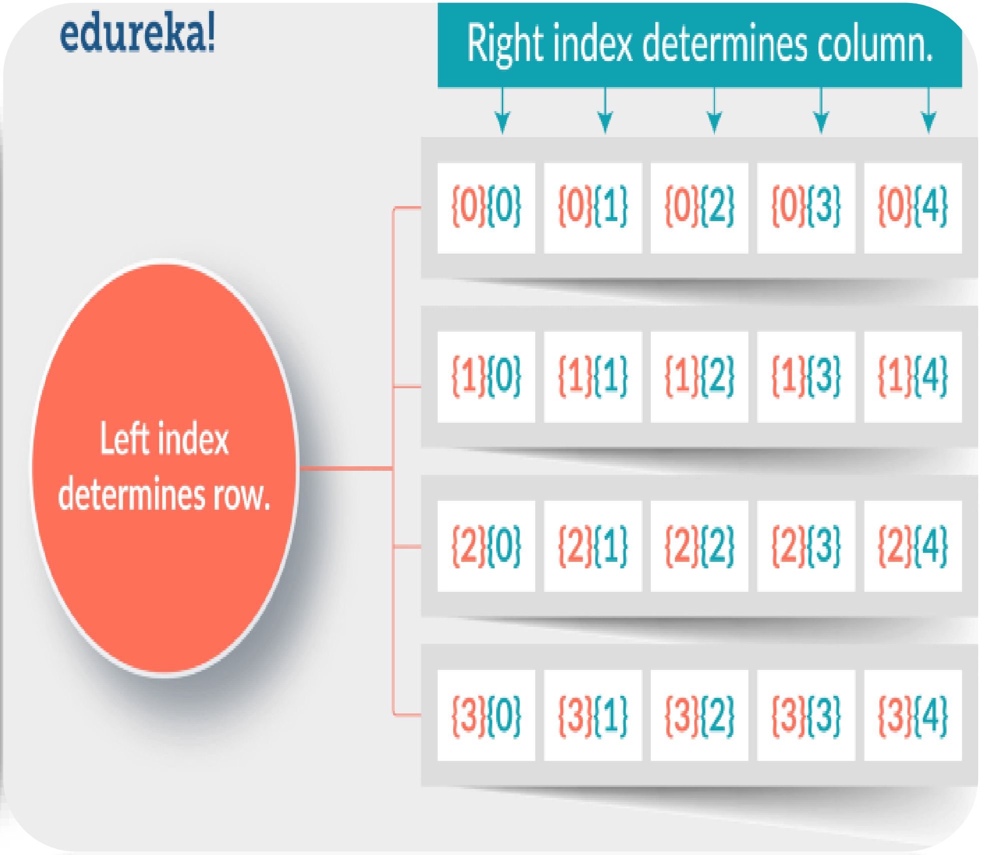
```
April has 30days
```

Multidimensional Array



This allocates a **4** by **5** array and assigns it to **Mul.**

```
int Mul[ ][ ] = new int[4][5];
```



```
class TwoDArray
{
  //-----------------------------------------------------------------
  //  Creates a 2D array of integers, fills it with increasing
  //  integer values, then prints them out.
  //-----------------------------------------------------------------
  public static void main (String[] args)
  {
    int[][] multarry = new int[4][5];
    int i,j,k=0;

    // Load the table with values
    for (i=0; i < 4;i++)
      for (j=0; j < 5; j++)
      {
        multarry[i][j]=k;
       k++;
      }
```

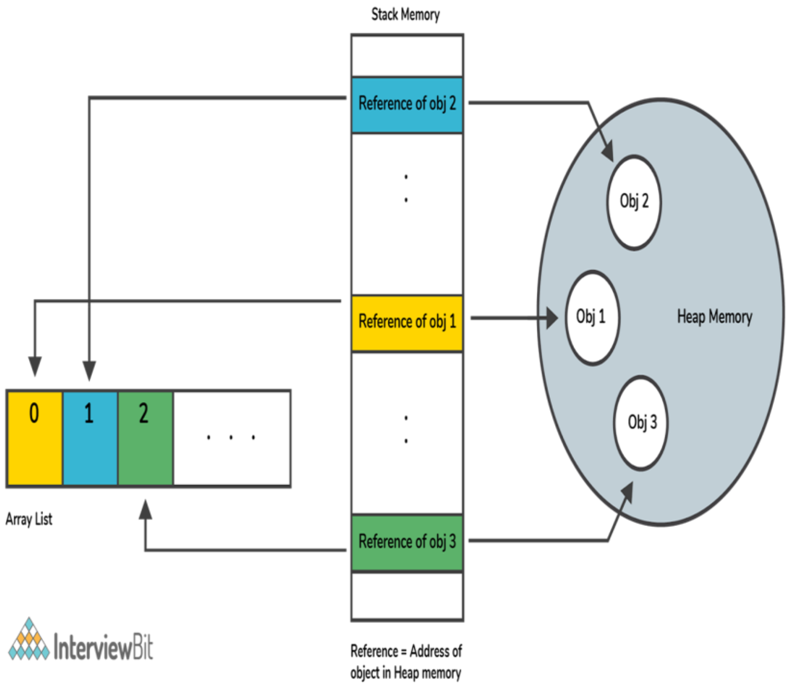
```
// Print the table
    for (i=0; i < 4;i++)
    {
      for (j=0; j < 5; j++)
      {
        System.out.print( multarry[i][j]+" ");
       }

         System.out.println();

    }
  }
}
```

OUTPUT

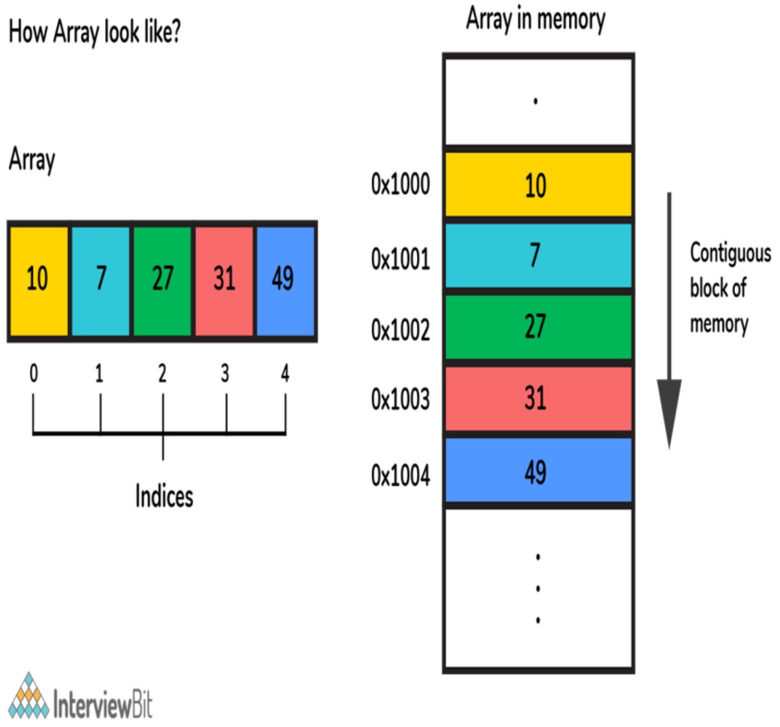
```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Array Vs ArrayList

Sorting of objects in Array List



Sorting values in Array



Passing Java Array to a Method

```
class PMethods{
public static void display(int y[])
   {
         System.out.println(y[0]);
         System.out.println(y[1]);
         System.out.println(y[2]);

   }
public static void main(String args[])
   {
   int x[] = { 1, 2, 3 };
   display(x);  //Passed array x to method display
   }
}
```

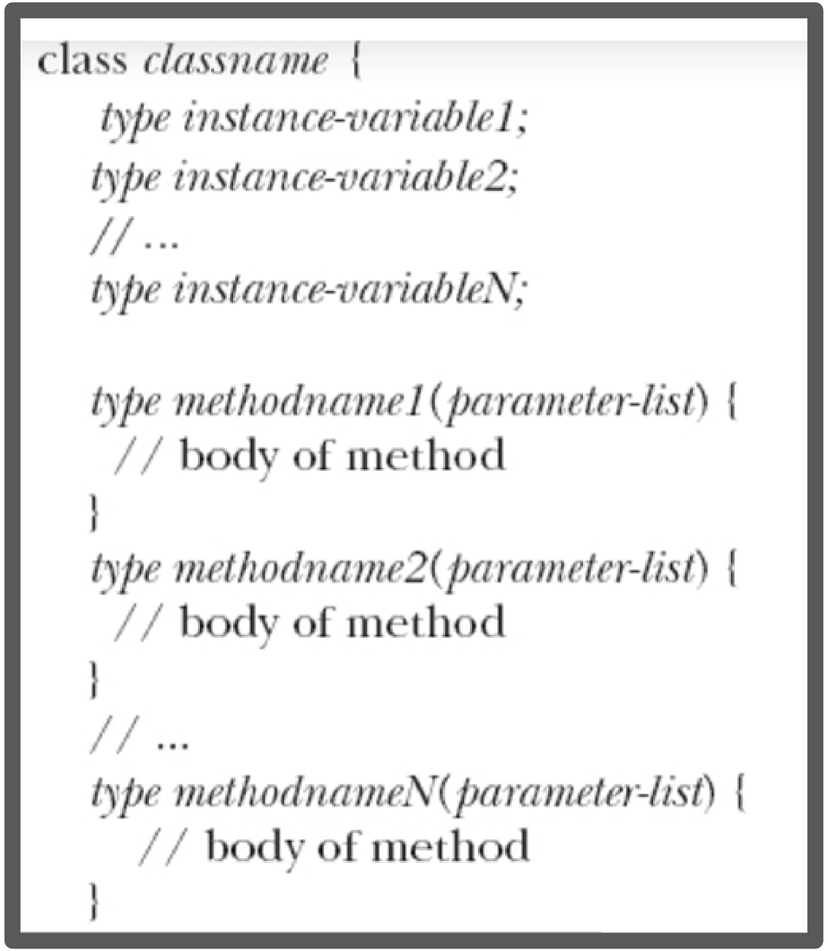
OUTPUT

```
1
2
3
```

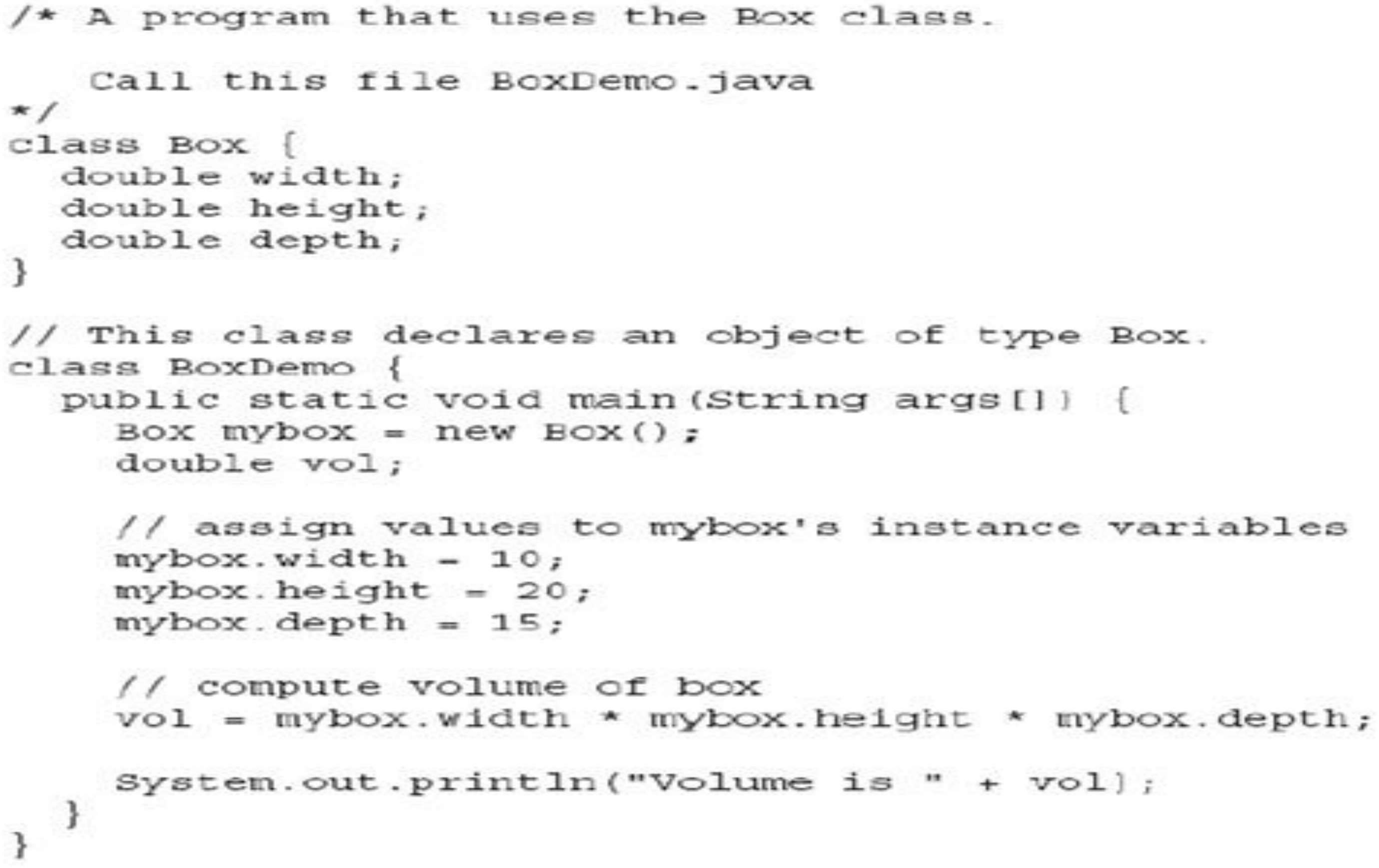
Class Fundamentals

- A class is that it defines a n**ew data type.**
- Once defined, this new type can be used **to create objects of that type.**
- A class is a **template for an object**, and an object is **an instance of a class**. Because an object is an instance of a class
- Two word **object and instance used interchangeably.**

The General Form of a Class



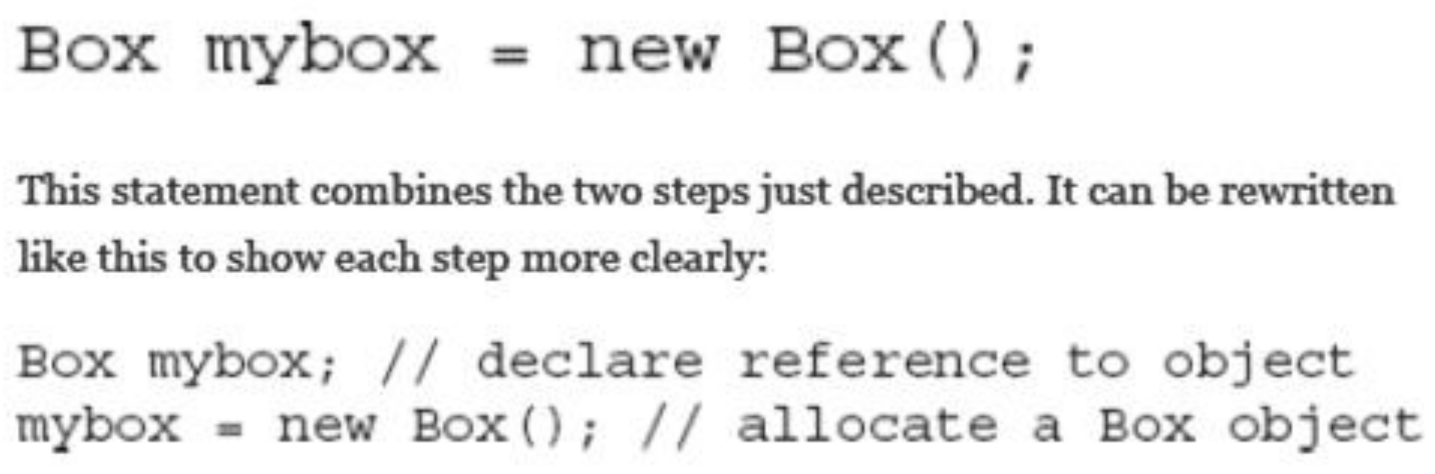
```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
    }
    // ...
    type methodnameN(parameter-list) {
       // body of method
    }
```



```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}

// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Volume is " + vol);
  }
}
```

Declaring Objects

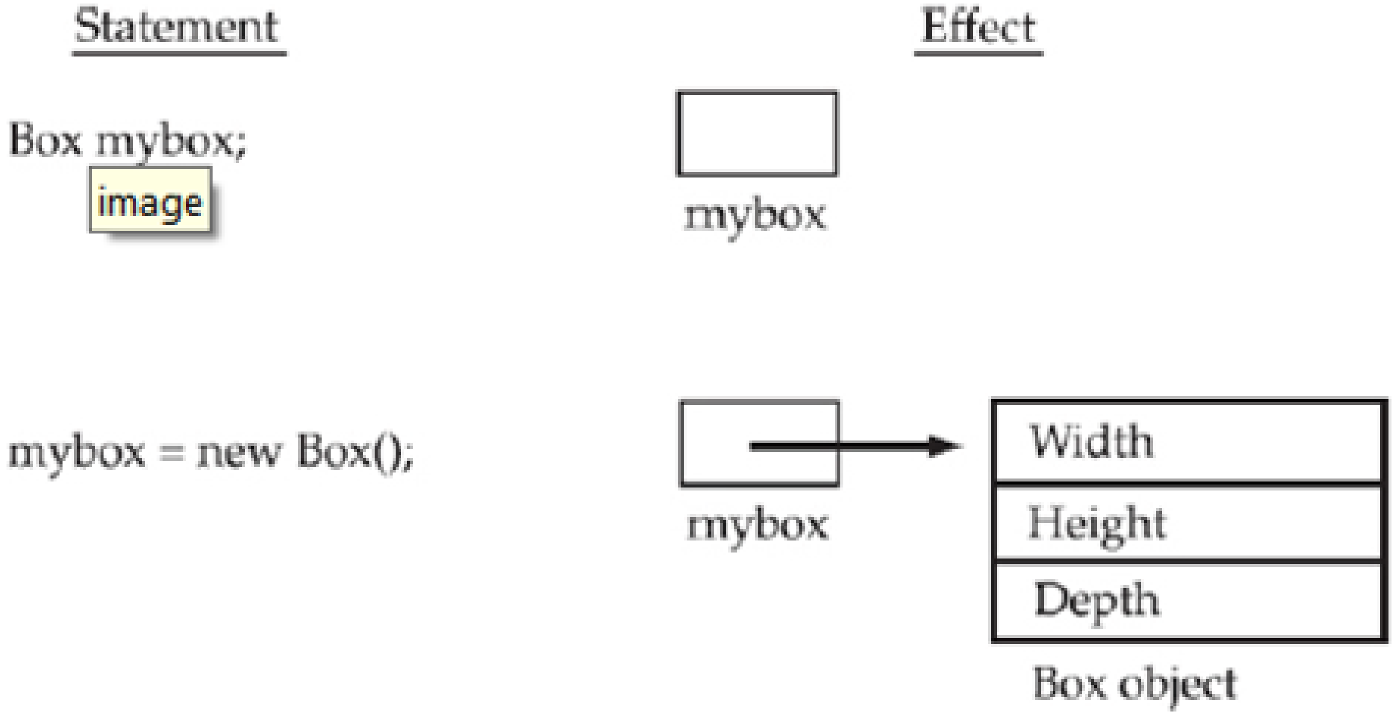
- The **new operator dynamically allocates** (that is, allocates at **run time) memory** for an object **and returns a reference to it**.
- This reference is, more or less, **the address in memory of the object allocated by new**
- **This reference is then stored in the variable.** Thus, in Java, all class **objects must be dynamically allocated.**



```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

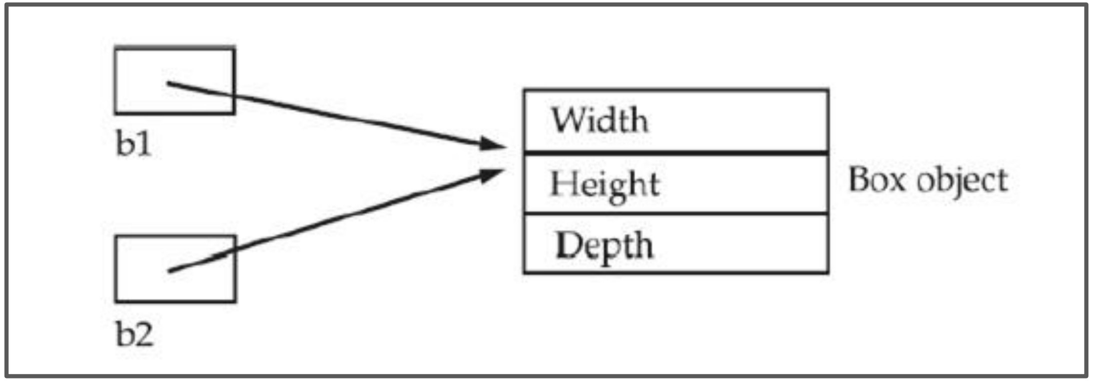


```
Box mybox;
```

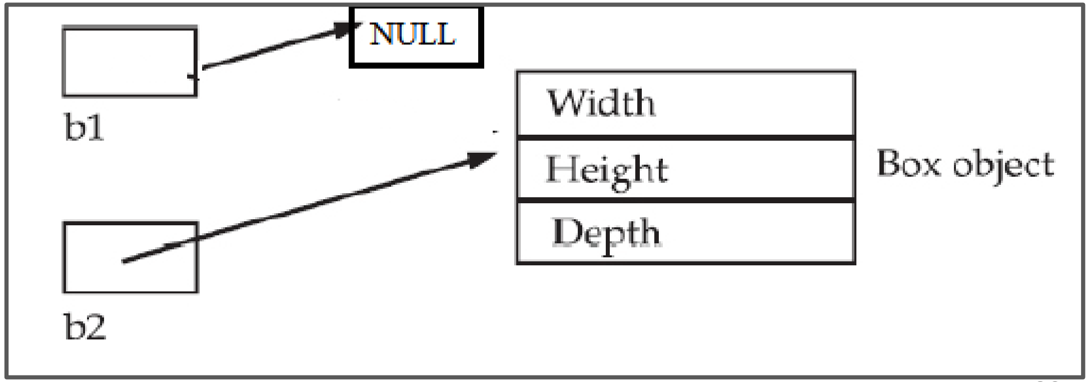
```
mybox = new Box();
```

Assigning Object Reference Variables

```
Box b1 = new Box();
Box b2 = b1;
```

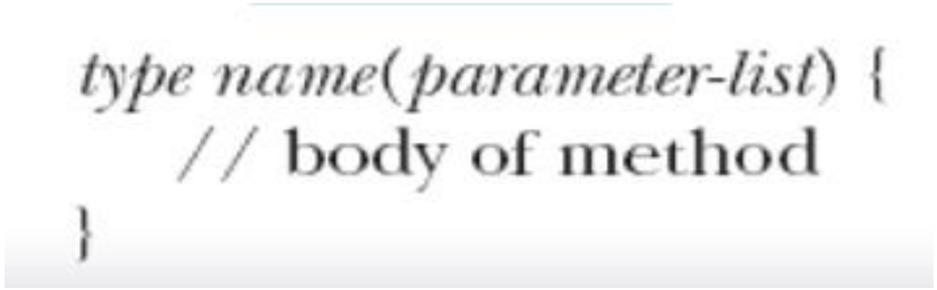


```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```



Introducing Methods

General form of a method

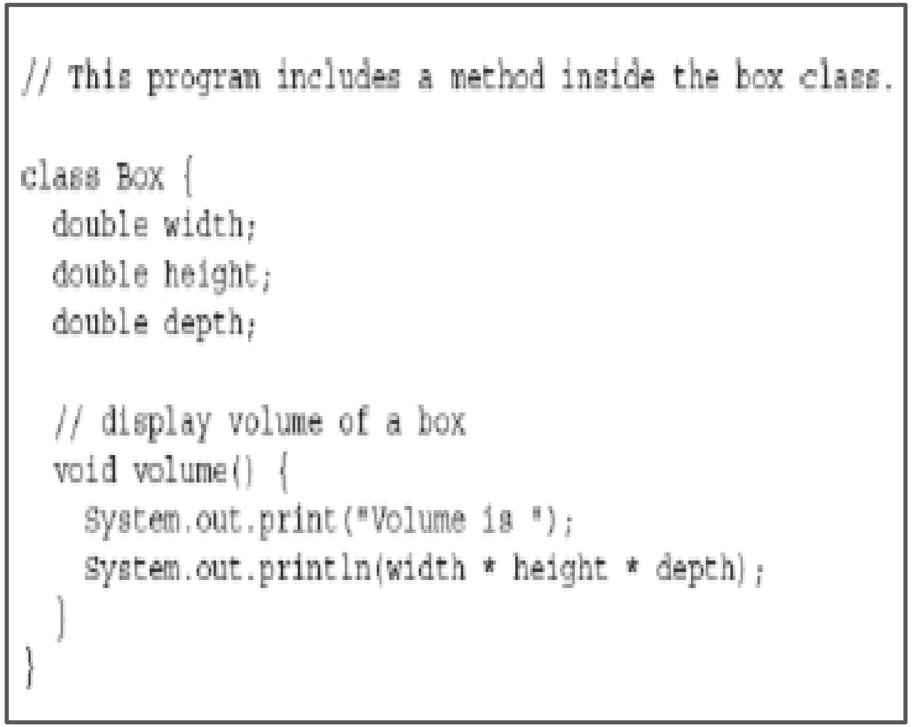


```
type name(parameter-list) {
    // body of method
}
```

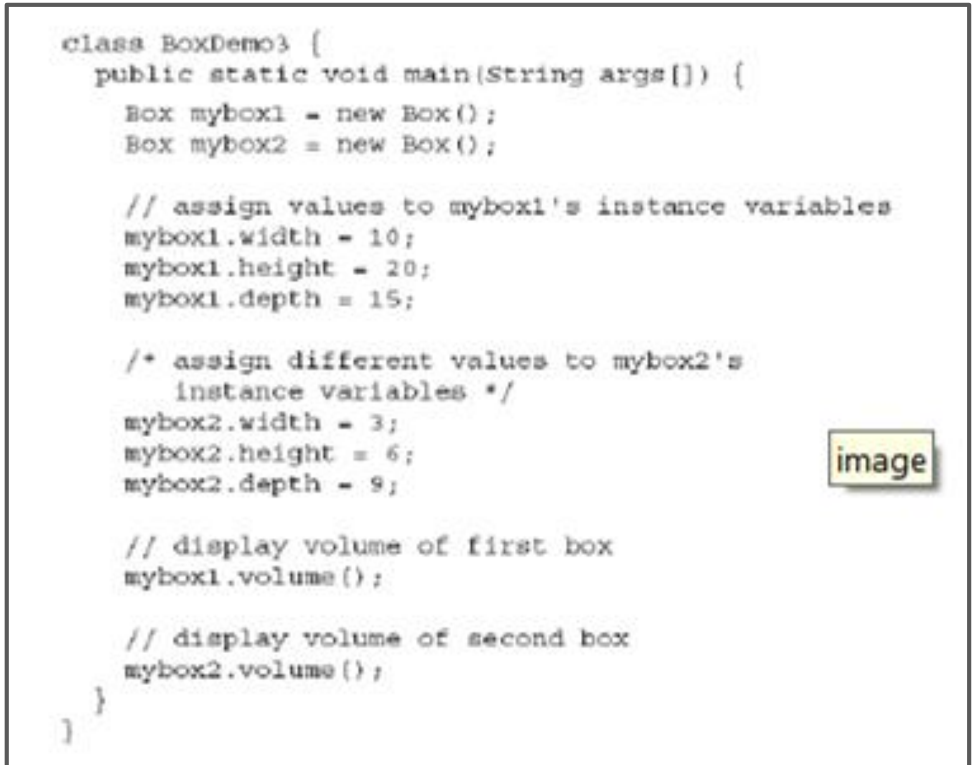
Adding a Method to the Class

- In fact, methods **define the interface to most classes.** This allows the class **implementor to hide the specific layout of internal data structures** behind cleaner method abstractions.
- In addition **to defining methods that provide access to data**,you can also define **methods that are used internally by the class itself.**

- **The instance variables** width, height, and depth **are referred to directly**,
- **without** preceding them with a**n object name or the dot operator.**
- When an instance variable is accessed by c**ode that is not part of the class** in which that instance variable is defined, it **must be done through an object, by use of the dot operator.**



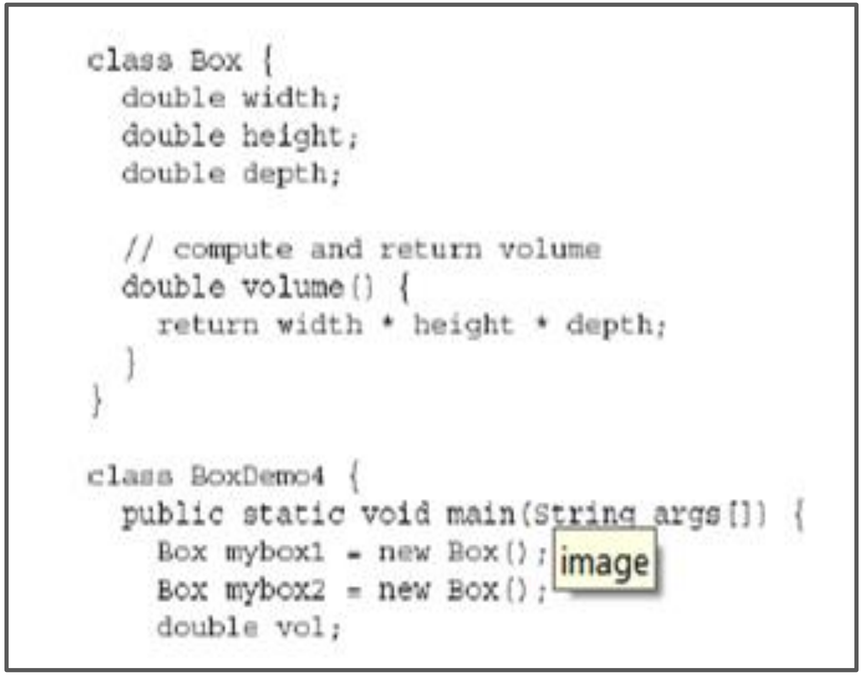
```
// This program includes a method inside the box class.

class Box {
  double width;
  double height;
  double depth;

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}
```



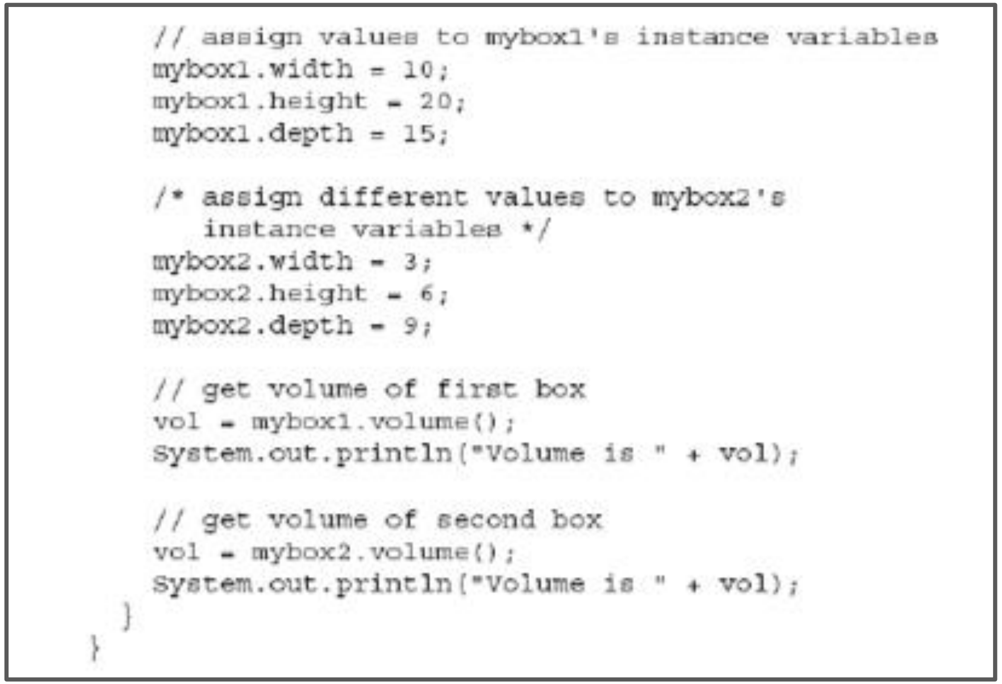
```
class BoxDemo3 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // display volume of first box
    mybox1.volume();

    // display volume of second box
    mybox2.volume();
  }
}
```

Returning a Value

- The type of **data returned by a method must be compatible with the return type** specified by the method.
- For example, if the return type of some method is boolean , you could not return an integer.
- The variable **receiving the value returned by a method** (such as vol in this case) must also be **compatible with the return type specified for the method**



```
class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
```



```
    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Adding a Method That Takes Parameters

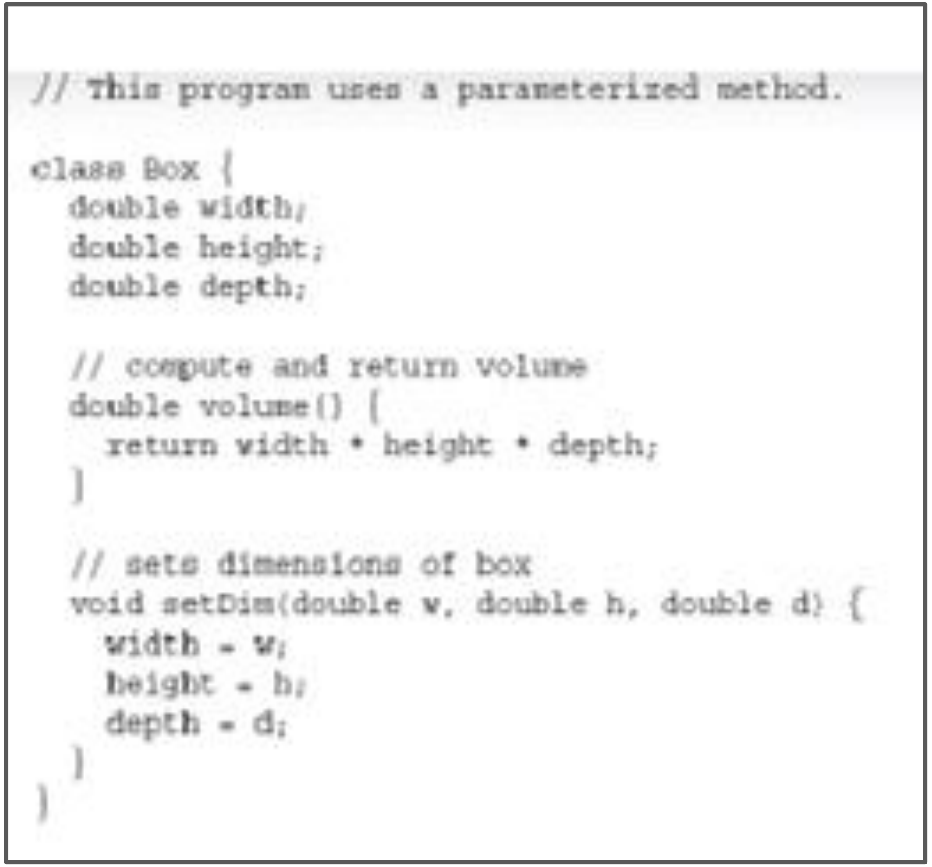
```
int square(int i)
{
  return i * i;
}
```

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

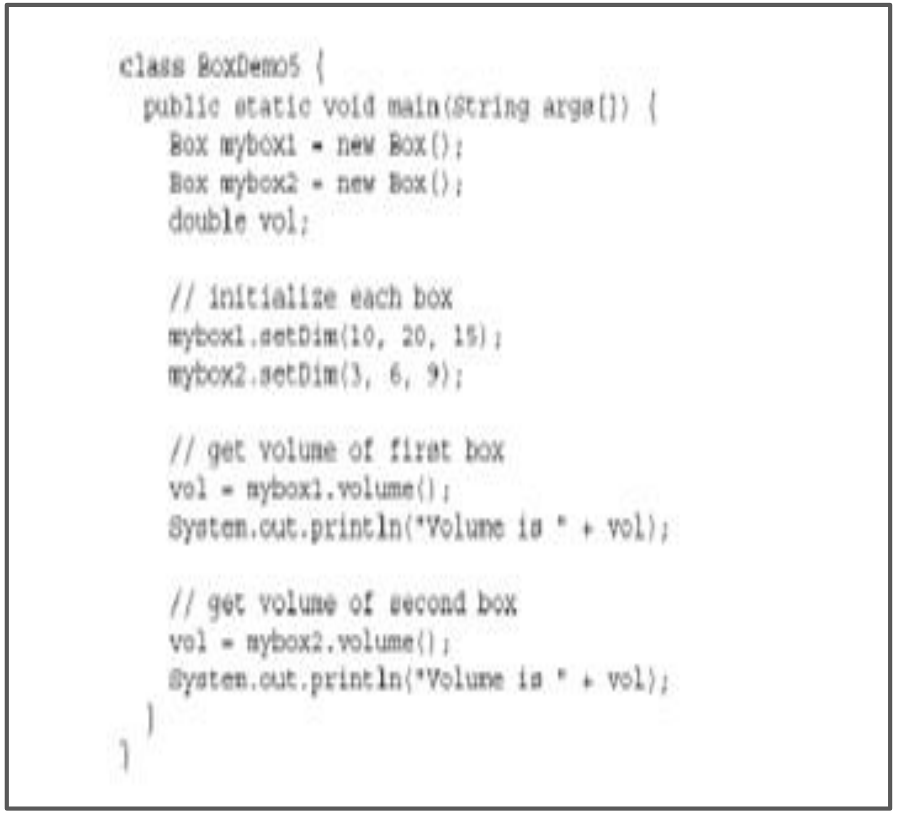
Returning a Value

- **A parameter** is a variable defined by a method **that receives a value when the method is called.**
- For example, **in square( ) , i is a parameter.**
- **An argument is a value that is passed to a method** when it is invoked. For example, **square(100) passes 100 as an argument.**
- **Inside square( ) , the parameter i receives that value.**

Modified program using parameterized methods



```
// This program uses a parameterized method.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }

  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}
```



```
class BoxDemo5 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Constructor

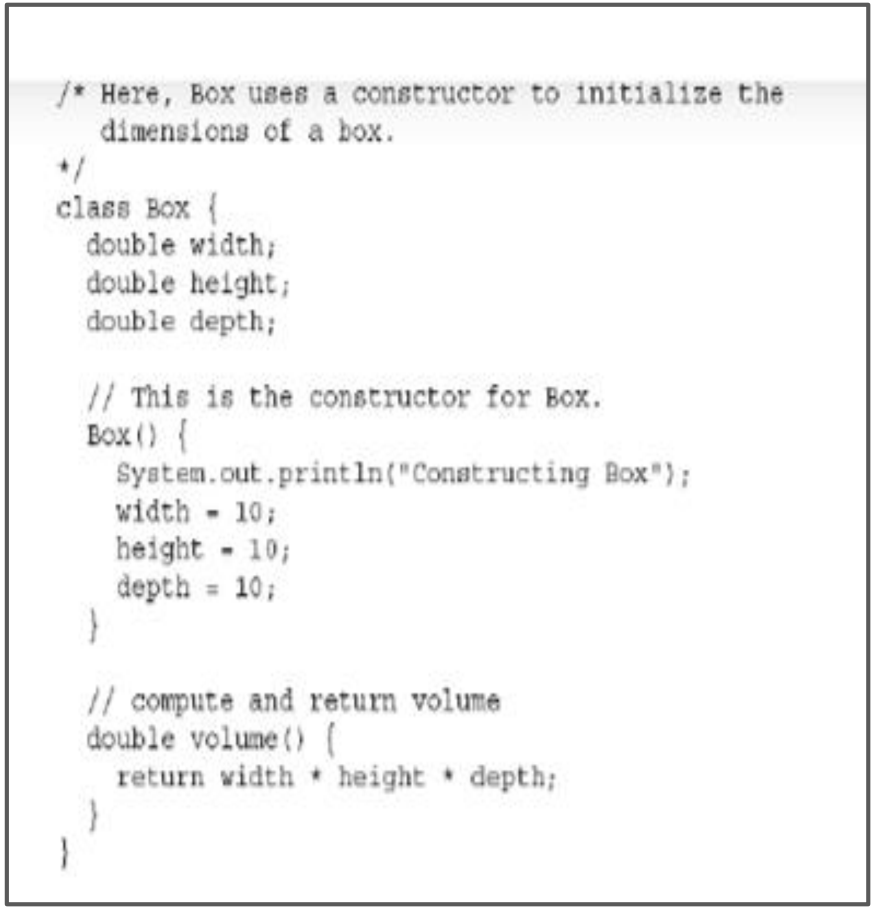
- A constructor **initializes an object immediately upon creation**.
- It has the **same name as the class** in which it resides and is syntactically similar to a method.
- Once defined, **the constructor is automatically called** when the object is created, **before the new operator completes**

Types of Constructors in Java

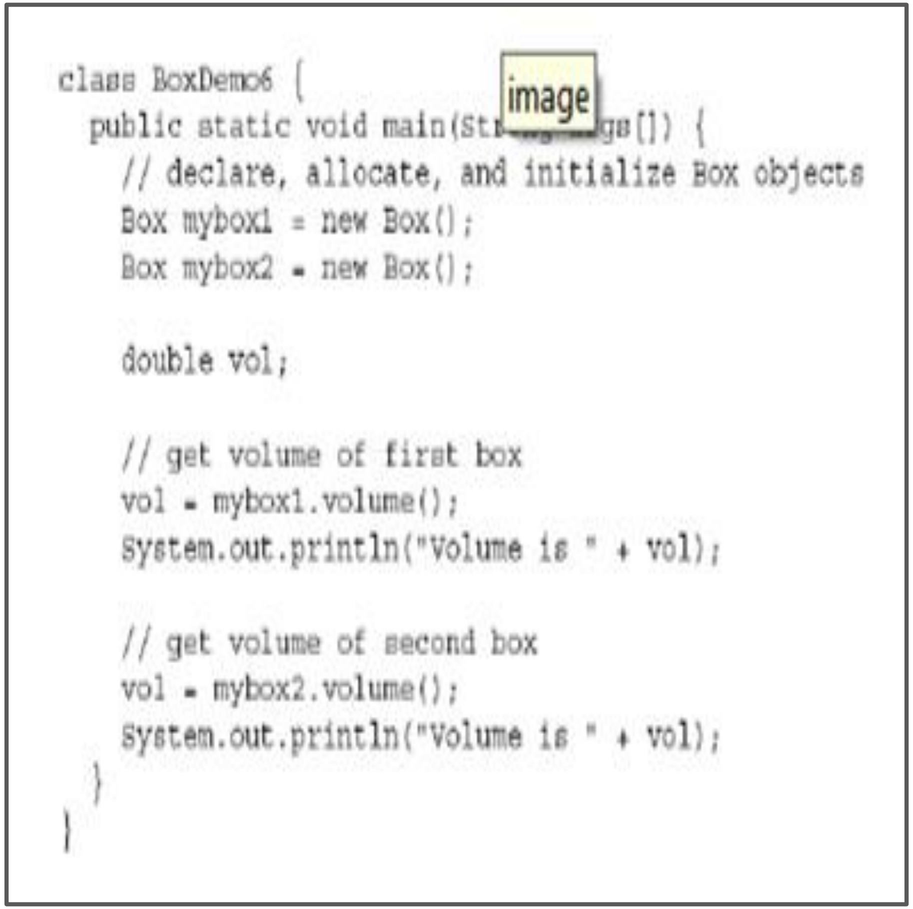
- Default Constructor
- Parameterized Constructor
- Copy Constructor

Default Constructor

- A constructor that has no parameters is known as default constructor.
- A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor.
- It is taken out. It is being overloaded and called a parameterized constructor.
- The default constructor changed into the parameterized constructor. But Parameterized constructor can’t change the default constructor. The default constructor can be implicit or explicit.



```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

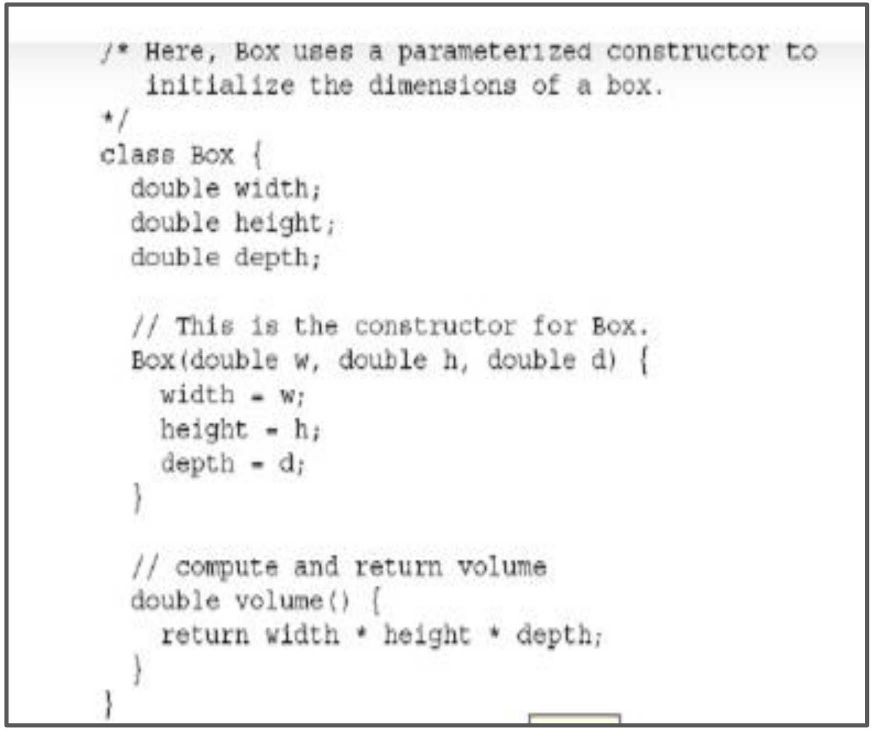


```
class BoxDemo6 {
  public static void main(St[illegible]gs[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

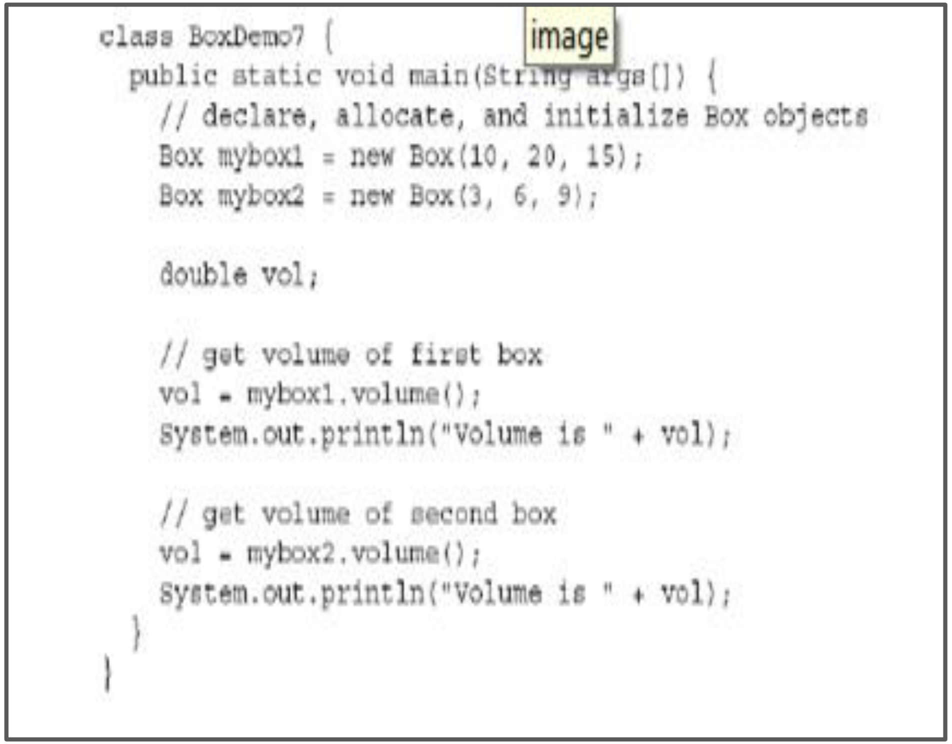
- The **default constructor automatically initializes all instance variables** to their **default values**,
    - which are zero for numeric types,
    - null for reference types ,
    - and false for boolean
- The default constructor **is often sufficient for simple classes**, but it usually won’t do for more sophisticated ones.
- Once you define **your own constructor, the default constructor is no longer used.**

Parameterized Constructor

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.



```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```



```
class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

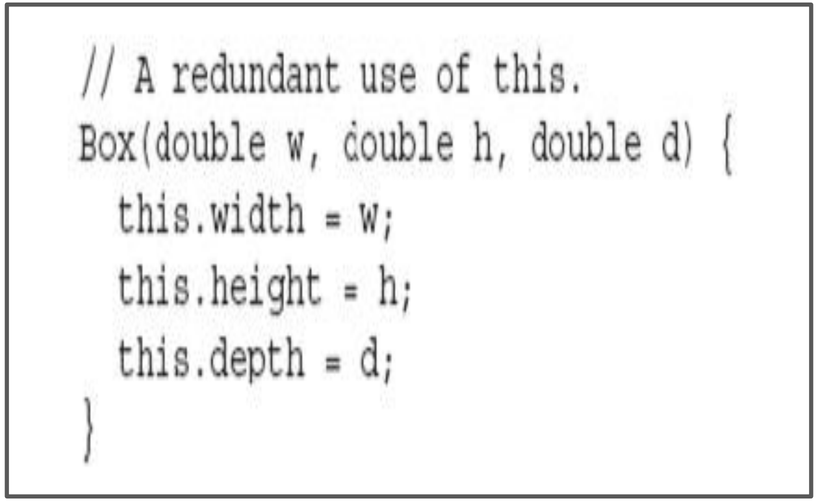
Copy Constructor in Java

copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

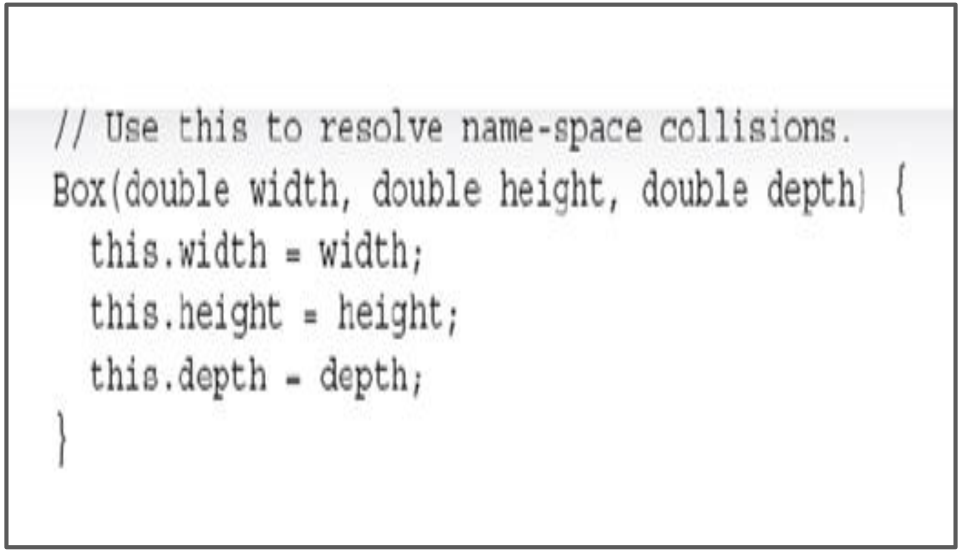
```
class Person {
        private String name;
        private int age;

        public Person(String name, int age) {
                this.name = name;
                this.age = age;
        }

        public Person(Person another) {
                this(another.name, another.age);
        }

        // Getters and setters for the instance variables
}
```

The this Keyword

The this Keyword and Instance Variable Hiding



```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```



```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

- Sometimes a method will need to refer **to the object that invoked it.** To allow this, Java defines the this keyword.
- This can be used **inside any method to refer to the current object.**
- That is, this is always **a reference to the object on which the method was invoked**.
- You can use **this any where a reference to an object of the current class' type is permitted.**

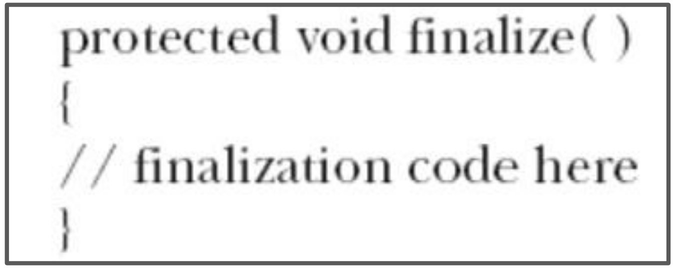
Garbage Collection

- It is **automatic deallocation.**
- when **no references to an object exist**, that object is assumed to be **no longer needed,** and the memory occupied **by the object can be reclaimed**.
- There is **no explicit need to destroy objects as in C++**.
- only occurs sporadically (If at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.
- **different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.**

The finalize( ) Method

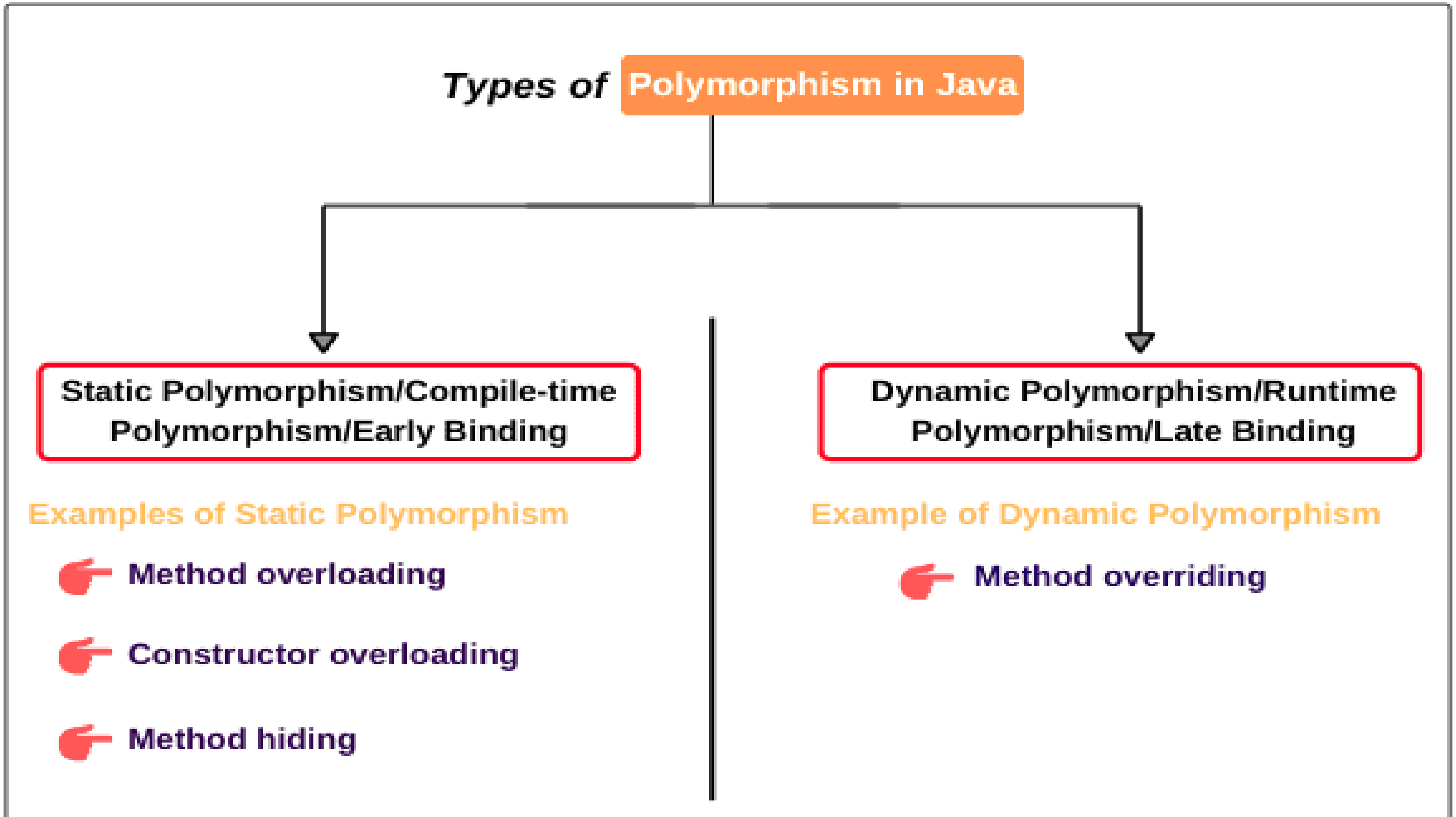
- Sometimes an object will **need to perform some action when it is destroyed**.
- For example, **if an object is holding some non-Java resource** such as a file handle or character font, then you might want to **make sure these resources are freed before an object is destroyed.**
- To handle such situations, Java provides a mechanism called finalization
- you can define **specific actions that will occur when an object is just about to be reclaimed by the garbage collector.**

- To add a finalizer to a class, you simply define **the finalize( ) Method.**
- The Java **run time calls that method whenever it is about to recycle an object of that class.**
- Inside the finalize( ) method, you will specify thos eactions that must be performed before an object is destroyed.



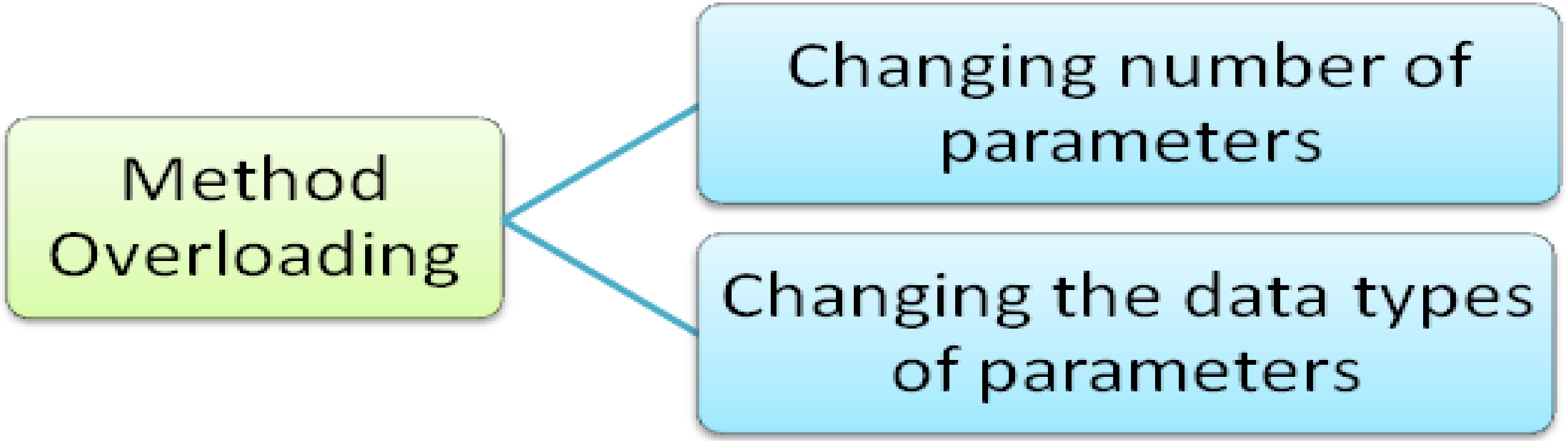
```
protected void finalize( )
{
// finalization code here
}
```

- It is important t**o understand that finalize( ) is only called just prior to garbage collection.**
- It is not called when **an object goes out-of-scope**
- For example. This means that you cannot know when—or even if—finalize( ) will be executed. Therefore, your program should provide **other means of releasing system resources, etc., used by the object.**
- **It must not rely on finalize( ) for normal program operation**

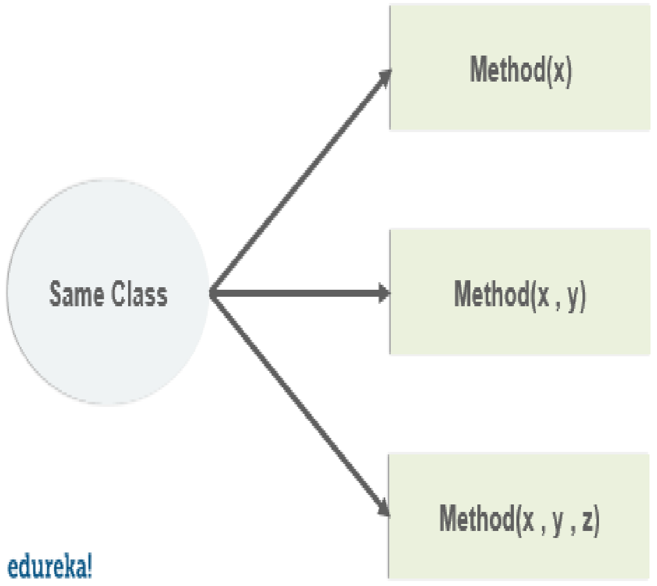

Types of Polymorphism in Java
Static Polymorphism/Compile-time Polymorphism/Early Binding
Dynamic Polymorphism/Runtime Polymorphism/Late Binding
Examples of Static Polymorphism
Method overloading
Constructor overloading
Method hiding
Example of Dynamic Polymorphism
Method overriding

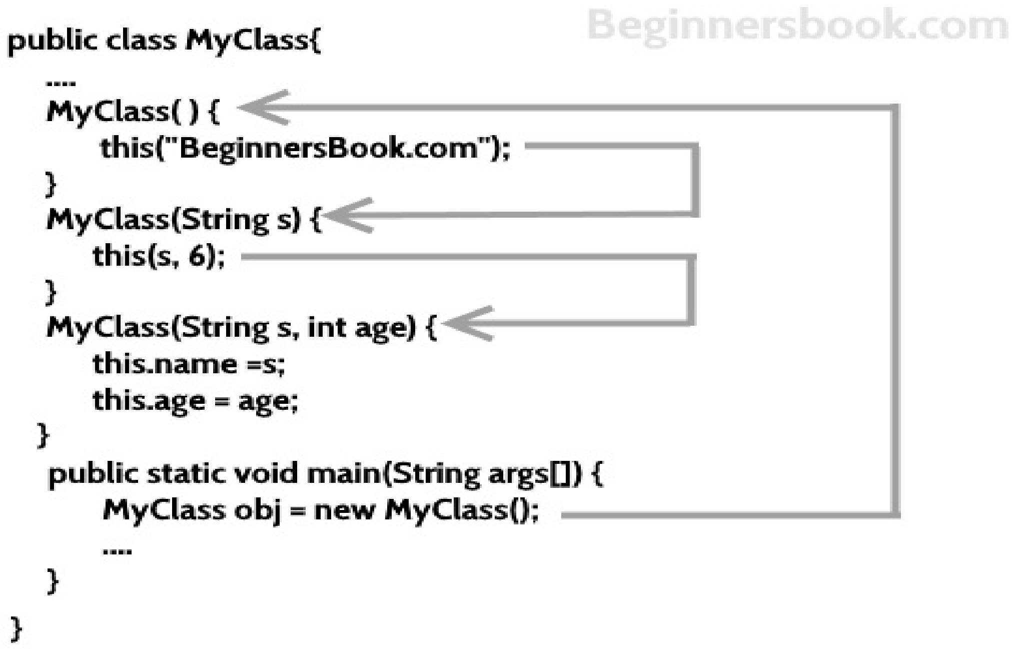
Method Overloading

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call
- The return **type alone is insufficient to distinguish two versions of a method.**


Method Overloading
Changing number of parameters
Changing the data types of parameters

Constructor Overloading





```
public class MyClass{
    ....
    MyClass( ) {
        this("BeginnersBook.com");
    }
    MyClass(String s) {
        this(s, 6);
    }
    MyClass(String s, int age) {
        this.name =s;
        this.age = age;
    }
    public static void main(String args[]) {
        MyClass obj = new MyClass();
        ....
    }
}
```

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

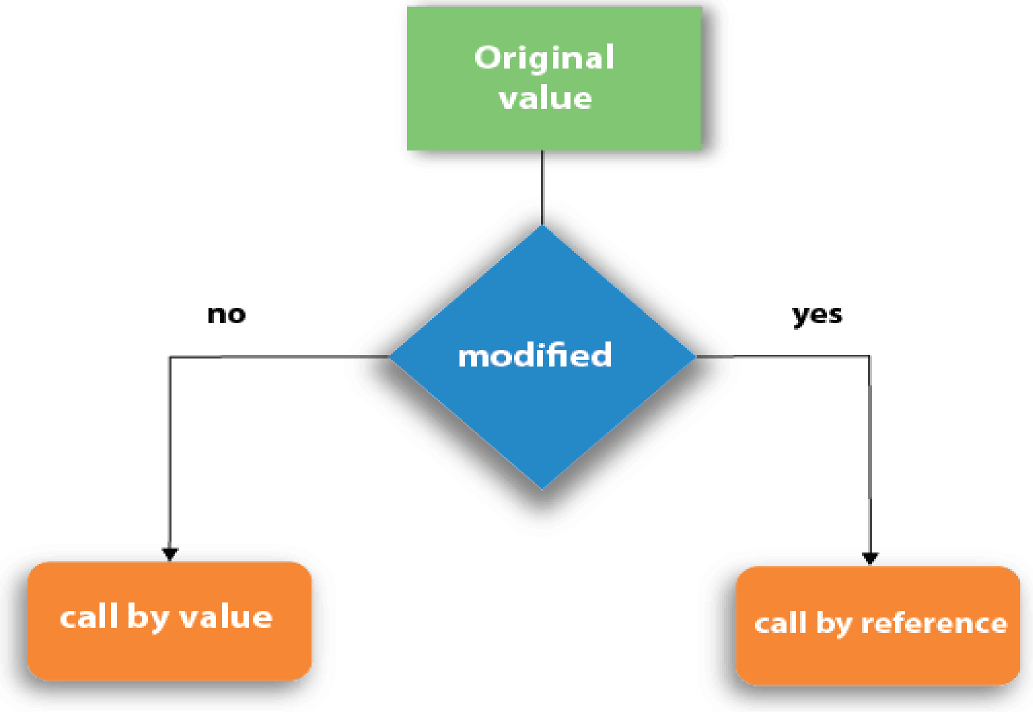
Objects as Parameters

- Objects, like primitive types, can be passed as parameters to methods in Java.
- When passing an object as a parameter to a method, a reference to the object is passed rather than a copy of the object itself.
- This means that any modifications made to the object within the method will have an impact on the original object.

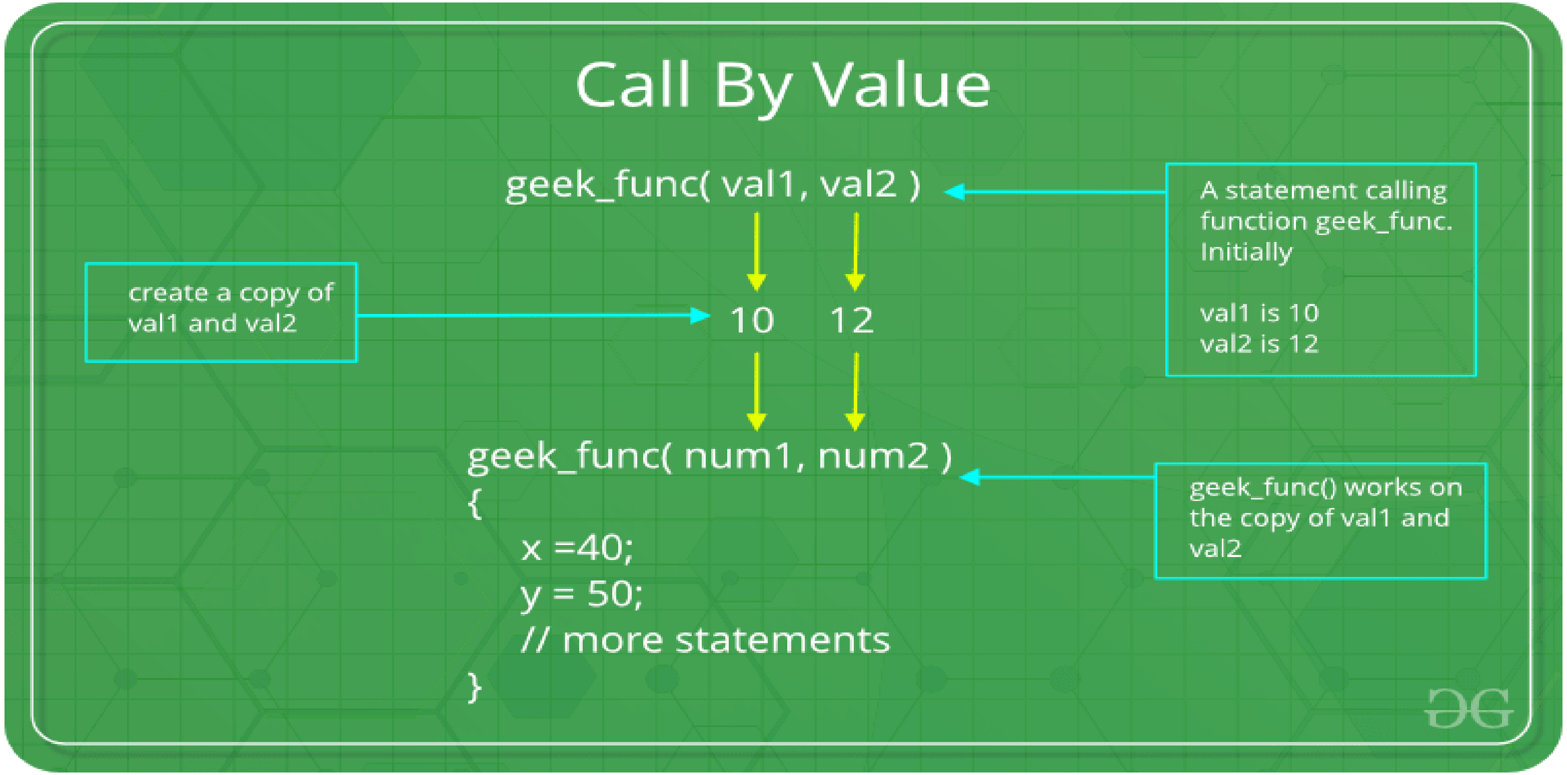
```
public class MyClass
{ // Fields or attributes
private int attribute1;
private String attribute2;
private double attribute3; // Constructor
public MyClass(int attribute1, String attribute2, double attribute3)
{
   this.attribute1 = attribute1;
   this.attribute2 = attribute2;
   this.attribute3 = attribute3;
} // Method with object as parameter
public void myMethod(MyClass obj)
{ // block of code to define this method
} // More methods
}
```

An object belonging to the "MyClass" class is used as a parameter in the myMethod() function. This enables the method to operate on the passed object and access its attributes and methods.

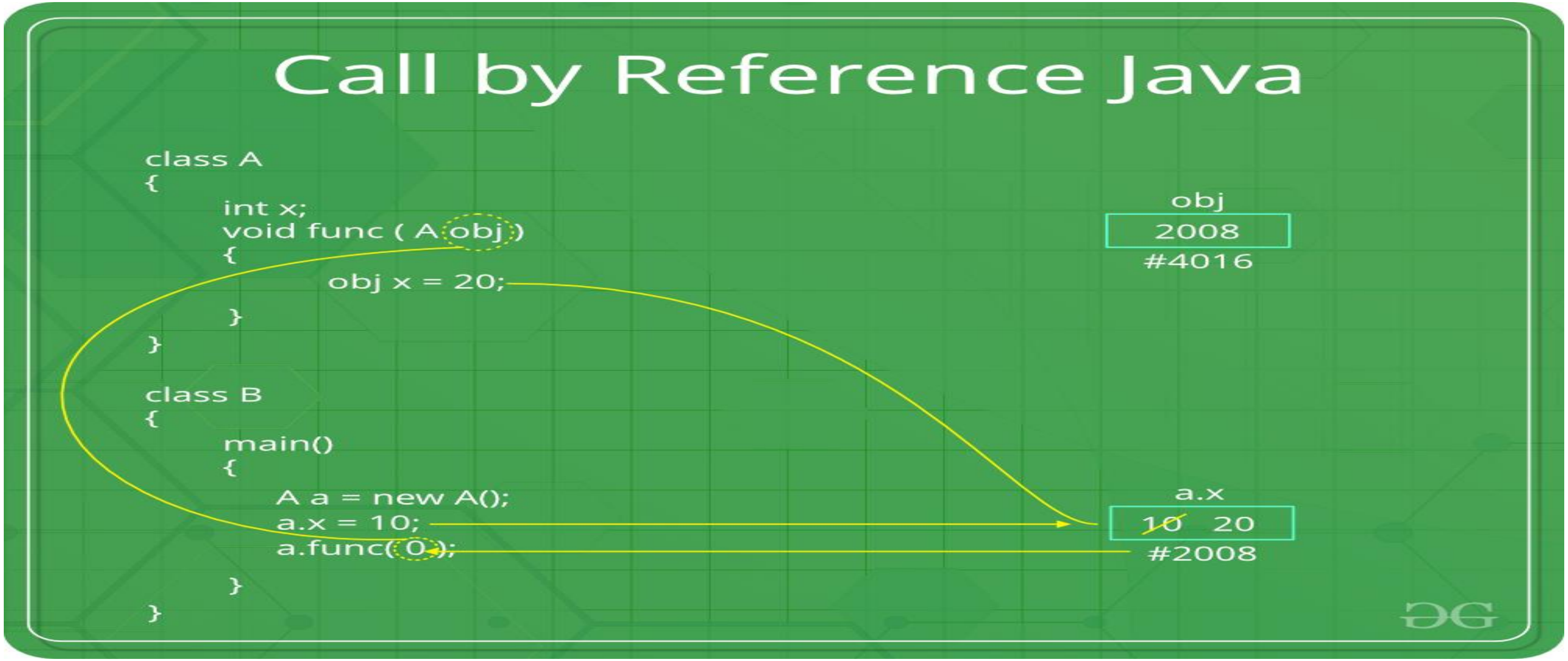
Argument Passing



- **Call by Value** : When a primitive type is passed to a method
- **Call by Reference** : objects are implicitly passed to a method


Call By Value
geek_func(val1, val2)
A statement calling function geek_func. Initially
val1 is 10
val2 is 12
create a copy of val1 and val2
10
12
geek_func(num1, num2)
{
x =40;
y = 50;
// more statements
}
geek_func() works on the copy of val1 and val2

Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as ***call by value***.



```
class A
{
        int x;
        void func ( A obj )
        {
                obj x = 20;

        }
}

class B
{
        main()
        {
            A a = new A();
            a.x = 10;
            a.func( 0 );

        }
}
```

Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **call by reference**. This method is efficient in both time and space.

Returning Objects

Similar to returning primitive types, Java methods can also return objects. A reference to the object is returned when an object is returned from a method, and the calling method can use that reference to access the object.

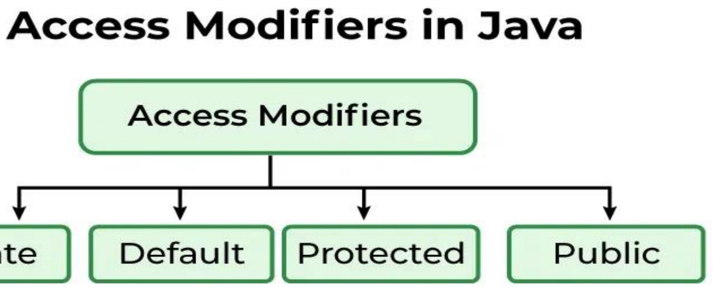
```
public class ObjectAsReturnValueExample
{
private int attribute1;
private String attribute2;
public ObjectAsReturnValueExample(int attribute1, String attribute2)
{
this.attribute1 = attribute1;
this.attribute2 = attribute2;
}
public ObjectAsReturnValueExample modifyObject()
{
 int newVariable1 = attribute1 + 10;
String newVariable2 = attribute2.toUpperCase();
return new ObjectAsReturnValueExample(newVariable1, newVariable2);
}
public static void main(String[] args)
{
ObjectAsReturnValueExample obj1 = new ObjectAsReturnValueExample(5, “Java");
ObjectAsReturnValueExample obj2 = obj1.modifyObject();
System.out.println("Object 1: attribute1 = " + obj1.attribute1 + ", attribute2 = " + obj1.attribute2);
System.out.println("Object 2: attribute1 = " + obj2.attribute1 + ", attribute2 = " + obj2.attribute2);
}}
```

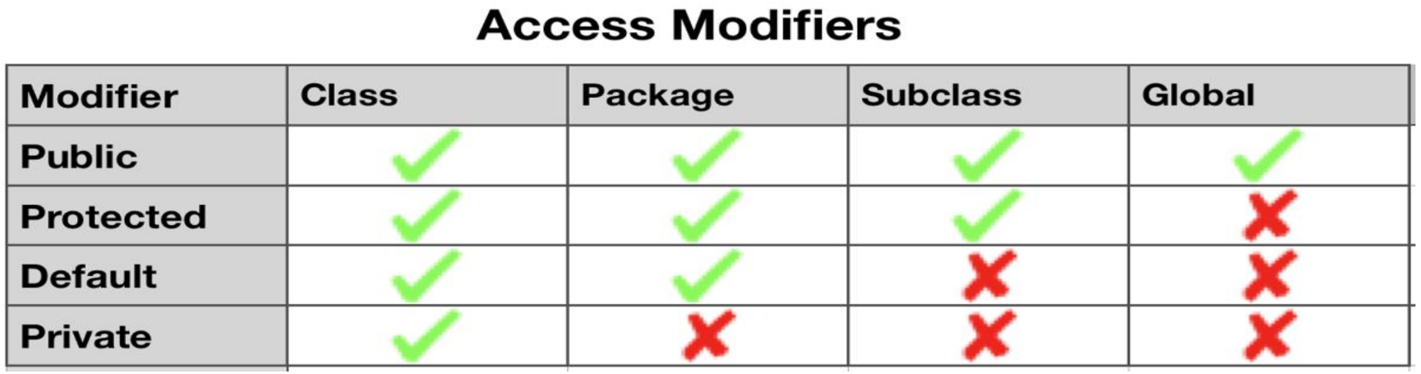
Output:

```
Object 1: attribute1 = 5, attribute2 = Java
Object 2: attribute1 = 15, attribute2 = JAVA
```

Introducing Access Control

In Java, Access modifiers helps to restrict the **scope of a class**, **constructor**, **variable**, **method**, or **data member**. It provides **security, accessibility**, etc. to the user depending upon the access modifier used with the element. In this article, let us learn about **Java Access Modifiers**, their types, and the uses of access modifiers.





Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

Default Access Modifier

When no access modifier is specified for a class, method, or data member, it is said to be having the **default** access modifier by default. The default access modifiers are accessible ***only within the same package***.

```
// default access modifier
package p1;
// Class DefaultAM is having // Default access
modifier
class DefaultAM
{
void display()
{
System.out.println("Hello World!");
}
}
```

```
// error while using class from p1
// This class is having different package with default
modifier
package p2;
import p1.*;
class Demo
{
public static void main(String args[])
{ // Accessing class from package p1
DefaultAM  d = new DefaultAM ();
d.display();
}
}
```

Private Access Modifier

The **private access modifier** is specified using the keyword **private**. The methods or data members declared as private are accessible ***only within the class in which they are declared***.

```
/ error while using class from different package with private access modifier
class A
{
private void display()
{
System.out.println(“Java");
}
}
class B
{
public static void main(String args[])
{
A obj = new A(); // Trying to access private method of another class
obj.display(); } }
```

Protected Access Modifier

The **protected access modifier** is specified using the keyword **protected**. The methods or data members declared as protected are ***accessible within the same package or subclasses in different packages***.

```
// protected access modifier
package p1;
public class A
{
protected void display()
{
System.out.println(“Java");
}
}
```

```
// protected modifier
package p2;
// importing all classes in package p1
import p1.*;
// Class B is subclass of A
class B extends A
{
public static void main(String args[])
{
B obj = new B();
obj.display();
}
}
```

Public Access Modifier

The **public access modifier** is specified using the keyword **public**.

•The public access modifier has the **widest scope** among all other access modifiers.

•Classes, methods, or data members that are declared as public are ***accessible from everywhere*** in the program. There is no restriction on the scope of public data members.

```
// public modifier
package p1;
public class A
{
public void display()
{ System.out.println(“Java");
}
}
```

```
//public access modifier
package p2;
import p1.*;
class B
{
public static void main(String args[])
{
A obj = new A();
obj.display();
}
}
```

Understanding static

- When a member is declared **static, it can be accessed before any objects of its class are create**d, and without reference to any object
- The most common example of a **static member is main( )**
- main( ) **is declared as static because it must be called before any objects exist**
- Instance variables declared as static are, **essentially, global variables**

Static variables

- When a variable is declared as static, then a **single copy of the variable is created and shared among all objects** at the class level.
- Static variables are, essentially, **global variables**.
- **All instances of the class share the same static variable**.
- We can create static variables at class-level only
- static variables are executed in order they are present in a program.
- Static variable can call by directly with the help of class only, we do not need to create object for the class

Methods declared as static have several restrictions:

- They can only **call other static methods**
- They must **only access static data**
- They cannot **refer to this or super in any way**
- **We can declare a static block which gets executed exactly once, when the class is first loaded**

Syntax to declare a static method:

```
 Access_modifier static void methodName()
{
     // Method body.
}
```

Syntax to call a static method:

```
className.methodName();
```

```
class StaticExample
{
static int num = 10;
static String str = “Java Programming";
static void display()
{
System.out.println("static number is " + num);
System.out.println("static string is " + str);
}
void nonstatic()
{
display(); // our static method can accessed in non static method
}
public static void main(String args[])
{
StaticExample obj = new StaticExample();
obj.nonstatic();
display(); // static method can called // directly without an object
} }
```

Output

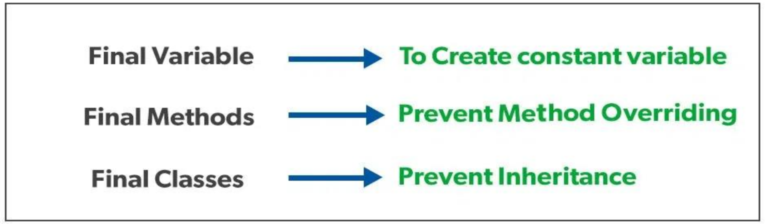
```
static number is 10
static string is Java Programming
static number is 10
static string is Java Programming
```

Introducing final

- The **final Keyword in Java** is used as a **non-access modifier** appli
- It is used to restrict a user in Java.

Final Variable

- A variable can be declared as final
- Doing so prevents its **contents from being modified**
- We **must initialize a final variable when it is declared**
- final int FILE_NEW = 1;
- final int FILE_OPEN = 2;
- Variables declared **as final do not occupy memory on a per-instance basis**
- The keyword final can also **be applied to methods, but its meaning is substantially different t**han when it is applied to variables



Final Variable →	To Create constant variable
Final Methods →	Prevent Method Overriding
Final Classes →	Prevent Inheritance

```
class ConstantExample
{
public static void main(String[] args)
{ // Define a constant variable PI
final double PI = 3.14159;
System.out.println("Value of PI: " +
PI);
}
}
```

Final classes

- When a class is declared with the *final* keyword in Java, it is called a **final class**.
- A final class cannot be **extended(inherited)**.

```
final class A
{
// methods and fields
}
// The following class is illegal
class B extends A
{
// COMPILE-ERROR! Can't subclass A
}
```

Final Method

When a method is declared with *final* keyword, it is called a **final method** in Java. A final method cannot be overridden.

```
class A
{
final void m1()
{
System.out.println("This is a final method.");
}
}
class B extends A
{
void m1()
{
// Compile-error! We can not override
System.out.println("Illegal!");
}
}
```

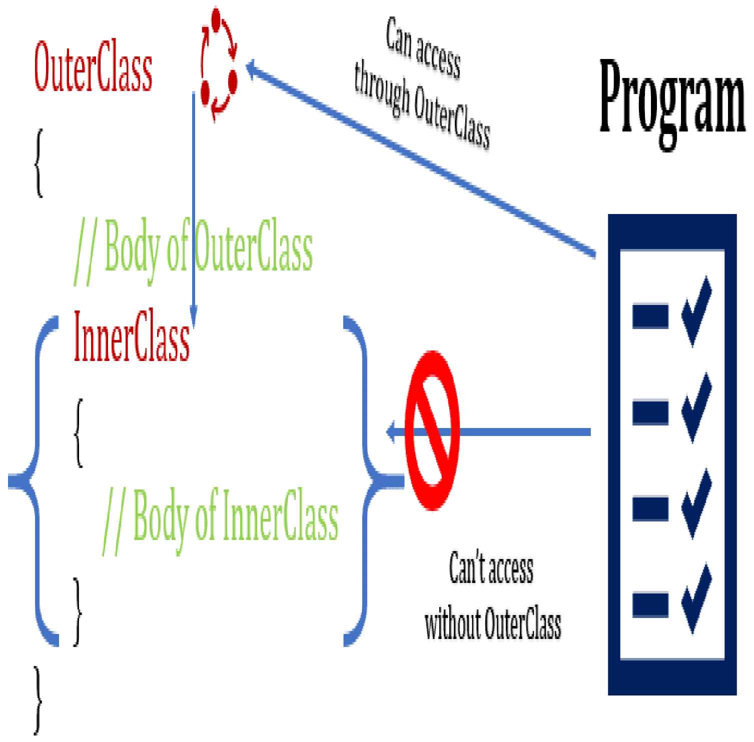
Introducing Nested and Inner Classes

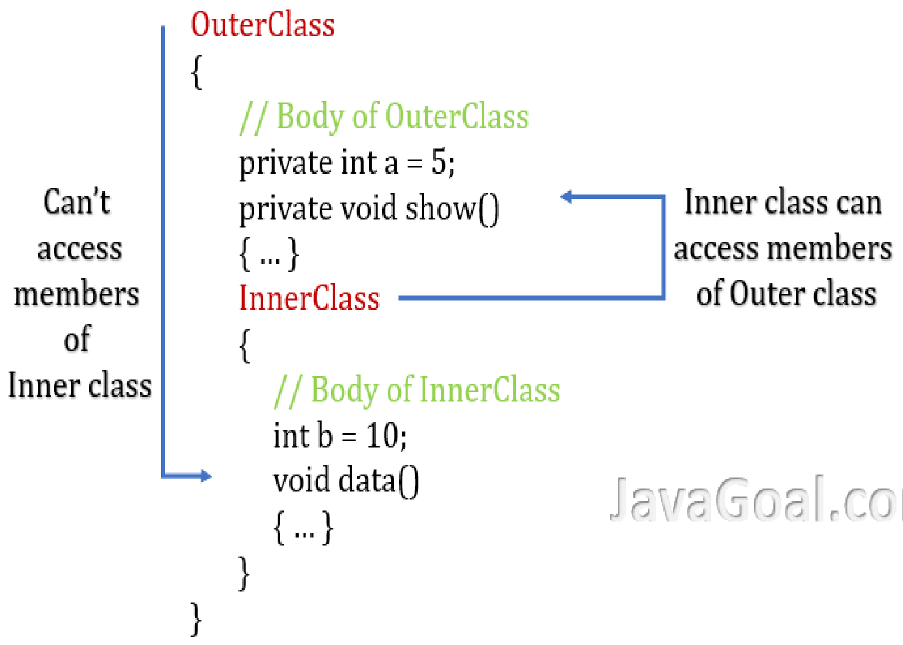
- It is possible to define **a class within another class**
- The **scope of a nested class is bounded by the scope of its enclosing class**
- If **class B is defined within class A, then B is known to A, but not outside of A**
- A nested class **has access to the members, including private members, of the class in which it is nested**

- However, **the enclosing class does not have access to the members of the nested class**
- There are two types of **nested classes: static and non-static**
- A **static nested class** is one which has the **static modifier applied**
- static innerclass must access its enclosing **class by creating an object.**

- The most important type of **nested class is the inner class**
- An inner class is **a non-static nested class**
- It has access to **all of the variables and methods of its outer class**

- It is important to **realize that class Inner is known only within the scope of class Outer**
- The Java compiler **generates an error message if any code outside of class Outer attempts to instantiate class Inner**





```
OuterClass
{
    // Body of OuterClass
    private int a = 5;
    private void show()
    { … }
    InnerClass
    {
        // Body of InnerClass
        int b = 10;
        void data()
        { … }
    }
}
```

Can't access members of Inner class
Inner class can access members of Outer class

```
class TestMemberOuter1{

 private int data=30;

 class Inner{

  void msg(){System.out.println("data is "+data);}
/)msg() complete

 }  // Inner class Complete
```

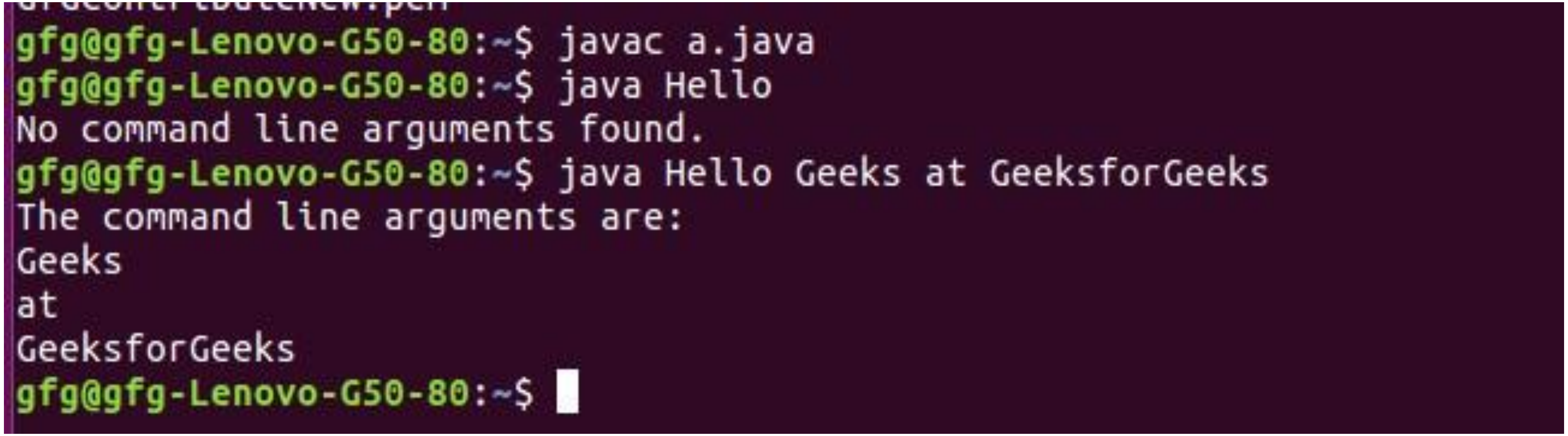
```
 public static void main(String args[]){

  TestMemberOuter1 obj=new
TestMemberOuter1();

  TestMemberOuter1.Inner in=obj.new Inner();

  in.msg();

 }

}
```

Command Line Argument

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt
- When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[]. It can be confirmed that they are actually wrapped up in an args array by checking the length of args using args.length.
- Internally, JVM wraps up these command-line arguments into the args[ ] array that we pass into the main() function. We can check these arguments using args.length method. JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.

Steps to Run COmmand Line Argument

- Save the program as Hello.java
- Open the command prompt window and compile the program- javac Hello.java
- After a successful compilation of the program, run the following command by writing the arguments- java Hello



```
gfg@gfg-Lenovo-G50-80:~$ javac a.java
gfg@gfg-Lenovo-G50-80:~$ java Hello
No command line arguments found.
gfg@gfg-Lenovo-G50-80:~$ java Hello Geeks at GeeksforGeeks
The command line arguments are:
Geeks
at
GeeksforGeeks
gfg@gfg-Lenovo-G50-80:~$ 
```

Var args-Variable length Arguments

A method with variable length arguments(Varargs) in Java can have zero or multiple arguments.

Variable length arguments are most useful when the number of arguments to be passed to the method is not known beforehand.

They also reduce the code as overloaded methods are not required.

```
public class Demo {

  public static void Varargs(String... str) {

    System.out.println("\nNumber of arguments are: " + str.length);

    System.out.println("The argument values are: ");

    for (String s : str)

      System.out.println(s);

  }
```

```
public static void main(String args[]) {
      Varargs("Apple", "Mango", "Pear");
      Varargs();
      Varargs("Magic");
   }
```

Any questions

Thank you