Object and classes, new keyword, stack vs heap,Scanner, constructor, method overloading, constructor overloading, usage this, static,

**Chapter 2 – Unit 1**

**Classes and Objects: Concepts & methods**

* An object is an instance of a class. A class is a template from which objects are created. So, an object is the instance (result) of a class.
* **Object Definitions:**
* An object is *a real-world entity*.
* An object is *a runtime entity*.
* The object is *an entity which has state and behavior*.
* The object is *an instance of a class*.
* **Class Definition :**
* A class is a group of objects which have common properties. It is a template from which objects are created. It is a logical entity. It can't be physical.
* A class in Java can contain:
* Fields
* Methods
* Constructors
* Blocks
* Nested class and interface
* **Syntax to declare a class:**

class <class\_name>

{

fields;

methods;

}

* **Syntax for Creating an Object:**

className objectname = new className();

* **Syntax for method definition in Java:**

AccessModifier returnType nameOfMethod (Parameter List)

{

// method body

}

**new Keyword**

* The Java new keyword is used to create an instance of the class. In other words, it instantiates a class by allocating memory for a new object and returning a reference to that memory. We can also use the new keyword to create the array object.
* **Points to remember**
* It is used to create the object.
* It allocates the memory at runtime.
* All objects occupy memory in the heap area.
* It invokes the object constructor.
* **Object and Class Example: main within the class:**

class Student

{

int id=45; //field or data member or instance variable

String name="siva"; //field or data member or instance variable

public static void main(String args[])

{

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

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

System.out.println(s1.name);

}

}

**Output**

45  
siva

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

class Student

{

int id=45;

String name="siva";

}

class TestStudent1

{

public static void main(String args[])

{

Student s1=new Student();

System.out.println(s1.id);

System.out.println(s1.name);

}

}

**Output**

45  
siva

* **Ways to initialize object:**
* There are 3 ways to initialize object in Java.
* By reference variable
* By method
* By constructor
* **1) Object and Class Example: Initialization through reference**
* Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

class Student

{

int id;

String name;

}

class TestStudent2

{

public static void main(String args[])

{

Student s1=new Student();

s1.id=45;

s1.name="Siva";

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

}

}

**Output**

45  
siva

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

class Student

{

int id;

String name;

}

class TestStudent3

{

public static void main(String args[])

{

//Creating object

Student s1=new Student();

//Initializing object

s1.id=45;

s1.name="Siva Kumar";

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

//Creating object

Student s2=new Student();

//Initializing object

s2.id=55;

s2.name="Ravi Kumar";

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

}

}

**Output**

45 Siva Kumar  
55 Ravi Kumar

**Example: Initialization through method**

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the read method. Here, we are displaying the state (data) of the objects by invoking the display() method.

class Student

{

int rollno;

String name;

void read(int r, String n)

{

rollno=r;

name=n;

}

void display()

{

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

}

}

class TestStudent4

{

public static void main(String args[])

{

Student s1=new Student();

s1.read(45,"Siva Kumar");

s1.display();

Student s2=new Student();

s2.read(55,"Ravi Kumar");

s2.display();

}

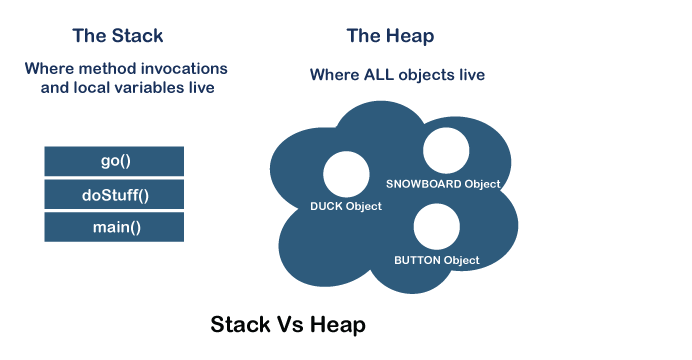
}

**Output**

45 Siva Kumar  
55 Ravi Kumar

**Stack Vs Heap Java**

* In Java, [memory management](https://www.javatpoint.com/memory-management-in-java) is a vital process.
* It is managed by Java automatically.
* **The JVM divides the memory into two parts: [stack](https://www.javatpoint.com/java-stack) memory and [heap](https://www.javatpoint.com/java-heap) memory.**
* From the perspective of Java, both are important memory areas but both are used for different purposes.
* **The major difference between Stack memory and heap memory is that the stack is used to store the order of method execution and local variables while the heap memory stores the objects and it uses dynamic memory allocation and deallocation.**
* In this section, we will discuss the differences between stack and heap in detail.



* The stack memory is a physical space (in RAM) allocated to each thread at run time. It is created when a thread creates. Memory management in the stack follows LIFO (Last-In-First-Out) order because it is accessible globally. It stores the variables, references to objects, and partial results. Memory allocated to stack lives until the function returns. If there is no space for creating the new objects, it throws the java.lang.StackOverFlowError. The scope of the elements is limited to their threads. The [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates a separate stack for each thread.
* It is created when the JVM starts up and used by the application as long as the application runs. It stores objects and JRE classes. Whenever we create objects it occupies space in the heap memory while the reference of that object creates in the stack. It does not follow any order like the stack. It dynamically handles the memory blocks. It means, we need not to handle the memory manually. For managing the memory automatically, [Java](https://www.javatpoint.com/java-tutorial) provides the [garbage collector](https://www.javatpoint.com/Garbage-Collection) that deletes the objects which are no longer being used. Memory allocated to heap lives until any one event, either program terminated or memory free does not occur. The elements are globally accessible in the application. It is a common memory space shared with all the threads. If the heap space is full, it throws the java.lang.OutOfMemoryError.

**Java User Input (Scanner)**

* **The Scanner class is used to get user input, and it is found in the java.util package.**
* To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.
* **Input Types:**
* In the example above, we used the nextLine() method, which is used to read Strings. To read other types, look at the table below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| nextBoolean() | Reads a boolean value from the user |
| nextByte() | Reads a byte value from the user |
| nextDouble() | Reads a double value from the user |
| nextFloat() | Reads a float value from the user |
| nextInt() | Reads a int value from the user |
| nextLine() | Reads a String value from the user |
| nextLong() | Reads a long value from the user |
| nextShort() | Reads a short value from the user |

**Example :**

import java.util.Scanner;

class sample

{

public static void main(String[] args)

{

Scanner s = new Scanner(System.in);

System.out.println("Enter name:");

String name = s.nextLine();

System.out.println("Enter age:");

int age = s.nextInt();

System.out.println("Enter salary:");

double salary = s.nextDouble();

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

}

}

**Output**

Enter name, age and salary:

Siva Kumar 35 75000  
Siva Kumar

35

75000

**Constructors**

* **A constructor is a block of codes similar to the method.**
* **It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created.**
* **At the time of calling constructor, memory for the object is allocated in the memory.**
* **It is a special type of method which is used to initialize the object.**
* Every time an object is created using the new() keyword, at least one constructor is called.
* It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
* **Rules for creating Java constructor:**
* 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
* **Types of Java constructors:**
* There are two types of constructors in Java:
* Default constructor (no-arg constructor)
* Parameterized constructor
* **Default Constructor:** A constructor is called "Default Constructor" when it doesn't have any parameter.
* **Syntax of default constructor:**

<class\_name>()

{

}

* **/\* 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 boxdemo

{

public static void main(String args[])

{

double vol;

box b1 = new box();

vol = b1.volume();

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

box b2 = new box();

vol = b2.volume();

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

}

}

**Output**

**When this program is run, it generates the following results:**

Constructing Box

Volume is 1000.0

Constructing Box

Volume is 1000.0

* **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 boxdemo

{

public static void main(String args[])

{

double vol;

box b1 = new box(10, 20, 15);

vol = b1.volume();

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

box b2 = new box(3, 6, 9);

vol = b2.volume();

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

}

}

**Output**

The output from this program is shown here:

Volume is 3000.0

Volume is 162.0

**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 this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
* **(OR)** **If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as Method Overloading.**
* **Method overloading is one of the ways that Java supports polymorphism.**
* **Different ways to overload the method:** There are two ways to overload the method in java.
* By changing number of arguments
* By changing the data type

**Example:**

// Demonstrate method overloading.

class Overload

{

void test()

{

System.out.println("No parameters");

}

void test(int a)

{

System.out.println("One Integer Parameter:”);

System.out.println("a: " + a);

}

void test(int a, int b)

{

System.out.println("Two Integer Parameters:”);

System.out.println("a and b: " + a + " " + b);

}

void test(double a)

{

System.out.println("One Double Parameter:”);

System.out.println("double a: " + a);

}

}

class Overloaddemo

{

public static void main(String args[])

{

Overload ob = new Overload();

// call all versions of test()

ob.test();

ob.test(10);

ob.test(10, 20);

ob.test(123.35);

}

}

**This program generates the following output:**

No parameters

One integer parameter:

a: 10

Two integer parameters:

a: 10 a and b: 10 20

One double parameter:

double a: 123.25

**Constructor overloading**

* In Java, we can overload constructors like methods.
* In constructor over loading, we create multiple constructors with the same name but with different parameters types or with different no of parameters.

**Example:**

/\* Here, Box defines three constructors to initialize the dimensions of a box various ways. \*/

class Box

{

double width;

double height;

double depth;

Box()

{

width = -1;

height = -1;

depth = -1;

}

Box(double len)

{

width = height = depth = len;

}

Box(double w, double h, double d)

{

width = w;

height = h;

depth = d;

}

// compute and return volume

double volume()

{

return width \* height \* depth;

}

}

class Overloaddemo

{

public static void main(String args[])

{

// create boxes using the various constructors

double vol;

Box b1 = new Box();

vol = b1.volume();

System.out.println("Volume of b1 is " + vol);

Box b2 = new Box(7);

box vol = b2.volume();

System.out.println("Volume of b2 is " + vol);

Box b3 = new Box(10, 20, 15);

vol = b3.volume();

System.out.println("Volume of b3 is " + vol);

}

}

**The output produced by this program is shown here:**

Volume of b1 is 3000.0

Volume of b2 is -1.0

Volume of b3 is 343.

**this keyword**

The usage of this keyword is:

* [this can be used to refer current class instance variable.](https://www.javatpoint.com/this1)
* [this can be used to invoke current class method (implicitly)](https://www.javatpoint.com/this2).
* [this() can be used to invoke current class constructor.](https://www.javatpoint.com/this3)
* [this can be passed as an argument in the method call.](https://www.javatpoint.com/this4)
* [this can be passed as argument in the constructor call.](https://www.javatpoint.com/this5)
* [this can be used to return the current class instance from the method.](https://www.javatpoint.com/this6)

### **this: to refer current class instance variable**

* The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
* Example : Understanding the problem without this keyword

class Student

{

int a,b,c;

Student(int a,int b,int c)

{

a=a;

b=b;

c=c;

}

void display()

{

System.out.println(a);

System.out.println(b);

System.out.println(c);

}

}

class TestThis1

{

public static void main(String args[])

{

Student s1=new Student(11,22,33);

s1.display();

}

}

**The output produced by this program is shown here:**

0  
0  
0

**Note :** 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 a,b,c;

Student(int a,int b,int c)

{

this.a=a;

this.b=b;

this.c=c;

}

void display()

{

System.out.println(a);

System.out.println(b);

System.out.println(c);

}

}

class TestThis1

{

public static void main(String args[])

{

Student s1=new Student(11,22,33);

s1.display();

}

}

**The output produced by this program is shown here:**

11  
22  
33

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

class Student

{

int a1,b1,c1;

Student(int a2,int b2,int c2)

{

a1=a2;

b1=b2;

c1=c2;

}

void display()

{

System.out.println(a1);

System.out.println(b1);

System.out.println(c1);

}

}

class TestThis1

{

public static void main(String args[])

{

Student s1=new Student(11,22,33);

s1.display();

}

}

**The output produced by this program is shown here:**

11  
22  
33

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

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example

class A

{

void m()

{

System.out.println("hello m");

}

void n()

{

System.out.println("hello n");

//m(); (OR) this.m()

this.m();

}

}

class TestThis4

{

public static void main(String args[])

{

A a=new A();

a.n();

}

}

**The output produced by this program is shown here:**

hello n  
hello m

3) this() : to invoke current class constructor

* The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Example1 :** 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);

}

}

**The output produced by this program is shown here:**

hello a

10

**Example 2 :** 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();

}

}

**The output produced by this program is shown here:**

5  
hello a

**Usage of static**

* The static is a non-access modifier in Java which is applicable for the following:
* Blocks
* Variables
* Methods
* nested classes
* To create a static member(block,variable,method,nested class), precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

## 1) static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

### Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

2) static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

**Note : Why is the Java main method static?**

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

3) Java static block

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

// Demonstrate static variables, methods, and blocks.

class UseStatic

{

static int a = 3;

static int b;

static void meth(int x)

{

System.out.println("x = " + x);

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

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

}

static

{

System.out.println("Static block initialized.");

b = a \* 4;

}

public static void main(String args[])

{

meth(42);

}

}

**Here is the output of the program:**

Static block initialized.

x = 42

a = 3

b = 12

**Note :** Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. **For example, if you wish to call a static method from outside its class, you can do so using the following general form: classname.method( )** Here, classname is the name of the class in which the static method is declared.

class UseStatic

{

static int a = 3;

static int b;

static void meth(int x)

{

System.out.println("x = " + x);

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

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

}

static

{

System.out.println("Static block initialized.");

b = a \* 4;

}

}

class staticdemo

{

public static void main(String args[])

{

UseStatic.meth(42);

}

}

**Here is the output of the program:**

Static block initialized.

x = 42

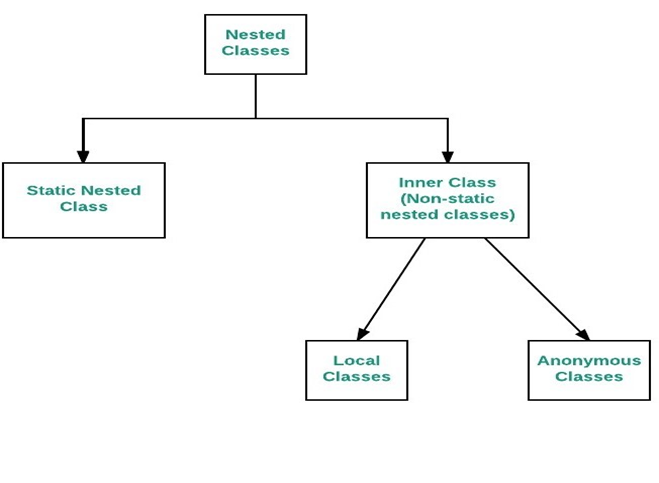
a = 3

b = 12

**Nested classes (Inner classes)**

In Java, it is possible to define a class within another class, such classes are known as nested classes. They enable you to logically group classes that are only used in one place, thus this increases the use of [encapsulation](https://www.geeksforgeeks.org/encapsulation-in-java/), and creates more readable and maintainable code.

* The scope of a nested class is bounded by the scope of its enclosing class. Thus in above example, class NestedClass does not exist independently of class OuterClass.
* A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.
* A nested class is also a member of its enclosing class.
* As a member of its enclosing class, a nested class can be declared private, public, protected, or package private(default).
* **Nested classes are divided into two categories:**
* static nested class : Nested classes that are declared static are called static nested classes.
* inner class : An inner class is a non-static nested class.

****

**Syntax:**

class OuterClass

{

...

class NestedClass

{

...

}

}

**Static nested classes**

* In the case of normal or regular inner classes, without an outer class object existing, there cannot be an inner class object. i.e., an object of the inner class is always strongly associated with an outer class object. But in the case of static nested class, Without an outer class object existing, there may be a static nested class object. i.e., an object of a static nested class is not strongly associated with the outer class object.  
  As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.
* **Syntax for accessing the nested static class methods;**

outerclassname.staticnestedclassmethods();

* **Syntax to create an object for the static nested class:**

outerclassname.staticnestedclassname staticnestedclassobjectname =

new outerclassname.staticnestedclassname();

// Java program to demonstrate accessing a static nested class

// outer class

class OuterClass

{

// static member

static int outer\_x = 10;

// instance(non-static) member

int outer\_y = 20;

// private member

private static int outer\_private = 30;

// static nested class

static class StaticNestedClass

{

void display()

{

// can access static member of outer class

System.out.println("outer\_x = " + outer\_x);

// can access display private static member of outer class

System.out.println("outer\_private = " + outer\_private);

// The following statement will give compilation error as static nested class cannot directly access non-static members.

// System.out.println("outer\_y = " + outer\_y);

}

}

}

// Driver class

public class StaticNestedClassDemo

{

public static void main(String[] args)

{

// accessing a static nested class

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

nestedObject.display();

}

}

**Output:**

outer\_x = 10

outer\_private = 30

**Inner classes**

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

outerclassname.innerclassname innerclassobject = outerclassobject.new innerclassname();

* There are two special kinds of inner classes :
* [Local inner classes](https://www.geeksforgeeks.org/local-inner-class-java/)
* [Anonymous inner classes](https://www.geeksforgeeks.org/anonymous-inner-class-java/)

// Java program to demonstrate accessing an inner class

// outer class

class OuterClass

{

// static member

static int outer\_x = 10;

// instance(non-static) member

int outer\_y = 20;

// private member

private int outer\_private = 30;

// inner class

class InnerClass

{

void display()

{

// can access static member of outer class

System.out.println("outer\_x = " + outer\_x);

// can also access non-static member of outer class

System.out.println("outer\_y = " + outer\_y);

// can also access a private member of the outer class

System.out.println("outer\_private = " + outer\_private);

}

}

}

// Driver class

public class InnerClassDemo

{

public static void main(String[] args)

{

// accessing an inner class

outerclass outerobject = new outerclass();

outerclassname.innerclassname innerclassobject = outerobject.new innerclass();

innerObject.display();

}

}

**Output:**

outer\_x = 10

outer\_y = 20

outer\_private = 30

**Type Casting or Type Conversion**

* **Converting one primitive datatype into another is known as type casting (type conversion) in Java.**
* You can cast the primitive datatypes in two ways namely, Widening and, Narrowing.
* **Widening Casting (automatically)** - converting a smaller type to a larger type size.
* Widening casting is done automatically when passing a smaller size type to a larger size type:

byte -> short -> char -> int -> long -> float -> double

* **Example :**

class sample

{

public static void main(String[] args)

{

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt);

System.out.println(myDouble);

}

}

**Output:**

9  
9.0

* **Narrowing Casting** (manually) - converting a larger type to a smaller size type.
* Narrowing casting must be done manually by placing the type in parentheses in front of the value:

double -> float -> long -> int -> char -> short -> byte

* **Example :**

class sample

{

public static void main(String[] args)

{

double myDouble = 9.78d;

int myInt = (int) myDouble; // Explicit casting: double to int

System.out.println(myDouble);

System.out.println(myInt);

}

}

**Output:**

9.78  
9

**Parameter passing mechanisms**

**Java programming languages support two parameter passing techniques namely: *pass-by-value* and *pass-by-reference*.**

1. **Pass By Value:** 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**.

**Example :** In case of call by value original value is not changed. Let's take a simple example:

class Operation

{

int data=50;

void change(int data)

{

data=data+100;//changes will be in the local variable only

}

public static void main(String args[])

{

Operation op=new Operation();

System.out.println("before change "+op.data);

op.change(50);

System.out.println("after change "+op.data);

}

}

**Output:**

Before change : 50

After change :50

1. **Call by reference(aliasing):** 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.

**Example**: In case of call by reference original value is changed if we made changes in the called method. **If we pass object in place of any primitive value**, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

class Operation

{

int data=50;

void change(Operation op)

{

op.data=op.data+100;//changes will be in the instance variable

}

public static void main(String args[])

{

Operation op=new Operation();

System.out.println("before change "+op.data);

op.change(op);//passing object

System.out.println("after change "+op.data);

}

}

**Output:**

Before change : 50

After change :150

**Note :** when we pass a reference, a new reference variable to the same object is created. So we can only change members of the object whose reference is passed. We cannot change the reference to refer to some other object as the received reference is a copy of the original reference.