NRI INSTITUTE OF INFORMATION SCIENCE & TECHNOLOGY BHOPAL

DEPARTMENT OF MASTER OF COMPUTER APPLICATION

LAB MANUAL

JAVA AND OOPS LAB . (MCA-206)

MASTER OF COMPUTER APPLICATION (MCA)

EXPERIMENT NO.-1

Print “Hello World”

Objective:

To learn the fundamental of JAVA programming:s

Underlying Concept/ procedure: Java is a programming language originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities than either C or C++. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java is as of 2012 one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users.

Platform-Independent

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is more closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM.

Simple

There are various features that makes the java as a simple language. Programs are easy to write and debug because java does not use the pointers explicitly. It is much harder to write the java programs that can crash the system but we can not say about the other programming languages. Java provides the bug free system due to the strong memory management. It also has the automatic memory allocation and deallocation system.

Object-Oriented

To be an Object Oriented language, any language must follow at least the four characteristics.

- Inheritance : It is the process of creating the new classes and using the behavior of the existing classes by extending them just to reuse the existing code and adding the additional features as needed.
- Encapsulation: : It is the mechanism of combining the information and providing the abstraction.

- Polymorphism: : As the name suggest one name multiple form, Polymorphism is the way of providing the different functionality by the functions having the same name based on the signatures of the methods.
- Dynamic binding : Sometimes we don't have the knowledge of objects about their specific types while writing our code. It is the way of providing the maximum functionality to a program about the specific type at runtime.

As the languages like Objective C, C++ fulfills the above four characteristics yet they are not fully object oriented languages because they are structured as well as object oriented languages. But in case of java, it is a fully Object Oriented language because object is at the outer most level of data structure in java. No stand alone methods, constants, and variables are there in java. Everything in java is object even the primitive data types can also be converted into object by using the wrapper class.

Robust

Java has the strong memory allocation and automatic garbage collection mechanism. It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features makes the java language robust.

Distributed

The widely used protocols like HTTP and FTP are developed in java. Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing codes on their local system.

Portable

The feature Write-once-run-anywhere makes the java language portable provided that the system must have interpreter for the JVM. Java also have the standard data size irrespective of operating system or the processor. These features makes the java as a portable language.

Dynamic

While executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

Secure

Java does not use memory pointers explicitly. All the programs in java are run under an area known as the sand box. Security manager determines the accessibility options of a class like reading and writing a file to the local disk. Java uses the public key encryption system to allow the java

applications to transmit over the internet in the secure encrypted form. The bytecode Verifier checks the classes after loading.

Performance

Java uses native code usage, and lightweight process called threads. In the beginning interpretation of bytecode resulted the performance slow but the advance version of JVM uses the adaptive and just in time compilation technique that improves the performance.

Multithreaded

As we all know several features of Java like Secure, Robust, Portable, dynamic etc; you will be more delighted to know another feature of Java which is **Multithreaded.**
Java is also a Multithreaded programming language. Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer.
Multithreading programming is a very interesting concept in Java. In multithreaded programs not even a single thread disturbs the execution of other thread. Threads are obtained from the pool of available ready to run threads and they run on the system CPUs. This is how Multithreading works in Java which you will soon come to know in details in later chapters.

Interpreted

We all know that Java is an interpreted language as well. With an interpreted language such as Java, programs run directly from the source code.
The interpreter program reads the source code and translates it on the fly into computations. Thus, Java as an interpreted language depends on an interpreter program.
The versatility of being **platform independent** makes Java to outshine from other languages. The source code to be written and distributed is platform independent.
Another advantage of Java as an interpreted language is its error debugging quality. Due to this any error occurring in the program gets traced. This is how it is different to work with Java.

Architecture-Neutral

The term architectural neutral seems to be weird, but yes Java is an architectural neutral language as well. The growing popularity of networks makes developers think distributed. In the world of network it is essential that the applications must be able to migrate easily to different computer systems. Not only to computer systems but to a wide variety of hardware architecture and Operating system architectures as well. The Java compiler does this by generating byte code instructions, to be easily interpreted on any machine and to be easily translated into native machine code on the fly. The compiler generates an architecture-neutral object file format to enable a Java application to execute anywhere on the network and then the compiled code is executed on many processors, given the

presence of the Java runtime system. Hence Java was designed to support applications on network. This feature of Java has thrived the programming language.

Java is an Object Oriented Language. As a language that has the Object Oriented feature Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter we will look into the concepts Classes and Objects.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.

Java Basic Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

1. Primitive Data Types
2. Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is a 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.

- Example : byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example : short s= 10000 , short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example : int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example : int a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example : float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values. generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example : double d1 = 123.4

boolean:

- boolean data type represents one bit of information.

- There are only two possible values : true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example : boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example . char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example : Animal animal = new Animal("giraffe");

Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicates octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
```

```
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Code:

```
class hello

{

public static void main(String args[ ])

{

System.out.println("Hello World");

}

}
```

Output: Hello World

Conclusion:

EXPERIMENT NO.-2

Java program with multiple statements (square root, multiplication, division etc.)

Objective:

To learn the fundamental of math functions in JAVA programming:

Underlying Concept/ procedure: Class Math

```
java.lang.Object
  └ java.lang.Math
```

---

public final class **Math**

extends Object

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of `Math` methods. Such higher-performance implementations still must conform to the specification for `Math`.

sqrt

```
public static double sqrt(double a)
```

Returns the correctly rounded positive square root of a `double` value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.

- If the argument is positive zero or negative zero, then the result is the same as the argument.

Otherwise, the result is the `double` value closest to the true mathematical square root of the argument value.

Parameters:

`a` - a value.

Returns:

the positive square root of `a`. If the argument is NaN or less than zero, the result is NaN.

Code:

```
import java.lang.Math;

class squareroot

{

public static void main(String args[ ])

{

double x=5;

double y;

y = Math.sqrt(x);

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

}

}
```

Output: y = 2.23607

Conclusion:

EXPERIMENT NO.-3

Typecasting

Objective:

To learn the fundamental of typecasting in JAVA programming:

Underlying Concept/ procedure: **Type Casting** refers to changing an entity of one datatype into another. This is important for the type conversion in developing any application. If you will store a int value into a byte variable directly, this will be illegal operation. For storing your calculated int value in a byte variable you will have to change the type of resultant data which has to be stored.

Code:

```
class typewrap

{

public static void main(string args[])

{

char c = 'x';

byte b = 50;

short s=1996;

int i=123456789;

long l =1234567654321L;

float f1 =3.142F;

float f2=1.2e-5F;



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

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

```
System.out.println("s="+s);

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

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

System.out.println("f1="+f1);

System.out.println("f2="+f2);

System.out.println("d2="+d2);


System.out.println(" ");

System.out.println("types converted");


short s1=(short)b;

short s2=(short)i;

float n2=(float)i;

int m1=(int)f1;


System.out.println("(short)b="+s1);

System.out.println("(short)i="+n2);

System.out.println("(float)l="+n2);

System.out.println("(int)f1="+m1);

}

}
```

Output:

Conclusion:

EXPERIMENT NO.-4

Loops concept

Objective:

To learn the fundamental of loops in JAVA programming:

Underlying Concept/ procedure:

There may be a sitution when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)

{

  //Statements

}
```

When executing, if the boolean_expression result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

  public static void main(String args[]){

    int x= 10;



    while( x < 20 ){

      System.out.print("value of x : " + x );

      x++;

      System.out.print("\n");

    }

  }

}
```

This would produce following result:

```
value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19
```

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do

{

  //Statements

}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test {

  public static void main(String args[]){

    int x= 10;



    do{

      System.out.print("value of x : " + x );

      x++;

      System.out.print("\n");

    }while( x < 20 );

  }

}
```

This would produce following result:

```
value of x : 10

value of x : 11
```

```
value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19
```

The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)

{

  //Statements

}
```

Here is the flow of control in a for loop:

The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.

After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.

The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step,then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {

   public static void main(String args[]){



      for(int x = 10; x < 20; x = x+1){

         System.out.print("value of x : " + x );

         System.out.print("\n");

      }

   }

}
```

This would produce following result:

```
value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19
```

Enhanced for loop in Java:

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)

{

  //Statements

}
```

Declaration . The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

Expression . This evaluate to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {

  public static void main(String args[]){

    int [] numbers = {10, 20, 30, 40, 50};



    for(int x : numbers ){

      System.out.print( x );

      System.out.print(",");

    }

    System.out.print("\n");

    String [] names ={"James", "Larry", "Tom", "Lacy"};

    for( String name : names ) {

      System.out.print( name );

      System.out.print(",");

    }
```

```
  }

}
```

This would produce following result:

```
10,20,30,40,50,

James,Larry,Tom,Lacy,
```

The break Keyword:

The break keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Example:

```
public class Test {

   public static void main(String args[]){

      int [] numbers = {10, 20, 30, 40, 50};



      for(int x : numbers ){

         if( x == 30 ){

               break;

         }

         System.out.print( x );

         System.out.print("\n");

      }
```

```
  }

}
```

This would produce following result:

```
10

20
```

The continue Keyword:

The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.

In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Example:

```
public class Test {

  public static void main(String args[]){

    int [] numbers = {10, 20, 30, 40, 50};



    for(int x : numbers ){

      if( x == 30 ){

            continue;

      }

      System.out.print( x );

      System.out.print("\n");
```

```
    }
  }
}
```

This would produce following result:

```
10
20
40
50
```

Code:

```
class factorial
{
public static void main(String args[])
{
int fact=1,i;
for(i=4;i>=1;i--)
{
fact=fact*i;
}
System.out.println("factorial of 4:"+fact);
}
}
```

Output:

Conclusion:

EXPERIMENT NO.-5

Multiple classes

Objective:

To learn the fundamental of multiple classes in JAVA programming:

Underlying Concept/ procedure:

You've now added extra methods to the basic program structure, but it is also possible to add extra classes too. In the basic program structure you had one class, which was declared public. Since a public class must always have the same name as the file in which it is contained, if your program is to include an extra public class, it should be in a separate file of the same name. Additional classes do not contain a main method.

One advantage of using additional classes is if you've written some methods for use in one program that you think may be useful in another. By writing these methods in a separate public class (in a separate file) any program may make use of them in exactly the same way as usual just by adding the name of the class containing the method, to the beginning of the method name, i.e.

ClassName.methodName(argument);

Code:

```
class room

{

float length;

float breadth;

void getdata(float a, float b)

{

length = a;

breadth = b;

}

}// This class save as room.java


class roomarea // This class save as roomarea.java and compile, run.
```

```
{
public static void main(String args[ ])
{
float area;
room room1 = new room();
room1.getdata(20,10);
area = room1.length*room1.breadth;
System.out.println(“Area=” + area);
}
}
```

Output:

Conclusion:

EXPERIMENT NO.-6

Objective: To learn the concept of for loop statement.

Problem statement: Implement a program using nested for loop to display the following pyramid.

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

Underlying concepts: for statement:

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

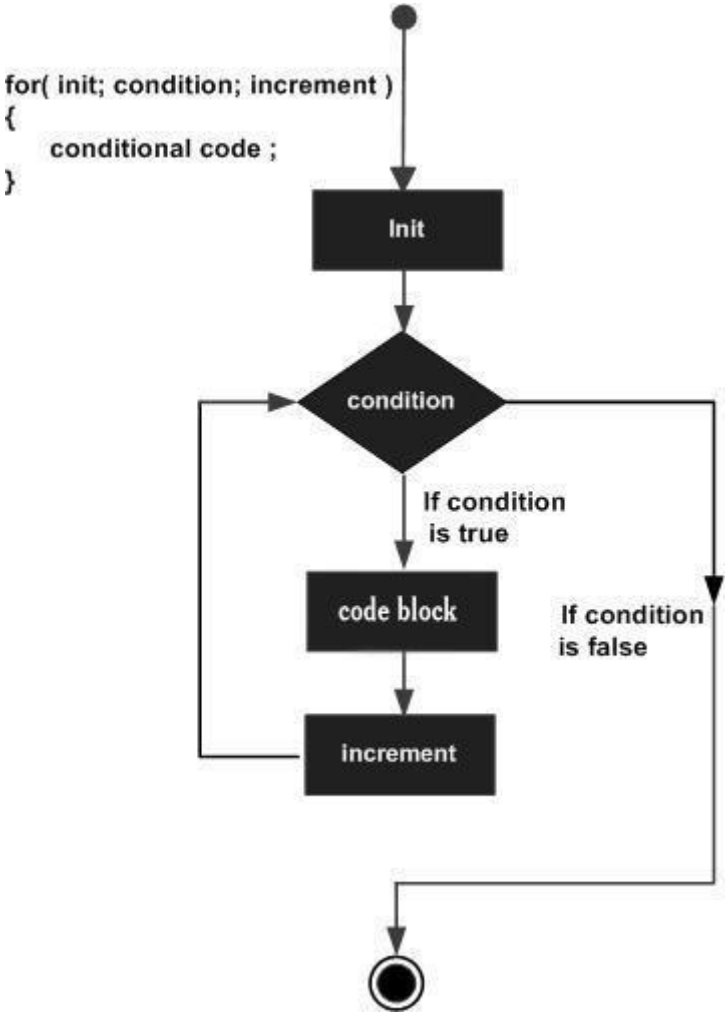
The syntax of a for loop in C++ is:

```
for ( init; condition; increment )
{
```

Here is the flow of control in a for loop:

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the**increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Algorithm

1. Input number of lines:n
2. i=1
3. if (i<=n) then
    a) display (40-i) spaces so that number displays in the middle of screen
    b) j=1
    c) if (j<=i)
        display j
        j=j+1
    d) goto (c)
4. i=i+1
5. goto (3)
6. stop

:

CODE :

```
#include<iostream.h>
void main()
{
        int n, i, j ;
        cout<<"Enter number of lines: ";
        cin>>n;

        for (i=1;i<=n;i++)
        {
                for(j=1;j<=40-I;j++)
                cout<<" ":
                for(j=1;j<=i;j++)
                        cout<< i<<" ";
                cout<<endl;
        }
}Output:
```

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

EXPERIMENT NO.-7

Objective: To learn the concept of call by reference and call by pointer.

Problem statement: Implement a program using call by reference and call by pointer to swap two variables.

Underlying concepts:

call by reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap(),** which exchanges the values of the two integer variables pointed to by its arguments.

call by pointer

The **call by pointer** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

Algorithm

1. Input a and b
2. t=a
3. a=b
4. b=t
5. display a and b

6. exit

Code:

```
#include <iostream.h>

// function declaration
void swapVal(int x, int y);
void swapRef(int &x, int &y);
void swapPtr(int *x, int *y);

void main ()
{
        // local variable declaration:
        int a=100, b=100;

        cout << "Before swap, value of a :" << a << endl;
        cout << "Before swap, value of b :" << b << endl;

          // calling a function to swap the values using call by value.
          swapVal(a, b);

          cout << "After swap (call by value) , value of a :" << a << endl;
          cout << "After swap (call by value), value of b :" << b << endl;

          swapRef(a, b);

          cout << "After swap (call by reference) , value of a :" << a << endl;
          cout << "After swap (call by reference), value of b :" << b << endl;
        swapPtr(&a, &b);

        cout << "After swap (call by pointer) , value of a :" << a << endl;
        cout << "After swap (call by pointer), value of b :" << b << endl;

  }
void swapVal(int x, int y)
{
        int t=x;
        x=y;
        y=t;
}
void swapRef(int &x, int &y)
{
        int t=x;
        x=y;
        y=t;
}
```

```
void swapPtr(int *x, int *y)
{
        Int* t;
        *t=*x;
        *x=*y;
        *y=*t;
}
```

Output:

```
Before swap, value of a: 100
Before swap, value of b: 200


After swap (call by value), value of a: 100
After swap (call by value), value of b: 200

After swap (call by reference), value of a: 200
After swap (call by reference), value of b: 100

After swap (call by pointer), value of a: 100
After swap (call by pointer), value of b: 200
```

EXPERIMENT NO.-8

Objective: To learn the concept of bubble sort.

Problem statement: Implement a program using bubble sort to sort an array in ascending order.

Underlying concepts:

Bubble Sort

Arranging a list in ascending or descending of in some other order is known as sorting. The simplest sorting algorithm is **bubble sort**. The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This process is repeated as many times as necessary, until the array is sorted

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:

(**5** **1** 4 2 8) → (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) → (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) → (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) → (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1** **4** 2 5 8) → (**1** **4** 2 5 8)
(1 **4** **2** 5 8) → (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)
Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

→

Third Pass:

(**1 2** 4 5 8) (**1 2** 4 5 8)
(1 **2 4** 5 8) (1 **2 4** 5 8)
(1 2 **4 5** 8) (1 2 **4 5** 8)
(1 2 4 **5 8**) (1 2 4 **5 8**)

The algorithm can be expressed as (0-based array):

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
      swapped = false
      for i = 1 to n-1 inclusive do
        /* if this pair is out of order */
        if A[i-1] > A[i] then
          /* swap them and remember something changed */
          swap( A[i-1], A[i] )
          swapped = true
        end if
      end for
    until not swapped
  end procedure
```

Code:

```
#include <iostream.h>

// function declaration
void bubbleSort(int *, int );

void main ()
{
        int a[10],I;
        cout<<"Enter 10 numbers:"<<endl;
        for(i=0; i<10; i++)
                cin>>a[i];


        bubbleSort(a,10);

        cout<<"List after sorting: "<<endl;

        for(i=0; i<10; i++)
                cout<<a[i]<<" ";
}
```

```
void bubbleSort(int *a, int n)
{
	int swapped=1;
	while (swapped==1)
	{
		for (i = 0 ; i< n-1 ;i++)
			if a[i] > a[i+1] // if this pair is out of order
			{// swap them and remember something changed
				t=a[i];
				a[i]=a[i+1];
				a[i+1]=t;
			}
			else

	}
}
```

Output:

```
Enter 10 numbers: 10 8 5 7 2 11 15 3 1 9
List after sorting: 1 2 3 5 7 8 9 10 11 15
```

EXPERIMENT NO.-9

Objective: To learn the concept of class.

Problem statement: Implement a program using class definition that defines a class ‘counter’ with a single member data ‘count’ and its constructor and destructor.

Underlying concepts: class and objects

An **object** has the same relationship to a **class** that a variable has to a data type. An object is said to be an instance of a class.

Defining the Class

Here’s the definition for a class:

```
class <class-name>//define a class
{
    private:
        <data-type1> <member-data-name1>;
        <data-type2> <member-data-name2>;
        <data-type3> <member-data-name3>;
    public:
        <return-type1> <member-function-name1>;
        <return-type2> <member-function-name2>;
}
```

The definition starts with the keyword **class**, followed by the class name, Like a structure, the body of the class is delimited by **braces** and terminated by a **semicolon**.

A key feature of object-oriented programming is **data hiding**. It means that data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class. The primary mechanism for hiding data is to put it in a class and make it **private**. Private data or functions can only be accessed from within the class. **Public** data or functions, on the other hand, are accessible from outside the class. Data hiding means hiding data from parts of the program

that don’t need to access it. More specifically, one class’s data is hidden from other classes. Data hiding is designed to protect well-intentioned programmers from honest mistakes. Programmers

who really want to can figure out a way to access private data, but they will find it hard to do so by accident.

The data items within a class are called **data members** (or sometimes member data). There can be any number of data members in a class, just as there can be any number of data items in a structure. **Member functions** are functions that are included within a class.

Usually the data within a class is private and the functions are public. This is a result of the way classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, there is no rule that says data must be private and functions public; in some circumstances you may find you'll need to use private functions and public data.

Code:

```
// object represents a counter variable
 #include<iostream.h>
class Counter
{
    private:
        unsigned int count;             //count
    public:
        Counter() : count(0)            //constructor
        { /*empty body*/ }
        void inc_count()                //increment count
        { count++; }
        int get_count()                 //return count
        { return count; }
};
void main()
{
     Counter c1, c2;                    //define and initialize
```

```
    cout << "\nc1=" << c1.get_count();        //display
    cout << "\nc2=" << c2.get_count();
    c1.inc_count();                          //increment c1
    c2.inc_count();                          //increment c2
    c2.inc_count();                          //increment c2
    cout << "\nc1=" << c1.get_count();        //display again
    cout << "\nc2=" << c2.get_count();
    cout << endl;
    return 0;
}
```

Output:

```
c1=0
c2=0
c1=1
c2=2
```

1. Implement a program using class definition that defines a class 'counter' with a single member function and its constructor and destructor.
2. Implement a class 'Distance' with feet and inches as its member functions and 'add_dist' as a member function that can add two objects of 'Distance' class.
3. Implement a program using operator overloading that defines a class 'counter' with unary ++ operator.
4. Implement a program using inheritance to derive a class advancedDistance from 'distance' class with that can also subtract two objects.
5. Implement a program using streams to handle a file with employee data.

EXPERIMENT NO.-10

Program: Write a Program in C++ to Two matrix Multiplication

```
#include <iostream>

using namespace std;

int main()

{

int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;

cout<<"enter the number of row=";

cin>>r;

cout<<"enter the number of column=";

cin>>c;

cout<<"enter the first matrix element=\n";

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)

{

cin>>a[i][j];

}

}

cout<<"enter the second matrix element=\n";

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)
```

```
{
cin>>b[i][j];
}
}
cout<<"multiply of the matrix=\n";
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
cout<<mul[i][j]<<" ";
}
```

```
cout<<"\n";
}
return 0;
}
```

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 2 3
1 2 3
1 2 3
enter the second matrix element=
1 1 1
2 1 2
3 2 1
 multiply of the matrix=
14 9 8
14 9 8
14 9 8
```