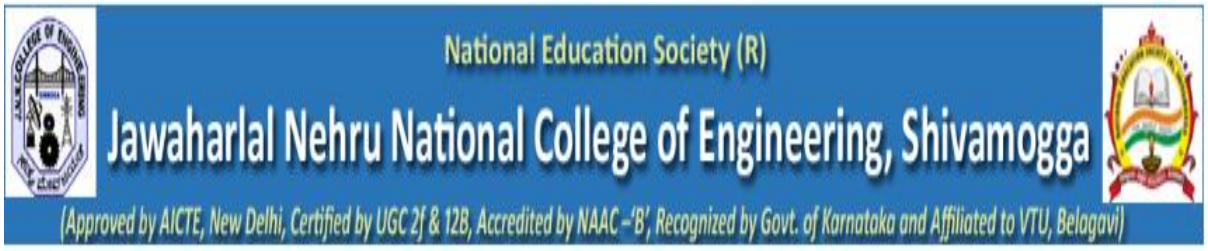


OBJECT ORIENTED CONCEPTS (OOC)

(18CS45)

COMMON TO BOTH

COMPUTER SCIENCE AND ENGINEERING

AND

INFORMATION SCIENCE ENGINEERING

MODULE 1: INTRODUCTION TO OBJECT ORIENTED CONCEPTS

Syllabus:

Introduction to Object Oriented Concepts: A Review of structures, Procedure–Oriented Programming system, Object Oriented Programming System, Comparison of Object Oriented Language with C, Console I/O, variables and reference variables, Function Prototyping, Function Overloading.

Class and Objects: Introduction, member functions and data, objects and functions.

INTRODUCTION TO OBJECT ORIENTED CONCEPTS

Review of the structures

- **The Need of the Structures**

Problem:

- There may be cases (when using groups of variables) where the value of one variable may influence the value of other variables logically.
- However, there is no language construct that actually places these variables in the same group. Thus, members of the wrong group may be accidentally sent to the function.
- Arrays could be used to solve this problem but this will not work if the variables are not of the same type.

Solution: Create a data type itself using structures.

```
struct date             // a structure to represent dates
{
        int d,m,y;
}

void next_day(struct date *);
```

A structure is a programming construct in C that allows us to put together variables that should be together.

- Library programmers use structures to create new data types.
- Application programs and other library programs use these new data types by declaring variables of this data type.

- **Creating New Data type using Structure**

This is a three-step process that is executed by the library programmer as follows:

Step 1 Put the structure definition and the prototypes of the associated functions in a header file. A header file contains the definition of a structure variable and the prototypes of its associated functions.

Step 2 Put the definition of the associated functions in a source code and create a library.

Step 3 Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

Note: Creation of a structure and creation of its associated functions are two separate steps that together constitute one complete process.

J.N.N. College of Engg.

- **Using Structures in Application Programs**

  Step 1 Include the header file provided by the library programmer in the source code.

  Step 2 Declare variables of the new data type in the source code.

  Step 3 Embed calls to the associated functions by passing these variables in the source code.

  Step 4 Compile the source code to get the object file.

  Step 5 Link the object file with the library provided by the library programmer to get the executable file or another library.

Overview of C++

- ✓ C++ extension was first invented by "Bjarne Stroustrup" in 1979.
- ✓ He initially called the new language " C with Classes".
- ✓ However in 1983 the name was changed to C++.
- ✓ c++ is an extension of the C language, in that most C programs are also c++programs.
- ✓ C++, as an opposed to C, supports "Object-Oriented Programming".

Object Oriented Programming System (OOPS)

- In OOPS we try to model real-world objects.
- Most real world objects have internal parts (Data Members) and interfaces (Member Functions) that enables us to operate them.

Object:

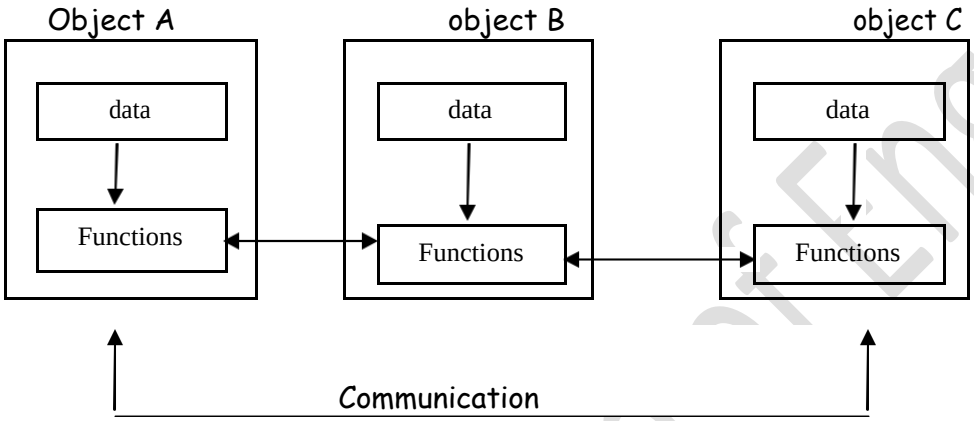
- ✓ Everything in the world is an object.
- ✓ An object is a collection of variables that hold the data and functions that operate on the data.
- ✓ The variables that hold data are called Data Members.
- ✓ The functions that operate on the data are called Member Functions.

The two parts of an object:

- ✓ Object = Data + Methods (Functions)
- ✓ In object oriented programming the focus is on creating the objects to accomplish a task and not creating the procedures (Functions).
- ✓ In OOPs the data is tied more closely to the functions and does not allow the data to flow freely around the entire program making the data more secure.
- ✓ Data is hidden and cannot be easily accessed by external functions.

J.N.N. College of Engg.

- ✓ Compliers implementing OOP does not allow unauthorized functions to access the data thus enabling data security.
- ✓ Only the associated functions can operate on the data and there is no change of bugs creeping into program.
- ✓ The main advantage of OOP is its capability to model real world problems.
- ✓ It follows Bottom Up approach in program design.



- ✓ Identifying objects and assigning responsibilities to these objects.
- ✓ Objects communicate to other objects by sending messages.
- ✓ Messages are received by the methods (functions) of an object.

Basic concepts (features) of Object-Oriented Programming

1. Objects
2. Classes
3. Data abstraction

Three pillars of OOP:

4. Data encapsulation
5. Inheritance
6. Polymorphism

❖ **Objects and Classes:**

- ➢ Classes are user defined data types on which objects are created.
- ➢ Objects with similar properties and methods are grouped together to form class.
- ➢ So class is a collection of objects.
- ➢ Object is an instance of a class.

J.N.N. College of Engg.

- **Data abstraction**
  - Abstraction refers to the act of representing essential features without including the background details or explanation.
  - **Ex:** Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.
  - **Ex**: #include <iostream>

```
int main( )
{
  cout << "Hello C++" <<endl;
  return 0;
}
```

  - Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.
- **Data encapsulation**
  - Information hiding
  - Wrapping (combining) of data and functions into a single unit (class) is known as data encapsulation.
  - Data is not accessible to the outside world, only those functions which are wrapped in the class can access it.
- **Inheritance**
  - Acquiring qualities.
  - Process of deriving a new class from an existing class.
  - Existing class is known as base, parent or super class.
  - The new class that is formed is called derived class, child or sub class.
  - Derived class has all the features of the base class plus it has some extra features also.
  - Writing reusable code.
  - Objects can inherit characteristics from other objects.

J. N. College of Engg.

- **Polymorphism**
  - The dictionary meaning of polymorphism is "having multiple forms".
  - Ability to take more than one form.
  - A single name can have multiple meanings depending on its context.
  - It includes function overloading, operator overloading.

The process of programming in an OOP involves the following basic steps:

1. Creating classes that define objects and behavior.
2. Creating objects from class definitions.
3. Establishing communications among objects.

Advantages of OOPS

- ➔ Data security
- ➔ Reusability of existing code
- ➔ Creating new data types
- ➔ Abstraction
- ➔ Less development time
- ➔ Reduce complexity
- ➔ Better productivity

Benefits of OOP

- ➔ Reusability
- ➔ Saving of development time and higher productivity
- ➔ Data hiding
- ➔ Multiple objects feature
- ➔ Easy to partition the work in a project based on objects.
- ➔ Upgrade from small to large systems
- ➔ Message passing technique for interface.
- ➔ Software complexity can be easily managed.

Applications of OOP

- ➔ Real time systems

J.N.N. College of Engg.

- ➔ Simulation and modeling
- ➔ Object oriented databases
- ➔ Hypertext, hypermedia
- ➔ AI (Artificial Intelligence)
- ➔ Neural networks and parallel programming
- ➔ Decision support and office automation systems
- ➔ CIM/CAD/CAED system

Difference between POP(Procedure Oriented Programming) and OOP(Object Oriented Programming)

Sl.No	POP	OOP
1.	Emphasis is on procedures (functions)	Emphasis is on data
2.	Programming task is divided into a collection of data structures and functions.	Programming task is divided into objects (consisting of data variables and associated member functions)
3.	Procedures are being separated from data being manipulated	Procedures are not separated from data, instead, procedures and data are combined together.
4.	A piece of code uses the data to perform the specific task	The data uses the piece of code to perform the specific task
5.	Data is moved freely from one function to another function using parameters.	Data is hidden and can be accessed only by member functions not by external function.
6.	Data is not secure	Data is secure
7.	Top-Down approach is used in the program design	Bottom-Up approach is used in program design
8.	Debugging is the difficult as the code size increases	Debugging is easier even if the code size is more

J.N.N. College of Engg.

Comparison of C with C++

Sl.No	C	C++
1.	It is procedure oriented language	It is object-oriented language
2.	Emphasis is on writing the functions which performs some specific tasks.	Emphasis is on data which uses functions to achieve the task.
3.	The data and functions are separate	The data and functions are combined
4.	Does not support polymorphism, inheritance etc.	Supports polymorphism, inheritance etc.
5.	They run faster	They run slower when compared to equivalent C program
6.	Type checking is not so strong	Type checking is very strong
7.	Millions of lines of code management is very difficult	Millions of lines of code can be managed very easily
8.	Function definition and declarations are not allowed within structure definitions	Function definitions and declarations are allowed within structure definitions.

Console Output/input in C++

Cin: used for keyboard input.

Cout: used for screen output.

Since Cin and Cout are C++ objects, they are somewhat "Intelligent".

- ✓ They do not require the usual format strings and conversion specifications.
- ✓ They do automatically know what data types are involved.
- ✓ They do not need the address operator and ,
- ✓ They do require the use of the stream extraction (>>) and insertion (<<) operators.

Extraction operator (>>):

- ✓ To get input from the keyboard we use the extraction operator and the object Cin.
- ✓ Syntax: Cin>> variable;
- ✓ No need for "&" infront of the variable.
- ✓ The compiler figures out the type of the variable and reads in the appropriate type.

J.N.N. College of Engg.

- Example:

```
#include<iostream.h>
Void main( )
{
        int x;
        float y;
        cin>> x;
        cin>>y;
}
```

Insertion operator (<<):

- To send output to the screen we use the insertion operator on the object Cout.
- Syntax: Cout<<variable;
- Compiler figures out the type of the object and prints it out appropriately.

Example:

```
#include<iostream.h>
void main( )
{
        cout<<5;
        cout<<4.1;
        cout<< "string";
        cout<< '\n';
}
```

Programs

Example using Cin and Cout

```
#include<iostream.h>
void main( )
{
        int a,b;
        float k;
        char name[30];
        cout<< "Enter your name \n";
        cin>>name;
        cout<< "Enter two Integers and a Float \n";
        cin>>a>>b>>k;
        cout<< "Thank You," <<name<<",you entered\n";
        cout<<a<<","<<b<<",and"<<k<<'/n';
}
```

Output:

```
Enter your name
Mahesh
```

J.N. College of Engg.

```
Enter two integers and a Float
10
20
30.5
        Thank you Mahesh, you entered
        10, 20 and 30.5
```

C++ program to find out the square of a number

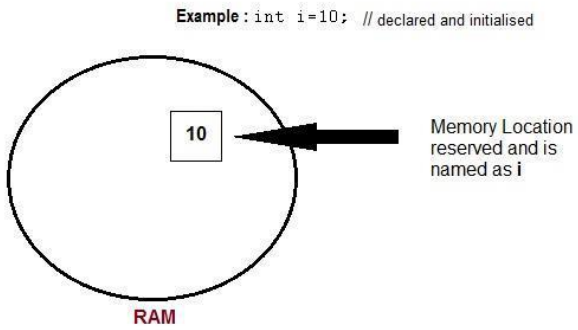
```
#include<iostream.h>
int main( )
{
        int i;
        cout<< "this is output\n";
        cout<< "Enter a number";
        cin>>i;
        cout<<i<< "Square is" << i*i<<"\n";
        return 0;
}
```

Output:

```
This is output
        Enter a number  5
         5 square is 25
```

Variables

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.



J.N.N. College of Engg.

Declaration and Initialization

- ✓ Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example **:**

```
int i;      // declared but not initialised
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;  // declaration
i = 10;  // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;       //initialization and declaration in same step
int i=10, j=11;
```

J.N. College of Engg.

- ✓ If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;        //compile time error, cannot redeclare a variable in same scope
```

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces, in which variable is declared that a variable exists, not outside it. we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which are once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the main() function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

Example : Only declared, not initialized

```
include <iostream>
int x;              // Global variable declared
int main()
{
        x=10;               // Initialized once
```

J.N.N. College of Engg.

```
        cout <<"first value of x = "<< x;
        x=20;              // Initialized again
        cout <<"Initialized again with value = "<< x;
}
```

Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example :

```
include <iostream>
int main()
{
        int i=10;
        if(i<20)       // if condition scope starts
        {
         int n=100;   // Local variable declared and initialized
        }// if condition scope ends
        cout << n;     // Compile time error, n not available here
}
```

Reference variable in C++

- ✓ When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

```
#include<iostream>
using namespace std;

int main()
{
  int x = 10;
```

J.N. College of Engg.

```
  // ref is a reference to
  x. int& ref = x;

  // Value of x is now changed to
  20 ref = 20;
  cout << "x = " << x << endl ;

  // Value of x is now changed to
  30 x = 30;
  cout << "ref = " << ref << endl ;

  return 0;
}
```

Output:

```
 x = 20
ref = 30
```

Functions in c++:

Definition: Dividing the program into modules, these modules are called as functions.

General form of function:

return_type function_name(parameter list)

{

Body of the function

}

Where,

return_type:

- ✓ What is the value to be return.
- ✓ Function can written any value except array.

Parameter_list: List of variables separated by comma.

The body of the function(code) is private to that particular function, it cannot be accessed outside the function.

J.N. College of Engg.

Components of function:

- Function declaration (or) prototype.
- Function parameters (formal parameters)
- Function definition
- Return statement
- Function call

Example:

```
#include<iostream.h>
int max(int x, int y); //prototype(consists of formal arguments)

void main( ) //Function caller
{
        int a, b, c;
        cout<< "enter 2 integers";
        cin>>a>>b;
        c=max(a,b); //function call
        cout<<c<<endl;
}

int max(int x, int y) // function definition
{
        if(x>y)
                return x; // function return
        else
                return y;
}
```

Function prototype:

```
int max(int x, int y);
```

- It provides the following information to the compiler.
- The name of the function
- The type of the value returned(default an integer)
- The number and types of the arguments that must be supplied in a call to the function.

J. N. College of Engg.

- Function prototyping is one of the key improvements added to the C++ functions.

When a function is encountered, the compiler checks the function call with its prototype so that correct argument types are used.

Consider the following statement:

```
int max(int x, int y);
```

- It informs the compiler that the function max has 2 arguments of the type integer.
- The function max( ) returns an integer value the compiler knows how many bytes to retrieve and how to interpret the value returned by the function.

Function definition:

- The function itself is returned to as function definition.
- The first line of the function definition is known as function declarator and is followed by function body.
- The declarator and declaration must use the same function name, number of arguments, the argument type and return type.
- The body of the function is enclosed in braces.
- C++ allows the definition to be placed anywhere in the program.

```
int max(int x, int y)       // function declaration, no semicolon
{
   if(x>y)  //function body
          return x;
   else
          return y;
}
```

Function call:

```
c= max (a, b) ;
```

- Invokes the function max( ) with two integer parameters, executing the call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is resumed to the statement following the function call. The max( ) returns the maximum of the parameters a and b. the return value is assigned to the local variable c in main( ).

Function parameters:

- The parameters specified in the function call are known as actual parameters and specified in the declarator are known as formal parameters.

J.N.N. College of Engg.

c=max(a,b);

- Here a and b are actual parameters. The parameters x and y are formal parameters. When a function call is made, a one to one correspondence is established between the actual and the formal parameters. In this case the value of the variable aa is assigned to the variable x and that of b to y. the scope of formal parameters is limited to the function only.

Function return:

- Functions can be grouped into two categories. Functions that do not have a return value(void) and functions that have a return value.

The statement: return x;// function return and
return y;//function return

ex: c=max(a,b);//function call

the value returned by the function max( ) is assigned to the local variable c in main( ).

- The return statement in a function need not be at the end of the function. It can occur anywhere in the function body and as soon as it is encountered , execution control will be returns to the caller.

Argument passing:

Two types

1. Call by value
2. Call by reference

➔ **Call by value:**

- ✓ The default mechanism of parameter passing(argument passing) is called call by value.
- ✓ Here we pass the value of actual arguments to formal parameters.
- ✓ Changes made to the formal parameters will not be affected the actual parameters.

Example 1:

```
#include<iostream.h>
void exchange(int x, int y);
 void main( )
{
        int a, b;
```

J.N.N. College of Engg.

```
        cout<< "enter values for a and b";
                                                    // 10 and 20
        cin>>a>>b;
        exchange(a,b);
        cout<<a<<b;                                 output: 10, 20
}

void exchange(int x, int y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
        cout<<x<<y;output:20,10
}
```

Example 2:

```
#include<iostream.h>
void main( )
{
        int a, b;
        cout<<" enter the value of a and b\n";          // 20 and 10
        cin>>a>>b;
        sub(a, b);
        getch( );
}
```

J.N.N. College of Engg.

```
void sub(int x, int y)
{
        int result;
        result=x-y;
        cout<<result;                    output: 10
}
```

Example 3:

```
#include<iostream.h>
void main( )
{
        int a=10, temp;
        temp=add(a);
        cout<<temp<<","<<a;
}
int add(int a)
{
        a=a+a;
        return a;
}
```

Output: 20

J.N.N. College of Engg.

➔ **Call by reference:**

- We pass address of an argument to the formal parameters.
- Changes made to the formal parameters will affect actual arguments.

Example 1:

```
#include<iostream.h>
void exchange(int *x, int *y);
 void main( )
{
        int a, b;
        cout<< "enter values for a and b";   //10, 20
        cin>>a>>b;
        exchange(&a,&b);
        cout<<a<<b;
}
void exchange(int *x, int *y)
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
        cout<<x<<y;             //  output: 20, 10

}
```

Example 2:

```
#include<iostream.h>
void main( )
{
        int a=10, temp;

        temp=add(&a);
        cout<<temp<<","<<a;
        getch();
}
int add(int *a)
{
        a=*a+*a;
        return a;
}
```

Output: 20

J. N. College of Engg.

Default arguments:

- Default values are specified when the function is declared.
- The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Example:

```
#include <iostream.h>

void add(int a=10, int b=20,int c=30);

void main( )
{
        add(1,2,3);
        add(1,2);
        add(1);
        add( );
}
void add(int a, int b, int c)
{

        cout<< a+b+c;

}
```

- A default argument is checked for type at the time of declaration and evaluated at the time of call.
- We must add defaults from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.

Example:

```
int mul (int i, int j=5, int k=10); //legal.
int mul (int i=5, int j); //illegal.
int mul (int i=0,int j, int k=10); //illegal.
int mul (int i=2, int j=5, int k=10); //legal.
```

- Default arguments are useful in situations where some arguments always have the same value.

J.N. College of Engg.

Classes & Objects

Structure of C++ Program

Programming language is most popular language after C Programming language. C++ is first Object oriented programming language.

➔ **Header File Declaration Section:**

- Header files used in the program are listed here.
- Header File provides Prototype declaration for different library functions.
- We can also include user define header file.
- Basically all preprocessor directives are written in this section.

➔ **Global declaration section:**

- Global Variables are declared here.
- Global Declaration may include
- Declaring Structure
- Declaring Class
- Declaring Variable

➔ **Class declaration section:**

- Actually this section can be considered as sub section for the global declaration section.
- Class declaration and all methods of that class are defined here

J.N.N. College of Engg.

➔ **Main function:**

- Each and every C++ program always starts with main function.
- This is entry point for all the function. Each and every method is called indirectly through main.
- We can create class objects in the main.
- Operating system calls this function automatically.

➔ **Method definition section**

- This is optional section. Generally this method was used in C Programming.

Class specification:

- A Class is way to **bind(combine)** the data and its associated functions together. it allows data and functions to be hidden.
- When we define a class, we are creating a new abstract data type that can be created like any other built-in data types.
- This new type is used to declare objects of that class.
- Object is an instance of class.

General form of class declaration is:

```
class  class_name
{
   access specifier: data

   access specifier: functions;

};
```

✓ The keyword class specifies that what follows is an abstract data of type class_name.the body of the class is enclosed in braces and terminated by semicolon.

J.N.N. College of Engg.

Access specifier can be of 3 types:

Private:

- Cannot be accessed outside the class.
- But can be accessed by the member functions of the class.

Public:

- Allows functions or data to be accessible to other parts of the program.

Protected:

- Can be accessed when we use inheritance.

Note:

- By default data and member functions declared within a class are private.
- Variables declared inside the class are called as data members and functions are called as member functions. Only member functions can have access to data members and function.

- The binding of functions and data together into a single class type variable is referred as ***Encapsulation***.

Example:

```
#include<iostream.h>
class student
{
   private:
            char name[10]; // private variables
            int marks1,marks2;

   public:
          void getdata( ) // public function accessing private members
            {
                    cout<<"enter name,marks in two subjects";
                    cin>>name>>marks1>>marks2;
             }
```

J. N. College of Engg.

```
        void display( ) // public function
         {
                cout<<"name:"<<name<<endl;
            cout<<"marks"<<marks1<<endl<<marks2;
         }
}; // end of class

void main( )
{
    student obj1;
     obj1.getdata( );
     obj1.display( );
}
```

Output:

```
Enter name,marks in two subjects
Mahesh    25    24

Name: Mahesh
Marks  25 24
```

In the above program,class name is student,with private data members name,marks1 and marks2,the public data members getdata( ) and display( ).

Functions,the getdata( ) accepts name and marks in two subjects from user and display( ) displays same on the output screen.

Scope resolution operator (::)

- ✓ It is used to define the member functions outside the class.
- ✓ Scope resolution operator links a class name with a member name in order to tell the compiler what class the member belongs to.
- ✓ Used for accessing global data.

Syntax to define the member functions outside the class using Scope resolution operator:

```
return_type  class_name  : :  function_name(actual arguments)
{
        function body
}
```

J.N.N. College of Engg.

Example:

```
#include<iostream.h>
class student
 {

    private:
                char name[10]; // private variables
                int marks1,marks2;
   public:
                 void getdata( );
                        void display( );
    };

  void student: :getdata( )
   {
      cout<<"enter name,marks in two subjects";
      cin>>name>>marks1>>marks2;
     }

 void student: :display( )
  {
          cout<<"name:"<<name<<endl;
          cout<<"marks"<<marks1<<endl<<marks2;
   }

void main( )
{
       student obj1;
        obj1.getdata( );
        obj1.display( );
}
```

Accessing global variables using scope resolution operator (: :)

Example:

```
#include<iostream.h>
int a=100; //  declaring global variable

class  x
 {
            int a;

   public:
              void f( )
            {
                a=20; // local variable
```

J. N. College of Engg.

```
            cout<<a; // prints value of a as 20
        }
};
void main( )
{
    x g;
    g.f( ); // this function prints value of a(local variable) as 20
   cout<<::a; // this statement prints value of a(global variable) as 100
}
```

In the above program, the statement ::a prints global variable value of a as 100.

Defining the functions with arguments(parameters):

```
#include<iostream.h>
class item
{
private:
            int number,cost;
public:
           void getdata(int a,int b );
           void display( );
 };

void item::getdata(int a,int b)
 {
          number=a;
          cost=b;
  }
  void item::display( )
  {
          cout<<"cost:"<<number<<endl;
          cout<<"number:"<<cost<<endl;
}
 void main( )
  {
          item i1;
          i1.getdata(10,20);
          i1.display( );
   }
```

output:

```
number:10
cost:20
```

J.N.N. College of Engg.

Access members

- Class members(variables(data) and functions) Can be accessed through ***an object and dot operator.***
- Private members can be accessed by the functions which belong to the same class.

The **format** for calling a member function is:

Object_name.function_name(actual arguments);

Example: accessing private members

```
#include<iostream.h>
class item
{
        Private:        int a;

         public:         int b;

};

void main( )
{
       item i1,i2;
        i1.a=10;  // illegal private member cannot be accessed outside the class

          i2.b=20;
       cout<<i2.b;  // this statement prints value of b as 20.
}
```

Note: private members cannot be accessed outside the class but public members can be accessed.

J.N.N. College of Engg.

Example: private members can be accessed by the functions which belongs to the same class

```
#include<iostream.h>
class item
{
            int a=10; // private member
 public:
          void display( )
        {
            cout<<a; // it prints a as10
        }
};

void main( )
{
    item i1;
   i1.display( );
}
```

Defining member functions

- We can define the function inside the class.
- We can define the function outside the class.

The member functions have some special characteristics:

- Several different classes can use same function name.
- Member function can access private data of the class.
- A member function can call another function directly, without using dot operator.

Defining the function inside the class:

- Another method of defining a member function is to replace the function declaration by actual function definition inside the class.

Example:

```
#include<iostream.h>
 class item
 {
    private:
                int cost,number;

     public:
```

J.N. College of Engg.

```
                    void getdata(int a,int b ) // defining function inside the class
                   {
                            number=a;
                            cost=b;
                     }
                    void display( )
                   {
                    cout<<"cost:"<<number<<endl;
                    cout<<"number:"<<cost<<endl;
                   }
          };


  void main( )
  {
           item i1;
           i1.getdata(10,30 );
           i1.display( );
  }
```

output:

number:10
cost:30

Function Overloading in C++

- ✓ Two or more functions have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different is called **function overloading**. An example of the function overloading is given below:

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define pi 3.14

class fn
{
     public:
       void area(int);  //circle
       void area(int,int);  //rectangle
       void area(float ,int,int);  //triangle
};

void fn::area(int a)
```

J.N.N. College of Engg.

```
{
     cout<<"Area of Circle:"<<pi*a*a;
}
void fn::area(int a,int b)
{
     cout<<"Area of rectangle:"<<a*b;
}
void fn::area(float t,int a,int b)
{
     cout<<"Area of triangle:"<<t*a*b;
}

void main()
{
    int ch;
    int a,b,r;
    clrscr();
    fn obj;
    cout<<"\n\t\tFunction Overloading";
    cout<<"\n1.Area of Circle\n2.Area of Rectangle\n3.Area of Triangle\n4.Exit\n:";
    cout<<"Enter your Choice:";
    cin>>ch;

    switch(ch)
    {
            case 1:
              cout<<"Enter Radious of the Circle:";
              cin>>r;
              obj.area(r);
              break;
            case 2:
              cout<<"Enter Sides of the Rectangle:";
              cin>>a>>b;
              obj.area(a,b);
              break;
            case 3:
              cout<<"Enter Sides of the Triangle:";
              cin>>a>>b;
              obj.area(0.5,a,b);
              break;
            case 4:
              exit(0);
    }
getch();
}
```

J.N.N. College of Engg.

Static data members

- Data members of class be qualified as static.

A static data member has certain special characteristics:

- It is initialized to zero when first object is created. No other initialization is permitted.
- Only one copy of the data member is created for the entire class and is shared by all the objects of class. no matter how many objects are created.
- Static variables are normally used to maintain values common to entire class objects.

Example

```
class item
{
                static int count; // static data member
                int number;
 public:
          void  getdata( )
          {
              number=a;
              count++;
          }

        void  putdata( )
         {
           cout<<"count value"<<count<<endl;
         }
};
void main( )
{
        item i1,i2,i3;  // count is initialized to zero
         i1.putdata( );
         i2.putdata( );
         i3.putdata( );
        i1.getdata( );
        i2.getdata( );
        i3.getdata( );
         i1.putdata( );    // display count after reading data
         i2.putdata( );
         i3.putdata( );
}
```

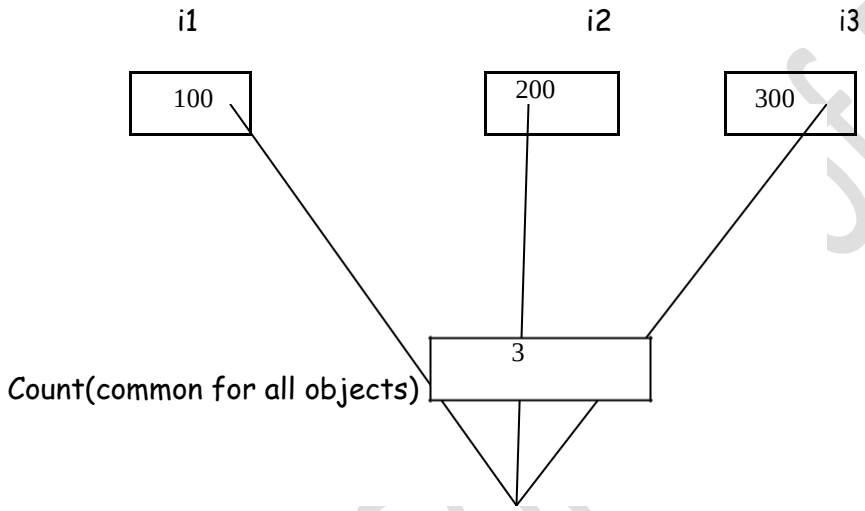
Output:

Count value 0

J. N. College of Engg.

Count value 0
Count value 0
Count value 3
Count value 3
Count value 3

In the above program,the static variable count is initialized to zero when objects are created.count is incremented whenever data is read into object.since three times getdata( ) is called,so 3 times count value is created.all the 3 objects will have count value as 3 because count variable is shared by all the objects,so all the last 3 statements in main( ) prints values of count value as 3.



Static member functions

- ✓ Like a static member variable, we can also have static member functions.

A member function that is declared as static has the following properties:

- ✓ A static member function can have access to only other static members declared in the same class.
- ✓ A static member function can be called using the class name, instead of objects.

Syntax:

class_name : : function_name ;

Example:

```
class item
{
```

```
        int number;
       static int count;

  public:
              void getdata(int a )
               {
                       number=a;
                      count++;
               }
              static void putdata( )
               {
                      cout<<"count value"<<count;
               }
};

void main( )
{
        item i1,i2;
         i1.getdata(10);
         i2.getdata(20);
          item::putdata( );
      // call static member function using class name with scope  resolution operator.
 }
```

Output:

Count value 2

- In the above program, we have one static data member count, it is initialized to zero, when first object is created, and one static member function putdata( ),it can access only static member.
- When getdata( ) is called,twice,each time, count value is incremented, so the value of count is 2.when static member function putdata( ) is called, it prints value of count as 2.

Inline functions:

- First control will move from calling to called function. Then arguments will be pushed on to the stack, then control will move back to the calling from called function.
- This process takes extra time in executing.
- To avoid this, we use inline function.
- When a function is declared as inline, compiler replaces function call with function code.

Example:

```
#include<iostream.h>
```

J.N.N. College of Engg.

```
void main( )
{
        cout<< max(10,20);
        cout<<max(100,90);
        getch( );
}
inline int max(int a, int b)
{
        if(a>b)
                return a;
        else
                return b;
}
```

Output: 20
100

Note: inline functions are functions consisting of one or two lines of code.

Inline function cannot be used in the following situation:

- If the function definition is too long or too complicated.
- If the function is a recursive function.
- If the function is not returning any value.
- If the function is having switch statement and goto statement.
- If the function having looping constructs such as while, for, do-while.
- If the function has static variables

J.N.N. College of Engg.

Questions

1. State the important features of object oriented programming. Compare object oriented programming with procedure oriented programming.
2. Give comparison of C and C++ with example
3. Write the general form of function. Explain different argument passing techniques with example
4. Define function overloading. Write a C++ program to define three overloaded functions to swap two integers, swap two floats and swap two doubles
5. Write a C++ program to overload the function area() with three overloaded function to find area of rectangle and area rectangle box and area of circle
6. Explain the working of inline functions with example
7. Write a C++ recursive program to find the factorial of a given number
8. Explain the use of scope resolution operator
9. Implement a C++ program to find prime number between 200 and 500 using for loop.
10. List a few areas of applications of OOP Technology.
11. What is class?how it is created? Write a c++ program to create a class called Employee with data members name age and salary. Display atleast 5 employee information
12. What is nested class? What is its use? Explain with example.
13. What is static data member?explain with example. What is the use of static members
14. Write a class rectangle which contains data items length and breadth and member functions setdata() getdata() displaydata(),area() to set length and breadth, to take user input,to display data and find area of rectangle.

J.N.N. College of Engg

MODULE 2: CLASS AND OBJECTS(CONTD) AND INTRODUCTION TO JAVA

Syllabus:

Class and Objects(Contd): objects and arrays, Namespaces, Nested classes, Constructors, Destructors.

Introduction to Java: Java and Java applications; Java Development Kit (JDK); Java is interpreted, Byte Code, JVM; Object-oriented programming; Simple Java programs.

Data types and other tokens: Boolean variables, int, long, char, operators, arrays, white spaces, literals, assigning values; Creating and destroying objects; Access specifiers.

Operators and Expressions: Arithmetic Operators, Bitwise operators, Relational operators, The Assignment Operator, The ? Operator; Operator Precedence; Logical expression; Type casting; Strings.

Control Statements: Selection statements, iteration statements, Jump Statements.

Class and Objects(Contd)

Arrays of Objects:

- It is possible to have arrays of objects.
- The syntax for declaring and using an object array is exactly the same as it is for any other type of array.
- A array variable of type class is called as an array of objects.

Program:

```
#include<iostream.h>
 class  c1
 {
                    int i;
        public:
                    void get_i(int j)
                     {
                              i=j;
                     }
                     void show( )
                     {
                              cout<<i<<endl;
                     }
 };
void main( )
{
              c1  obj[3];   // declared array of objects
            for(int i=0;i<3;i++)
                     obj[i].get_i(i);

            for(int i=0;i<3;i++)
                      obj[i].show( );
 }
```

In the above program,we have declared object obj as an array of objects[i.e created 3 objects].

J. N. N. College of Engg.

The following statement:

obj[i].get_i(i);

invokes get_i( ) function 3 times,each time it stores value of i in the index of obj[i].that is after the execution of complete loop,the array of object "obj" looks like this:

2
1
0

The following statement

obj[i].show( );

displays the array of objects contents:0,1,2

output: 0

1

2

Namespace

What is Namespace in C++?

Namespace is a new concept introduced by the ANSI C++ standards committee. For using identifiers it can be defined in the namespace scope as below.

Syntax:

using namespace std;

In the above syntax "std" is the namespace where ANSI C++ standard class libraries are defined. Even own namespaces can be defined.

Syntax:

```
namespace namespace_name
{
    //Declaration of variables, functions, classes, etc.
}
```

J. N. College of Engg.

Example :

```
#include <iostream.h>
using namespace std; namespace Own
{
      int a=100;
}


 int main()
{
      cout << "Value of a is:: " << Own::a;
      return 0;
}
```

Result :

Value of a is:: 100

In the above example, a name space "Own" is used to assign a value to a variable. To get the value in the "main()" function the "::" operator is used.

Nested Classes in C++

Nested class is a class defined inside a class that can be used within the scope of the class in which it is defined. In C++ nested classes are not given importance because of the strong and flexible usage of inheritance. Its objects are accessed using "Nest::Display".

J.N.N. College of Engg.

Example :

```
#include <iostream.h>
class Nest
{
      public:
            class Display
           {
               private:
                        int s;
               public:
                     void sum( int a, int b)
                   {
                            s =a+b;
                    }
                   void show( )
                  {
                        cout << "\nSum of a and b is:: " << s;
                  }
        };  //closing of inner class
}; //closing of outer class




void main()
{
     Nest::Display x;          // x is a object, objects are accessed using "Nest::Display".
       x.sum(12, 10);
      x.show();
}
```

Result : Sum of a and b is::22

In the above example, the nested class "Display" is given as "public" member of the class "Nest".

J.N. College of Engg.

Constructors

- ✓ Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructor's initialize values to data members after storage is allocated to the object.

```
class A
{
        int x;
         public:   A(); //Constructor
};
```

- ✓ While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.
- ✓ Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
        int i;
        public:
                A(); //Constructor declared
};

A::A()  // Constructor definition
{
        i=1;
}
```

Types of Constructors

Constructors are of three types :

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

J.N.N. College of Engg.

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ()
{
        Constructor Definition
}
```

Example :

```
class Cube
{
        int side;
        public:   Cube()                //constructor
                  {
                          side=10;
                   }
};

int main()
{
        Cube c;       //constructor is going to call
        cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};

int main()
{
 Cube c;
 cout << c.side;
}                Output : 0
```

J.N.N. College of Engg.

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Example :

```
class Cube
{
        int side;
        public:
                Cube(int x)
                {
                        side=x;
                }
};

int main()
{
        Cube c1(10);
        Cube c2(20);
        Cube c3(30);
        cout << c1.side;
        cout << c2.side;
        cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

copy constructor

What is copy constructor and how to use it in C++?

A **copy constructor** in C++ programming language is used to reproduce an identical copy of an original existing object.It is used to initialize one object from another of the same type.

Example :

```
#include<iostream>
```

J.N.N. College of Engg.

```
using namespace std;
class copycon
{
      int copy_a,copy_b; // Variable Declaration
      public:
            copycon(int x,int y)
           {
              //Constructor with Argument
                copy_a=x;
                copy_b=y; // Assign Values In Constructor
            }
           void Display()
          {
                      cout<<"\nValues :"<< copy_a <<"\t"<< copy_b;
          }
};

int main()
 {
          copycon obj(10,20);
          copycon obj2=obj; //Copy Constructor
         cout<<"\nI am Constructor";
          obj.Display(); // Constructor invoked.
          cout<<"\nI am copy Constructor";
          obj2.Display();
          return 0;
}
```

Result :

```
I am Constructor
Values:10 20
I am Copy Constructor
Values:10 20
```

Constructor Overloading

- ✓ Just like other member functions, constructors can also be overloaded. In fact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.
- ✓ You can have any number of Constructors in a class that differ in parameter list.

J. N. College of Engg.

```
class Student
{
        int rollno;
        string name;
        public:
                Student(int x)
                {
                        rollno=x;
                        name="None";
                }
                Student(int x, string str)
                {
                        rollno=x ;
                        name=str ;
                }
};

int main()
{
        Student A(10);
        Student B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Destructors

- ✓ Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.
- ✓ The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

J.N.N. College of Engg.

```
class A
{
 public:
            ~A();
};
```

Destructors will never have any arguments.

Example to see how Constructor and Destructor is called

```
class A
{
A()
 {
  cout << "Constructor called";
 }

~A()
 {
  cout << "Destructor called";
 }
};

int main()
{
 A obj1;   // Constructor Called
 int x=1
 if(x)
  {
   A obj2; // Constructor Called
  } // Destructor Called for
obj2 } // Destructor called for
obj1
```

J.N.N. College of Engg.

Introduction to JAVA

Basic concepts of object oriented programming

Object:

This is the basic unit of object oriented programming. That is both data and method that operate on data are bundled as a unit called as object. **It is a real world entity (Ex:a person, book, tables, chairs etc…)**

Class:

Class is a **collection of objects** or class is a **collection of instance variables and methods**. When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction:

Data abstraction refers to, **providing only essential information** to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is ***placing the data and the methods/functions*** that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance:

One of the most useful aspects of object-oriented programming is **code reusability**. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object oriented programming since this feature helps to reduce the code size.

Polymorphism:

The ability to use a method/function in different ways in other words giving different meaning for method/ functions is called polymorphism. Poly refers many. That is a single method/function functioning in many ways different upon the usage is called polymorphism.

Java History:

Java is a general-purpose object oriented programming language developed by sun Microsystems of USA in the year 1991. The original name of Java is Oak. Java was designed for the development of the software for consumer electronic devices like TVs, VCRs, etc.

Introduction: Java is a general purpose programming language. We can develop two types of Java application. They are:

J.N.N. College of Engg

(1). Stand alone Java application.
(2). Web applets.

Stand alone Java application: Stand alone Java application are programs written in Java to carry out certain tasks on a certain stand alone system. Executing a stand-alone Java program contains two phases:

(a) Compiling source coded into bytecode using javac compiler.
(b) Executing the bytecodede program using Java interpreter.

Java applet: Applets are small Java program developed for Internet application. An applet located on a distant computer can be downloaded via Internet and execute on local computer.

Java and Internet:

Java is strongly associated with Internet. Internet users can use Java to create applet programs and run them locally using a "Java enabled Browser" such as "hotjave". They can also use a Java enabled browser to download an applet locating on any computer any where in the internet and run them locally.

Internet users can also set their web-sites containing Java applets that could be used by other remote users of Internet. The ability of the Java applets to hitch a ride on the information makes Java a unique programming language for Internet.

Java Environment:

Java environment includes a large number of development tools and hundreds of classes and methods. The Java development tools are part of the systems known as Java development kit (JDK) and the classes and methods are part of the Java standard library known as Java standard Library (JSL) also known as application program interface (API).

Java Features (Java Buzz Words):

(1) Compiled and Interpreted
(2) Architecture Neutral/Platform independent and portable
(3) Object oriented
(4) Robust and secure.
(5) Distributed.
(6) Familiar, simple and small.
(7) Multithreaded and interactive.
(8) High performance
(9) Dynamic and extendible.

1. **Compiled and Interpreted**
   Usually a computer language is either compiled or interpreted. Java combines both these approaches; first java compiler translates source code into bytecode instructions. Bytecodes are not machine instructions and therefore, in the second

JNN College of Engg.

stage, java interpreter generates machine code that can be directly executed by the machine that is running the java program.

2. **Architecture Neutral/Platform independent and portable**
   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 closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM.

3. **Object oriented**
   In java everything is an Object. Java can be easily extended since it is based on the Object model.java is a pure object oriented language.

4. **Robust and secure.**
   Java is a robust language; Java makes an effort to eliminate error situations by emphasizing mainly on compile time error checking and runtime checking. Because of absence of pointers in java we can easily achieve the security.

5. **Distributed.**
   Java is designed for the distributed environment of the internet.java applications can open and access remote objects on internet as easily as they can do in the local system.

6. **Familiar, simple and small.**
   Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.

7. **Multithreaded and interactive.**
   With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

8. **High performance**
   Because of the intermediate bytecode java language provides high performance

9. **Dynamic and extendible.**
   Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Java Development kits(java software:jdk1.6): Java development kit comes with a number of
Java development tools. They are:

(1) Appletviewer: Enables to run Java applet.
(2) javac:Java compiler.

J.N.N. College of Engg.

(3) java :Java interpreter.
(4) javah :Produces header files for use with native methods.
(5) javap :Java disassembler.
(6) javadoc : Creates HTML documents for Java source code file.
(7) jdb :Java debugger which helps us to find the error.

Java Building and running Process:

1. **Open the notepad and type the below program Simple Java program:**

Example:

```
class Sampleone
{
        public static void main(String args[])
        {
                System.out.println("Welcome to JAVA");
        }
}
```

Description:

(1) **Class declaration:** "class sampleone" declares a class, which is an object-oriented construct. Sampleone is a Java identifier that specifies the name of the class to be defined.
(2) **Opening braces:** Every class definition of Java starts with opening braces and ends with matching one.
(3) **The main line:** the line " public static void main(String args[]) " defines a method name main. Java application program must include this main. This is the starting point of the interpreter from where it starts executing. A Java program can have any number of classes but only one class will have the main method.
(4) **Public:** This key word is an access specifier that declares the main method as unprotected and therefore making it accessible to the all other classes.
(5) **Static:** Static keyword defines the method as one that belongs to the entire class and not for a particular object of the class. The main must always be declared as static.
(6) **Void:** the type modifier void specifies that the method main does not return any value.
(7) **The println:** It is a method of the object out of system class. It is similar to the printf or cout of c or c++.

2. Save the above program with .java extension, here file name and class name should be same, ex: Sampleone.java,
3. Open the command prompt and **Compile** the above program

   javac Sampleone.java

   From the above compilation the java compiler produces a bytecode(.class file)
4. Finally run the program through the **interpreter java**

   Sapleone.java

J.N.N. College of Engg.

Output of the program:

Welcome to JAVA

Implementing a Java program: Java program implementation contains three stages. They are:

1. Create the source code.
2. Compile the source code.
3. Execute the program.

(1) Create the source code:

1. Any editor can be used to create the Java source code.
2. After coding the Java program must be saved in a file having the same name of the class containing main() method.
3. Java code file must have .Java extension.

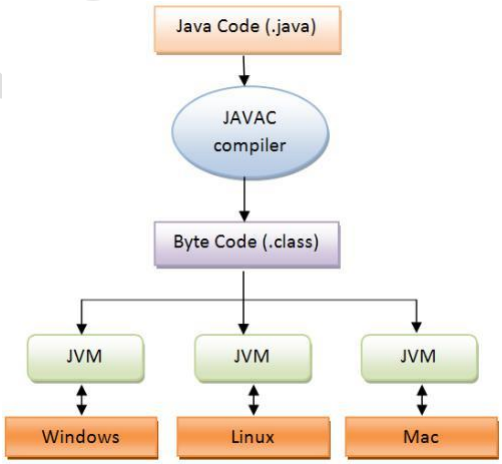
(2) Compile the source code:

1. Compilation of source code will generate the bytecode.
2. JDK must be installed before completion.
3. Java program can be compiled by typing javac <filename>.java
4. It will create a file called <filename>.class containing the bytecode.

(3) Executing the program:

1. Java program once compiled can be run at any system.
2. Java program can be execute by typing Java <filename>

JVM(Java Virtual Machine)



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 closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM(**Java**

J.N.N College of Engg.

Virtual Machine). A Java virtual machine (JVM) is a virtual machine that can execute Java bytecode. It is the code execution component of the Java software platform.

More Examples:

Java program with multiple lines:

Example:

```
import java.lang.math;
class squreroot
{
        public static void main(String args[])
        {
                double x = 5;
                double y;
                y = Math.sqrt(x);
                System.out.println("Y = " + y);
        }
}
```

Java Program structure: Java program structure contains six stages. They are:

(1) Documentation section: The documentation section contains a set of comment lines describing about the program.

(2) Package statement: The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the class defined here belong to the package.

(3) Import statements: Import statements instruct the compiler to load the specific class belongs to the mentioned package.

(4) Interface statements: An interface is like a class but includes a group of method deceleration. This is an optional statement.

(5) Class definition: A Java program may contain multiple class definition The class are used to map the real world object.

(6) Main method class: The main method creates objects of various classes and establish communication between them. On reaching to the end of main the program terminates and the control goes back to operating system.

Java command line arguments: Command line arguments are the parameters that are supplied to the application program at the time when they are invoked. The main() method of Java program will take the command line arguments as the parameter of the args[ ] variable which is a string array.

Example:

```
Class Comlinetest
{
```

J.N.N. College of Engg.

```
        public static void main(String args[ ] )
        {
                int count, n = 0;
                string str;
                count = args.length;
                System.out.println ( " Number of arguments :" + count);
                While ( n < count )
                {
                        str = args[ n ];
                        n = n + 1;
                        System.out.println( n + " : " + str);
                }
        }
}
```

Run/Calling the program:

```
javac Comlinetest.java
java Comlinetest Java c cpp fortran
```

Output:

```
1 : Java
2 : c
3 : cpp
4 : fortran
```

Java API:

Java standard library includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

(a) Language support Package.
(b) Utilities packages.
(c) Input/output packages
(d) Networking packages
(e) AWT packages.
(f) Applet packages.

Java Tokens

Constants: Constants in Java refers to fixed value that do not change during the execution of program. Java supported constants are given below:

(1) **Integer constants:** An integer constant refers to a sequence of digits. There are three types of integer namely decimal integer, octal integer and hexadecimal integer. For example: 123 -321
(2) **Real constants:** Any real world number with a decimal point is known as real constants. For example : 0.0064 12e-2 etc.
(3) **Single character constants:** A single character constant contains a single character enclosed with in a pair of single quotes. For ex: 'm' '5'
(4) **String constants :** A string constant is a sequence of character enclosed with double quotes. For ex: "hello" "java" etc.

J.N.N. College of Engg.

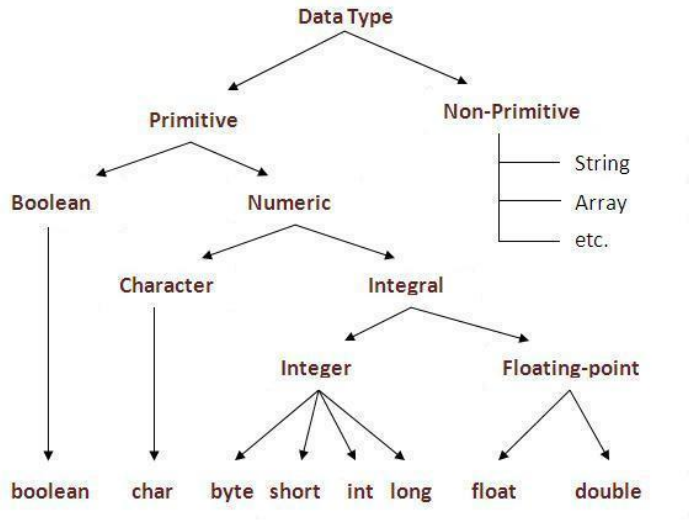
(5) **Backslash character constants:** Java supports some backslash constants those are used in output methods. They are :

1. \b	Backspace
2. \f	Form feed
3. \n	New Line
4. \r	Carriage return.
5. \t	Horizontal tab.
6. \'	Single quotes.
7. \"	Double quotes
8. \\	Back slash

Data Types in Java:

In java, data types are classified into two catagories :

1. Primitive Data type
2. Non-Primitive Data type



Data Type	Default Value	Default size

boolean	False	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Integers Type: Java provides four types of Integers. They are byte, sort, Int, long. All these are sign, positive or negative.

Byte: The smallest integer type is byte. This is a signed 8-bit type that has a range from -128 to 127. Bytes are useful for working with stream or data from a network or file. They are also useful for working with raw binary data. A byte variable is declared with the keyword "byte".

```
byte b, c;
```

Short: Short is a signed 16-bit type. It has a range from -32767 to 32767. This data type is most rarely used specially used in 16 bit computers. Short variables are declared using the keyword short.

```
short a, b;
```

int: The most commonly used Integer type is int. It is signed 32 bit type has a range from -2147483648 to 2147483648.

```
int a, b, c;
```

long: Long is a 64 bit type and useful in all those occasions where Int is not enough. The range of long is very large.

```
long a, b;
```

Floating point types: Floating point numbers are also known as real numbers are useful when evaluating a expression that requires fractional precision. The two floating-point data types are float and double.

float: The float type specifies a single precision value that uses 32-bit storage. Float keyword is used to declare a floating point variable.

```
float a, b;
```

double: Double DataTips is declared with **double** keyword and uses 64-bit value.

Characters: The Java data type to store characters is char. **char data type** of Java uses Unicode to represent characters. Unicode defines a fully international character set that can have all the characters of human language. Java char is 16-bit type. The range is 0 to 65536.

Boolean: Java has a simple type called **boolean** for logical values. It can have only one of two possible values. They are true or false.

Key Words: Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most statement contains an expression that contains the action carried out on data. The compiler recognizes the tokens

J. N. College of Engg.

for building up the expression and statements. Smallest individual units of programs are known as tokens. Java language includes five types of tokens. They are

(a) Reserved Keyword
(b) Identifiers
(c) Literals.
(d) Operators
(e) Separators.

(1) **Reserved keyword:** Java language has 60 words as reserved keywords. They implement specific feature of the language. The keywords combined with operators and separators according to syntax build the Java language.

(2) **Identifiers:** Identifiers are programmer-designed token used for naming classes methods variable, objects, labels etc. The rules for identifiers are

1. They can have alphabets, digits, dollar sign and underscores.
2. They must not begin with digit.
3. Uppercase and lower case letters are distinct.
4. They can be any lengths.
5. Name of all public method starts with lowercase.
6. In case of more than one word starts with uppercase in next word.
7. All private and local variables use only lowercase and underscore.
8. All classes and interfaces start with leading uppercases.
9. Constant identifier uses uppercase letters only.

(3) **Literals:** Literals in Java are sequence of characters that represents constant values to be stored in variables. Java language specifies five major types of Literals. They are:

1. Integer Literals.
2. Floating-point Literals.
3. Character Literals.
4. String Literals.
5. Boolean Literals.

(4) **Operators:** An operator is a symbol that takes one or more arguments and operates on them to produce an result.

(5) **Separators:** Separators are the symbols that indicates where group of code are divided and arranged. Some of the operators are:

1. Parenthases()
2. Braces{ }
3. Brackets [ ]
4. Semicolon ;
5. Comma ,
6. Period .

J.N. College of Engg.

Java character set: The smallest unit of Java language are its character set used to write Java tokens. This character are defined by unicode character set that tries to create character for a large number of character worldwide.

The Unicode is a 16-bit character coding system and currently supports 34,000 defined characters derived from 24 languages of worldwide.

Variables: A variable is an identifier that denotes a storage location used to store a data value. A variable may have different value in the different phase of the program. To declare one identifier as a variable there are certain rules. They are:

1. They must not begin with a digit.
2. Uppercase and lowercase are distinct.
3. It should not be a keyword.
4. White space is not allowed.

Declaring Variable: One variable should be declared before using. The syntax is

Data-type variblaname1, variablename2,. . . . . variablenameN;

Initializing a variable: A variable can be initialize in two ways. They are

(a) Initializing by Assignment statements.
(b) Initializing by Read statements.

Initializing by assignment statements: One variable can be initialize using assignment statements. The syntax is :

Variable-name = Value;

Initialization of this type can be done while declaration.

Initializing by read statements: Using read statements we can get the values in the variable.

Scope of Variable: Java variable is classified into three types. They are

(a) Instance Variable
(b) Local Variable
(c) Class Variable

J.N. College of Engg.

Instance Variable: Instance variable is created when objects are instantiated and therefore they are associated with the object. They take different values for each object.

Class Variable: Class variable is global to the class and belongs to the entire set of object that class creates. Only one memory location is created for each class variable.

Local Variable: Variable declared inside the method are known as local variables. Local variables are also can be declared with in program blocks. Program blocks can be nested. But the inner blocks cannot have same variable that the outer blocks are having.

Arrays in Java

Array which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Declaring Array Variables:

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

dataType[] arrayRefVar; or dataType arrayRefVar[];

Example:

The following code snippets are examples of this syntax:

int[] myList; or int myList[];

Creating Arrays:

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

arrayRefVar = new dataType[arraySize];

The above statement does two things:

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

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

dataType[] arrayRefVar = new dataType[arraySize];

J.N. College of Engg.

Alternatively you can create arrays as follows:

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

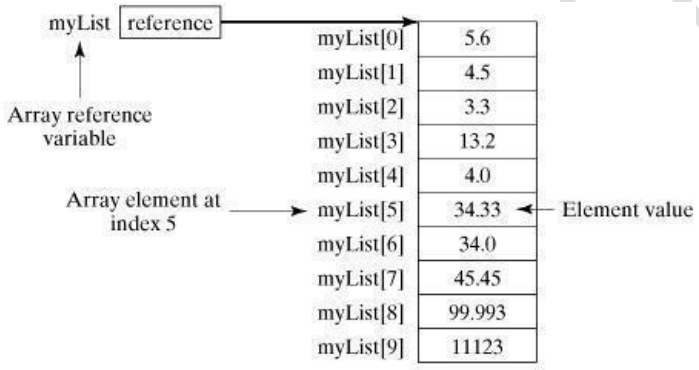
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:

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

Example:

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

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

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

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

This would produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

The foreach Loop

JDK 1.5 introduced a new for loop known as for-each loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example:

The following code displays all the elements in the array myList:

```
public class TestArray {

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

     // Print all the array elements

     for (double element: myList)
     {
       System.out.println(element);
     }
  }
}
```

J. N. College of Engg.

This would produce the following result:

1.9
2.9
3.4
3.5

Type Casting: It is often necessary to store a value of one type into the variable of another type. In these situations the value that to be stored should be casted to destination type. Type casting can be done in two ways.

Type Casting

Assigning a value of one type to a variable of another type is known as **Type Casting**.

Example :

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

- Widening Casting(Implicit)

- Narrowing Casting(Explicitly done)

Widening or Automatic type converion

Automatic Type casting take place when,

- the two types are compatible

J.N.N. College of Engg.

- the target type is larger than the source type

Example :

```
public class Test
{
    public static void main(String[] args)
    {
      int i = 100;
      long l = i;       //no explicit type casting required
      float f = l; //no explicit type casting required
      System.out.println("Int value "+i);
      System.out.println("Long value "+l);
      System.out.println("Float value "+f);
    }

}
```

Output :

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

Example :

```
public class Test
{
    public static void main(String[] args)
    {
      double d = 100.04;

      long l = (long)d; //explicit type casting required int
      i = (int)l; //explicit type casting required

      System.out.println("Double value "+d);
      System.out.println("Long value "+l);
      System.out.println("Int value "+i);

    }

}
```

Output :

```
Double value 100.04
Long value 100
Int value 100
```

J.N.N. College of Engg.

Java operators:

Java operators can be categorized into following ways:

(1) Arithmetic operator
(2) Relational operator
(3) Logical operator
(4) Assignment operator
(5) Increment and decrement operator
(6) Conditional operator
(7) Bitwise operator
(8) Special operator.

Arithmetic operator: The Java arithmetic operators are:

+	: Addition
-	: Subtraction
•	: Multiplication
/	: Division
%	: Remainder

Relational Operator:

<	: Is less then
<=	: Is less then or equals to
➢	: Is greater then
>=	: Is grater then or equals to
==	: Is equals to
!=	: Is not equal to

Logical Operators:

&&	: Logical AND
||	: Logical OR
!	: Logical NOT

Assignment Operator:

+=	: Plus and assign to
-=	: Minus and assign to
*=	: Multiply and assign to.
/=	: Divide and assign to.
%=	: Mod and assign to.
=	: Simple assign to.

Increment and decrement operator:

++	: Increment by One {Pre/Post)
--	: Decrement by one (pre/post)

Conditional Operator: Conditional operator is also known as ternary operator. The conditional operator is :

Exp1 ? exp2 : exp3

Bitwise Operator: Bit wise operator manipulates the data at Bit level. These operators are used for tasting the bits. The bit wise operators are:

& : Bitwise AND

J.N.N. College of Engg.

!	: Bitwise OR
^	: Bitwise exclusive OR
~	: One's Complement.
<<	: Shift left.
>>	: Shift Right.
>>>	: Shift right with zero fill

Example:

Bitwise operator works on bits and performs bit-by-bit operation. **Assume if a = 60; and b = 13;** now in binary format they will be as follows:

```
a = 0011  1100
b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011  1101
a^b = 0011  0001
 ~a  = 1100  0011
```

>>	**Binary Left Shift Operator.** The left operands value is moved left by the number of bits specified by the right operand.	
<<	**Binary Right Shift Operator**. The left operands value is moved right by the number of bits specified by the right operand.	
>>>	**Shift right zero fill Operator**. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	

J.N.N. College of Engg.

Special Operator:

Instanceof operator: The instanceof operator is a object refrence operator that returns true if the object on the right hand side is an instance of the class given in the left hand side. This operator allows us to determine whether the object belongs to the particular class or not.

Person instanceof student

The expression is true if the person is a instance of class student.

Dot operator: The dot(.) operator is used to access the instance variable or method of class object.

Example Programs

```
class arithmeticop
{
	public static void main(String args[])
	{
		float a=20.5f;
		float b=6.4f;
		System.out.println("a = " + a);
		System.out.println("b = " + b );
		System.out.println("a + b = " + (a+b));
	}
}
```

A << 2 will give 240 which is 1111 0000
A >> 2 will give 15 which is 1111
A >>>2 will give 15 which is 0000 1111

J.N.N. College of Engg.

```
class Bitlogic
{
        public static void main(String args[])
        {
String binary[] = {"0000","0001","0010","0011","0100","0101","0110","0111","1000","1001","1010",
                                "1011","1100","1101","1110","1111"};
                int a = 3;
                int b = 6;
                int c = a | b;
                int d = a & b;
                int e = a ^ b;
                int f = (~a&b)|(a & ~b);
                System.out.println("a or b :"+binary[c]);
                System.out.println("a and b : "+binary[d]);
                System.out.println("a xor b :  "+binary[e]);
                System.out.println("(~a&b)|(a & ~b) :  "+binary[f]);
        }
}
```

Control Statements

Decision making statements:

1.Simple If statement:

The general form of single if statement is :

```
If ( test expression)
{
        statement-Block;
}
statement-Blocks;
```

2. If- Else statement:

The general form of if-else statement is

```
If ( test expression)
{
        statement-block1;
}
else
{
        statement-block2
}
```

3. Else-if statement:

The general form of else-if statement is:

```
If ( test condition)
{
        statement-block1;
}
else if(test expression2)
{
        statement-block2;
}
else
{
```

J. N. College of Engg.

```
                statement block3;
        }
```

4. Nested if – else statement:

The general form of nested if-else statement is:

```
        If ( test condition)
        {
                if ( test condition)
                {
                        statement block1;
                }
                else
                {
                        statement block2;
                }
        }
        else
        {
                statement block 3
        }
```

5. The switch statements:

The general form of switch statement is:

```
        Switch ( expression)
        {
                case value-1:
                        block-1;
                        break;
                case value-2:
                        block-2;
                        break;
                .....
                default:
                        default block;
                        break;
        }
```

Loops In Java: In looping a sequence of statements are executed until a number of time or until some condition for the loop is being satisfied. Any looping process includes following four steps:

(1) Setting an initialization for the counter.
(2) Execution of the statement in the loop
(3) Test the specified condition for the loop.
(4) Incrementing the counter.

Java includes three loops. They are:

(1) While loop:

The general structure of a while loop is:

```
        Initialization
        While (test condition)
        {
                body of the loop
        }
```

J.N.N. College of Engg.

(2) Do loop:

The general structure of a do loop is :

```
Initialization
do
{
        Body of the loop;
}
while ( test condition);
```

(3) For loop :

The general structure of for loop is:

```
For ( Initialization ; Test condition ; Increment)
{
        body of the loop;
}
```

More about Loops:

Break Statement: Using "break" statement we can jump out from a loop. The "break" statement will cause termination of the loop.

Continue statement: The "continue" statement will cause skipping some part of the loop.

Labeled loops: We can put a label for the loop. The label can be any Java recognized keyword. The process of giving label is

```
Label-name : while (condition)
                {
                    Body ;
                }
```

Example Program for label break statement

```
class BreakTest
{
    public static void main(String args[])
    {
        boolean t= true;
        first:
        {
            second:
            {
                third:
                {
                    System.out.println("Third stage");
                    if(t)
                        break second;
                        System.out.println("Third stage complete");
                }
                System.out.println("Second stage");
            }
            System.out.println("First stage");
        }
    }
}
```

J. N. College of Engg.

Example Program for continue statement

```
class ContinueTest
{
	public static void main(String args[])
	{
		outer:  for(int i=0;i<10;i++)
			{
				for(int j = 0; j<10;j++)
				{
					if(j>i)
					{
						System.out.println("\n");
						continue outer;
					}
					System.out.print("  "+(i*j));
				 }
			 }

	//System.out.println(" ");
	}
}
```

Example Program for return statement

```
class ReturnTest
{
	public static void main(String args[])
	{
		boolean t = true;
		System.out.println("Before the return");
		if(t) return;
		System.out.println("After return");
	}
}
```

Java Access Specifiers/Modifiers

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Public Access Modifier - public:

J.N.N. College of Engg.

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However if the public class we are trying to access is in a different package, then the public class still need to be imported.

Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Advantages of JAVA:

- It is an open source, so users do not have to struggle with heavy license fees each year.
- Platform independent.
- Java API's can easily be accessed by developers.
- Java perform supports garbage collection, so memory management is automatic.
- Java always allocates objects on the stack.
- Java embraced the concept of exception specification.
- Multi-platform support language and support for web-services.
- Using JAVA we can develop dynamic web applications.
- It allows you to create modular programs and reusable codes.

J.N.N. College of Engg.

Questions

1. List & explain the characteristics features of java language.
2. Briefly discuss about the java development tool kit.
3. Explain the process of building and running java application program .
4. Explain the following: a)JVM b)Type casting.
5. Class Example{

```
public static void main(String s[]) {
        int a;
        for(a=0;a<3;a++){
        int b=-1;
        System.out.println(" "+b);
         b=50;
        System.out.println(" "+b);

        }
}}
```

What is the output of the above code? If you insert another 'int b' outside the for loop what is the output.

6. With example explain the working of >> and >>>.
7. What is the default package & default class in java?
8. Write a program to calculate the average among the elements {4, 5, 7, 8}, using for each in java. How for each is different from for loop?
9. Briefly explain any six key consideration used for designing JAVA language.
10. Discuss three OOP principles.
11. How compile once and run anywhere is implemented in JAVA language?
12. List down various operators available in JAVA language.
13. What is polymorphism?explain with an example.
14. Explain the different access specifiers in java, with examples.
15. a)int num,den;

```
if(den!=0&&num|den>2)
{
}
```

b)int num,den;

```
if(den!=0&num|den==2)
{
}
```

Compare & explain the above two snippets.

16. Write a note on object instantiation.
17. Explain type casting in JAVA
18. With a program explain break, continue and return keyword in java
19. List the characteristics of a constructor. Implement a C++ program to define a suitable parameterized constructor with default values for the class distance with data members feet and inches.
20. What is parameterized constructor. Explain different ways of passing parameters to the constructor.

J.N.N. College of Engg.

MODULE-3: CLASSES, INHERITANCE AND EXCEPTION HANDLING

Syllabus:

Classes: Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.

Inheritance: inheritance basics, using super, creating multi level hierarchy, method overriding.

Exception handling: Exception handling in Java.

1. CLASSES:

Definition

A class is a template for an object, and defines the data fields and methods of the object. The class methods provide access to manipulate the data fields. The "data fields" of an object are often called "instance variables."

Example Program:

```
Program to calculate Area of Rectangle
class Rectangle
{
        int length;             //Data Member or instance Variables
        int width;
        void getdata(int x,int y)       //Method
        {
                length=x;
                width=y;
        }
        int rectArea()                  //Method
        {
                return(length*width);
        }
}

class RectangleArea
{
    public static void main(String args[])
    {
          Rectangle rect1 =new Rectangle(); //object creation
              rect1.getdata(10,20); //calling methods using object with dot(.)
         int area1=rect1.rectArea();
        System.out.println("Area1="+area1);
    }
}
```

- ✓ After defining a class, it can be used to create objects by instantiating the class. Each object occupies some memory to hold its instance variables (i.e. its state).
- ✓ After an object is created, it can be used to get the desired functionality together with its class.

Creating instance of a class/Declaring objects:

```
Rectangle rect1 =new Rectangle()
Rectangle rect2=new Rectangle()
```

- ✓ The above two statements declares an **object rect1 and rect2 is of type Rectangle class using new operator** , this operator dynamically allocates memory for an object and returns a refernce to it.in java all class objects must be dynamically allocated.

We can also declare the object like this:

```
Rectangle rect1;          // declare reference to object.
rect1 =new Rectangle()    // allocate memory in the Rectangle object.
```

The Constructors:

- ✓ A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.
- ✓ Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

J.N.N. College of E...

- ✓ All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass
{
     int x;

   // Following is the constructor
   MyClass()
  {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

```
class ConsDemo
{
   public static void main(String args[])
  {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

Parameterized Constructor:

- ✓ Most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parentheses after the constructor's name.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass
{
   int x;

   // Following is the Parameterized constructor
   MyClass(int i )
   {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

```
class ConsDemo
{
   public static void main(String args[])
 {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce following result:

```
10 20
```

static keyword

The **static keyword** is used in java mainly for memory management. We may apply static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

static variable

Example Program without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
class Counter
{
        int count=0;//will get memory when instance is created
        Counter()
        {
                count++;
                System.out.println(count);
        }
}

Class MyPgm
{
        public static void main(String args[])
        {
                        Counter c1=new Counter();
                        Counter c2=new Counter();
                        Counter c3=new Counter();
                }
        }
```

Output: 1
1
1

Example Program with static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

J.N. College of Engg.

```
class Counter
{
        static int count=0;//will get memory only once and retain its value

        Counter()
        {
                count++;
                System.out.println(count);
        }
 }
Class MyPgm
{
        public static void main(String args[])
        {
                Counter c1=new Counter();
                Counter c2=new Counter();
                Counter c3=new Counter();
        }
}
```

Output:1
2
3

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 object of a class.
- ✓ A static method can be invoked without the need for creating an instance of a class.
- ✓ static method can access static data member and can change the value of it.

//Program to get cube of a given number by static method

```
class Calculate
{
        static int cube(int x)
        {
                return x*x*x;
        }
}
```

J.N.N. College of Engg.

```
Class MyPgm
{
        public static void main(String args[])
        {
                //calling a method directly with class (without creation of object)
                int result=Calculate.cube(5);
                System.out.println(result);
        }
}
```

Output:125

this keyword

- ✓ this keyword can be used to refer current class instance variable.
- ✓ If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

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

```
class student
{
        int id;
        String name;

        student(int id,String name)
        {
                id = id;
                name = name;
        }
        void display()
        {
                System.out.println(id+" "+name);
        }
}
Class MyPgm
{
        public static void main(String args[])
        {
                student s1 = new student(111,"Anoop");
```

J. N. College of Engg.

```
                student s2 = new student(321,"Arayan");
                s1.display();
                s2.display();
        }
}
```

Output: 0 null
0 null

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
//example of this keyword
class Student
{
        int id;
        String  name;

        student(int id,String name)
        {
                this.id = id;
                this.name = name;
        }
         void display()
        {
                System.out.println(id+" "+name);
        }
}
   Class MyPgm
   {
        public static void main(String args[])
        {
                Student s1 = new Student(111,"Anoop");
                Student s2 = new Student(222,"Aryan");
                s1.display();
                s2.display();
        }
   }
```

Output111 Anoop
222 Aryan

J. N. N. College of Engg.

Inner class

- ✓ It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.
- ✓ One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.

Example of Inner class

```
class Outer
{
        public void display()
        {
                Inner in=new Inner();
                in.show();
        }

        class Inner
        {
                public void show()
                {
                        System.out.println("Inside inner");
                }
        }
}

class Test
{
        public static void main(String[] args)
        {
                Outer ot=new Outer();
                ot.display();
        }
}
```

Output:

Inside inner

J.N.N. College of Engg.

Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memories occupied by the object are released. This technique is called **Garbage Collection**. This is accomplished by the JVM.

Can the Garbage Collection be forced explicitly?

No, the Garbage Collection cannot be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method. But this does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

---

finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used. **finalize()** method is called by garbage collection thread before collecting object. It's the last chance for any object to perform cleanup utility.

Signature of **finalize()** method

```
protected void finalize()
{
 //finalize-code
}
```

J. N. College of Engg.

gc() Method

gc() method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only requests the JVM for garbage collection. This method is present in **System** and **Runtime** class.

Example for gc() method

```
public class Test
{

   public static void main(String[] args)
   {
      Test t = new Test();
      t=null;
      System.gc();
   }
   public void finalize()
   {
      System.out.println("Garbage Collected");
   }
}
```

Output :

Garbage Collected

Inheritance:

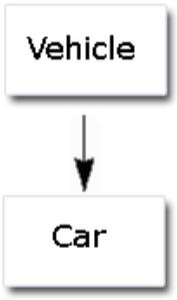
- ✓ As the name suggests, inheritance means to take something that is already made. It is one of the most important features of Object Oriented Programming. It is the concept that is used for **reusability** purpose.
- ✓ Inheritance is the mechanism through which we can derive classes from other classes.
- ✓ The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class.
- ✓ To derive a class in java the keyword **extends** is used. The following kinds of inheritance are there in java.

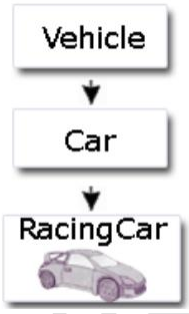
J. N. N. College of Engg.

Types of Inheritance

1. **Single level/Simple Inheritance**
2. **Multilevel Inheritance**
3. **Multiple Inheritance (Java doesn't support Multiple inheritance but we can achieve this through the concept of Interface.)**

Pictorial Representation of Simple and Multilevel Inheritance





Simple Inheritance **Multilevel Inheritance**

Single level/Simple Inheritance

- ✓ When a subclass is derived simply from its parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and its parent class. It is also called single inheritance or one level inheritance.

Example

```
class A
{
      int x;
      int y;
      int get(int p, int q)
     {
           x=p;
           y=q;
           return(0);
      }
```

```
    void Show()
     {
          System.out.println(x);
     }
  }
  class B extends A
  {
        public static void main(String args[])
        {
          A a = new A();
            a.get(5,6);
            a.Show();
        }
     void display()
     {
          System.out.println("y");
    //inherited "y" from class A
     }
}
```

- ✓ The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class A
{

}

class B extends A          //B is a subclass of super class A.

{

}
```

J.N.N. College of Engg.

Multilevel Inheritance

- ✓ When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance.
- ✓ The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class.
- ✓ Multilevel inheritance can go up to any number of level.

```
class A
{
      int x;
      int y;
      int get(int p, int q)
      {
          x=p;
          y=q;
          return(0);
      }
      void Show()
      {
          System.out.println(x);
      }
}
```

```
class B extends A
{
      void Showb()
      {
          System.out.println("B");
      }
}

class C extends B
{
      void display()
      {
           System.out.println("C");
      }
      public static void main(String args[])
      {
           A a = new A();
             a.get(5,6);
```

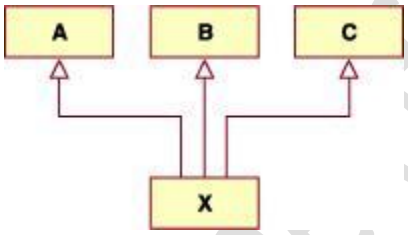
```
                a.Show();
        }
}
```

OUTPUT

```
5
```

Multiple Inheritance

- ✓ The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance. Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface.



- ✓ Here you can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

Java does not support multiple Inheritance

In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.

super keyword

- ✓ The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java.
- ✓ The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class.

Example: Suppose class A is the super class that has two instance variables as int a and float b. class B is the subclass that also contains its own data members

J N N College of Engg

named a and b. then we can access the super class (class A) variables a and b inside the subclass class B just by calling the following command.

super.member;

- ✓ Here member can either be an instance variable or a method. This form of super most useful to handle situations where the local members of a subclass hides the members of a super class having the same name. The following example clarifies all the confusions.

Example:

```
class A
{
        int a;
        float b;
        void Show()
        {
                System.out.println("b in super class: " + b);
        }

}
class B extends A
{
        int a;
        float b;
        B( int p, float q)
        {
             a = p;
            super.b = q;
        }
        void Show()
        {
            super.Show();
            System.out.println("b in super class: " + super.b);
            System.out.println("a in sub class:    " + a);
       }
}

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

```
                    B subobj = new B(1, 5);
                       subobj.Show();
        }
}
```

OUTPUT

```
b in super class: 5.0
b in super class: 5.0
a in sub class:   1
```

Use of super to call super class constructor: The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.

super(param-list);

- ✓ Here parameter list is the list of the parameter requires by the constructor in the super class. super must be the first statement executed inside a super class constructor. If we want to call the default constructor then we pass the empty parameter list. The following program illustrates the use of the super keyword to call a super class constructor.

Example:

```
class A
{
        int a;
        int b;
        int c;
        A(int p, int q, int r)
    {
           a=p;
           b=q;
           c=r;
     }
}

class B extends A
{
           int d;
           B(int l, int m, int n, int o)
           {
```

College of Engg

```
            super(l,m,n);
            d=o;
        }

        void Show()
        {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println("c = " + c);
            System.out.println("d = " + d);
        }
}
class Mypgm
{
        public static void main(String args[])
        {
            B b = new B(4,3,8,7);
            b.Show();
        }
 }
```

OUTPUT

a = 4
b = 3
c = 8
d = 7

Method Overriding

- ✓ Method overriding in java means a subclass method overriding a super class method.
- ✓ Superclass method should be non-static. Subclass uses extends keyword to extend the super class. In the example **class B** is the sub class and **class A** is the super class. **In overriding methods of both subclass and superclass possess same signatures**. Overriding is used in modifying the methods of the super class. In overriding return types and constructor parameters of methods should match.

Below example illustrates method overriding in java.

J N N College of Engg.

Example:

```
class A
{
            int i;
           A(int a, int b)
           {
                  i = a+b;
           }
           void add()
           {
                System.out.println("Sum of a and b is: " + i);
           }
}
class B extends A
{
            int j;
            B(int a, int b, int c)
            {
                 super(a, b);
                 j = a+b+c;
            }
        void add()
        {
              super.add();
              System.out.println("Sum of a, b and c is: " + j);
        }
}
class MethodOverriding
{
          public static void main(String args[])
          {
                  B b = new B(10, 20, 30);
                    b.add();
           }
}
```

OUTPUT

Sum of a and b is: 30
Sum of a, b and c is: 60

J. N. N. College of Engg.

Method Overloading

- ✓ Two or more methods have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different is called **method overloading**. An example of the method overloading is given below:

Example:

```
class MethodOverloading
{
          int add( int a,int b)
          {
               return(a+b);
          }
          float add(float a,float b)
          {
               return(a+b);
          }
          double add( int a, double b,double c)
          {
              return(a+b+c);
          }
}
class MainClass
{
          public static void main( String arr[] )
          {
                MethodOverloading mobj = new MethodOverloading
                (); System.out.println(mobj.add(50,60));
                System.out.println(mobj.add(3.5f,2.5f));
                System.out.println(mobj.add(10,30.5,10.5));
          }
}
OUTPUT
110
6.0
51.0
```

Abstract Class

- ✓ **abstract keyword** is used to make a class abstract.
- ✓ Abstract class can't be instantiated with new operator.
- ✓ We can use abstract keyword to create an abstract method; an abstract method doesn't have body.
- ✓ If classes have abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- ✓ Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

Example Program:

```
abstract Class AreaPgm
{
        double dim1,dim2;
        AreaPgm(double x,double y)
        {
                dim1=x;
                dim2=y;
        }
        abstract double area();
}
class rectangle extends AreaPgm
{
        rectangle(double a,double b)
        {
                super(a,b);
        }
        double area()
        {
                System.out.println("Rectangle Area");
                return dim1*dim2;
        }
}
class triangle extends figure
{
        triangle(double x,double y)
        {
                super(x,y);

        }
        double area()
        {
                System.out.println("Traingle Area");
```

J. N. College of Engg.

```
            return dim1*dim2/2;
        }
}
class MyPgm
{
        public static void main(String args[])
        {

        AreaPgm a=new AreaPgm(10,10); // error, AreaPgm is a abstract class.

                rectangle r=new rectangle(10,5);
                System.out.println("Area="+r.area());

                triangle t=new triangle(10,8);
                AreaPgm ar;
                ar=obj;
                System.out.println("Area="+ar.area());
        }
}
```

final Keyword In Java

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) final variable: If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example: There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike
{
         final int speedlimit=90;//final variable
        void run()
        {
                 speedlimit=400;
         }
}
```

J.N.N. College of Engg.

```
Class MyPgm
{
        public static void main(String args[])
        {
                Bike obj=new  Bike();
               obj.run();
        }
}
```

Output:Compile Time Error

2) final method: If you make any method as final, you cannot override it.

Example:

```
class Bike
{
        final void run()
        {
                System.out.println("running");
        }
}
class Honda extends Bike
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}

Class MyPgm
{
        public static void main(String args[])
        {
                Honda honda= new Honda();
                honda.run();
        }
}
```

Output:Compile Time Error

J.N.N. College of Engg.

3) final class: If you make any class as final, you cannot extend it.

Example:

```
final class Bike
{

}

class Honda extends Bike
{
          void run()
          {
                System.out.println("running safely with 50kmph");
           }
}

Class MyPgm
{

          public static void main(String args[])
         {
                  Honda honda= new Honda();
                        honda.run();
         }
}
```

Output:Compile Time Error

J.N.N. College of Engg.

Exception handling:

Introduction

An **Exception**, It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions. The abnormal event can be an error in the program.

Errors in a java program are categorized into two groups:

1. **Compile-time errors** occur when you do not follow the syntax of a programming language.
2. **Run-time errors** occur during the execution of a program.

Concepts of Exceptions

An exception is a run-time error that occurs during the exception of a java program.

Example: If you divide a number by zero or open a file that does not exist, an exception is raised.

In java, exceptions can be handled either by the java run-time system or by a user-defined code. When a run-time error occurs, an exception is thrown.

The unexpected situations that may occur during program execution are:

- Running out of memory
- Resource allocation errors
- Inability to find files
- Problems in network connectivity

Exception handling techniques:

Java exception handling is managed via five keywords they are:

1. try:
2. catch.
3. throw.
4. throws.
5. finally.

Exception handling Statement Syntax

Exceptions are handled using a try-catch-finally construct, which has the Syntax.

```
try
{
  <code>
}
catch (<exception type1> <parameter1>)
{
  // 0 or more<statements>

}
finally
{
  // finally block<statements>
}
```

1. try Block: The java code that you think may produce an exception is placed within a try block for a suitable catch block to handle the error.

If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error.

Again if the matching catch handler is not found execution proceeds with the finally block and the default exception handler throws an exception.

2. catch Block: Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed (Though the catch block throws an exception).

3. finally Block: A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Generally finally block is used for freeing resources, cleaning up, closing connections etc.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest
{
          public static void main(String args[])
         {
                 try
                {
                          int a[] = new int[2];
                          System.out.println("Access element three :" + a[3]);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                          System.out.println("Exception thrown  :" + e);
                }
                System.out.println("Out of the block");
       }
}
```

This would produce following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
  // code
}
catch(ExceptionType1 e1)
{
  //Catch block
}
catch(ExceptionType2 e2)
{
  //Catch block
}
catch(ExceptionType3 e3)
{
  //Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try.

Example: Here is code segment showing how to use multiple try/catch statements.

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

                try
                {
                        int a=args.length;
                        System.out.println("a="+a);
                         int b=50/a;
                         int c[]={1}
                }
                catch (ArithmeticException e)
                {
                      System.out.println ("Division by zero");
                }
```

```
                catch (ArrayIndexOutOfBoundsException e)
                {
                        System.out.println (" array index out of bound");
                }
        }
}
```

OUTPUT
Division by zero
array index out of bound

Nested try Statements

- ✓ Just like the multiple catch blocks, we can also have multiple try blocks. These try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-catch block within another try-block. The program structure for nested try statement is:

Syntax

```
try
{
   // statements
   // statements

   try
   {
      // statements
      // statements
   }
   catch (<exception_two> obj)
   {
      // statements
   }

   // statements
   // statements
}
catch (<exception_two> obj)
{
   // statements
}
```

- ✓ Consider the following example in which you are accepting two numbers from the command line. After that, the command line arguments, which are in the string format, are converted to integers.
- ✓ If the numbers were not received properly in a number format, then during the conversion a NumberFormatException is raised otherwise the control goes to the next try block. Inside this second try-catch block the first number is divided by the second number, and during the calculation if there is any arithmetic error, it is caught by the inner catch block.

Example

```
class Nested_Try
{
   public static void main (String args [ ] )
   {
      try
       {
            int a = Integer.parseInt (args [0]);
            int b = Integer.parseInt (args [1]);
            int quot = 0;

            try
            {
               quot = a / b;
               System.out.println(quot);
            }
            catch (ArithmeticException e)
            {
               System.out.println("divide by zero");
            }
       }
       catch (NumberFormatException e)
       {
          System.out.println ("Incorrect argument type");
       }
   }
}
```

The output of the program is: If the arguments are entered properly in the command prompt like:

OUTPUT

```
java Nested_Try 2 4 6
 4
```

If the argument contains a string than the number:

OUTPUT

```
java  Nested_Try 2 4  aa
 Incorrect argument type
```

If the second argument is entered zero:

OUTPUT

```
java Nested_Try 2 4 0
 divide by zero
```

throw Keyword

- ✓ **throw keyword** is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown.
- ✓ Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

Syntax : **throw** *ThrowableInstance*

Creating Instance of Throwable class

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with **new** operator.

new NullPointerException("test");

This constructs an instance of NullPointerException with name test.

Example demonstrating throw Keyword

```
class Test
{
        static void avg()
        {
                try
                {
                        throw new ArithmeticException("demo");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception caught");
                }
        }
        public static void main(String args[])
        {
                avg();
        }
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

---

throws Keyword

- ✓ Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws** keyword.

Syntax :

```
type method_name(parameter_list) throws exception_list
{
        //definition of method
}
```

NOTE : It is necessary for all exceptions, except the exceptions of type **Error** and **RuntimeException**, or any of their subclass.

J.N.N. College of Engg.

Example demonstrating throws Keyword

```
class Test
{
        static void check() throws
        ArithmeticException {
                System.out.println("Inside check function");
                throw new ArithmeticException("demo");
        }

        public static void main(String args[])
        {
                try
                {
                        check();
                }
                catch(ArithmeticException e)
                {
                        System.out.println("caught" + e);
                }
        }
}
```

finally

- ✓ The finally clause is written with the try-catch statement. It is guaranteed to be executed after a catch block or before the method quits.

Syntax

```
try
{
   // statements
}
```

catch (<exception> obj)

```
{
   // statements
}
finally
{
   //statements
}
```

N. N. College of Engg.

- ✓ Take a look at the following example which has a catch and a finally block. The catch block catches the ArithmeticException which occurs for arithmetic error like divide-by-zero. After executing the catch block the finally is also executed and you get the output for both the blocks.

Example:

```
class Finally_Block
{
   static void division ( )
   {
      try
      {
         int num = 34, den = 0;
         int quot = num / den;
      }
      catch(ArithmeticException e)
      {
         System.out.println ("Divide by zero");
      }
      finally
      {
         System.out.println ("In the finally block");
      }
   }
class Mypgm
{
   public static void main(String args[])
   {
         Finally_Block   f=new Finally_Block();
         f.division ( );
    }
}
```

OUTPUT

```
Divide by zero
In the finally block
```

Java's Built in Exceptions

Java defines several exception classes inside the standard package **java.lang**.

- ✓ The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java **Unchecked RuntimeException**.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of **Java Checked Exceptions** Defined in java.lang.

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating your own Exception Subclasses

- ✓ Here you can also define your own exception classes by extending **Exception**. These exception can represents specific runtime condition of course you will have to throw them yourself, but once thrown they will behave just like ordinary exceptions.
- ✓ When you define your own exception classes, choose the ancestor carefully. Most custom exception will be part of the official design and thus checked, meaning that they extend Exception but not RuntimeException.

Example: Throwing User defined Exception

```
public class MyException extends Exception
{
    String msg = "";
    int marks=50;
    public MyException()
    {
    }
    public MyException(String str)
    {
       super(str);
```

J.N.N. College of

```
    }
    public String toString()
    {
        if(marks <= 40)
        msg = "You have failed";
        if(marks > 40)
        msg = "You have Passed";

        return msg;

    }
}


 class test
 {
    public static void main(String args[])
       {
            test t = new test();
            t.dd();
       }
    public void add()
     {
          try
           {
              int i=0;
              if( i<40)
              throw new MyException();
           }
           catch(MyException ee1)
           {
              System.out.println("Result:"+ee1);
           }
     }
}
```

OUTPUT

Result: You have Passed

J. N. N. College of Engg.

Chained Exception

- ✓ Chained exceptions are the exceptions which occur one after another i.e. most of the time to response to an exception are given by an application by throwing another exception.
- ✓ Whenever in a program the first exception causes an another exception, that is termed as **Chained Exception**. Java provides new functionality for chaining exceptions.
- ✓ Exception chaining (also known as "nesting exception") is a technique for handling the exception, which occur one after another i.e. most of the time

  is given by an application to response to an exception by throwing another exception.
- Typically the second exception is caused by the first exception. Therefore chained exceptions help the programmer to know when one exception causes another.

The **constructors** that support chained exceptions in **Throwable** class are:

Throwable initCause(Throwable)

Throwable(Throwable)

Throwable(String, Throwable)

Throwable getCause()

J.N.N. College of Engg

Questions

1. Distinguish between Method overloading and Method overriding in JAVA, with suitable examples.
2. What is super? Explain the use of super with suitable example
3. Write a JAVA program to implement stack operations.
4. What is an Exception? Give an example for nested try statements.
5. Write a program in java to implement a stack that can hold 10 integer values.
6. What is mean by instance variable hiding ?how to overcome it?
7. Define exception .demonstrate the working of nested try blocks with suitable example?
8. Write short notes on

   i) Final class ii) abstract class
9. Write a java program to find the area and volume of a room. Use a base class rectangle with a constructor and a method for finding the area. Use its subclass room with a constructor that gets the value of length and breadth from the base class and has a method to find the volume. Create an object of the class room and obtain the area and volume.
10. Explain i) Instance variables ii) Class Variables iii) Local Variables
11. Distinguish between method overloading and method overriding? How does java decide the method to call?
12. Explain the following with example.

    i) Method overloading ii) Method overriding

J.N.N. College of Engg.

13. Write a java program to find the distance between two points whose coordinates are given. The coordinates can be 2-dimensional or 3-dimensional (for comparing the distance between 2D and a 3D point, the 3D point, the 3D x and y components must be divided by *z*). Demonstrate method overriding in this program.

14. With an example explain static keyword in java.
15. Why java is not support concept of multiple inheritance? Justify with an example program.
16. Write a short note on:
    1. this keyword
    2. super keyword
    3. final keyword
    4. abstract
17. Illustrate constructors with an example program

J.N.N. College of Engg.

MODULE-4: PACKAGES, INTERFACES AND MULTI THREADED PROGRAMMING

Syllabus:

Packages and Interfaces: Packages, Access Protection, Importing Packages, Interfaces.

Multi Threaded Programming: Multi Threaded Programming: What are threads? How to make the classes threadable ; Extending threads; Implementing runnable; Synchronization; Changing state of the thread; Bounded buffer problems, producer consumer problems.

Packages in JAVA

- ✓ A **java package** is a group of similar types of classes, interfaces and sub-packages.
- ✓ Package in java can be categorized in two form,
  - built-in package
  - and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple
{
        public static void main(String args[])
        {
                System.out.println("Welcome to package");
        }
}
```

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

*1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

J.N. College of Engg.

Example of package that import the packagename.*

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

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

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

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import

package.classname //save by A.java

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

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

Output:Hello

J. N. College of Engg.

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

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

//save by B.java
package mypack;
class B
{
        public static void main(String args[])

        {
                pack.A obj = new pack.A();//using fully qualified
                name obj.msg();
        }
}
```

Output:Hello

Access Modifiers/Specifiers

The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

1) private access modifier

The private access modifier is accessible only within class.

J. N. College of Engg.

2) default access modifier

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

4) public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Understanding all java access modifiers by a simple table.

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

J.N.N. College of Engg.

Interface in java

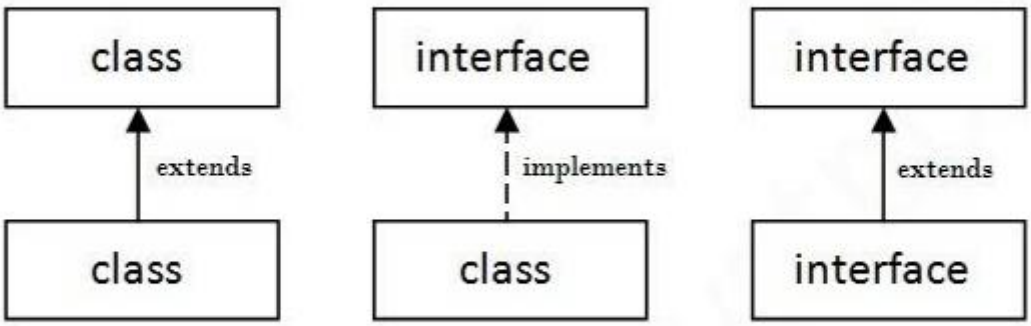
- ✓ An **interface in java** is a blueprint of a class. It has static final variables and abstract methods.
- ✓ The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface does not contain method body. It is used to achieve abstraction and multiple inheritance in Java.
- ✓ It cannot be instantiated just like abstract class.
- ✓ Interface fields are public, static and final by default, and methods are public and abstract.

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



College of Engg.

Example 1

In this example, Printable interface has only one method, its implementation is provided in the Pgm1 class.

```
interface printable
{
        void print();
}

class Pgm1 implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }
}

 class IntefacePgm1
{
        public static void main(String args[])
        {
                Pgm1 obj = new Pgm1 ();
                obj.print();
         }
}
```

Output:

```
Hello
```

Example 2

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

```
//Interface declaration: by first user

interface Drawable
{
        void draw();
}
```

J.N. College of Engg.

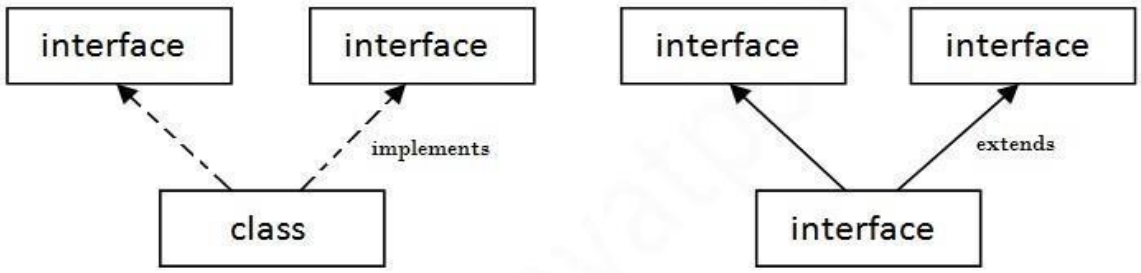
```
//Implementation: by second user

class Rectangle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing rectangle");
        }
}

class Circle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing circle");
        }
}

//Using interface: by third user
class TestInterface1
{
        public static void main(String args[])
        {
                //In real scenario, object is provided by method e.g. getDrawable()
                Drawable d=new Circle();

                d.draw();
        }
}
```

Output:

drawing circle

Multiple inheritance in Java by interface

- ✓ If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

J.N.N. College of Engg.

Example

```
interface Printable
{
        void print();
}

interface Showable
{
        void show();
}

class Pgm2 implements Printable,Showable
{
        public void print()
        {
                System.out.println("Hello");
        }

        public void show()
        {
                System.out.println("Welcome");
        }
}

Class InterfaceDemo
{
        public static void main(String args[])
        {
                Pgm2 obj = new Pgm2 ();
                obj.print();
                obj.show();
        }
}
```

Output:
Hello
Welcome

Multiple inheritance is not supported through class in java but it is possible by interface, why?

- ✓ As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity.
- ✓ But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

J. N. N. College of Engg.

Example

```
interface Printable
{
        void print();
}

interface Showable
{
        void print();
}

class InterfacePgm1 implements Printable,
Showable {
        public void print()
        {
                System.out.println("Hello");
        }
}
class InterfaceDemo
{
        public static void main(String args[])
        {
                InterfacePgm1 obj = new InterfacePgm1 ();
                obj.print();
         }
}
```

Output:
Hello

- ✓ As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

- ✓ A class implements interface but one interface extends another interface .

```
interface Printable
{
        void print();
}
interface  Showable extends Printable
{
        void show();
}
```

J.N. College of Engg.

```
class InterfacePgm2 implements Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public void show()
        {
                System.out.println("Welcome");
        }

Class InterfaceDemo2
{
        public static void main(String args[])
        {
                InterfacePgm2 obj = new InterfacePgm2 ();
                obj.print();
                obj.show();
         }
}
```

Output:
Hello
Welcome

Program to implement Stack

```
    public class StackDemo
    {
                private static final int capacity = 3;
                int arr[] = new int[capacity];
                int top = -1;

        public void push(int pushedElement)
        {
                if (top < capacity - 1)
                {
                        top++;
                        arr[top] = pushedElement;
            System.out.println("Element " + pushedElement  + " is pushed to Stack !")
        ;
                printElements();
                }
                else
                 {
                        System.out.println("Stack Overflow !");
                }
        }

        public void pop()
        {
                if (top >= 0)
```

J. N. N. College of Engg.

```
            {
                top--;
                System.out.println("Pop operation done !");
            }
            else
            {
                System.out.println("Stack Underflow !");
            }
    }
            public void printElements()
            {
                if (top >= 0)
                {
                    System.out.println("Elements in stack :");
                    for (int i = 0; i <= top; i++)
                    {
                        System.out.println(arr[i]);
                    }

                }
            }

class MyPgm
{

            public static void main(String[] args)
            {
                StackDemo stackDemo = new StackDemo();

                stackDemo.pop();
                stackDemo.push(23);
                stackDemo.push(2);
                stackDemo.push(73);
                stackDemo.push(21);
                stackDemo.pop();
                stackDemo.pop();
                stackDemo.pop();
                stackDemo.pop();
            }

}
```

Output

```
Stack Underflow !
Element 23 is pushed to Stack !
Elements in stack :
23
Element 2 is pushed to Stack !
Elements in stack :
23
2
Element 73 is pushed to Stack !
Elements in stack :
23
2
73
Stack Overflow !
Pop operation done !
Pop operation done !
Pop operation done !
Stack Underflow !
```

J. N. N. College of Engg.

What are threads?

- Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. Multitasking threads require less overhead than multitasking processes.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- The **Thread** class defines several methods that help manage threads (shown below)

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

- In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**.
- Next, the program displays information about the thread. The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed.
- Next, a loop counts down from five, pausing one second between each line.
- The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the
  delay period in milliseconds.

Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

These displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. The general form of sleep() is:

static void sleep(long milliseconds) throws InterruptedException

The number of milliseconds to suspend is specified in milliseconds. This method may throw an **InterruptedException**.

Creating a Thread

There are two different ways to create threads.

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

Thread defines several constructors. :

Thread(Runnable threadOb, String threadName)

In this constructor, threadOb is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**.
The **start( )** method is shown here:

void start( )

J.N.N. College of Engg.

Example:

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {


        System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
    }
  }
```

a new **Thread** object is created by the following statement:

t = new Thread(this, "Demo Thread");

Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. After calling **start( )**, **NewThread**'s constructor

returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish.

Output:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread Class

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

Example:

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

- The child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.
- Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

  public Thread(String threadName)

  Here, threadName specifies the name of the thread.

Creating Multiple Threads

For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
   try {
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive( ) and join( )

- To make main to finish last First, you can call **isAlive( )** on the thread.
- This method is defined by **Thread**, and its general form is shown here:

  final boolean isAlive( )
- The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.
- While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

  final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates.

Here is an improved version of the preceding example that uses **join( )** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}

class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");

     System.out.println("Thread One is alive: "
                        + ob1.t.isAlive());
     System.out.println("Thread Two is alive: "
                        + ob2.t.isAlive());
     System.out.println("Thread Three is alive: "
                        + ob3.t.isAlive());
     // wait for threads to finish
    try {
       System.out.println("Waiting for threads to finish.");
       ob1.t.join();
       ob2.t.join();
       ob3.t.join();
     } catch (InterruptedException e) {
       System.out.println("Main thread Interrupted");
     }

     System.out.println("Thread One is alive: "
                        + ob1.t.isAlive());
     System.out.println("Thread Two is alive: "
                        + ob2.t.isAlive());
     System.out.println("Thread Three is alive: "
                        + ob3.t.isAlive());

     System.out.println("Main thread exiting.");
   }
 }
```

Output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to **join( )** return, the threads have stopped executing.

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**. This is its general form:

  final void setPriority(int level)

  - Here, level specifies the new priority setting for the calling thread.

J.N.N. College of Engg.

- The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**.
- Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. T
- hese priorities are defined as **static final** variables within **Thread**.

- You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

The following example demonstrates two threads at different priorities, One thread is set two levels above the normal priority, as defined by **Thread.NORM_PRIORITY**, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread madeit through the loop is then displayed.

```
// Demonstrate thread priorities.
class clicker implements Runnable {
  long click = 0;
  Thread t;
  private volatile boolean running = true;

  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }

  public void run() {
    while (running) {
      click++;
    }
  }

  public void stop() {
    running = false;
  }

  public void start() {
    t.start();
  }
}

class HiLoPri {
  public static void main(String args[]) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start();
    hi.start();
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
       System.out.println("Main thread interrupted.");
    }

    lo.stop();
    hi.stop();

    // Wait for child threads to terminate.
    try {
      hi.t.join();
      lo.t.join();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    System.out.println("Low-priority thread: " + lo.click);
    System.out.println("High-priority thread: " + hi.click);
  }
}
```

Output:

The higher-priority thread got the majority of the CPU time.
Low-priority thread: 4408112
High-priority thread: 589626904

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.

Using Synchronized Methods

- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

The following program has three simple classes.

- The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

- The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string.

- Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

J.N.N. College of Engg.

Example:

```
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

    public Caller(Callme targ, String s) {
      target = targ;
      msg = s;
      t = new Thread(this);
      t.start();
    }

    public void run() {
      target.call(msg);
    }
  }
```

```
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]
]
]
```

As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings.

In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a **race condition**, because the three threads are racing each other to complete the method.

To fix the preceding program, you must serialize access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
synchronized void call(String msg) {
...
```

After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

The synchronized Statement

You simply put calls to the methods defined by this class inside a **synchronized** block. This is the general form of the **synchronized** statement:

```
synchronized(object) {
//      statements to be synchronized
}
```

- Here, object is a reference to the object being synchronized.
- Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;

  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}

class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

College of Engg.

Interthread Communication

Java supports interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException final
void notify( )
final void notifyAll( )
```

The following sample program that incorrectly implements a simple form of the producer/ consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}

  class Producer implements Runnable {
    Q q;

    Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
    }

    public void run() {
      int i = 0;

      while(true) {
        q.put(i++);
      }
    }
  }
```

```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}

class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here.

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while(!valueSet)
      try {
        wait();

     } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

      System.out.println("Got: " + n);
      valueSet = false;
      notify();
      return n;
  }
  synchronized void put(int n) {
    while(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}

class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells the **Consumer** that it should now remove it.

Here is some output from this program:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

J.N.N. College of Engg.

Questions

1. What is an interface? Write a program to illustrate multiple inheritance using interfaces.
2. Explain packages in java.
3. What are access specifiers? Explain with an example.
4. Define thread. Explain types of thread
5. Explain synchronized with different block
6. Explain multi threading . write a java program that create two thread with message.
7. Explain the role procedure & consumer problem
8. Explian briefly
   i) Isalive ( ) ii) Join ( ) iii) thread priority iv)suspending & stopping thread

J.N.N. College of Engg.

MODULE-5: EVENT HANDLING AND SWINGS

Syllabus:

Event Handling: Two event handling mechanisms; The delegation event model; Event classes; Sources of events; Event listener interfaces; Using the delegation event model; Adapter classes; Inner classes.

Swings: The origins of Swing; Two key Swing features; Components and Containers; The Swing Packages; A simple Swing Application; Create a Swing Applet; Jlabel and ImageIcon; JTextField;The Swing Buttons; JTabbedpane; JScrollPane; JList; JComboBox; JTable.

Event Handling

The Delegation Event Model

- Delegation event model defines standard and consistent mechanisms to generate and process events.
- A source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to
  listeners that want to receive them.

Events:

- An event is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources:

- A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method. General form:

  public void addTypeListener(TypeListener el)
- Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.
- A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

  public void removeTypeListener(TypeListener el)
- Here, Type is the name of the event, and el is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

Event Listeners

- A *listener* is an object that is notified when an event occurs.
- It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

J.N.College of Engg.

Event Classes

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-1 Main Event Classes in **java.awt.event**

ActionEvent Class

- An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- **ActionEvent** has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

- You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

  String getActionCommand( )
- The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

  int getModifiers( )
- The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here:

  long getWhen( )

AdjustmentEvent Class

- An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events.

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

- **AdjustmentEvent** constructor:

  AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)

  Here, *src* is a reference to the object that generated this event. The *id* specifies the event. The type of the adjustment is specified by *type*, and its associated data is *data*.

ComponentEvent Class

- A **ComponentEvent** is generated when the size, position, or visibility of a component is changed.
- There are four types of component events.

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

- **ComponentEvent** has this constructor:

  ComponentEvent(Component *src*, int *type*)

  Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.
- **ComponentEvent** is the superclass either directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.
- The **getComponent( )** method returns the component that generated the event. It is shown here:

  Component getComponent( )

ContainerEvent Class

- A **ContainerEvent** is generated when a component is added to or removed from a container.
- There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**.
- **ContainerEvent** is a subclass of **ComponentEvent** and has this constructor:

  ContainerEvent(Component *src*, int *type*, Component *comp*)

  Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*.

- You can obtain a reference to the container that generated this event by using the **getContainer( )** method, shown here:

  Container getContainer( )
- The **getChild( )** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

  Component getChild( )

FocusEvent Class

- A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.
- **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:

  FocusEvent(Component *src*, int *type*)

  FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)

  FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

  Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**.
- You can determine the other component by calling **getOppositeComponent( )**, shown here:

  Component getOppositeComponent( )

  The opposite component is returned.
- The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

  boolean isTemporary( )

  The method returns **true** if the change is temporary. Otherwise, it returns **false**.

InputEvent Class

- It is the superclass for component input events.
- Its subclasses are **KeyEvent** and **MouseEvent**.
- **InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event.

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

- To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

  boolean isAltDown( )
  boolean isAltGraphDown( )
  boolean isControlDown( )
  boolean isMetaDown( )
  boolean isShiftDown( )

N.N. College of Eng.

- You can obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:

  int getModifiers( )

ItemEvent Class

- An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

- **ItemEvent** has this constructor:

  ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

  Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.
- The **getItem( )** method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

  Object getItem( )
- The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

  temSelectable getItemSelectable( )

KeyEvent Class

- A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and**KEY_TYPED**.
- There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

- The **VK** constants specify *virtual key codes*
- **KeyEvent** is a subclass of **InputEvent**. Here is one of its constructors:

  KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

  Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*
- **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

  char getKeyChar( )

  int getKeyCode( )

- If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

MouseEvent Class

- There are eight types of mouse events.

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

- **MouseEvent** is a subclass of **InputEvent**. Here is one of its constructors:
  MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*)
  Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
- **getX( )** and **getY( ):** These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:
  int getX( )
  int getY( )
- **getPoint( )** method to obtain the coordinates of the mouse.
  Point getPoint( )
- The **translatePoint( )** method changes the location of the event. Its form is shown here:
  void translatePoint(int *x*, int *y*)
  Here, the arguments *x* and *y* are added to the coordinates of the event.
- The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:
  int getClickCount( )
- The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:
  boolean isPopupTrigger( )
- **getButton( )** method, shown here:
  int getButton( )
  It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **MouseEvent**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

J.N.N College of Eng

MouseWheelEvent Class

- The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**.
- **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

- Here is one of the constructors defined by **MouseWheelEvent**:
  MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*, int *scrollHow*, int *amount*, int *count*)
  Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred.
- To obtain the number of rotational units, call **getWheelRotation( )**, shown here: **int getWheelRotation( )**

TextEvent Class

- These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.
- The one constructor for this class is shown here:
  TextEvent(Object *src*, int *type*)
  Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

WindowEvent Class

- There are ten types of window events.

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

- **WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors.
  WindowEvent(Window *src*, int *type*)
  WindowEvent(Window *src*, int *type*, Window *other*)
  WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
  WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)
  other specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *to State* specifies the new state that the window will have when a window state change occurs.

- **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:

  Window getWindow( )

Sources of Events

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-3 Commonly Used Event Listener Interfaces

ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

J.N.N. College of Engg.

This interface defines two methods.

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

The WindowListener Interface

This interface defines seven methods.

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces.

```
//      Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
 /*<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener,
MouseMotionListener
{
        String msg = "";
        int mouseX = 0, mouseY = 0; // coordinates
        of mouse public void init()
```

```
{
        addMouseListener(this);
        addMouseMotionListener(this);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me)
 {
        //      save coordinates
        mouseX = 0; mouseY = 10;
        msg = "Mouse clicked.";
        repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me)
{
        //      save coordinates
        mouseX = 0; mouseY = 10;
        msg = "Mouse entered.";
        repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me)
{
        //      save coordinates
        mouseX = 0; mouseY = 10;
        msg = "Mouse exited.";
        repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me)
{
        //      save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
         msg = "Down";
        repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me)
{
        //      save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up"; repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
```

J. N. College of Engg.

```
            //      save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "*";
            showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
            repaint();
        }
        // Handle mouse moved.
        public void mouseMoved(MouseEvent me)
        {
            // show status
            showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
        }
        //      Display msg in applet window at
        current X,Y location. public void
        paint(Graphics g)
        {
            g.drawString(msg, mouseX, mouseY);
        }
}
```

- It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer.

  Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
- It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
- The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces.
- Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:
  void addMouseListener(MouseListener *ml*)
  void addMouseMotionListener(MouseMotionListener *mml*)

Handling Keyboard Events

- When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler.
- When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed.
- If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked.

```
//      Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet> */
public class SimpleKey extends Applet implements KeyListener
{
        String msg = "";
        int X = 10, Y = 20; // output coordinates
        public void init()
        {
                addKeyListener(this);
        }
        public void keyPressed(KeyEvent ke)
        {
                showStatus("Key Down");
        }
        public void keyReleased(KeyEvent ke)
        {
                showStatus("Key Up");
        }
        public void keyTyped(KeyEvent ke)
        {
                msg += ke.getKeyChar();
        }

        // Display keystrokes.
        public void paint(Graphics g)
        {
                g.drawString(msg, X, Y);
        }
}
```

J. N. College of Engg.

Adapter Classes

An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

```
//      Demonstrate
an adapter. import
java.awt.*; import
java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300
height=100> </applet>
*/
public class AdapterDemo extends Applet
{
        public void init()
        {
                addMouseListener(new MyMouseAdapter(this));
                addMouseMotionListener(new
                MyMouseMotionAdapter(this));
        }
}

class MyMouseAdapter extends MouseAdapter
{
        AdapterDemo adapterDemo;
        public MyMouseAdapter(AdapterDemo adapterDemo)
        {
                this.adapterDemo = adapterDemo;
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {
                adapterDemo.showStatus("Mouse clicked");
        }
}
class MyMouseMotionAdapter extends MouseMotionAdapter
```

I. N. College of Engg.

```
{
        AdapterDemo adapterDemo;
        public MyMouseMotionAdapter(AdapterDemo adapterDemo)
        {
                this.adapterDemo = adapterDemo;
        }
        // Handle mouse dragged.
        public void mouseDragged(MouseEvent me)
        {
                adapterDemo.showStatus("Mouse dragged");
        }
}
```

- It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored.
- The program has three classes.

  AdapterDemo extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events.

  MyMouseAdapter extends **MouseAdapter** and overrides the **mouseClicked( )** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class.

  MyMouseMotionAdapter extends **MouseMotionAdapter** and overrides the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

Inner Classes

An *inner class* is a class defined within another class, or even within an expression.

```
//      Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200
height=100> </applet>
*/
public class InnerClassDemo extends Applet
{
        public void init()
        {
                addMouseListener(new MyMouseAdapter());
        }
        class MyMouseAdapter extends MouseAdapter
        {
                public void mousePressed(MouseEvent me)
                {
```

```
                showStatus("Mouse Pressed");
            }
        }
}
```

- Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**.
- Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **showStatus( )** method directly.

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name.

```
//      Anonymous inner  class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200
height=100> </applet>
*/
public class AnonymousInnerClassDemo extends Applet
{
        public void init()
        {
                addMouseListener(new MouseAdapter()
                {
                        public void mousePressed(MouseEvent me)
                        {
                        showStatus("Mouse Pressed");
                        }
                });
        }
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**.
The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class.

J. N. College of Engg.

Swings

Introduction

Swing contains a set of classes that provides more powerful and flexible GUI components than those of **AWT**. **Swing** provides the look and feel of modern Java GUI. Swing library is an official Java GUI tool kit released by Sun Microsystems. It is used to create graphical user interface with Java.

Swing is a set of program component s for **Java** programmers that provide the ability to create graphical user interface (GUI) components, such as buttons and scroll bars, that are independent of the windowing system for specific operating system . **Swing** components are used with the **Java** Foundation Classes (JFC).

The Origins of Swing

The original Java GUI subsystem was the Abstract Window Toolkit (AWT).

AWT translates it visual components into platform-specific equivalents (peers).

Under AWT, the look and feel of a component was defined by the platform.

AWT components are referred to as **heavyweight**.

Swing was introduced in 1997 to fix the problems with AWT.

Swing offers following key features:

1. Platform Independent
2. Customizable
3. Extensible
4. Configurable
5. Lightweight

- ✓ Swing components are **lightweight** and don't rely on peers.
- ✓ Swing supports a pluggable look and feel.
- ✓ Swing is built on AWT.

J. N. College of Engg.

Model-View-Controller

One component architecture is MVC - Model-View-Controller.

The **model** corresponds to the state information associated with the component.

The **view** determines how the component is displayed on the screen.

The **controller** determines how the component responds to the user.

Swing uses a modified version of MVC called "Model-Delegate". In this model the view (look) and controller (feel) are combined into a "delegate". Because of the Model-Delegate architecture, the look and feel can be changed without affecting how the component is used in a program.

Components and Containers

A component is an independent visual control: a button, a slider, a label, ...

A container holds a group of components.

In order to display a component, it must be placed in a container.

A container is also a component and can be contained in other containers. Swing applications create a containment-hierarchy with a single top-level container.

Components

Swing components are derived from the **JComponent** class. The only exceptions are the four top-level containers: JFrame, JApplet, JWindow, and JDialog.

JComponent inherits AWT classes Container and Component.

All the Swing components are represented by classes in the javax.swing package.

All the component classes start with **J**: JLabel, JButton, JScrollbar, ...

Containers

There are two types of containers:

//Top-level which do not inherit JComponent, and

J.N.N. College of Engg.

// Lightweight containers that do inherit JComponent.

Lightweight components are often used to organize groups of components.

Containers can contain other containers.

All the component classes start with **J**: JLabel, JButton, JScrollbar, ...

Top-level Container Panes

Each top-level component defines a collection of "panes". The top-level pane is **JRootPane**.

JRootPane manages the other panes and can add a menu bar.

There are three panes in JRootPane: 1) the glass pane, 2) the content pane, 3) the layered pane.

The content pane is the container used for visual components. The content pane is an instance of JPanel.

The Swing Packages:

Swing is a very large subsystem and makes use of many packages. These are the packages used by Swing that are defined by Java SE 6.

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

Some of the Swing Packages are:

javax.swing javax.swing.border javax.swing.colorchooser javax.swing.event javax.swing.filechooser javax.swing.plaf javax.swing.plaf.basic javax.swing.plaf.metal javax.swing.plaf.multi	javax.swing.plaf.synth javax.swing.table javax.swing.text javax.swing.text.html javax.swing.text.html.parser javax.swing.text.rtf javax.swing.tree javax.swing.undo

J.N.N. College of Engg.

A simple Swing Application;

There are two ways to create a frame:

By creating the object of Frame class (association)
By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

By creating the object of Frame class

```
import javax.swing.*;
public class FirstSwing
{
        public static void main(String[] args)
        {
                JFrame f=new JFrame(" MyApp");
                        //creating instance of JFrame and title of the frame is
                MyApp. JButton b=new JButton("click");
                   //creating instance of JButton  and name of the button is click.
               b.setBounds(130,100,100, 40);    //x axis, y axis, width, height
               f.add(b);                         //adding  button  in JFrame
               f.setSize(400,500);              //400  width  and 500  height
               f.setLayout(null);                //using  no layout  managers
              f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
              f.setVisible(true);                //making the frame visible
        }
}
```

Output:

J.N.N. College of Engg.

Explanation:

- The program begins by importing javax.swing. As mentioned, this package contains the components and models defined by Swing.
- For example, javax.swing defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing. Next,
- the program declares the FirstSwing class
- It begins by creating a JFrame, using this line of code:
  JFrame f = new JFrame("My App");
- This creates a container called f that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

Next, the window is sized using this statement:

```
f.setSize(400,500);
```

- The setSize( ) method (which is inherited by JFrame from the AWT class Component) sets the dimensions of the window, which are specified in pixels. in this example, the width of the window is set to 400 and the height is setto 500.
- By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated.
  - If want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call setDefaultCloseOperation( ), as the program does:

```
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Swing by inheritance

We can also inherit the JFrame class, so there is no need to create the instance of JFrame class explicitly.

J.N.N. College of Engg.

```
import javax.swing.*;
public class MySwing extends JFram      //inheriting JFrame
{
   JFrame f;
   MySwing()
   {
        JButton b=new JButton("click");//create
        button b.setBounds(130,100,100,  40);

        add(b);//adding button on
        frame setSize(400,500);
        setLayout(null);
        setVisible(true);
   }
   public static void main(String[] args)
   {
       new MySwing();
   }
}
```

Jlabel, JTextField and JPassword

- JLabel is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at JLabel a bit mor closely.

- JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors. Here are three of them:

  JLabel(Icon icon)

  JLabel(String str)

  JLabel(String str, Icon icon, int align)

J. N. N. College of Engg.

- JTextField is the simplest Swing text component. It is also probably its most widely used text component. JTextField allows you to edit one line of text. It is derived from JTextComponent, which provides the basic functionality common to Swing text components.
- Three of JTextField's constructors are shown here:

  JTextField(int cols)

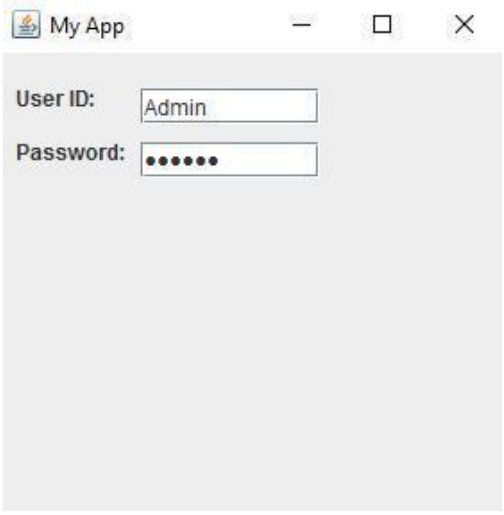
  JTextField(String str, int cols)

  JTextField(String str)
- Here, str is the string to be initially presented, and cols is the number ofcolumns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.
- JPasswordField is a lightweight component that allows the editing of a single line of text where the view indicates something was typed, but does not show the original characters.

```
import javax.swing.*;
public class JTextFieldPgm
{

     public static void main(String[] args)
     {
          JFrame f=new JFrame("My App");
          JLabel namelabel= new JLabel("User ID: ");
           namelabel.setBounds(10, 20, 70, 10);
          JLabel passwordLabel = new JLabel("Password: ");
         passwordLabel.setBounds(10, 50, 70, 10);

                JTextField userText = new JTextField();
                userText.setBounds(80, 20, 100, 20);

JPasswordField  passwordText = new JPasswordField();
                passwordText.setBounds(80, 50, 100, 20);

                f.add(namelabel);
                f.add( passwordLabel);
                f.add(userText);
                f.add(passwordText);

                f.setSize(300, 300);
                f.setLayout(null);
                f.setVisible(true);

     }

}
```

J.N. N College of Engg.

Output:



ImageIcon with JLabel

- JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors. Here are three of them:

JLabel(Icon icon)

JLabel(String str)

JLabel(String str, Icon icon, int align)

Here, str and icon are the text and icon used for the label.

The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING.

- These constants are defined in the Swing Constants interface, along with several others used by the Swing classes. Notice that icons are specified by objects of type Icon, which is an interface defined by Swing.
- The easiest way to obtain an icon is to use the ImageIcon class. ImageIcon implements Icon and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor.

ImageIcon(String filename)

J.N.N. College of Engg.

- It obtains the image in the file named filename. The icon and text associated with the label can be obtained by the following methods:

  Icon getIcon( )

  String getText( )

The icon and text associated with a label can be set by these methods:

void setIcon(Icon icon)

void setText(String str)

Here, icon and str are the icon and text, respectively. Therefore, using setText( ) it is possible to change the text inside a label during program execution.

```
import javax.swing.*;

public class PgmImageIcon
{

      public static void main(String[] args)
      {

            JFrame jf=new
            JFrame("Image Icon");
            jf.setLayout(null);

           ImageIcon icon = new ImageIcon("JNNCE.jpg");
          JLabel label1 = new JLabel("Welocme to JNNCE",
          icon,JLabel.RIGHT);
          label1.setBounds(20, 30,267, 200);
          jf.add(label1);

          jf.setSize(300,400);
          jf.setVisible(true);
      }

}
```

J. N. N. College of Engg.

The Swing Buttons:

There are four types of Swing Button

1. JButton
2. JRadioButton
3. JCheckBox
4. JComboBox

JButton class provides functionality of a button. A JButton is the Swing equivalent of a Button in AWT. It is used to provide an interface equivalent of a common button.

JButton class has three constuctors,

JButton(Icon *ic*)

JButton(String *str*)

JButton(String *str*, Icon *ic*)

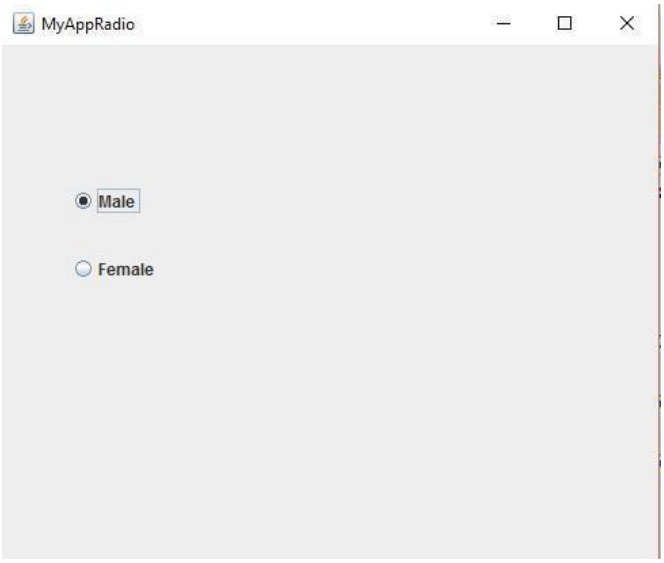
```
import javax.swing.*;

class FirstSwing
{
    public static void main(String args[])
    {
        JFrame jf=new JFrame("My App");

        JButton  jb=new  JButton("Next");
        jb.setBounds(30, 100, 100, 50);

        JButton  jb1=new JButton("Prev");
        jb1.setBounds(30, 200, 100, 50);

        jf.add(jb);
        jf.add(jb1);

        jf.setSize(300, 600);
        jf.setLayout(null);
        jf.setVisible(true);

    }
}
```

J. N. N. College of Engg.

A **JRadioButton** is the swing equivalent of a RadioButton in AWT. It is used to represent multiple option single selection elements in a form. This is performed by grouping the JRadio buttons using a ButtonGroup component. The ButtonGroup class can be used to group multiple buttons so that at a time only one button can be selected.

```
import javax.swing.*;

import javax.swing.*;
public class RadioButton1
{
    public static void main(String args[])
    {

       JFrame f=new JFrame("MyAppRadio");
     JRadioButton r1=new JRadioButton("Male ");
   JRadioButton r2=new JRadioButton("Female");

      r1.setBounds(50, 100, 70, 30);
        r2.setBounds(50,150,70,30);

        ButtonGroup bg=new ButtonGroup();
        bg.add(r1);
        bg.add(r2);

        f.add(r1);
        f.add(r2);

        f.setSize(500,500);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

J.N.N. College of Engg.



A **JCheckBox** is the Swing equivalent of the Checkbox component in AWT. This is sometimes called a ticker box, and is used to represent multiple option selections in a form.

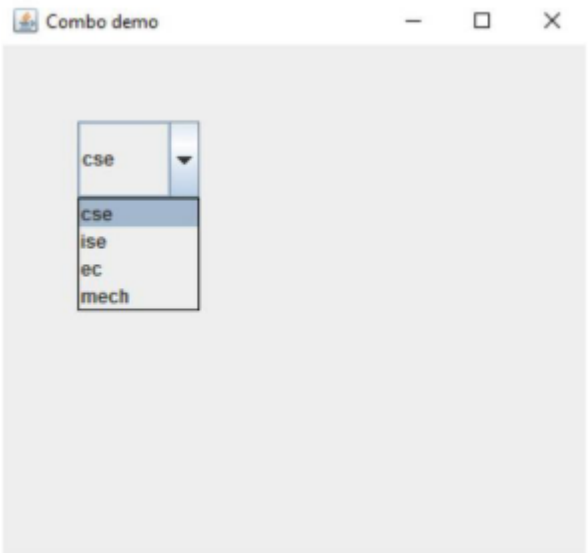
```
import javax.swing.*;

class FirstSwing
{
     public static void main(String args[])
     {
          JFrame jf=new JFrame("CheckBox");

          JCheckBox jb=new JCheckBox("JAVA");
          jb.setBounds(30, 100, 100, 50);

          JCheckBox jb1=new
          JCheckBox("Python");
          jb1.setBounds(30, 200, 100, 50);

          jf.add(jb);
          jf.add(jb1);

          jf.setSize(300, 600);
          jf.setLayout(null);
          jf.setVisible(true);

     }
}
```

J.N.N. College of Engg.

The **JComboBox** class is used to create the combobox (drop-down list). At a time only one item can be selected from the item list.

```
import java.awt.*;
import javax.swing.*;
public class Comboexample
{public static void main(String[] args)
    {
        JFrame f=new JFrame("Combo demo");
        String Branch[]={"cse","ise","ec","mech"};
        JComboBox jc=new JComboBox(Branch);
        jc.setBounds(50,50,80,50);
        f.add(jc);
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

J. N. N. College of Engg.



JTable and JScrollPane:

The JTable class is used to display the data on two dimensional tables of cells.

Commonly used Constructors of JTable class:

- ✓ JTable(): creates a table with empty cells.

  JTable(Object[ ][ ] rows, Object[ ] columns): creates a table with the specified data.

- ✓ JScrollPane is a lightweight container that automatically handles the scrolling of another component.

- ✓ The component being scrolled can either be an individual component, such as a table, or a group of components contained within another lightweight container, such as a JPanel.

- ✓ In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because JScrollPane automates scrolling, it usually eliminates the need to manage individual scroll bars.

- ✓ The viewable area of a scroll pane is called the viewport.

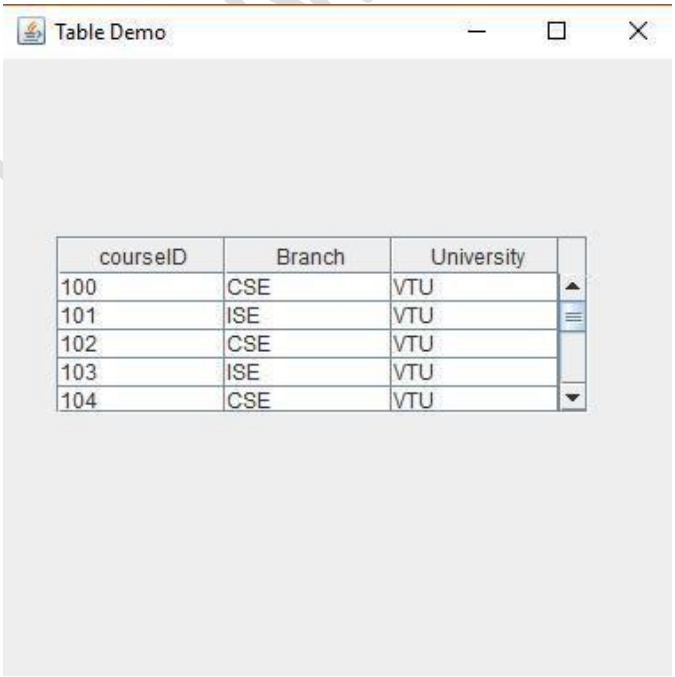
- ✓ It is a window in which the component being scrolled is displayed.

- ✓ Thus, the view port displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport.

- ✓ In its default behavior, a JScrollPane will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

J.N.N. College of Engg.

```
import javax.swing.*;
public class TableExample1
{
      public static void main(String[] args)
      {
            // TODO   Auto-generated   method   stub
            JFrame f=new JFrame("Table Demo");

            String data[][]={
                                          {"100","CSE","VTU"}
                                          ,
                                          {"101","ISE","VTU"}
                                          ,
                                          {"102","CSE","VTU"}
                                          ,
                                          {"103","ISE","VTU"}
                                          ,
                                          {"105","ISE","VTU"}
                                          ,
                                          {"106","ISE","VTU"}
                                    };
    String column[]={"courseID","Branch","University"};

            JTable jt=new JTable(data,column);
            JScrollPane js=new
            JScrollPane(jt);
            js.setBounds(30,100,300,100);
            f.add(js);

             f.setSize(300,400);
             f.setLayout(null);
             f.setVisible(true);

      }
}
```



N. College of Engg.

JTabbedpane

- ✓ JTabbedPane encapsulates a tabbed pane. It manages a set of components by linking them with tabs.
- ✓ Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI.
- ✓ Given the complex nature of a tabbed pane, they are surprisingly easy to create and use. JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane.
- ✓ The other two constructors let you specify the location of the tabs, which can be along any of the four sides.
- ✓ JTabbedPane uses the SingleSelectionModel model. Tabs are added by calling **addTab( ) method**. Here is one of its forms:

  void addTab(String name, Component comp)

- ✓ Here, **name** is the name for the tab, and **comp** is the component that should be added to the tab. Often, the component added to a tab is a JPanel that contains a group of related components. This technique allows a tab to hold a set of components.

```
import javax.swing.*;

public class MainClass

{
  public static void main(String[] a)
  {
    JFrame f = new JFrame("JTab");

    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    f.add(new JTabbedPaneDemo());

    f.setSize(500, 500);

    f.setVisible(true);

  }

}
```

J.N.N. College of Engg.

```
class JTabbedPaneDemo extends JPanel
{
                JTabbedPaneDemo()

                {
                        makeGUI();
                }

               void makeGUI()

                {
                                JTabbedPane jtp = new JTabbedPane();

                                jtp.addTab("Cities", new CitiesPanel());

                                jtp.addTab("Colors", new ColorsPanel());

                                jtp.addTab("Flavors", new FlavorsPanel());

                                add(jtp);
                }
}

class CitiesPanel extends JPanel
{
  public CitiesPanel()
  {
    JButton b1 = new JButton("NewYork");

    add(b1);

    JButton b2 = new JButton("London");

    add(b2);

    JButton b3 = new JButton("Hong Kong");

    add(b3);

    JButton b4 = new JButton("Tokyo");

    add(b4);
  }

}

class ColorsPanel extends JPanel
{
  public ColorsPanel()
  {
   JCheckBox cb1 = new
```

J.N. College of Engg.

```
    JCheckBox("Red"); add(cb1);

    JCheckBox cb2 = new

    JCheckBox("Green"); add(cb2);
    JCheckBox cb3 = new

    JCheckBox("Blue"); add(cb3);
  }
}

class FlavorsPanel extends JPanel
{

  public FlavorsPanel()
  {
    JComboBox jcb = new JComboBox();

    jcb.addItem("Vanilla");

    jcb.addItem("Chocolate");

    jcb.addItem("Strawberry");

    add(jcb);
  }
}
```

JList:

- ✓ In Swing, the basic list class is called JList.
- ✓ It supports the selection of one or more items from a list.
- ✓ Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed.
- ✓ JList is so widely used in Java that it is highly unlikely that you have not seen one before.

JList provides several constructors. The one used here is

JList(Object[ ] items)

- ✓ This creates a JList that contains the items in the array specified by items.
- ✓ JList is based on two models. The first is ListModel. This interface defines how access to the list data is achieved.
- ✓ The second model is the ListSelectionModel interface, which defines methods that determine what list item or items are selected.

```
import java.awt.FlowLayout;

import javax.swing.*;

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

JFrame frame = new JFrame("JList");

String[] selections = { "green", "red", "orange", "dark
blue" };
JList list = new JList(selections);

list.setSelectedIndex(1);

frame.add(new JScrollPane(list));

frame.setSize(300, 400);
frame.setLayout(new FlowLayout());
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setVisible(true);
}

}
```

J.N. College of Engg.

Difference between Java AWT and java Swing

	Java AWT	Java Swing
1.	AWT components are platform-dependent.	Java swing components are platform-independent.
2.	AWT components are heavyweight.	Swing components are lightweight.
3.	AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
4.	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5.	AWT doesn't follows MVC(Model View Controller)	Swing follows MVC .where model represents data, view represents presentation and controller acts as an interface between model and view.

J.N.N. College of Engg.

Questions

1. Write a swing applet program to demonstrate with two JButtons named JNNCE and VTU. When either of button pressed, it should display respective label with its icon. Refer the image icon "jnnce.gif" and "vtu.gif". set the initial label is "press the button"
2. Explain JscrollPane with an example.
3. Explain IComboBox with an example.
4. Name & Explain the different types of Swing Buttons with syntax.
5. Write the steps to create J-table.write a program to create a table with column heading "fname,lname,age" and insert at least five records in the table and display.
6. Differentiate between AWT and Swings?
7. Explain the MVC architecture of swings?
8. Describe the different types of swing button?
9. What is a swing ? explain the components and containers in the swings
10. Explain the following with an example for each
    i)JTextField class ii)JButton class iii)JComboBox Class
11. Explain the following swing buttons.
    A. JButton
    B. JChekBoxes **C**. JRadioButton
12. Explain the concept of JComboBox and JTable.
13. Write a program which displays the contents of an array in the tabular format.

J.N.N. College of Engg.