ANNAMALAI UNIVERSITY

WITH COURAGE AND FAITH

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E.(CSE) Big Data Analytics - III SEMESTER

19BDCP 30 - OBJECT ORIENTED PROGRAMMING LAB MANUAL

Annamalai University
Faculty of Engineering and Technology
Department of Computer Science and Engineering
B. E. (CSE) - Big Data Analytics
III Semester
19BDCP308 - Object Oriented Programming Lab

List of Exercises

Cycle 1 : C++

1. Classes and Objects
2. Constructors and Destructors
3. Passing/Returning Objects to/from a Function
4. Unary and Binary Operator Overloading
5. Multiple Inheritance
6. Friend Function and Virtual Function
7. Data Conversion Between Objects of Different Classes
8. File Operations
9. Stack using Standard Template Library

Cycle 2 : JAVA

10. Classes and Objects
11. String Functions
12. Creation of Packages and Interfaces
13. Exception Handling
14. Multithreading
15. Abstract Window Toolkit

Staff-in-charge: Dr. L. R. Sudha

Ex. No. : 1 **Classes and Objects**
Date :

Aim:

To write a C++ program to show how to create a class and how to create objects.

Theoretical Concepts:

Classes

A class is a blueprint that defines the variables and the methods common to all objects of a certain kind. A class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects.

A class in C++ is a user-defined type or data structure declared with keyword ***class*** that has data and functions (also called member variables and member functions) as its members whose access is governed by the three access specifiers ***private***, ***protected*** or ***public***. By default access to members of a C++ class is private. The private members are not accessible outside the class; they can be accessed only through methods of the class. The public members form an interface to the class and are accessible outside the class.

A class is defined in C++ using keyword ***class*** followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end as shown in the syntax.

Syntax:
```
class ClassName
   {
   // some data
   // some functions
   };
```

Objects

In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. Instances of a class data type are known as objects. All the public members of the class can be accessed through object. Object is a runtime entity, it is created at runtime. In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. To use the data and access functions defined in the class, you need to create objects using the following syntax.

Syntax

```
ClassName ObjectName-1, ObjectName-2, …, ObjectName-N;
```

Accessing Data Members and Member Functions

The data members and member functions of class can be accessed using the dot(“.“) operator with the object.

Syntax

```
ObjectName.FunctionName(Parameter);
```

For example if the name of object is obj and you want to access the member function with the name `printName()` then you have to write `obj.printName()`

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. In C++, this access control is given by access specifiers **private**, **protected** or **public**.

Syntax

```
ObjectName.PublicMember
```

For example if the name of object is obj and you want to access the data member with the name totalmarks then you have to write `obj.totalmarks` to access the content.

Program:

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

class Distance
{
private:
     int feet;
     float inches;
public:
     void setdist(int ft, float in)
          {
          feet = ft;
          inches = in;
          }

     void getdist()
          {
          cout << "\nEnter feet: ";
          cin >> feet;
          cout << "Enter inches: ";
          cin >> inches;
          }

     void showdist()
          {
          cout << feet << "\'-" << inches << "\" ";
          }
};

int main()
     {
     Distance dist1, dist2;
     clrscr();
     dist1.setdist(11, 6.25);
     dist2.getdist();
     clrscr();
     cout << "\ndist1 = ";
     dist1.showdist();
     cout << "\ndist2 = ";
     dist2.showdist();
     getch();
     return 0;
     }
```

Sample Input and Output:

```
Enter feet: 10
Enter inches: 4.75
dist1 = 11'-6.25"
dist2 = 10'-4.75"
```

Result:

Thus, a C++ program has been written to show how to create a class and how to create objects.

Ex. No. : 2 **Constructors and Destructors**
Date :

Aim:

To write a C++ program to implement constructors and destructor for initializing and destroying objects.

Theoretical Concepts:

Constructors

The process of creating and deleting objects in C++ is a vital task. Each time an instance of a class is created the constructor method is called. Constructor is a special member function of class and it is used to initialize the objects of its class. It is treated as a special member function because its name is the same as the class name. These constructors get invoked whenever an object of its associated class is created. It is named as "constructor" because it constructs the value of data member of a class. Initial values can be passed as arguments to the constructor function when the object is declared. This can be done in two ways: (1) By calling constructor explicitly, (2) By calling constructor implicitly. C++ offers four types of constructors. These are:

1. Do nothing constructor
2. Default constructor
3. Parameterized constructor
4. Copy constructor

Do nothing constructor

Do nothing constructors are that type of constructor which does not contain any statements. Do nothing constructor is the one which has no argument in it and no return type.

Syntax

```
class-name ()
     { }
```

Default constructor

The default constructor is the constructor which doesn't take any argument. It has no parameter but a programmer can write some initialization statement there. A default constructor is very important for initializing object members, that even if we do not define a constructor explicitly, the compiler automatically provides a default constructor implicitly.

Syntax

```
class-name ()
      {
       // body of the function
      }
```

Parameterized constructor

A default constructor does not have any parameter, but programmers can add and use parameters within a constructor if required. This helps programmers to assign initial values to an object at the time of creation.

Syntax

```
class-name (data-type variable-1, data-type variable-2, . . . ,
data-type variable-n)
      {
       // body of the function
      }
```

Copy constructor

C++ provides a special type of constructor which takes an object as an argument and is used to copy values of data members of one object into another object. In this case, copy constructors are used to declaring and initializing an object from another object. The process of initializing through a copy constructor is called the copy initialization.

Syntax

```
class-name (class-name &)
      {
       // body of the function
      }
```

There can be many constructors in a class. It can be overloaded based on different types or number of arguments but it cannot be inherited or virtual. Different constructors will be called depends on different types or number of arguments passed. Whenever we create an object, the default constructor of that class is invoked automatically to initialize the members of the class. If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e. the order of invocation is that the base class"s default constructor will be invoked first and then the derived class"s default constructor will be invoked.

Destructors

As the name implies, destructors are used to destroy the objects that have been created by the constructor within the C++ program. Like constructor, destructor (also known as deconstructor) is also a member function of a class that has the same name as the class name preceded by a tilde(~) operator. It helps to deallocate the memory of an object. It is called while object of the class is freed or deleted. In a class, there is always a single destructor without any parameters so it can"t be overloaded. It is always called in the reverse order of the constructor. If a class inherited by another class and both the classes have destructor then the destructor of the child class is called first, followed by the destructor of the parent or base class.

Syntax

```
~class-name ()
     {
      // body of the function
     }
```

Program:

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

class Counter
{
private:
     unsigned int count;
public:
     Counter() : count(0)
          {  }

     Counter(int c)
          {
          count = c;
          }

     Counter (Counter &c)
          {
          count = c.count;
          }

     void inc_count()
          {
          count++;
          }

     void dec()
          {
          count--;
          }

     void show()
          {
          cout<<"Count is "<<count<<endl;
          }

     ~Counter()
          {
          cout<<"Object destroyed\n";
          }
};

int main()
     {
     Counter c1, c2(20),c3(c2);
     clrscr();
     c1.inc_count();
     c2.inc_count();
     c3.dec();
     c1.show();
     c2.show();
     c3.show();
     getch();
     return 0;
     }
```

Sample Input and Output:

```
Count is 1
Count is 21
Count is 19
Object destroyed
Object destroyed
Object destroyed
```

Result:

Thus, a C++ program has been written to implement constructors and destructor for initializing and destroying objects.

Ex. No. : 3 **Passing/Returning Objects to/from a Function**
Date :

Aim:

To write a C++ program to pass objects to a function and return objects from a function.

Theoretical Concepts:

Passing Objects to a Function

In C++, we can pass class"s objects as arguments and also return them from a function the same way we pass and return other variables. No special keyword or header file is required to do so. The objects of a class can be passed as arguments to member functions as well as non-member functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object. To pass an object as an argument, we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax

Declaration

```
function_name(class_name object_name);
```

Calling

```
function_name(object_name);
```

On the other hand, in pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Syntax

Declaration

```
function_name(class_name &object_name);
```

Calling

```
function_name(object_name);
```

Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

Returning Objects from a Function

A function can also return objects either by value or by reference. When an object is returned by value from a function, a temporary object is created within the function, which holds the return value. This value is further assigned to another object in the calling function.

The syntax for defining a function that returns an object by value is

```
class_name function_name (parameter_list)
{
// body of the function
}
```

In the case of returning an object by reference, no new object is created, rather a reference to the original object in the called function is returned to the calling function.

The syntax for defining a function that returns an object by reference is

```
class_name& function_name (parameter_list)
{
//body of the function
}
```

Program:

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

class Distance
{
private:
     int feet;
     float inches;
```

```
public:
     Distance() : feet(0), inches(0.0)
          { }
     Distance(int ft, float in) : feet(ft), inches(in)
          { }
     void getdist()
          {
          cout << "\nEnter feet: ";
          cin >> feet;
          cout << "Enter inches: ";
          cin >> inches;
          }
     void showdist()
          {
          cout << feet << "\'-" << inches << "\"";
          }
     Distance add_dist(Distance);
};

Distance Distance::add_dist(Distance d2)
     {
     Distance temp;
     temp.inches = inches + d2.inches;
     if(temp.inches >= 12.0)
          {
          temp.inches -= 12.0;
          temp.feet = 1;
          }
     temp.feet += feet + d2.feet;
     return temp;
     }

int main()
     {
     Distance dist1, dist3;
     Distance dist2(11, 6.25);
     clrscr();
     dist1.getdist();
     dist3 = dist1.add_dist(dist2);
     cout << "\ndist1 = ";
     dist1.showdist();
     cout << "\ndist2 = ";
     dist2.showdist();
     cout << "\ndist3 = ";
     dist3.showdist();
```

```
                getch();
                return 0;
                }
```

Sample Input and Output:

```
Enter feet: 12
Enter inches: 3

dist1 = 12'-3"
dist2 = 11'-6.25"
dist3 = 23'-9.25"
```

Result:

Thus, a C++ program has been written to pass objects to a function and return objects from a function.

Ex. No. : 4 **Unary and Binary Operator Overloading**
Date :

Aim:

To write C++ programs to overload unary operator and binary operator.

Theoretical Concepts:

Operator Overloading

C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator overloading is the method by which we can change the function of some specific operators to do some different task. Operator overloading is a compile polymorphic technique where a single operator can perform multiple functionalities. This can be done by declaring the function as in syntax below. Operator overloading can be done by using three approaches, they are

1. Overloading unary operator.
2. Overloading binary operator.
3. Overloading binary operator using a friend function.

Syntax

```
friend Return_Type classname :: operator op(Argument list)
{
    // body of the function
}
```

Operator function must be either non-static (member function) or friend function. In the above syntax `friend` is a keyword for friend function and it can be optional, `Return_Type` is value type to be returned to another object, operator `op` is the function, where the `operator` is a keyword and `op` is the operator to be overloaded.

Unary Operator Overloading

Unary operator is an operator that operates on a single operand and returns a new value. The unary operators that can be overloaded are

1. Unary plus(+) and unary minus(-)
2. Increment(++) and decrement(--)
3. Address of operator(&)
4. Bitwise not(~) and logical not(!)
5. dereference *
6. new and delete

Binary Operator Overloading

Binary operator is an operator that operates with two operands and returns a new value. The binary operators that can be overloaded are

1. Arithmetic operators (+, -, *, /, %)
2. Logical operators (&&, ||)
3. Relational operators (==, !=, >, <, >=, <=)
4. Bitwise operators (&, |, ^, <<, >>)
5. Assignment operators (=, +=, -=, *=, /=, %=, &= , |= , ^= , <<== , >>=)

Rules to Define the Operator Function

1. In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
2. In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
3. All the class members should be public if operator overloading is implemented.
4. Operators that cannot be overloaded are
    a) ?: (conditional)
    b) . (member selection)
    c) .* (member selection with pointer-to-member)
    d) :: (scope resolution)
    e) sizeof (object size information)
5. Operators cannot be used to overload when declaring that function as friend function are
    a) = (assignment)
    b) () (function call)
    c) [] (subscripting)
    d) -> (class member access operator)

Program 1:

```
// Unary operator overloading
#include <iostream.h>
#include <conio.h>

class Counter
{
private:
      unsigned int count;
public:
      Counter() : count(0)
            { }
```

```
        unsigned int get_count()
             {
             return count;
             }
        void operator ++ ()
             {
             ++count;
             }
};

int main()
     {
     Counter c1, c2;
     clrscr();
     cout << "\nc1=" << c1.get_count();
     cout << "\nc2=" << c2.get_count();
     ++c1;
     ++c2;
     ++c2;
     cout << "\nc1=" << c1.get_count();
     cout << "\nc2=" << c2.get_count() << endl;
     getch();
     return 0;
     }
```

Sample Input and Output:

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

Program 2:

```
// Binary operator overloading
#include <iostream.h>
#include <conio.h>

class Distance
{
private:
     int feet; float inches;
public:
     Distance() : feet(0), inches(0.0)
          { }
     Distance(int ft, float in) : feet(ft), inches(in)
          { }
```

```
        void getdist()
             {
             cout << "\nEnter feet: "; cin >> feet;
             cout << "Enter inches: "; cin >> inches;
             }
        void showdist() const
             {
             cout << feet << "\'-" << inches << "\"";
             }
        Distance operator + ( Distance ) const;
   };

   Distance Distance::operator + (Distance d2) const
        {
        int f = feet + d2.feet;
        float i = inches + d2.inches;
        if(i >= 12.0)
             {
             i -= 12.0;
             f++;
             }
        return Distance(f,i);
        }

   int main()
        {
        Distance dist1, dist3, dist4;
        clrscr();
        dist1.getdist();
        Distance dist2(11, 6.25);
        dist3 = dist1 + dist2;
        dist4 = dist1 + dist2 + dist3;
        cout << "\n dist1 = ";
        dist1.showdist();
        cout << "\n dist2 = ";
        dist2.showdist();
        cout << "\n dist3 = ";
        dist3.showdist();
        cout << "\n dist4 = ";
        dist4.showdist();
        getch();
        return 0;
        }
```

Sample Input and Output:

```
Enter feet: 14
Enter inches: 3
dist1 = 14'-3"
dist2 = 11'-6.25"
dist3 = 25'-9.25"
dist4 = 51'-6.5"
```

Result:

Thus, C++ programs have been written to overload unary operator and binary operator.

Ex. No. : 5 **Multiple Inheritance**
Date :

Aim:

To write a C++ program to inherit a class from multiple classes.

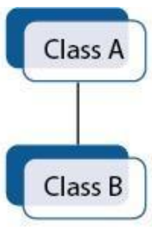
Theoretical Concepts:

Inheritance

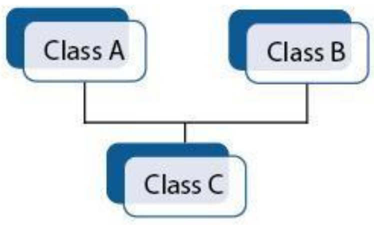
In C++, inheritance is a process in which one object acquires all the properties and behaviours of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviours which are defined in other class. The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

C++ supports five types of inheritance:

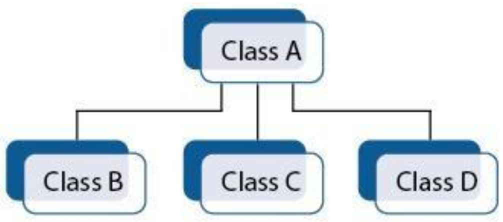
1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance



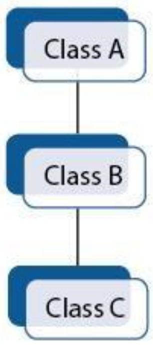
Single Inheritance



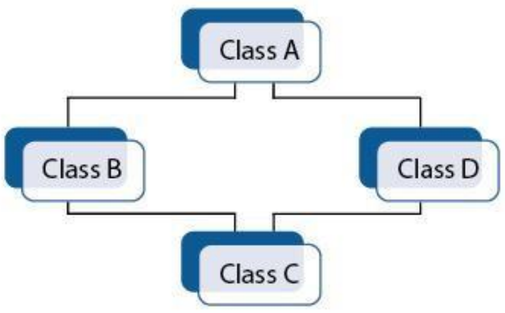
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

The Syntax of Derived class is

```
class derived_class_name :: visibility-mode base_class_name
{
    // body of the derived class.
}
```

where,
`derived_class_name` - It is the name of the derived class.
`visibility-mode` - The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited.
`base_class_name` - It is the name of the base class.

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.

Multilevel inheritance is a process of deriving a class from another derived class.

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

Hybrid inheritance is a combination of more than one type of inheritance.

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

Visibility modes can be classified into three categories:
1. **Public:** When the member is declared as public, it is accessible to all the functions of the program.
2. **Private:** When the member is declared as private, it is accessible within the class only.
3. **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Multiple Inheritance

Multiple inheritance is a feature of C++ where a class can inherit from more than one classes. The constructors of inherited classes are called in the same order in which they are inherited. In multiple inheritance, a class is derived from more than one base class, using a comma-separated list. The most difficult to avoid complication that arises when using multiple inheritance is that sometimes the programmers interested in using this technique to extend the existing code are forced to learn some of the implementation's details. The second trivial problem that might appear when using this technique is the creation of ambiguities. This can be addressed in various ways, including using virtual inheritance.

Program:

```
#include <iostream.h>
#include <conio.h>
const int LEN = 80;
class student
{
private:
     char school[LEN];
     char degree[LEN];
public:
     void getedu()
      {
      cout<<"\n Student Class";
      cout << " \n Enter name of school or university: ";
      cin >> school;
      cout << " Enter highest degree earned \n";
      cout << " (Highschool, Bachelor\'s, Master\'s, PhD): ";
      cin >> degree;
      }
     void putedu()
          {
          cout<<"\n Student Class";
          cout << "\n School or university: " << school;
          cout << "\n Highest degree earned: " << degree;
          }
};

class employee
{
private:
     char name[LEN];
     unsigned long number;
public:
     void getdata()
          {
          cout<<"\n Employee Class";
          cout << "\n Enter last name: ";
          cin >> name;
          cout << " Enter number: ";
          cin >> number;
          }
     void putdata()
          {
          cout<<"\n Employee Class";
          cout << "\n Name: " << name;
          cout << "\n Number: " << number;
          }
};

class manager : private employee, private student
{
private:
     char title[LEN];
     double dues;
public:
     void getdata()
          {
          employee::getdata();
          cout << " Enter title: ";
          cin >> title;
          cout << " Enter golf club dues: ";
          cin >> dues;
          student::getedu();
          }

     void putdata()
          {
          employee::putdata();
          cout << "\n Title: " << title;
          cout << "\n Golf club dues: " << dues;
          student::putedu();
          }
};

class scientist : private employee, private student
{
private:
     int pubs;
public:
     void getdata()
          {
          employee::getdata();
          cout << " Enter number of pubs: ";
          cin >> pubs;
          student::getedu();
          }
```

```
        void putdata()
             {
             employee::putdata();
             cout << "\n Number of publications: " << pubs;
             student::putedu();
             }
   };

   int main()
        {
        manager m1;
        scientist s1;
        clrscr();
        cout << "\n\t\t\t Enter data for manager";
        m1.getdata();
        cout << "\n\t\t\t Enter data for scientist ";
        s1.getdata();
        cout << "\n\t\t\t Data on manager ";
        m1.putdata();
        cout << "\n\t\t\t Data on scientist ";
        s1.putdata();
        getch();
        return 0;
        }
```

Sample Input and Output:

```
                  Enter data for manager
       Employee Class
       Enter last name: Ram
       Enter number: 11
       Enter title: Vice-President
       Enter golf club dues: 10000

       Student Class
       Enter name of school or university: Annamalai
       Enter highest degree earned
       (Highschool, Bachelor's, Master's, PhD): Master's

                  Enter data for scientist
       Employee Class
       Enter last name: Ravana
       Enter number: 99
       Enter number of pubs: 100
```

```
 Student Class
 Enter name of school or university: MIT
 Enter highest degree earned
(Highschool, Bachelor's, Master's, PhD): Ph.D.

                     Data on manager
 Employee Class
 Name: Ram
 Number: 11
 Title: Vice-President
 Golf club dues: 10000
 Student Class
 School or university: Annamalai
 Highest degree earned: Master's
                     Data on scientist
 Employee Class
 Name: Ravana
 Number: 99
 Number of publications: 100
 Student Class
 School or university: MIT
Highest degree earned: Ph.D.
```

Result:

Thus, a C++ program has been written to inherit a class from multiple classes.

Ex. No. : 6 **Friend Function and Virtual Function**
Date :

Aim:

To write C++ programs to show how to implement friend function and virtual function.

Theoretical Concepts:

Friend Function

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function. The complier knows a given function is a friend function by the use of the keyword `friend`. For accessing the data, the declaration of a friend function should be made inside the body of the class starting with keyword `friend`.

Declaration of friend function in C++

```
class class_name
{
    ...
    friend return_type function_name(argument/s);
    ...
}
```

Now, you can define the friend function as a normal function to access the data of the class. No `friend` keyword is used in the definition.

```
return_type function_name(argument/s)
{
    ...
    // Private and protected data of class_name can be accessed
    // from this function because it is a friend function of
    // class_name.
    ...
}
```

Friend functions have the following properties:

1. Friend of the class can be member of some other class.
2. Friend of one class can be friend of another class or all the classes in one program, such a friend is known as GLOBAL FRIEND.
3. Friend can access the private or protected members of the class in which they are declared to be friend, but they can use the members for a

specific object.

4. Friends are non-members hence do not get “this” pointer.
5. Friends can be friend of more than one class hence they can be used for message passing between the classes.
6. Friend can be declared anywhere (in public, protected or private section) in the class.

Virtual Function

A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword. It is used to tell the compiler to perform dynamic linkage or late binding on the function.
There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function. A 'virtual' is a keyword preceding the normal declaration of a function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Declaration of virtual function in C++

```
class base_class_name
{
    ...
    virtual return_type function_name(argument/s);
    ...
}
```

Rules of Virtual Function

1. Virtual functions must be members of some class.
2. Virtual functions cannot be static members.
3. They are accessed through object pointers.
4. They can be a friend of another class.
5. A virtual function must be defined in the base class, even though it is not used.
6. The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
7. We cannot have a virtual constructor, but we can have a virtual destructor.

Pure Virtual Function

A virtual function is not used for performing any task. It only serves as a placeholder. When the function has no definition, such function is known as "do-nothing" function. The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class. A class containing the pure virtual function cannot be used to declare the objects of its own; such classes are known as abstract base classes. The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Declaration of pure virtual function in C++

```
class base_class_name
{
    ...
    virtual return_type function_name(argument/s)=0;
    ...
}
```

Program 1:

```
//Friend function
#include <iostream.h>
#include <conio.h>

class beta;

class alpha
{
private:
      int data;
public:
      alpha() : data(3)
            { }
      friend int frifunc(alpha, beta);
};

class beta
{
private:
      int data;
public:
      beta() : data(7)
            { }
```

```
     friend int frifunc(alpha, beta);
};

int frifunc(alpha a, beta b)
     {
     return( a.data + b.data );
     }

int main()
     {
     alpha aa;
     beta bb;
     clrscr();
     cout << frifunc(aa, bb) << endl;
     getch();
     return 0;
     }
```

Sample Input and Output:

```
10
```

Program 2:

```
// Virtual Function
#include <iostream.h>
#include <conio.h>

class Base
{
public:
     virtual void show()
          {
          cout << "Base\n";
          }
};

class Derive1 : public Base
{
     public: void show()
          {
          cout << "Derived 1\n";
          }
};
```

```
class Derive2 : public Base
{
     public: void show()
          {
          cout << "Derived 2\n";
          }
};

int main()
     {
     Derive1 dv1;
     Derive2 dv2;
     Base* ptr;
     clrscr();
     ptr = &dv1;
     ptr->show();
     ptr = &dv2;
     ptr->show();
     getch();
     return 0;
 }
```

Sample Input and Output:

```
Derived 1
Derived 2
```

Result:

Thus, C++ programs to show how to implement friend function and virtual function have been written.

Ex. No. : 7 **Data Conversion Between Objects of Different Classes**
Date :

Aim:

To write a C++ program to convert data between objects of different classes.

Theoretical Concepts:

Type Conversion

Type Conversion refers to conversion from one type to another. The main idea behind type conversion is to make variable of one type compatible with variable of another type to perform an operation. In C++, there are two types of type conversion i.e. implicit type conversion and explicit type conversion.

Implicit Type Conversion

Implicit type conversion or automatic type conversion is done by the compiler on its own. There is no external trigger required by the user to typecast a variable from one type to another. This occurs when an expression contains variables of more than one type. So, in those scenarios automatic type conversion takes place to avoid loss of data. In automatic type conversion, all the data types present in the expression are converted to data type of the variable with the largest data type. Implicit conversions can lose information such as signs can be lost when signed type is implicitly converted to unsigned type and overflow can occur when long is implicitly converted to float. Below is the order of the automatic type conversion. You can also say, smallest to largest data type for type conversion.

char → short int → int → unsigned int → long → unsigned → long long → float → double → long double

Explicit Type Conversion

Explicit type conversion or type casting is user defined type conversion. In explicit type conversion, the user converts one type of variable to another type. Explicit type conversion can be done in two ways in C++:

1. Converting by assignment
2. Conversion using cast operator

Converting by assignment

In this type conversion the required type is explicitly defined in front of the expression in parenthesis. Data loss happens in explicit type casting. It is considered as forceful casting.

Conversion using cast operator

Cast operator is an unary operator which forces one data type to be converted into another data type. There are four types of casting in C++, i.e. static cast, dynamic cast, const cast and reinterpret cast.

Static Cast – This is the simplest type of cast which can be used. It not only performs upcasts, but also downcasts. It is a compile time cast. Checks are not performed during the runtime to guarantee that an object being converted is a full object of the destination type.

Dynamic Cast – It ensures that a result of the type conversion points to the valid, complete object of the destination pointer type.

Const Cast – manipulates that whether the object needs to be constant or non-constant. It ensures that either the constant needs to be set or to be removed.

Reinterpret Cast – converts any pointer type to any other pointer type, even of unrelated classes. It does not check if the pointer type and data pointed by the pointer is same or not.

Object Conversion

The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator where values of all data members of right hand object to the objects on the left hand side are equal. The objects in this case are of same data type. But if object are of different data types we must apply conversion rules for assignment.

There are three different types of situations that arise where data type conversions are between incompatible types.

1. Conversion from built in type to class type
    - Copy constructor is used (destination class).
2. Conversion from class type to built in type
    - Overload casting or assignment operator is used (source class).
3. Conversion from one class to another class type
    - Overload casting or assignment operator is used (source class).
    - Copy constructor is used (destination class).

Syntax of copy constructor

```
Class_name(const class_name &object_name)
{
// body of the conversion function.
}
```

Syntax for overload casting operator

```
operator typename()
{
  // body of the conversion function.
}
```

Program:

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

enum bool {false,true};

class time12
{
private:
     bool  pm;
     int  hrs;
     int mins;
public:
     time12() : pm(true), hrs(0), mins(0)
          { }

     time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
          { }

     void display()
          {
          cout << hrs << ":";
          if(mins < 10)
               cout << "0";
          cout << mins << " ";
          char *am_pm = pm ? "p.m." : "a.m.";
          cout << am_pm;
          }
};
```

```
class time24
{
private:
      int hours;
      int minutes;
      int seconds;
public:
      time24() : hours(0), minutes(0), seconds(0)
            { }

      time24(int h, int m, int s) : hours(h), minutes(m),
      seconds(s)
            { }

      void display()
            {
            if(hours < 10)
                  cout << "0";
            cout << hours << ":";
            if(minutes < 10)
                  cout << "0";
            cout << minutes << ":";
            if(seconds < 10)
                  cout << "0";
            cout << seconds;
            }
      operator time12();
};

time24::operator time12()
      {
      int hrs24 = hours;
      bool pm = hours < 12 ? false : true;
      //rounding minutes based on seconds
      int roundMins = seconds < 30 ? minutes : minutes+1;
      if(roundMins == 60)
            {
            roundMins=0;
            ++hrs24;
            if(hrs24 == 12 || hrs24 == 24) pm = (pm==true) ?
            false : true;
            }
      int hrs12 = (hrs24 < 13) ? hrs24 : hrs24-12;
```

```
        if(hrs12==0)
             {
             hrs12=12;
             pm=false;
             }
        return time12(pm, hrs12, roundMins);
        }

int main()
        {
        int h, m, s;
        clrscr();
        cout << "\n \t\t\t 24-hours time format \n";
        cout << " Enter the Hours (0 to 23): ";
        cin >> h;
        cout << " Enter the Minutes(0 to 59): ";
        cin >> m;
        cout << " Enter the Seconds(0 to 59): ";
        cin >> s;
        if(h > 23 || m > 59 || s >59 )
             { cout<<"\n Wrong input"; }
        else
             {
             time24 t24(h, m, s);
             cout << "\n You have entered: ";
             t24.display();
             time12 t12 = t24;
             cout << "\n\n 12-hours time format: ";
             t12.display();
             }
        getch();
        return 0;
        }
```

Sample Input and Output:

```
                     24-hours time format
 Enter the Hours (0 to 23): 23
 Enter the Minutes(0 to 59): 40
 Enter the Seconds(0 to 59): 32
 You have entered: 23:40:32
 12-hours time format: 11:41 p.m.
```

Result:

Thus, a C++ program has been written to convert data between objects of different classes.

Ex. No. : 8 **File Operations**
Date :

Aim:

To write a C++ program to perform file operations.

Theoretical Concepts:

File Operations

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream header file.

1. **ofstream**: Stream class to write on files
2. **ifstream**: Stream class to read from files
3. **fstream**: Stream class to both read and write from/to files.

Basic operations in File Handling:

1. Creating a file: `open()`
2. Reading data: `read()`
3. Writing new data: `write()`
4. Closing a file: `close()`

Creating a file

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating.

Syntax for file creation:

```
FilePointer.open("Path",ios::mode);
```

Mode flags and it‟s descriptions are

1. ios::app - Append mode. All output to that file to be appended to the end.
2. ios::ate - Open a file for output and move the read/write control to the end of the file.
3. ios::in - Open a file for reading.
4. ios::out - Open a file for writing.
5. ios::trunk - If the file already exists, its contents will be truncated before opening the file.
6. ios::nocreate - Opens the file only if it already exists
7. ios::noreplace - Opens the file only if it does not already exist
8. ios::binary - Opens the file in binary mode

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

Closing a file

Close a file is done by

Syntax:

```
FilePointer.close()
```

There are few important functions to be used with file streams like:

`tellp()` - It tells the current position of the put pointer.

Syntax:

```
filepointer.tellp()
```

`tellg()` - It tells the current position of the get pointer.

Syntax:

```
filepointer.tellg()
```

`seekp()` - It moves the put pointer to mentioned location.

Syntax:

```
filepointer.seekp(no of bytes,reference mode)
```

`seekg()` - It moves get pointer(input) to a specified location.

Syntax:

```
filepointer.seekg(no of bytes,reference point)
```

`put()` - It writes a single character to file.
`get()` - It reads a single character from file.
eof() - Returns true if a file open for reading has reached the end.

Note: For `seekp` and `seekg` three reference points are passed:
`ios::beg` - beginning of the file
`ios::cur` - current position in the file
`ios::end` - end of the file

Binary files

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...). File streams include two member functions specifically designed to input and output binary data sequentially: write and read. The first one (write) is a member function of ostream inherited by ofstream. And read is a member function of istream that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

```
write ( memory_block, size );
read ( memory_block, size );
```

where, memory_block is of type "pointer to char" (char*) and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

Program:

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

class person
{
protected:
     char name[80];
```

```
        int age;
    public:
        void getData()
            {
            cout << "\n Enter name: ";
            cin >> name;
            cout << "\n Enter age: ";
            cin >> age;
            }
        void showData()
            {
            cout << "\n Name: " << name;
            cout << "\n Age: " << age;
            }
    };

    int main()
        {
        char ch;
        person pers;
        fstream file;
        file.open("GROUP.DAT", ios::app | ios::out | ios::in |
        ios::binary );
        clrscr();
        do
            {
            cout << "\n\t\t\t Enter person\'s data";
            pers.getData();

            file.write( (char*)(&pers), sizeof(pers) );
            cout << "Enter another person (y/n)? ";
            cin >> ch;
            } while(ch=='y');

        file.seekg(0);

        file.read( (char*)(&pers), sizeof(pers) );
        while( !file.eof() )
            {
            cout << "\n Person:"; pers.showData();
            file.read( (char*)(&pers), sizeof(pers) );
            }
        getch();
        return 0;
        }
```

Sample Input and Output:

```
                    Enter person's data
      Enter name: R.Ragupathy

      Enter age: 42
      Enter another person (y/n)? y

                    Enter person's data
      Enter name: U.Anandh

      Enter age: 43
      Enter another person (y/n)? n

      Person:
      Name: R.Ragupathy
      Age: 42
      Person:
      Name: U.Anandh
      Age: 43
```

Result:

Thus, a C++ program has been written to perform file operations.

Ex. No. : 9 **Stack using Standard Template Library**
Date :

Aim:

To write a C++ program to make use of standard template library of stack to perform stack operations.

Theoretical Concepts:

Standard Template Library - Stack

Introduction

Stack is a data structure designed to operate in LIFO (Last in First out) context. In stack elements are inserted as well as get removed from only one end. Stack class is container adapter. Container is an object that holds data of same type. Stack can be created from different sequence containers. If container is not provided it uses default deque container. Container adapters do not support iterators therefore we cannot use them for data manipulation. However they support push() and pop() member functions for data insertion and removal respectively.

Definition

Below is definition of std::stack from <stack> header file

```
template <class T, class Container = deque<T> > class stack;
```

Parameters

T – Type of the element contained. T may be substituted by any other data type including user-defined type.
Container – Type of the underlying container object.

Member types

Following member types can be used as parameters or return type by member functions.

Member types and it"s definitions are

1. **value_type** - T (First parameter of the template)
2. **container_type** - Second parameter of the template
3. **size_type** - size_t

4. **reference** - value_type&
5. **const_reference** - const value_type&

Functions from <stack>

Below is list of all methods from <stack> header.

Constructors

Methods and it‟s descriptions are

1. **stack::stack** - default constructor : Constructs an empty stack object, with zero elements.
2. **stack::stack** - copy constructor : Constructs a stack with copy of each elements present in another stack.
3. **stack::stack** - move constructor : Constructs a stack with the contents of other using move semantics.

Destructor

stack::~stack - Destroys stack by deallocating container memory.

Member functions

Methods and it‟s descriptions are

1. **stack::emplace** - Constructs and inserts new element at the top of stack.
2. **stack::empty** - Tests whether stack is empty or not.
3. **stack::operator =** - copy version : Assigns new contents to the stack by replacing old ones.
4. **stack::operator =** - move version : Assigns new contents to the stack by replacing old ones.
5. **stack::pop** - Removes top element from the stack.
6. **stack::push** - copy version : Inserts new element at the top of the stack.
7. **stack::push** - move version : Inserts new element at the top of the stack.
8. **stack::size** - Returns the total number of elements present in the stack.
9. **stack::swap** - Exchanges the contents of stack with contents of another stack.
10. **stack::top** - Returns a reference to the topmost element of the stack.

Non-member overloaded functions

Methods and it‟s descriptions are

1. **Operator ==** - Tests whether two stacks are equal or not.
2. **Operator !=** - Tests whether two stacks are equal or not.

3. **Operator <** - Tests whether first stack is less than other or not.
4. **Operator <=** - Tests whether first stack is less than or equal to other or not.
5. **Operator >** - Tests whether first stack is greater than other or not.
6. **Operator >=** - Tests whether first stack is greater than or equal to other or not.
7. **Swap** - Exchanges the contents of two stack.

Program:

```
// Important Note
// To execute this program use gdb online C++ compiler
// available at https://www.onlinegdb.com/online_c++_compiler
//
#include <iostream>
#include <stack>

using namespace std;

int main(void)
     {
     stack<int> s;

     for (int i = 0; i < 5; ++i)
          s.push(i + 1);

     while (!s.empty())
          {
          cout << s.top() << endl;
          s.pop();
          }

     return 0;
     }
```

Sample Input and Output:

```
5
4
3
2
1
```

Result:

Thus, a C++ program has been written to make use of standard template library of stack to perform stack operations.

Ex. No. : 10 **Classes and Objects**
Date :

Aim:

To write Java programs to show how to create a class and how to create object.

Theoretical Concepts:

Classes

In Java, a class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access.
2. **Class name:** The name should begin with an initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class‟s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Syntax

```
Modifiers class ClassName extends SuperClassName implements
InterfaceName1,InterfaceName2, ..., InterfaceNamen {
   // variables
   // methods
}
```

Below Table summarizes the access modifiers.

Modifier	Class	Constructor	Method	Data/Variables
public	Yes	Yes	Yes	Yes
protected		Yes	Yes	Yes
default	Yes	Yes	Yes	Yes
private		Yes	Yes	Yes
static			Yes	
final	Yes		Yes	

Constructors are used for initializing new objects. Fields or variables provide the state of the class and its objects, and methods are used to implement the behaviour of the class and its objects. There are various types of classes that are used in real time applications such as nested classes, anonymous classes.

Objects

It is a basic unit of object oriented programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behaviour:** It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Definitions of Object

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behaviour.
- The object is an instance of a class.

There are three steps when creating an object from a class

1. **Declaration** – A variable declaration with a variable name with an object type.
2. **Instantiation** – The 'new' keyword is used to create the object.
3. **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

Syntax

```
className object = new className();
```

Source File Declaration Rules

These rules are essential when declaring classes, import statements and package statements in a source file.

1. There can be only one public class per source file.
2. A source file can have multiple non-public classes.
3. The public class name should be the name of the source file as well which should be appended by .java at the end. For example: the class name is public class Employee{} then the source file should be as Employee.java.
4. If the class is defined inside a package, then the package statement should be the first statement in the source file.
5. If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
6. Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Program 1:

```
// Important Note
// create Box class in a java file and name it as Box.java
// compile in prompt using the following command
// javac file_name i.e. javac Box.java
// if no errors run using following command
// java class_name i.e. java Box
// main method is present inside the same class

class Box {
      // member variables or instance variables
      double width;
      double height;
      double depth;

      // constructor
      Box(){
            width=10; height=20; depth=15;
            }
            //end of Box constructor

      // volume method
      void volume(){
            double vol;
            vol = width*height*depth;
            System.out.println("Volume  is "+vol);
            }
            // end of volume method
```

```
          // main method
          public static void main(String args[]) {
               Box mybox = new Box();
               mybox.volume();
               }
               // end of main method

          }
          // end of Box Class
```

Sample Input and Output 1:

Volume is 3000.0

Program 2:

```
     // Important Note
     // create Employee class in a java file and name it as
     // Employee.java

     public class Employee {

          String name;
          int age;
          String designation;
          double salary;

          // This is the constructor of the class Employee
          public Employee(String name) {
               this.name = name;
               }

          // Assign the age of the Employee to the variable age.
          public void empAge(int empAge) {
               age = empAge;
               }

          // Assign the designation to the variable designation.
          public void empDesignation(String empDesig) {
               designation = empDesig;
               }

          // Assign the salary to the variable salary.
          public void empSalary(double empSalary) {
               salary = empSalary;
               }
     // Print the Employee details
     public void printEmployee() {
          System.out.println("Name: "+ name );
          System.out.println("Age: " + age );
          System.out.println("Designation: " +designation );
          System.out.println("Salary: " + salary);
          }

     } // end of Employee class

// Important Note
// create EmployeeTest class in a java file and name it as
// EmployeeTest.java
// compile in prompt using the following command
// javac file_name i.e. javac EmployeeTest.java
// After successful compilation, you can find both
// Employee.class and EmployeeTest.class files in the folder.
// run using following command
// java class_name i.e. java EmployeeTest
// main method is present in the another class

public class EmployeeTest {

     public static void main(String args[]) {
          // Create two objects using constructor
          Employee empOne = new Employee("James Smith");
          Employee empTwo = new Employee("Mary Anne");

          // Invoking methods for each object created
          empOne.empAge(26);
          empOne.empDesignation("Senior Software Engineer");
          empOne.empSalary(1000);
          empOne.printEmployee();

          empTwo.empAge(21);
          empTwo.empDesignation("Software Engineer");
          empTwo.empSalary(500);
          empTwo.printEmployee();
          }
          // end of main method

     }
     //end of EmployeeTest class
```

Sample Input and Output 2:

```
Name: James Smith
Age: 26
Designation: Senior Software Engineer
Salary: 1000.0
Name: Mary Anne
Age: 21
Designation: Software Engineer
Salary: 500.0
```

Result:

Thus, Java programs have been written to show how to create a class and how to create object.

Ex. No. : 11 **String Manipulations**
Date :

Aim:

To write a Java program to perform string manipulations using string class.

Theoretical Concepts:

Java String

String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

There are two ways to create a String object:

1. **By string literal** : Java String literal is created by using double quotes.
   For Example: `String s="Welcome";`
2. **By new keyword** : Java String is created by using a keyword "new".
   For example: `String s=new String("Welcome");`

Summary of String Class Constructors

Constructor and Description
`String()` Initializes a newly created String object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new String by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new String by decoding the specified array of bytes using the specified charset.
`String(byte[] bytes, int offset, int length)` Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] bytes, int offset, int length, Charset charset)` Constructs a new String by decoding the specified subarray of bytes using the specified charset.
`String(byte[] bytes, int offset, int length, String charset)` Constructs a new String by decoding the specified subarray of bytes using the specified charset.

`String(byte[] bytes, String charsetName)`
Constructs a new String by decoding the specified array of bytes using the specified charset.

`String(char[] value)`
Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

`String(char[] value, int offset, int count)`
Allocates a new String that contains characters from a subarray of the character array argument.

`String(int[] codePoints, int offset, int count)`
Allocates a new String that contains characters from a subarray of the Unicode code point array argument.

`String(String original)`
Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

`String(StringBuffer buffer)`
Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

`String(StringBuilder builder)`
Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

Summary of String Class Methods

Modifier and Type	Method and Description
`char`	**`charAt(int index)`** Returns the char value at the specified index.
`int`	**`codePointAt(int index)`** Returns the character (Unicode code point) at the specified index.
`int`	**`codePointBefore(int index)`** Returns the character (Unicode code point) before the specified index.
`int`	**`codePointCount(int beginIndex, int endIndex)`** Returns the number of Unicode code points in the specified text range of this String.
`int`	**`compareTo(String anotherString)`** Compares two strings lexicographically.

`int`	**`compareToIgnoreCase(String str)`** Compares two strings lexicographically, ignoring case differences.
`String`	**`concat(String str)`** Concatenates the specified string to the end of this string.
`boolean`	**`contains(CharSequence s)`** Returns true if and only if this string contains the specified sequence of char values.
`boolean`	**`contentEquals(CharSequence cs)`** Compares this string to the specified CharSequence.
`boolean`	**`contentEquals(StringBuffer sb)`** Compares this string to the specified StringBuffer.
`static String`	**`copyValueOf(char[] data)`** Returns a String that represents the character sequence in the array specified.
`static String`	**`copyValueOf(char[] data, int offset, int count)`** Returns a String that represents the character sequence in the array specified.
`boolean`	**`endsWith(String suffix)`** Tests if this string ends with the specified suffix.
`boolean`	**`equals(Object anObject)`** Compares this string to the specified object.
`boolean`	**`equalsIgnoreCase(String anotherString)`** Compares this String to another String, ignoring case considerations.
`static String`	**`format(Locale l, String format, Object... args)`** Returns a formatted string using the specified locale, format string, and arguments.
`static String`	**`format(String format, Object... args)`** Returns a formatted string using the specified format string and arguments.
`byte[]`	**`getBytes()`** Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
`byte[]`	**`getBytes(Charset charset)`** Encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.

`byte[]`	**`getBytes(String charsetName)`** Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
`void`	**`getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`** Copies characters from this string into the destination character array.
`int`	**`hashCode()`** Returns a hash code for this string.
`int`	**`indexOf(int ch)`** Returns the index within this string of the first occurrence of the specified character.
`int`	**`indexOf(int ch, int fromIndex)`** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
`int`	**`indexOf(String str)`** Returns the index within this string of the first occurrence of the specified substring.
`int`	**`indexOf(String str, int fromIndex)`** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
`String`	**`intern()`** Returns a canonical representation for the string object.
`boolean`	**`isEmpty()`** Returns true if, and only if, length() is 0.
`int`	**`lastIndexOf(int ch)`** Returns the index within this string of the last occurrence of the specified character.
`int`	**`lastIndexOf(int ch, int fromIndex)`** Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
`int`	**`lastIndexOf(String str)`** Returns the index within this string of the last occurrence of the specified substring.
`int`	**`lastIndexOf(String str, int fromIndex)`** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

`int`	**`length()`** Returns the length of this string.
`boolean`	**`matches(String regex)`** Tells whether or not this string matches the given regular expression.
`int`	**`offsetByCodePoints(int index, int codePointOffset)`** Returns the index within this String that is offset from the given index by codePointOffset code points.
`boolean`	**`regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`** Tests if two string regions are equal.
`boolean`	**`regionMatches(int toffset, String other, int ooffset, int len)`** Tests if two string regions are equal.
`String`	**`replace(char oldChar, char newChar)`** Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String`	**`replace(CharSequence target, CharSequence replacement)`** Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String`	**`replaceAll(String regex, String replacement)`** Replaces each substring of this string that matches the given regular expression with the given replacement.
`String`	**`replaceFirst(String regex, String replacement)`** Replaces the first substring of this string that matches the given regular expression with the given replacement.
`String[]`	**`split(String regex)`** Splits this string around matches of the given regular expression.
`String[]`	**`split(String regex, int limit)`** Splits this string around matches of the given regular expression.
`boolean`	**`startsWith(String prefix)`** Tests if this string starts with the specified prefix.

`boolean`	**`startsWith(String prefix, int toffset)`** Tests if the substring of this string beginning at the specified index starts with the specified prefix.
`CharSequence`	**`subSequence(int beginIndex, int endIndex)`** Returns a new character sequence that is a sub sequence of this sequence.
`String`	**`substring(int beginIndex)`** Returns a new string that is a substring of this string.
`String`	**`substring(int beginIndex, int endIndex)`** Returns a new string that is a substring of this string.
`char[]`	**`toCharArray()`** Converts this string to a new character array.
`String`	**`toLowerCase()`** Converts all of the characters in this String to lower case using the rules of the default locale.
`String`	**`toLowerCase(Locale locale)`** Converts all of the characters in this String to lower case using the rules of the given Locale.
`String`	**`toString()`** This object (which is already a string!) is itself returned.
`String`	**`toUpperCase()`** Converts all of the characters in this String to upper case using the rules of the default locale.
`String`	**`toUpperCase(Locale locale)`** Converts all of the characters in this String to upper case using the rules of the given Locale.
`String`	**`trim()`** Returns a copy of the string, with leading and trailing whitespace omitted.
`static String`	**`valueOf(boolean b)`** Returns the string representation of the boolean argument.
`static String`	**`valueOf(char c)`** Returns the string representation of the char argument.
`static String`	**`valueOf(char[] data)`** Returns the string representation of the char array argument.

`static String`	**`valueOf(char[] data, int offset, int count)`** Returns the string representation of a specific subarray of the char array argument.
`static String`	**`valueOf(double d)`** Returns the string representation of the double argument.
`static String`	**`valueOf(float f)`** Returns the string representation of the float argument.
`static String`	**`valueOf(int i)`** Returns the string representation of the int argument.
`static String`	**`valueOf(long l)`** Returns the string representation of the long argument.
`static String`	**`valueOf(Object obj)`** Returns the string representation of the Object argument.

Program:

```
// Important Note
// create StringDemo class in a java file and name it as
// StringDemo.java
// compile in prompt using the following command
// javac file_name i.e. javac StringDemo.java
// After successful compilation, run using following command
// java class_name i.e. java StringDemo

class StringDemo {

     public static void main(String args[]) {
          String strOb1 = "Annamalai";
          String strOb2 = "University";
          String strOb3 = strOb1;

          // String Display
          System.out.println("String 1 is : " + strOb1);
          System.out.println("String 2 is : " + strOb2);
          System.out.println("String 3 is : " + strOb3);

          //String Length
          System.out.println
          ("Length of String 1 is : " + strOb1.length());
```

```
        //Character at position
        System.out.println
        ("Char at index 3 in String 1 is : " +
        strOb1.charAt(3));

        //String Comparison
        if(strOb1.equals(strOb2))
             System.out.println
             ("String 1 is equal to String 2");
        else
             System.out.println
             ("String 1 is not equal to String 2");

        if(strOb1.equals(strOb3))
             System.out.println
             ("String 1 is equal to  String 3");
        else
             System.out.println
             ("String 1 is not equal to  String 3");

        // Substring finding
        String substr;
        substr = strOb1.substring (4,9);
        System.out.println
        ("Susbstring at 4 to 9 in String 1 is : "+substr);

        // String reversal
        StringBuffer str=new StringBuffer("Annamalai");
        System.out.println
        ("Reversed string of " + str +" is : "
        + str.reverse());

        //String concatenation
        String s3;
        s3=strOb1.concat(strOb2);
        System.out.println
        ("Concatenated string of String 1"
        + " and Srting 2 is : " + s3);
        }
        // end of main method
}
// end of StringDemo class
```

Sample Input and Output:

```
String 1 is : Annamalai
String 2 is : University
String 3 is : Annamalai
Length of String 1 is : 9
Char at index 3 in String 1 is : a
String 1 is not equal to String 2
String 1 is equal to String 3
Susbstring at 4 to 9 in String 1 is : malai
Reversed string of Annamalai is : ialamannA
Concatenated string of String 1 and Srting 2 is : AnnamalaiUniversity
```

Result:

Thus, a Java program has been written to perform string manipulations using string class.

Ex. No. : 12 **Creation of Packages and Interfaces**
Date :

Aim:

To write Java programs to create a package and use it in a class, and create an interface and implement it in some classes.

Theoretical Concepts:

Packages

A package is a namespace that organizes a set of related classes and interfaces i.e. it is a group of similar types of classes, interfaces and sub-packages.

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.

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. are in the Java API library, which are free to use, included in the Java Development Environment. The library contains components for managing input, database programming, and much much more. The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package. To use a class or a package from the library, you need to use the import keyword:

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Creating a user defined package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for the code. The general form of the package statement is

Syntax

```
package pkg; // Single package

package pkg1[.pkg2[.pkg3]]; //Multileveled package
```

Here, `pkg` is the name of the package. A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as `package java.awt.image;` needs to be stored in java/awt/image.

How does the Java run-time system look for packages that you create?

1. The Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
2. You can specify a directory path or paths by setting the CLASSPATH environmental variable. For example, consider the following package specification.

   ```
   package MyPack;
   ```

   In order for a program to find MyPack, one of two things must be true. Either the program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.

Create a java file with package and name it as the public class name which contains main method. Compile in prompt using the following command.

```
javac –d . classname_with _main.java
```

After successful compilation, run using following command.

```
java userdefined_packagename.classname_with_main
```

Access Protection

Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses

The three access specifiers namely private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. But, a class has only two possible access levels: default and public. The following table is applicable only to members of classes for access.

	Private	**No modifier**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Interfaces

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. It is used to achieve abstraction and multiple inheritance in Java. It cannot be instantiated just like the abstract class. Since Java 8, we can have default and static methods in an interface. Since Java 9, we can have private methods in an interface. An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. A class can extend another class, an interface can extend another interface, but a class can implement an interface. A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extend portion of the declaration.

Syntax:

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract by default.
}
```

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks as follows.

Syntax:

```
access class classname [extends superclass] [implements interface
[,interface...]] {
// class-body
}
```

Here, `access` is either public or not used. Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma.

Program 1:

```
// Simple user defined Package creation
// create CustomerBalance class in a java file and name it as
// CustomerBalance.java
// For compilation and execution
// compile in prompt using the following command
// D:\MyJava>javac –d . CustomerBalance.java
// After successful compilation, run using following command
// D:\MyJava>java Mypackage. CustomerBalance

package Mypackage;

class Balance {
      String name;
      double bal;
      Balance(String n, double b) {
            name = n;
            bal = b;
            }

      void show() {
            if(bal<0)
                  System.out.print("***** ");
            System.out.println(name + ": $" + bal);
            }

}
```

```
public class CustomerBalance {

     public static void main(String args[]) {
          Balance current[] = new Balance[3];
          current[0] = new Balance("Raman", 123.23);
          current[1] = new Balance("Ravi", 157.02);
          current[2] = new Balance("Rani", -12.33);
          for(int i=0; i<3; i++)
          current[i].show();
          }

}
```

Sample Input and Output 1:

```
Raman: $123.23
Ravi: $157.02
 ***** Rani: $-12.33
```

Program 2:

```
//  User defined Package in directory structure
// create Balance class in a java file and name it as
// Balance.java and Store it in Mypackage folder
// create AccountBalance class in a java file and name it as
// AccountBalance.java
// For compilation and execution
// goto Mypackage folder using cd command as follows
// D:\MyJava>cd Mypackage
// compile Balance.java in prompt using the following command
// D:\MyJava\Mypackage>javac Balance.java
// execute cd.. command as follows to back to parent folder
// D:\MyJava\Mypackage>cd ..
// compile in prompt using the following command
// D:\MyJava>javac -cp d:\MyJava AccountBalance.java
// After successful compilation, run using following command
// D:\MyJava>java –cp d:\MyJava AccountBalance
// class path is indicated as cp in the above commands
// it can also be set as environment variable CLASSPATH
// Balance.java under the folder Mypackage

package Mypackage;

public class Balance {
     String name;
      double bal;
```

```
	public Balance(String n, double b) {
		name = n;
		bal = b;
		}

	public void show() {
		if(bal<0)
			System.out.print("***** ");
		System.out.println(name + ": $" + bal);
		}

}
// end of Balance.java

//AccountBalance.java

import Mypackage.*;

public class AccountBalance {

	public static void main(String args[]) {
		Balance current[] = new Balance[3];
		current[0] = new Balance("J. Fielding", 123.23);
		current[1] = new Balance("Will Tell", 157.02);
		current[2] = new Balance("Tom Jackson", -12.33);
		for(int i=0; i<3; i++)
			current[i].show();
		}

} // end of AccountBalance.java
```

Sample Input and Output 2:

```
J. Fielding: $123.23
Will Tell: $157.02
 ***** Tom Jackson: $-12.33
```

Program 3:

```
// interface creation and implementation
import java.lang.*;

interface shape{
	public void draw();
	public double getarea();
}
```

```
class circle implements shape{
     double radius;

     public circle(double r){
          this.radius=r;
          }

     public double getarea(){
          return Math.PI*radius*radius;
          }

     public void draw(){
          System.out.println("Drawing  circle");
          }

}
class rectangle implements shape{
     double width;
     double height;

     public rectangle (double w, double h){
          this.width = w;
          this.height = h;
          }

     public double getarea(){
          return width*height;
          }

     public void draw(){
          System.out.println("Drawing rectangle");
          }
}

public class testinterface{

     public static void main(String args[]){
          shape s= new circle(10);
          s.draw();
          System.out.println("Area = "+s.getarea());
          shape s2 = new rectangle(10,10);
          s2.draw();
          System.out.println("Area ="+s2.getarea());
          } //end of main method
} // end of testinterface class
```

Sample Input and Output 3:

```
Drawing  circle
Area = 314.1592653589793
Drawing rectangle
Area =100.0
```

Result:

Thus, Java programs have been written to create a package and use it in a class, and create an interface and implement it in some classes.

Ex. No. : 13 **Exception Handling**
Date :

Aim:

To write a Java program to handle divide by zero exception.

Theoretical Concepts:

Exception Handling

Exception

An exception is an event, which occurs during the execution of a program, which disrupts the normal flow of the program's instructions.

Errors

Errors are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Reasons

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Types of exceptions

There are two types of exceptions in Java:
1. Checked exceptions
2. Unchecked exceptions

Checked exceptions

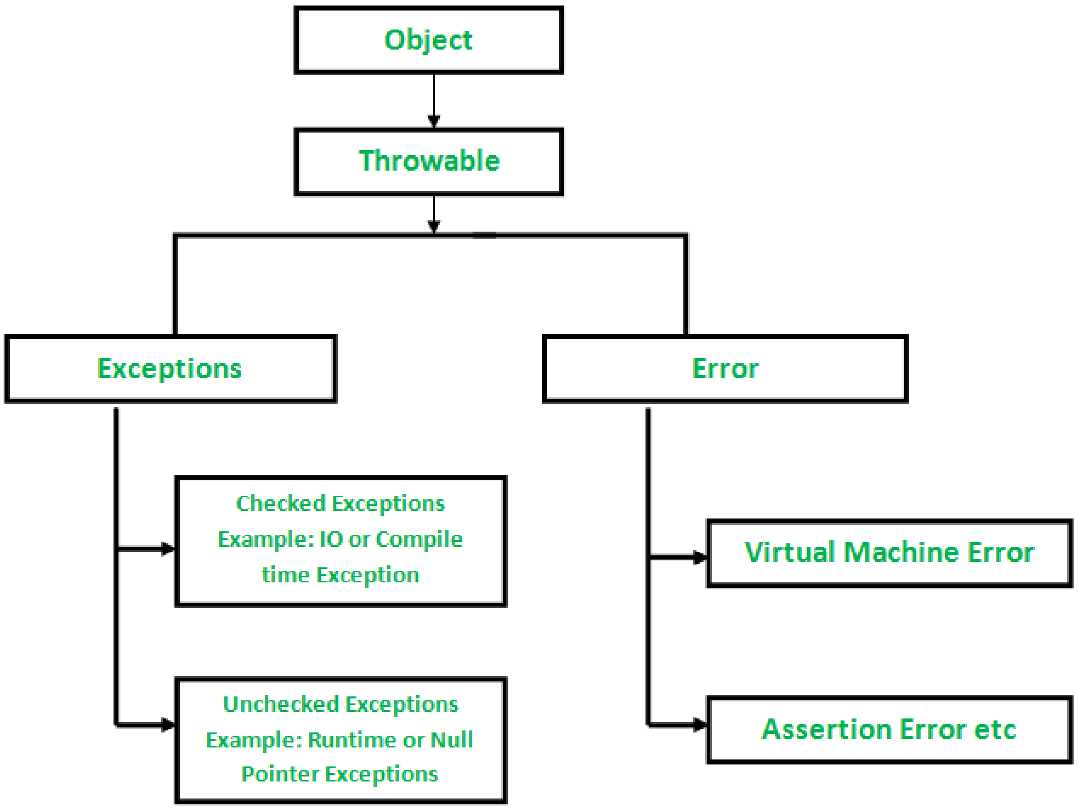
All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it‟s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException,NullPointerException, ArrayIndexOutOfBoundsException etc.

Exception Hierarchy

All exception and errors types are sub classes of class Throwable, which is base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error are used by the Java run-time system (JVM) to indicate errors having to do with the run-time environment itself (JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

1. **try** - The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means we can't use try block alone.

2. **catch** - The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

3. **finally** - The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.

4. **throw**- The "throw" keyword is used to throw an exception.

5. **throws** - The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may be an exception in the method. It is always used with method signature.

Syntax

```
try {

   // Protected code

} catch (ExceptionType1 e1) {

   // Catch block

} catch (ExceptionType2 e2) {

   // Catch block

} catch (ExceptionType3 e3) {

   // Catch block

} finally {

   // The finally block always executes.

}
```

Program:

```
class ZeroException {
    public static void main(String args[]) {
       int d, a;

       try {
           // monitor a block of code.
           d = 0;
           a = 42 / 3;
           System.out.println(" No error and a = " +a);
           a = 42 / d;
           System.out.println(" This will not be printed.");
           }
       catch (ArithmeticException e) {
           // catch divide-by-zero error
           System.out.println
                 (" User message is : Division by zero.");
           System.out.println
                 (" Actual message is : " + e.getMessage());
           }
       finally {
           System.out.println
                 (" Finally after catch statement.");
           }
       }
       //end of main method
    }
    // end of ZeroException class
```

Sample Input and Output:

```
No error and a = 14
User message is : Division by zero.
Actual message is : / by zero
Finally after catch statement.
```

Result:

Thus, a java program is written to handle divide by zero exception.

Ex. No. : 14 **Multithreading**
Date :

Aim:

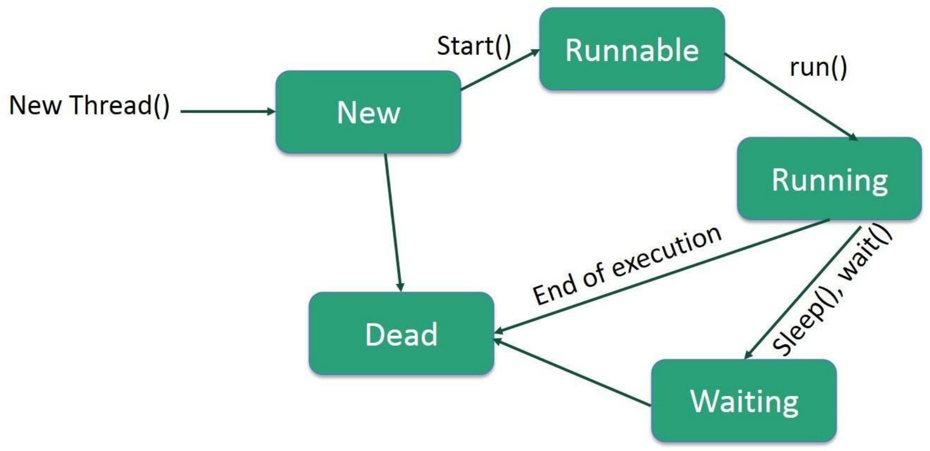
To write a Java program to display addition and multiplication table using multiple threads.

Theoretical Concepts:

Threads

A thread is a light-weight smallest part of a process that can run concurrently with the other parts (other threads) of the same process. Threads are independent because they all have separate path of execution that‟s the reason if an exception occurs in one thread, it doesn‟t affect the execution of other threads. All threads of a process share the common memory.

Life Cycle of a Thread



A thread can be in one of the following states:

1. **New** – A thread that has not yet started is in this state.
2. **Runnable** – A thread executing in the Java virtual machine is in this state.
3. **Blocked** – A thread that is blocked waiting for a monitor lock is in this state.
4. **Waiting** – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
5. **Timed waiting** – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
6. **Terminated** – A thread that has exited is in this state.

A thread can be in only one state at a given point in time.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5). Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Multithreading

Multithreading: It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

Multitasking: Ability to execute more than one task at the same time is known as multitasking.

Multiprocessing: It is same as multitasking, however in multiprocessing, more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

Parallel Processing: It refers to the utilization of multiple CPUs in a single computer system.

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the `java.lang.Thread` class. This class overrides the `run()` method available in the Thread class. A thread begins its life inside `run()` method. We create an object of our new class and call `start()` method to start the execution of a thread. `Start()` invokes the `run()` method on the Thread object.

Thread creation by implementing the Runnable Interface

We create a new class which implements `java.lang.Runnable` interface and override `run()` method. Then we instantiate a Thread object and call `start()` method on this object.

Thread Class vs Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn‟t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like `yield()`, `interrupt()` etc. that are not available in Runnable interface.

Program:

```
// Creation of multiple threads

class Add extends Thread {

    public void run(){
       System.out.println("Addition thread started");

       try {
          for(int i = 1; i<=5; i++) {
             int j = 5;
             System.out.println(i+" + " +j+" = "+(i+j));
             Thread.sleep(1000);
             }
          }
       catch (InterruptedException e) {
          System.out.println("Addition thread interrupted");
          }

       System.out.println("Addition thread terminated");
       }
       // end of run method

    }
    //end of Add class
```

```
class Multi extends Thread{

    public void run() {
       System.out.println
             ("\t\t\t\t Multiplication thread started ");

       try{
          for(int i = 1; i<=5; i++) {
              int j = 5;
              System.out.println
                    ("\t\t\t\t "+i+" X "+j+ " = "+(i*j));
              Thread.sleep(500);
              }
          }
       catch (InterruptedException e) {
          System.out.println
              ("\t\t\t\t Multiplication thread interrupted");
          }

       System.out.println
             ("\t\t\t\t Multiplication thread terminated");
       }
       // end of run method

    }
    // end of Multi class

class MultiThread{

      public static void main(String args[])
           {
           Add a = new Add();
           Multi m = new Multi();
           a.start();
           m.start();
           }
           // end of main method

      }
      //end of MultiThread class
```

Sample Input and Output:

```
Addition thread started
1 + 5 = 6
                                        Multiplication thread started
                                        1 X 5 = 5
                                        2 X 5 = 10
2 + 5 = 7
                                        3 X 5 = 15
                                        4 X 5 = 20
3 + 5 = 8
                                        5 X 5 = 25
                                        Multiplication thread terminated
4 + 5 = 9
5 + 5 = 10
Addition thread terminated
```

Result:

Thus, a Java program has been written to display addition and multiplication table using multiple threads.

Ex. No. : 15 **Abstract Window Toolkit**
Date :

Aim:

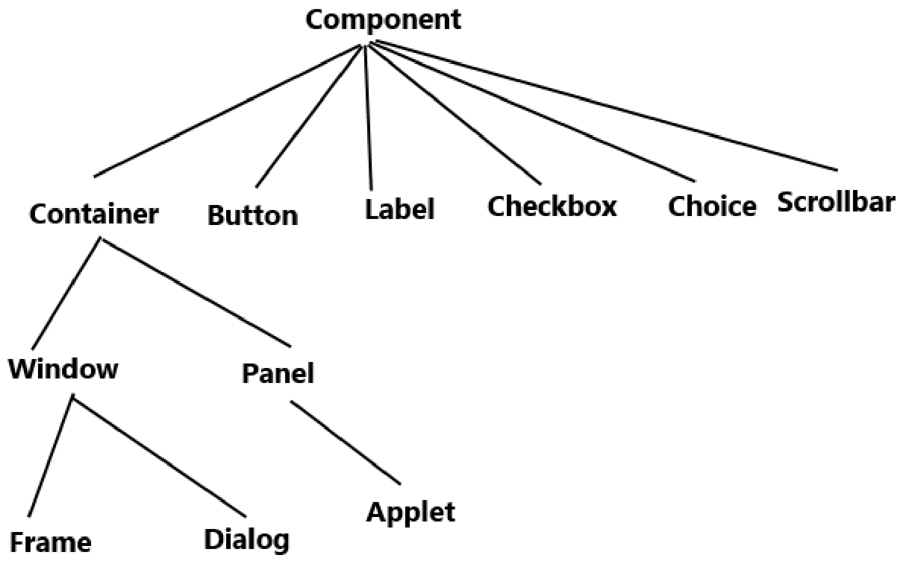
To write a Java program to show how to develop simple applications using AWT.

Theoretical Concepts:

Abstract Window Toolkit

Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java. Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS. The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

AWT class hierarchy



Container - The Container is a component in AWT that can contain other components like buttons, textfields, labels etc. The classes that extend Container class are known as container such as Frame, Dialog and Panel.

Window - The window is the container that has no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel - The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame - The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Dialog - The Dialog is the container that has border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.

Applet - Applet is a special type of container that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side. Applet class extends Panel.

Button - This class creates a labelled button.

Label - A Label object is a component for placing text in a container.

Check Box - A check box is a graphical component that can be in either an on (true) or off (false) state.

Choice - A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

Scroll Bar - A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

Check Box Group - The CheckboxGroup class is used to group the set of checkbox.

List- The List component presents the user with a scrolling list of text items.

Text Field - A TextField object is a text component that allows for the editing of a single line of text.

Text Area - A TextArea object is a text component that allows for the editing of a multiple lines of text.

Canvas - A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.

Image - An Image control is superclass for all image classes representing graphical images.

Methods of Component Class

Method	Description
public void add(Component c)	Inserts a component on this component.
public void setSize(int width,int height)	Sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	Defines the layout manager for the component.
public void setVisible(boolean status)	Changes the visibility of the component, by default false.

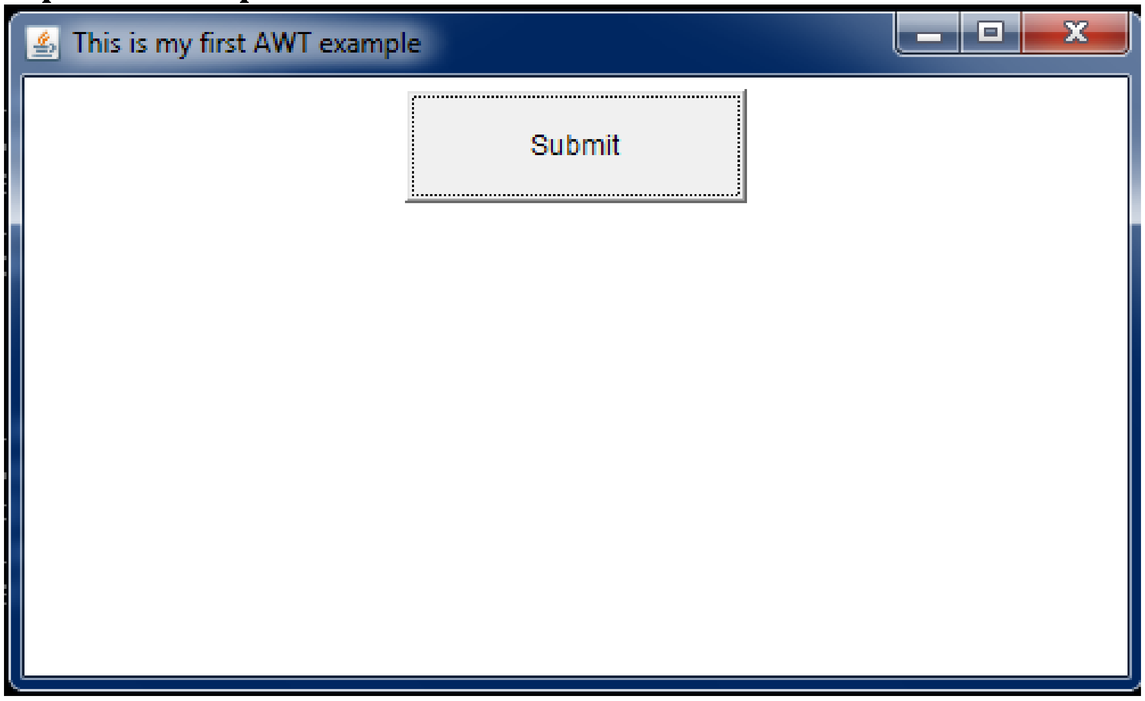
To create simple AWT application, you need a frame. There are two ways to create a frame in AWT.

1. By extending Frame class (inheritance)
2. By creating the object of Frame class (association)

Program 1:

```
// extended (inheritance) the Frame class
// thus class "SimpleExample1" would behave like a Frame

import java.awt.*;
import java.awt.event.*;

public class SimpleExample1 extends Frame{
     SimpleExample1(){
          // adding windows listener to close application
          addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                    dispose();
                    System.exit(0);
                    //calling the exit method is a must
                    }
               });

          Button b=new Button("Submit");

          // setting button position on screen
          b.setBounds(50,50,50,50);
          b.setPreferredSize(new Dimension(150, 50));

          //adding button into frame
          add(b);
          //Setting Frame width and height
          setSize(500,300);

          //Setting the title of Frame
          setTitle("This is my first AWT example");

          //Setting the layout for the Frame
          setLayout(new FlowLayout());

          // By default frame is not visible so set the
          // visibility to true to make it visible.
          setVisible(true);
          }
          // end of constructor

     public static void main(String args[]){
          // Creating the instance
          SimpleExample1 fr=new SimpleExample1();
          } // end of main method

     }// end of SimpleExample1 class
```

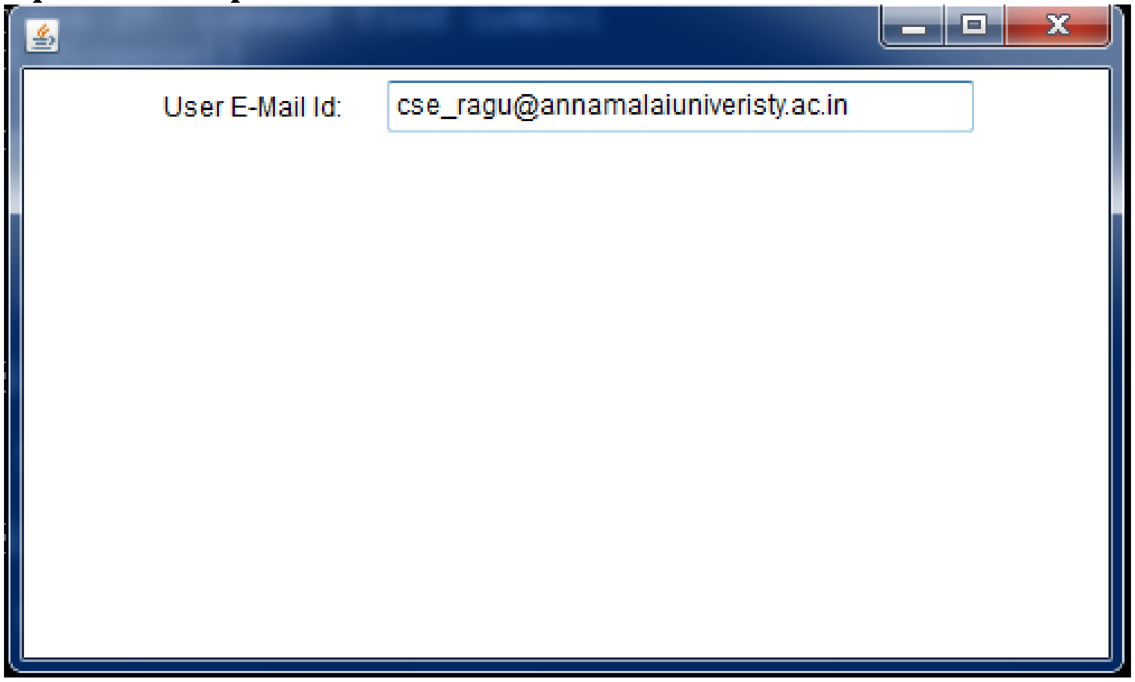
Sample Input and Output 1:



Program 2:

```
// creating the object of Frame class (association)

import java.awt.*;
import java.awt.event.*;

public class SimpleExample2 extends WindowAdapter{

      //Creating Frame
      Frame fr=new Frame();

      SimpleExample2() {
            // Add windows action listener to frame
            fr.addWindowListener(this);

            //Creating a label
            Label lb = new Label("User E-Mail Id: ");

            //adding label to the frame
            fr.add(lb);

            //Creating Text Field
            TextField t = new
            TextField("cse_ragu@annamalaiuniveristy.ac.in");

            //adding text field to the frame
            fr.add(t);

            //setting frame size
            fr.setSize(500, 300);

            //Setting the layout for the frame
            fr.setLayout(new FlowLayout());

            // By default frame is not visible so set the
            // visibility to true to make it visible.
            fr.setVisible(true);
            }

      public void windowClosing(WindowEvent e) {
            fr.dispose();
            System.exit(0);
            //calling the exit method is a must
            }
public static void main(String args[]) {
     SimpleExample2 ex = new SimpleExample2();
     }
     // end of main method
}
// end of SimpleExample2 class
```

Sample Input and Output 2:



User E-Mail Id: cse_ragu@annamalaiuniveristy.ac.in

Result:

Thus, Java programs have been written to show how to develop simple applications using AWT.