SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

21CSC101T

Object Oriented Design and Programming

UNIT-3

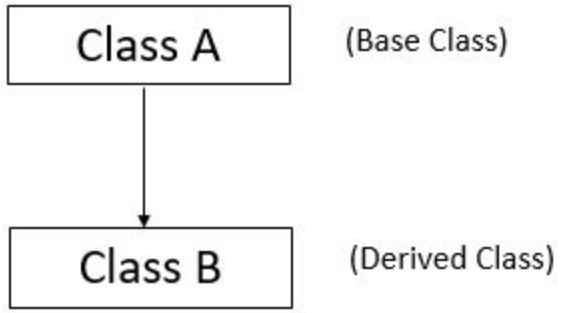
SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Unit 3 :

Single and Multiple Inheritance - Multilevel inheritance - Hierarchical - Hybrid Inheritance - Advanced Functions - Inline - Friend - Virtual -Overriding - Pure virtual function -Abstract class and Interface -UML State Chart Diagram - UML Activity Diagram

Inheritence

- The capability of a class to **derive properties and characteristics from another class** is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.



- When derived class inherits the base class, it means, ***the derived class inherits all the properties of the base class***, *without changing the properties of base class and may add new features to its own*.
- These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.
- **Sub Class:** The class that inherits properties from another class is called **Subclass** or **Derived Class or Child Class**
- **Super Class**: The class whose properties are inherited by a subclass is called **Base Class** or **Superclass or Parent Class.**

Advantages of Inheritence

- **Reusability**: inheritance helps the code to be reused in many situations.
- Using the concept of inheritance,the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.
- **Saves Time and Effort:**The above concept of reusability achieved by inheritance saves the programmer time and effort.The main code written can be reused in various situations as needed.

Modes of Inheritence

- The **visibility mode** (private, public or protected) in the definition of the derived class specifies whether the features of the base class are privately derived, publicly derived or protected derived.
- The visibility modes control the access-specifier to be for inheritable members of base-class, in the derived class.
  - Public Mode
  - Private Mode
  - Protected Mode

Publicly Visbility Modes of Inheritence

- Public Visibility mode gives the **least privacy** to the attributes of the base class.
- If the visibility mode is public, it means that the derived class can access the **public and protected members** of the base class, but not the private members of the base class.

Publicly Visbility Modes of Inheritence

```
#include<iostream>
using namespace std;
class base
{
int i,j;
public:
    void set(int a,int b)
        {
        i=a;
        j=b;
        }
    void show()
        {
        cout<<"Base Class Values:"<<i<<" "<<j<<"\n";
        }
};
class derived:public base
{
int k;
public:
    derived(int x)
        {
        k=x;
        }
    void showk(){
    cout<<"Derived Class Value :"<<k<<"\n";
    }
};
```

```
int main()
{
derived ob(7);
ob.set(1,3); //access member of base
ob.show(); //access member of base
ob.showk(); //uses member of derived class
}
```

```
Base Class Values:1 3
Derived Class Value :7
```

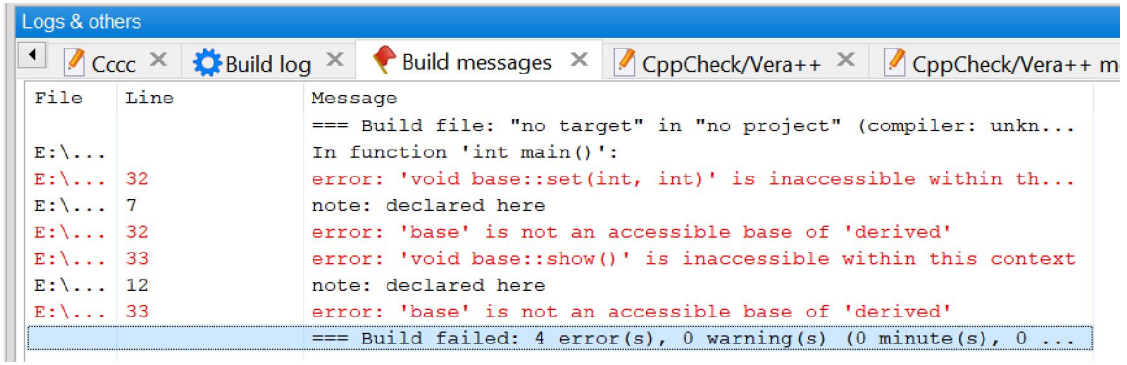
Privately Visbility Modes of Inheritence

- Public Visibility mode gives the **most privacy** to the attributes of the base class.
- If the visibility mode is private, that means that the derived class can **privately access the public and protected members of the base class.**

Privately Visbility Modes of Inheritence

```
#include<iostream>
using namespace std;
class base
{
int i,j;
public:
    void set(int a,int b)
        {
        i=a;
        j=b;
        }
    void show()
        {
        cout<<"Base Class Values:"<<i<<" "<<j<<"\n";
        }
};
class derived:private base
{
int k;
public:
    derived(int x)
        {
        k=x;
        }
    void showk(){
    cout<<"Derived Class Value :"<<k<<"\n";
    }
};
```

```
int main()
{
derived ob(7);
ob.set(1,3); //access member of base
ob.show(); //access member of base
ob.showk(); //uses member of derived class
}
```

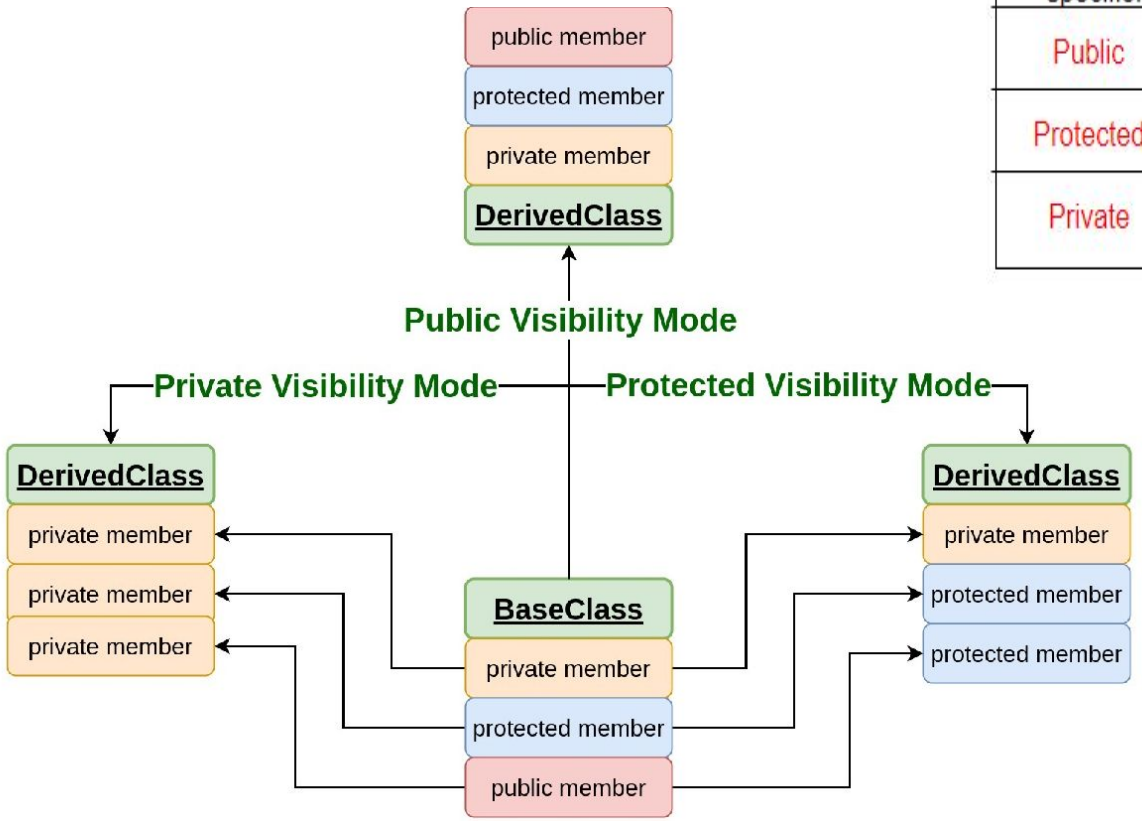


Protected Visbility Modes of Inheritence

- Protected visibility mode is somewhat between the public and private modes.
- If the visibility mode is protected, that means the derived class **can access the public and protected members of the base class protectively.**

Visbility Modes of Inheritence

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)



Types of Inheritance

Inheritance

01 Definition

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is

02 Syntax

```
class Subclass_name : access_mode Superclass_name
```

03 Types

Single Inheritance, Multiple Inheritance, Hierarchical Inheritance, Multilevel Inheritance, and Hybrid Inheritance (also known as Virtual Inheritance)

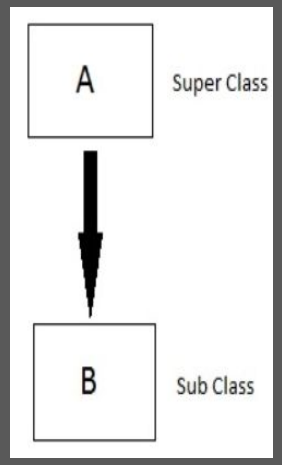
Note : All members of a class except Private, are inherited

04 Advantages

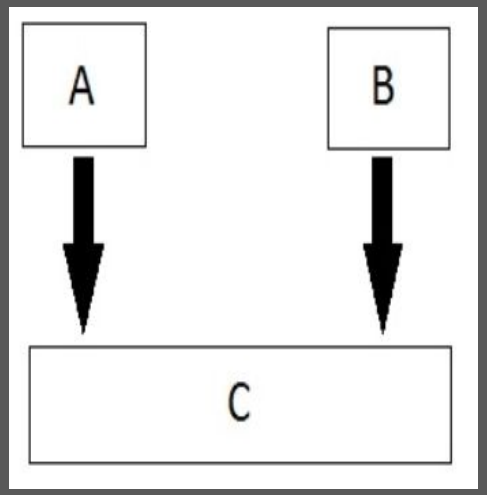
1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Inheritance Types

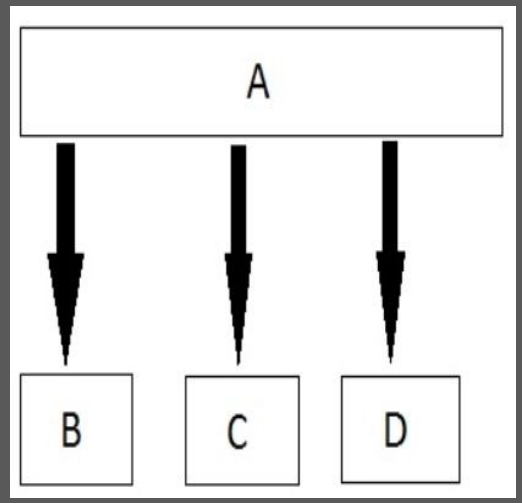
01 Single



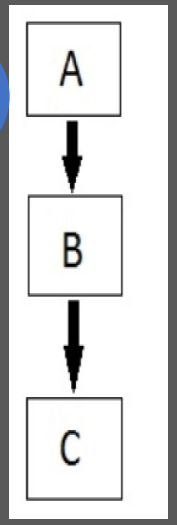
02 Multiple



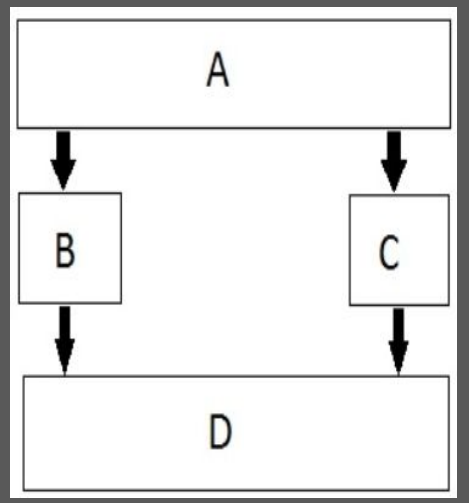
03 Hierarchical



04 Multilevel



05 Hybrid



Modes of Inheritance

01 Public

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

02 Protected

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

03 private

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

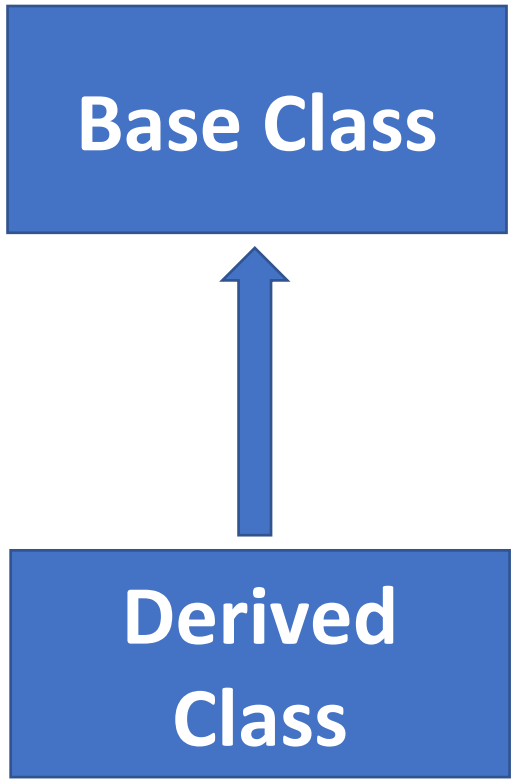
Note:

The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example

Inheritance Access Matrix

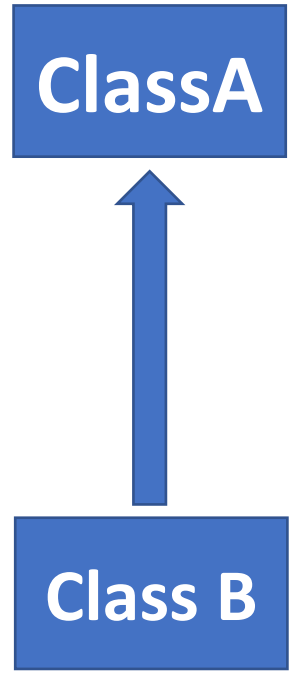
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

SINGLE INHERITANCE



Example

Base Class



Derived Class

Single Inheritance

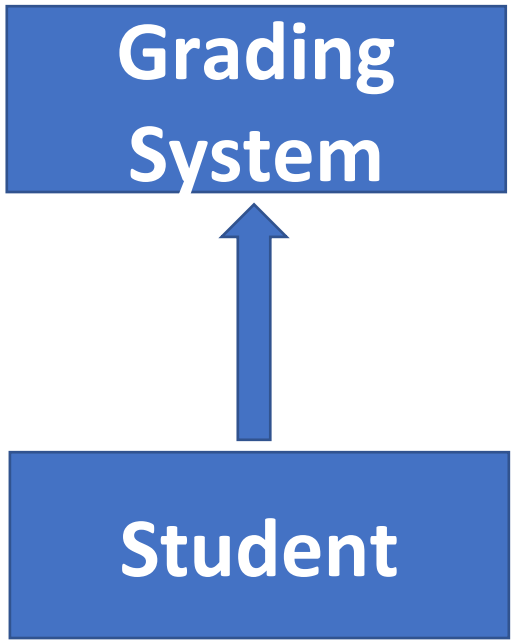
In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only. Based on the visibility mode used or access specifier used while deriving, the properties of the base class are derived. Access specifier can be private, protected or public.

Syntax:

```
class Classname   // base class
{
    ..........
};
class classname: access_specifier baseclassname
{
…
};
```

```
#include <iostream>
using namespace std;
class base    //single base class
{  public:
     int x;
       void getdata()
       {
        cout << "Enter the value of x = ";
        cin >> x;
        }
 };
class derived : public base    //single derived
{
    int y;
   public:
      void readdata()
      {
     cout << "Enter the value of y = ";
     cin >> y;
      }
```

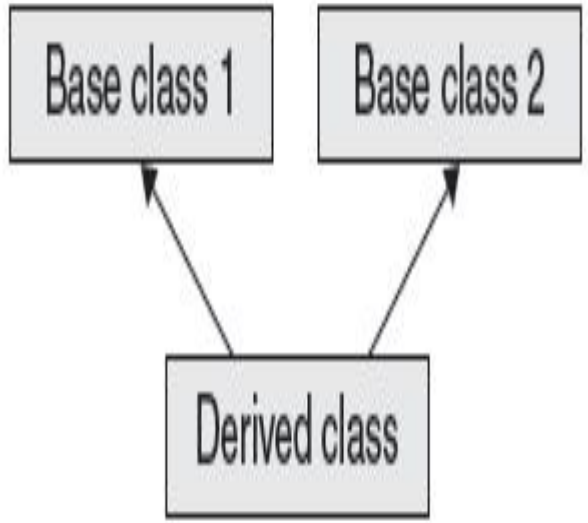
Applications of Single Inheritance



1. University Grading System
2. Employee and Salary

Multiple Inheritance

Base Class



Base Class

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.

Syntax:

```
class A   // base class
{
    ..........
};
class B
{
    ..........
}
class c : access_specifier A, access_specifier B   // derived class
{
    ..........
} ;
```

Example:

```
#include using namespace std;
class sum1
{
      protected: int n1;
};
class sum2
{
      protected: int n2;
};
class show : public sum1, public sum2
{
public: int total()
{
cout<<“enter n1”;
cin>>n1;


cout<<“enter n2”;
```

Applications of Multiple Inhertiance

- Distributed Database

MCQ Questions

- What are the things are inherited from the base class?

A. Constructor and its destructor

B. Operator=() members

C. Friends

D. All of the above

MCQ Questions

- What are the things are inherited from the base class?

A. Constructor and its destructor

B. Operator=() members

C. Friends

D. **All of the above**

MCQ Questions

```
using namespace std;
 class Base1 {
 public:
     Base1()
     { cout << " Base1" << endl;  }
};
class Base2 {
 public:
     Base2()
     { cout << "Base2" << endl;  }
};
  class Derived: public Base1, public Base2 {
   public:
     Derived()
     {  cout << "Derived" << endl;  }
};
```

```
int main()
{
   Derived d;
   return 0;
}
```

A. Compiler Dependent
B. Base1 Base2 Derived
C. Base2 Base1 Derived
D. Compiler Error

MCQ Questions

```
using namespace std;
 class Base1 {
 public:
     Base1()
     { cout << " Base1" << endl;  }
};
class Base2 {
 public:
     Base2()
     { cout << "Base2" << endl;  }
};
  class Derived: public Base1, public Base2 {
   public:
     Derived()
     {  cout << "Derived" << endl;  }
};
```

```
int main()
{
   Derived d;
   return 0;
}
```

A. Compiler Dependent

B**. Base1 Base2 Derived**

C. Base2 Base1 Derived

D. Compiler Error

MCQ Questions

class ABC is derived from Class X, class Y and class Z . This is _____ inheritance.

A. Multiple
B. Multilevel
C. Hierarchical
D. Single

MCQ Questions

class ABC is derived from Class X, class Y and class Z . This is _____ inheritance.

A. **Multiple**
B. Multilevel
C. Hierarchical
D. Single

Multilevel Inheritance

A derived class can be derived from another derived class. A child class can be the parent of another class.

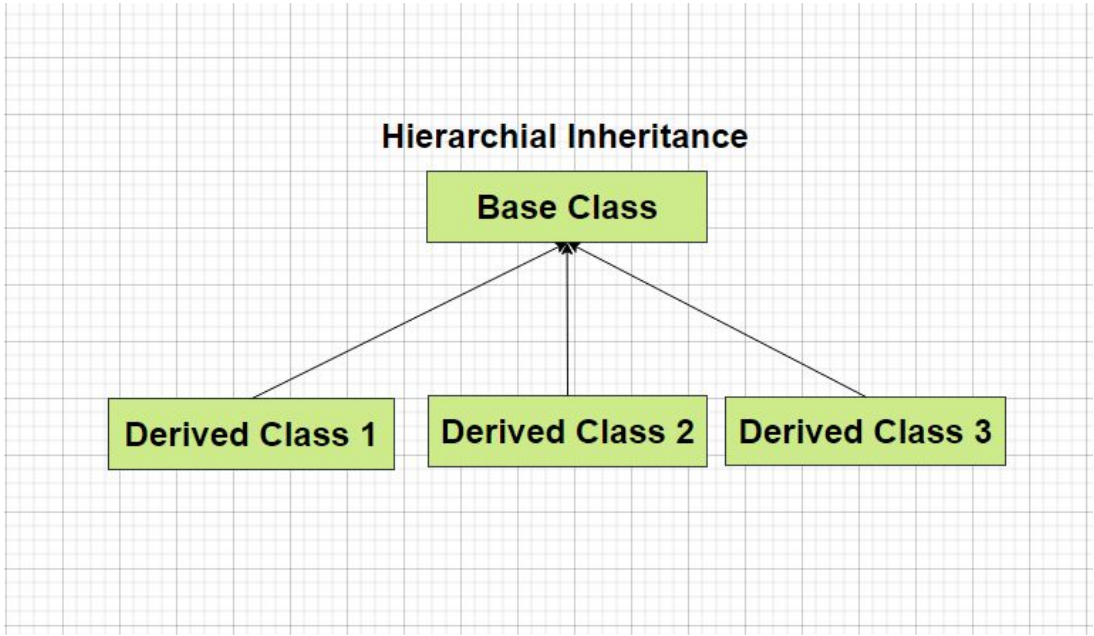
Syntax:

```
class A   // base class
{
   ..........
};
class B
{
    ..........
}
class C : access_specifier B
// derived class
{
   ...........
} ;
```

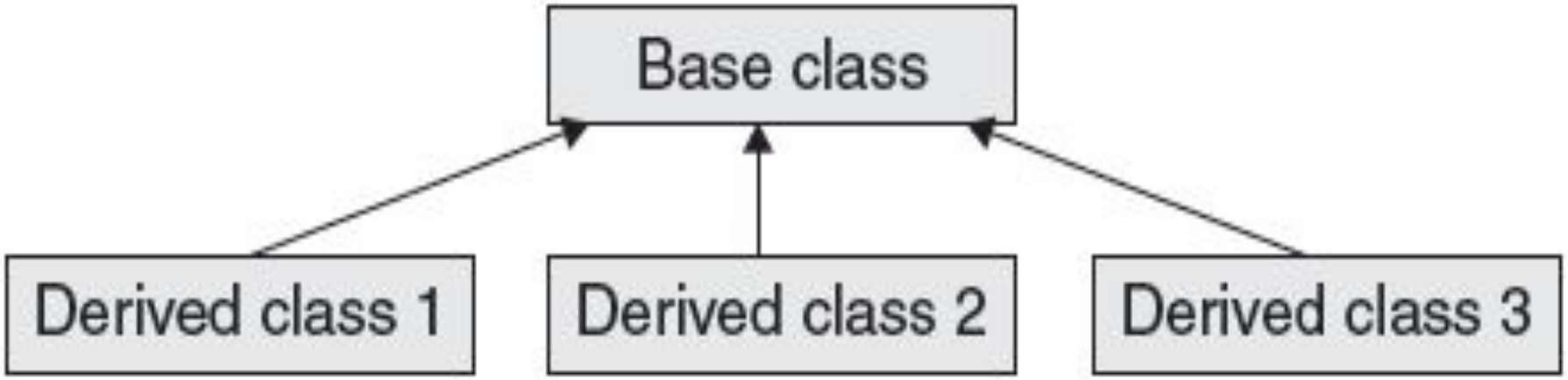
```
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle";
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
```

Hierarichal Inheritance

- In Hierarichal Inheritance we have several classes that are derived from a common base class (or parent class).
- Here in the diagram Class 1, Class 2 and Class 3 are derived from a common parent class called Base Class.



Hierarchical Inheritance



How to implement Hierarchal Inheritance in C++

```
class A {
 // Body of Class A
};  // Base Class

class B : access_specifier A
{
 // Body of Class B
}; // Derived Class

class C : access_specifier A
{
 // Body of Class C
}; // Derived Class
```

- In the example present in the left we have class A as a parent class and class B and class C that inherits some property of Class A.
- While performing inheritance it is necessary to specify the access_specifier which can be public, private or protected.

Example

```
#include <iostream>
using namespace std;
class A //single base class
{
    public:
        int x, y;
        void getdata()
        {
              cout << "\nEnter value of x and y:\n";
              cin >> x >> y;
        }
};
class B : public A //B is derived from class base
{
   public:
        void product()
        {
              cout << "\nProduct= " << x * y;
        }
};
```

```
class C : public A //C is also derived from
class base
{
   public:
        void sum()
        {
       cout << "\nSum= " << x + y;
        }
};
int main()
{
   B obj1;        //object of derived class B
   C obj2;        //object of derived class C
   obj1.getdata();
   obj1.product();
   obj2.getdata();
   obj2.sum();
   return 0;
}
```

Example of **Hierarchical Inheritance**

```
class Car {
  public:
    int wheels = 4;
    void show() {
      cout << "No of wheels : "<<wheels ;
    }
};

class Audi: public Vehicle {
  public:
    string brand = "Audi";
    string model = “A6”;
};
```

```
class Volkswagen: public Car {
  public:
    string brand = "Volkswagen";
    string model = “Beetle”;
};

class Honda: public Car {
  public:
    string brand = "Honda";
    string model = “City”;
};
```

MCQ

1. What is the minimum number of levels for a implementing multilevel inheritance?
   a) 1
   b) 2
   c) 3
   d) 4

Ans(C)

2. In multilevel inheritance one class inherits _____
a) Only one class
b) More than one class
c) At least one class
d) As many classes as required

Ans(a)

3. Can abstract classes be used in multilevel inheritance?
a) Yes, always
b) Yes, only one abstract class
c) No, abstract class doesn’t have constructors
d) No, never

Ans(a)

4. How many abstract classes can be used in multilevel inheritance?
a) Only 1
b) Only 2
c) At least one less than number of levels
d) Can’t be used

Ans(c)

5. Is it compulsory for all the classes in multilevel inheritance to have constructors defined explicitly if only last derived class object is created?

a) Yes, always

b) Yes, to initialize the members

c) No, it not necessary

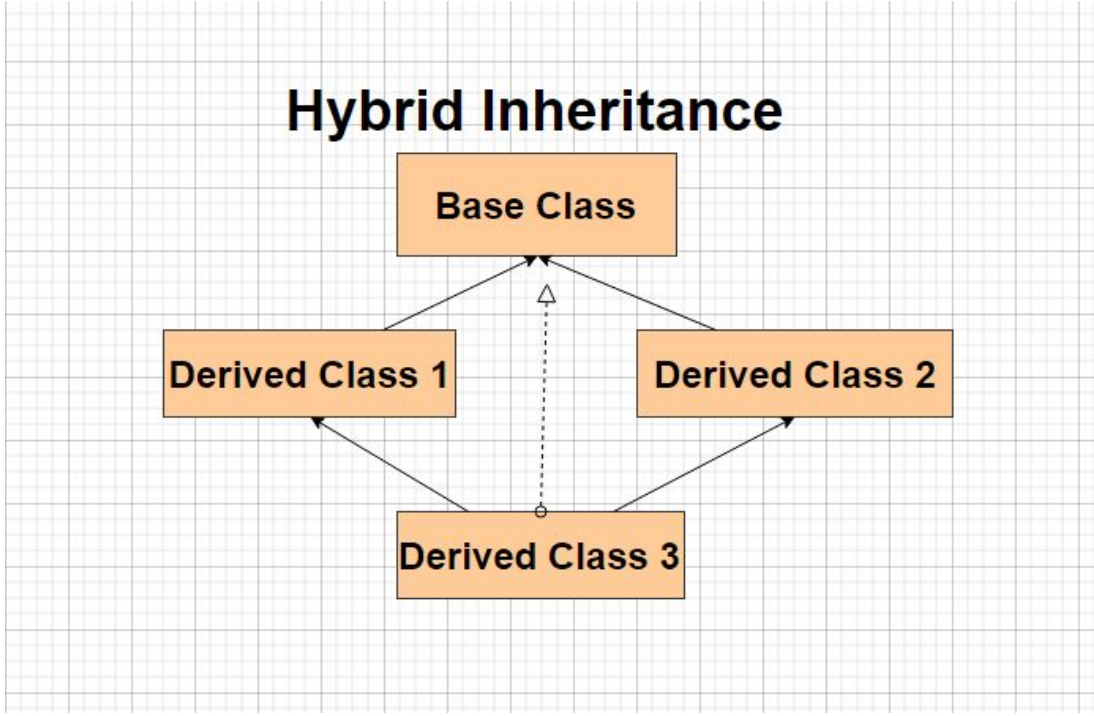
d) No, Constructor must not be defined

Ans(c)

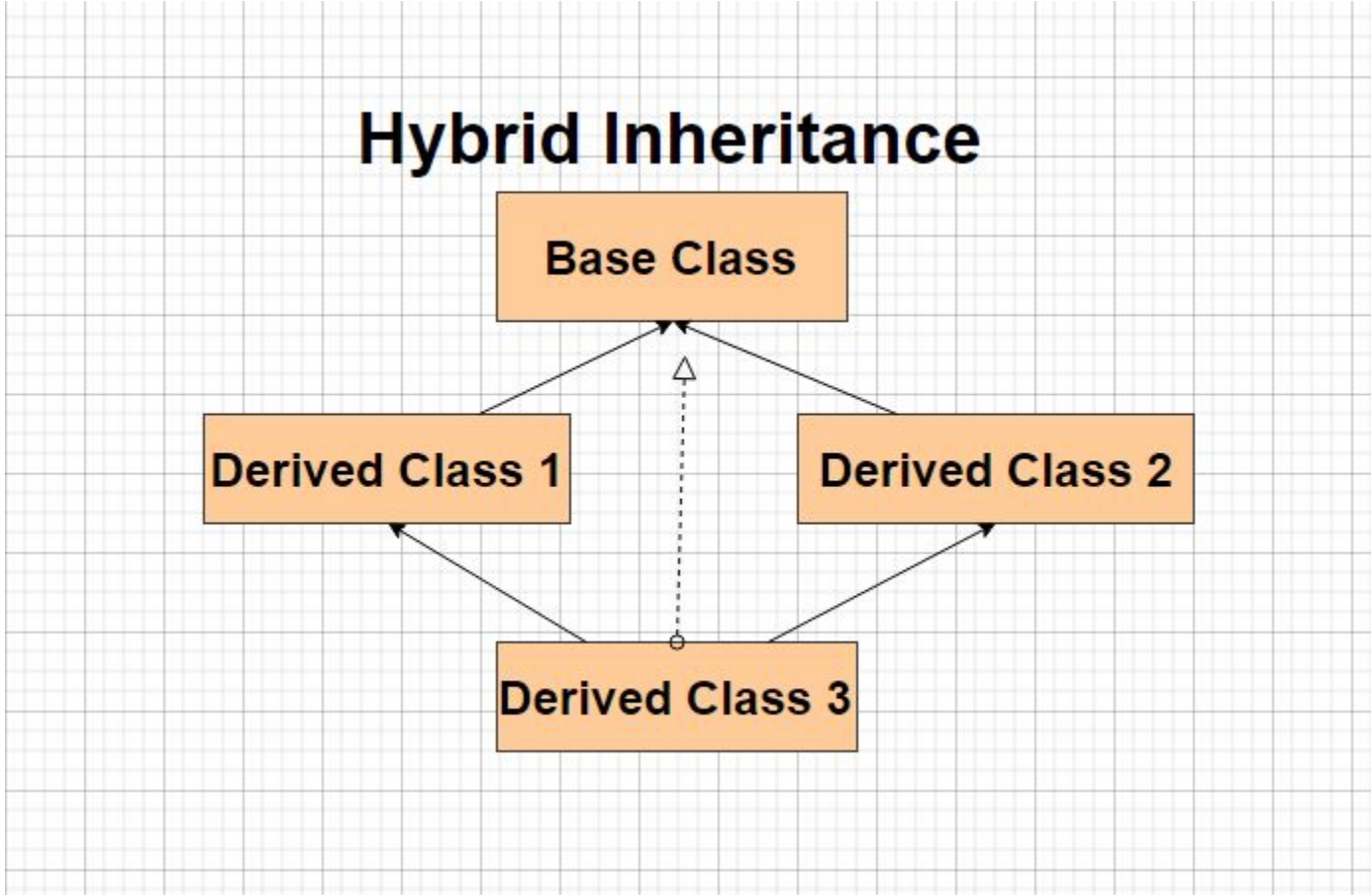
Hybrid Inheritance

Hybrid Inheritance

- Hybrid Inheritance involves derivation of more than one type of inheritance.
- Like in the given image we have a combination of hierarichal and multiple inheritance.
- Likewise we can have various combinations.



Diagramatic Representation of Hybrid Inheritance



How to implement Hybrid Inheritance in C++

```
class A
{
  // Class A body
};
class B : public A
{
    // Class B body
} ;
class C
{
  // Class C body
};
class D : public B, public C
{   // Class D body
};
```

- Hybrid Inheritance is no different than other type of inheritance.
- You have to specify the access specifier and the parent class in front of the derived class to implement hybrid inheritance.

Access Specifiers

In C++ we have basically three types of access specifiers :

- Public : Here members of the class are accessible outside the class as well.
- Private : Here members of the class are not accessible outside the class.
- Protected : Here the members cannot be accessed outside the class, but can be accessed in inherited classes.

Example of Hybrid Inheritance

```
class A
{
        public:
        int x;
};

class B : public A
{
        public:
        B()
        {
          x = 10;
        }
};
```

```
class C
{       public:
        int y;
        C()
        {
            y = 4;
        }
};
class D : public B, public C
{       public:
        void sum()
        {
            cout << "Sum= " << x + y;
        }
};
```

Order of Constructor Call

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Points to Remember

- Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.

- To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Example

```
class Base
{
   int x;
   public:
Base()
   {
cout<<"Base default constructor";
   }
};

class Derived : public Base
{
   int y;
   public:
Derived()
   {
cout<"Derived def. constructor";
   }
```

Order of Constructor Call

Example :

```
class Base
{
   int x;
   public:
   // parameterized constructor
   Base(int i)
   {
      x = i;
      cout<<"BaseParameterized Constructor\n";
   }
};

class Derived : public Base
{
   int y;
   public:
   // parameterized constructor
```

Note:

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

01 Concept

- Constructors and Destructors are never inherited and hence never overridden.
- Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class.

02 Static Function

- They are inherited into the derived class.
- If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden.
- Static Member functions can never be virtual.

03 Limitation

Derived class can inherit all base class methods except:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Calling base and derived class method using base reference

Example :

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

class Foo
{
public:
   int x;

   virtual void printStuff()
   {
 cout<<"BaseFoo                    printStuff called"<<endl;
   }
};

class Bar : public Foo
{
public:
   int y;

   void printStuff()
```

Calling base and derived class method using derived reference

Example :

```
#include <iostream>

class Base{
  public:
    void foo()
        {
            std::cout<<"base";
        }
};
class Derived : public Base
{
  public:
    void foo()
        {
            std::cout<<"derived";
        }
};
```

Advanced Functions: Inline, Friend, Virtual function and Overriding

Inline Member Function

Inline functions are used in C++ to reduce the overhead of a normal function call.

A member function that is both declared and defined in the class member list is called an inline member function.

The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

The advantages of using inline member functions are:

1. **The size of the object code is considerably reduced.**
2. **It increases the execution speed, and**
3. **The inline member function are compact function calls.**

Inline Member Function

Syntax:

```
class user_defined_name
{
        private:
                ------------
        public:
inline return_type function_name(parameters);
inline retrun_type function_name(parameters);
                ---------------
                ---------------
};
```

Syntax

```
Inline return_type function_name(parameters)
{
        ----------------
        ----------------
}
```

Inline function and classes

- It is also possible to define the inline function inside the class.

- All the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here.

- If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword

```
#include <iostream>
using namespace std;
class operation
{
   int a,b,add;

public:
   void get();
   void sum();
   };
inline void operation :: get()
{
   cout << "Enter first value:";
   cin >> a;
   cout << "Enter second value:";
   cin >> b;
}
inline void operation :: sum()
{
   add = a+b;
   cout << "Addition of two numbers: " << a+b << "\n";
}

int main()
{
   cout << "Program using inline function\n";
   operation s;
   s.get();
   s.sum();
     return 0;
}
```

Output:

Enter first value: 45 Enter second value: 15 Addition of two numbers: 60 Difference of two numbers: 30 Product of two numbers: 675 Division of two numbers: 3

Friend Function

1. The main concepts of the object oriented programming paradigm are data hiding and data encapsulation.

2. Whenever data variables are declared in a private category of a class, these members are restricted from accessing by non – member functions.

3. The private data values can be neither read nor written by non – member functions.

4. If any attempt is made directly to access these members, the compiler will display an error message as “inaccessible data type”.

5. The best way to access a private data member by a non – member function is to change a private data member to a public group.

6. When the private or protected data member is changed to a public category, it violates the whole concept or data hiding and data encapsulation.

7. To solve this problem, a friend function can be declared to have access to these data members.

8. Friend is a special mechanism for letting non – member functions access private data.

9. The keyword friend inform the compiler that it is not a member function of the class.

Friend Function

Granting Friendship to another Class

1. A class can have friendship with another class.

2. For Example, let there be two classes, first and second. If the class first grants its friendship with the other class second, then the private data members of the class first are permitted to be accessed by the public members of the class second. But on the other hand, the public member functions of the class first cannot access the private members of the class second.

01

Syntax

```
class second;forward declaration
class first
    {
        private:
            ----------
```

Friend Function

Two classes having the same Friend

1. A non – member function may have friendship with one or more classes.

2. When a function has declared to have friendship with more than one class, the friend classes should have forward declaration.

3. It implies that it needs to access the private members of both classes.

01

Syntax

friend return_type function_name(parameters);

02

Example

friend return_type fname(first one, second two)
{}

03

Note:

where friend is a keyword used as a function modifier. A friend declaration is valid only within or outside the class definition.

Friend Function

Syntax:

```
class second;forward declaration
    class first
    {
        private:
            ----------
        public:
            friend return_type fname(first one,
second two);
    };
    class second
    {
        private:
            ----------
        public:
            friend return_type fname(first one,
second two);
    };
```

Case 2:

```
class sample
{
    private:
        int x;
        float y;
    public:
        virtual void display();
        virtual static int sum();     //error
}
int sample::sum()
{ }
```

Friend Function Example

Example:

```
class sample
{
	private:
		int x;
	public:
		void getdata();
		friend void display(sample abc);
};
void sample::getdata()
{
	cout<<"Enter a value for x\n"<<endl;
	cin>>x;
}
void display(sample abc)
{
	cout<<"Entered Number is  "<<abc.x<<endl;
}

void main()
{
```

Friend Function Example

Example:

```
class first
{
	friend class second;
	private:
		int x;
	public:
		void getdata();
};

class second
{
	public:
		void disp(first temp);
};

void first::getdata()
{
	cout<<"Enter a Number ?"<<endl;
	cin>>x;
}
```

Friend Function Example

```
class second;//Forward Declaration
class first
{
	private:
		int x;
	public:
void getdata();
void display();
friend int sum(first one,second two);
};
class second
{
	private:
		int y;
	public:
		void getdata();
		void display();
friend int sum(first one,second two);
};
void first::getdata()
{
```

Virtual function

- **Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform Late Binding on this function.**
- **Virtual Keyword is used to make a member function of the base class Virtual. Virtual functions allow the most specific version of a member function in an inheritance hierarchy to be selected for execution. Virtual functions make polymorphism possible.**

Key:

- **Only the Base class Method's declaration needs the Virtual Keyword, not the definition.**
- **If a function is declared as virtual in the base class, it will be virtual in all its derived classes.**

Syntax

01

virtual return_type function_name (arg);

Example

02

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

Note:

03

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

Virtual function features

Case 1:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual void display();
                virtual int sum();
}
virtual void sample::display()        //Error
{ }
```

Case 2:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual void display();
                virtual static int sum(); //error
}
int sample::sum()
{ }
```

Virtual function features

Case 3:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual sample(int x,float y);
//constructor
                void display();
                int sum();
}
```

Case 4:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual ~sample(int x,float y);
//invalid
                void display();
                int sum();
}
```

Virtual function features

Case 5:

```
class sample_1
{
        private:
                int x;
                float y;
        public:
                virtual int sum(int x,float y);     //error
};
class sample_2:public sample_1
{
        private:
                int z;
        public:
                virtual float sum(int xx,float yy);
};
```

Case 6:

```
virtual void display()    //Error, non member
function
{
        ----------------
        ----------------
}
```

Virtual Function Example

Example:

```
class Point
{
protected:
        float length,breath,side,radius,area,height;
};
class Shape: public Point
{
        public:
                virtual void getdata()=0;
                virtual void display()=0;
};
class Rectangle:public Shape
{
public:
        void getdata()
        {
                cout<<"Enter the Breadth Value:"<<endl;
                cin>>breath;
                cout<<"Enter the Length Value:"<<endl;
                cin>>length;
        }
        void display()
        {
                area = length * breath;
                cout<<"The Area of the Rectangle is:"<<area<<endl;
        }
};
 class Square:public Shape
{
public:
void getdata()
{
        cout<<"Enter the Value of the Side of the Box:"<<endl;
```

Difference in invocation for virtual and non virtual function

Example:

```
class Base
{    public:
   void show()
   {
      cout << "Base class";
   }
};
class Derived:public Base
{    public:
   void show()
   {
      cout << "Derived Class";
   }
}
int main()
{
   Base* b;      //Base class pointer
   Derived d;    //Derived class object
   b = &d;
   b->show();    //Early Binding Occurs
}
```

```
class Base
{    public:
   virtual void show()
   {
```

Difference in invocation for virtual and non virtual function

Example:

```
#include<iostream>
using namespace std;
class Base {
public:
  void foo()
  {
        std::cout << "Base::foo\n";
  }
  virtual void bar()
  {
        std::cout << "Base::bar\n";
  }
};

class Derived : public Base {
public:
  void foo()
  {
        std::cout << "Derived::foo\n";
  }
  virtual void bar()
  {
        std::cout << "Derived::bar\n";
```

Override

Example:

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

class Base {
public:

   // user wants to override this in the derived class
   virtual void func()
   {
      cout << "I am in base" << endl;
   }
};

class derived : public Base {
public:

   // did a mistake by putting   an argument "int a"

   void func(int a) override
   {
      cout << "I am in derived class" << endl;
   }
};
```

Pure Virtual function

- Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function.
- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.
- Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Syntax

01

Example

02

Note:

03

Pure Virtual function

Example:

```
class pet
{
private:
        char name[5];
public:
virtual void getdata()=0;
virtual void display()=0;
};
class fish:public pet
{
private:
        char environment[10];
        char food[10];
public:
        void getdata();
        void display();
};
class dog: public pet
{
        private:
                char environment[10];
                char food[10];
        public:
                void getdata();
```

Pure Virtual function

Example:

```
void dog::getdata()
{
        cout<<"Enter the Dog Environment required"<<endl;
        cin>>environment;
        cout<<"Enter the Dog Food require"<<endl;
        cin>>food;
}
void dog::display()
{
        cout<<"Dog Environment="<<environment<<endl;
        cout<<"Dog Food="<<food<<endl;
        cout<<"-------------------------------------"<<endl;
}
void main()
{
        pet *ptr;
        fish f;
        ptr=&f;
        ptr->getdata();
        ptr->display();
        dog d;
        ptr=&d;
        ptr->getdata();
        ptr->display();
```

Abstract Class

- **Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.**
- **Abstract class can have normal functions and variables along with a pure virtual function.**
- **Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.**
- **Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.**

Pure virtual function

Example:

```
/Abstract base class
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
   public:
   void show()
   {
      cout << "Implementation of Virtual Function in Derived class\n";
   }
};




int main()
{
   Base obj;   //Compile Time Error
```

Abstract Class

Abstract Class

- Abstract Class is a class which contains atleast one Pure Virtual function(abstract method) in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Abstract Class

Pure Virtual Functions

- Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function,

```
virtual void funname() = 0;
```

Example 1

```
class Base //Abstract base class
{
public:
virtual void show() = 0; //Pure Virtual Function
};
class Derived:public Base
{
public:
void show()
{
cout <<“Implementation of Virtual Function in Derived class”;
}
};
```

```
int main()
{
Base obj; //Compile Time Error
Base *b;
Derived d;
b = &d;
b->show();
}
```

Output : Implementation of Virtual Function in Derived class

In the above example Base class is abstract, with pure virtual show() function, hence we cannot create object of base class.

Example2

```
// C++ program to calculate the area of a square and a circle
#include <iostream>
using namespace std;
// Abstract class
class Shape {
  protected:
   float dimension;
  public:
   void getDimension() {
      cin >> dimension;
   }
// pure virtual Function
   virtual float calculateArea() = 0;
};
```

```
// Derived class
class Square : public Shape {
  public:
float calculateArea() {
  return dimension * dimension; }
};
// Derived class
class Circle : public Shape {
  public:
   float calculateArea() {
      return 3.14 * dimension * dimension;
   }
};
```

```
int main() {
   Square square;
   Circle circle;
   cout << "Enter the length of the square: ";
   square.getDimension();
   cout << "Area of square: " <<
              square.calculateArea() << endl;
  cout << "\nEnter radius of the circle: ";
   circle.getDimension();
   cout << "Area of circle: " <<
              circle.calculateArea() << endl;
  return 0;
}
```

Output

Enter the length of the square: 4

Area of square: 16

Enter radius of the circle: 5

Area of circle: 78.5

What will be the output of the following program?

```
#include <iostream>
#include <string>
using namespace std;
class Exam
{
   int a;
   public:
       virtual void score() = 0;
};
class FirstSemExam: public Exam
{
   public:
      void score(){
            cout<<“First Semester Exam"<<endl;
     }
};
 int main(int argc, char const *argv[])
{
    Exam e;
    e.score();
    return 0;
}
```

a) Class B
b) Error
c) Segmentation fault
d) No output

For abstract class Exam object cannot be instantiated

Quiz

1. Which class supports run-time polymorphism?

   a. Base class b. abstract class c. derived class d. subclass

2. Identify the syntax for pure virtual function

   a. void area();

   b. void area()=0;

   c. void area()=1;

   d. pure void area();

3. Can we create object for abstract class? State Yes/No.

No

4. Identify the incorrect statement with respect to abstract class

   a. all functions must be pure virtual function

   b. Incomplete type

   c. foundation for derived class

   d. supports run-time polymorphism

5. If there is an abstract method in a class then, _____

   a. Class may or may not be abstract class

   b. Class is generic

   c. Class must be abstract class

   d. Class must be public

Practice Questions

1.Given an abstract base class Car with member variables and functions:

String name;

- //abstract methods

Price()

Discount()

totalseats()

You have to write a class Tata which extends the class Car and uses the member functions and variables to implement price, discount of 20% and total seats.

2. Create an abstract class 'Bank' with an abstract method 'getBalance'. $100, $150 and $200 are deposited in banks A, B and C respectively. 'BankA', 'BankB' and 'BankC' are subclasses of class 'Bank', each having a method named 'getBalance'. Call this method by creating an object of each of the three classes.

Interface

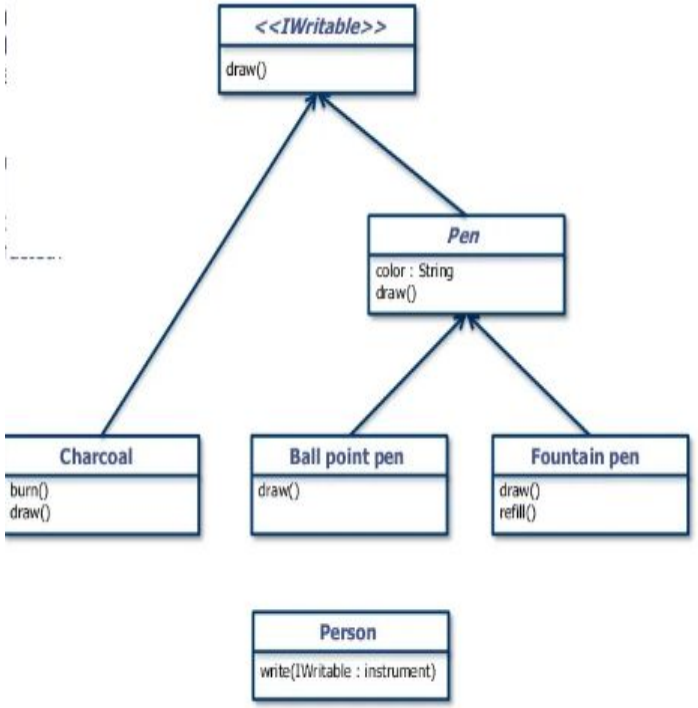
- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function.
- A pure virtual function is specified by placing "= 0" in its declaration

- Every method declared by an object specifies the method's name, the object/value it takes as parameters, and the method's return valuew. this is known as the operation **Signature**
- The set of all signatures defined by an object's methods is called the **interface** to the object .An object's interface characterizes the complete set of requests that can be sent to the object.
- An interface is like an abstract class that cannot be instantiated,
- Interface are better suited to situations in which your applications require many possibly unrelated object types to provide certain functionality
- Explicitly implementing interface in a class enables us to define a set of methods that are mandatory for that class.
- Interface definition begins with the keyword **interface**

Interface Example

- Fountain pen and ball point pen are used for writing, they are related entities. What should be the abstract class for these entities ?

- We can Write using pen but we can also write with charcoal. Over here what is the common behaviour between pen and charcoal that can be abstracted?



```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

When to use Interface?

- Use an interface when an immutable contract is really intended .
- Interfaces are better suited in situations when your applications require many possibly unrelated object types to provide certain functionality.
- Interfaces are better in situations in which you do not need to inherit implementation from a base class.
- Interfaces can be used for multiple inheritance.

Difference between abstract class and interface

Feature	Interface	Abstract class
Multiple Inheritance	A class may implement several interfaces.	A class may inherit only one abstract class.
Default implementation	An interface is purely abstract, it cannot provide any code, just the signature.	An abstract class can provide complete, default code and/or just the details that have to be overridden.
Access modifiers	An interface cannot have access modifiers for the method, properties etc. Everything is assumed as public.	An abstract class can contain access modifiers for the methods, properties etc.
Core vs. Peripheral	Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface.	An abstract class defines the core identity of a class and there it is used for related objects.
Homogeneity	If various implementations only share method signatures then it is better to use Interfaces.	If various implementations are of the same kind and use common behavior or status then abstract class is better to use.

Difference between abstract class and interface

Feature	Interface	Abstract class
Adding functionality (Versioning)	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.	If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.
Fields and Constants	No fields can be defined in interfaces	An abstract class can have fields and constants defined

What is similar to interface in c++

A. methods

B. instance of class

C. pure abstract class

D. friend function

Which of the following is used to implement the c++ interfaces?

- A. absolute variables
- **B. abstract classes**
- C. Constant variables
- D. default variables

Practice Questions- MCQ

Which of the following Combines two concurrent activities and re-introduces them to a flow where only one activity can be performed at a time?

A. Joint symbol

B. Fork symbol

C. Decision symbol

D. Note symbol

Ans: A

State chart Diagram

State diagram

- A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioural** diagram and it represents the behaviour using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams**.

Uses of state chart diagram

- State chart diagrams are useful to model reactive systems

  -Reactive systems can be defined as a system that responds to external or internal events.

- State chart diagram describes the flow of control from one state to another state.

Purpose

Following are the main purposes of using State chart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

Difference between state diagram and flowchart

- The basic purpose of a **state diagram** is to portray various changes in state of the class and not the processes or commands causing the changes.

- However, a **flowchart** on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

When to use State charts

- So the main usages can be described as:
- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- Forward and reverse engineering.

How to draw state charts

Before drawing a State chart diagram we must have clarified the following points:

- ✔ Identify important objects to be analysed.
- ✔ Identify the states.
- ✔ Identify the events.

Elements of state chart diagrams

- Initial State: This shows the starting point of the state chart diagram that is where the activity starts.

Elements of state chart diagrams

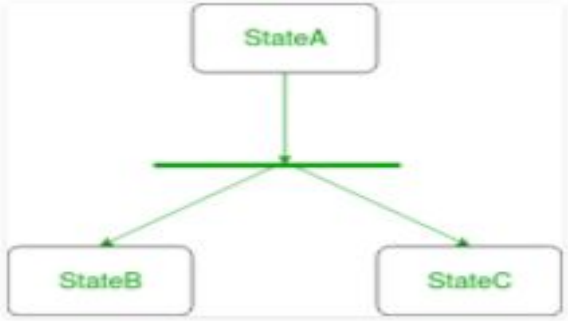
- State: A state represents a condition of a modelled entity for which some action is performed. The state is indicated by using a rectangle with rounded corners and contains compartments

Elements of state chart diagrams

- **Composite state –** We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.

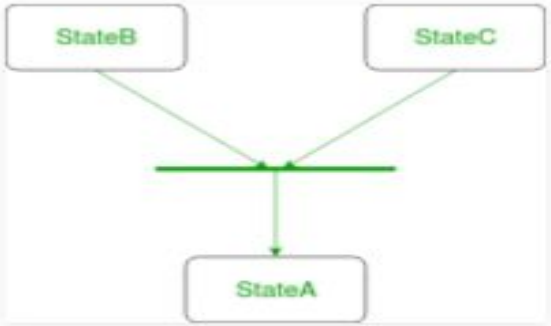
Elements of state chart diagrams

- **Fork –** We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



Elements of state chart diagrams

- **Join** – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.

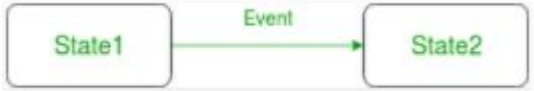


Elements of state chart diagrams

- Transition: It is indicated by an arrow. Transition is a relationship between two states which indicates that Event/ Action an object in the first state will enter the second state and performs certain specified actions.

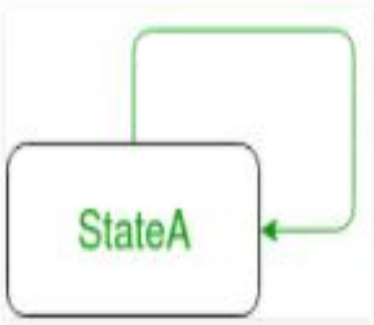
Elements of state chart diagrams

- **Transition –** We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.



Elements of state chart diagrams

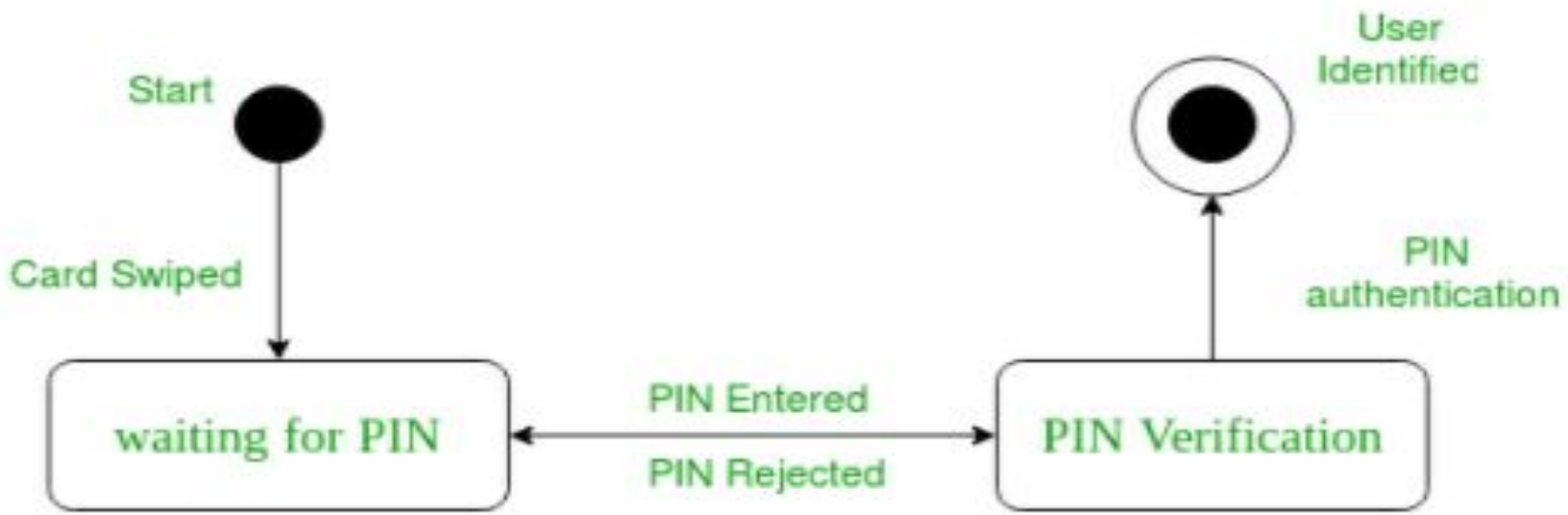
- **Self transition –** We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.



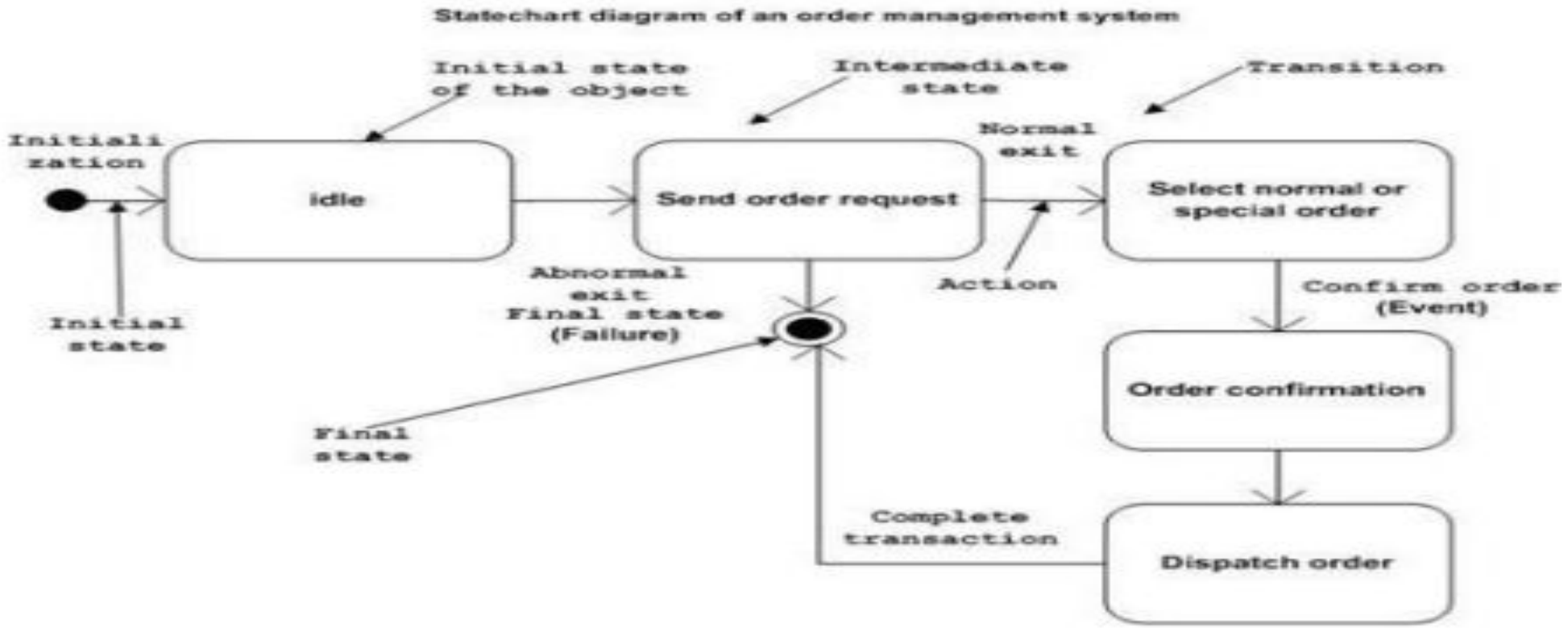
Elements of state chart diagrams

- Final State: The end of the state chart diagram is represented by a solid circle surrounded by a circle.

Example state chart for ATM card PIN Verification



Example state chart for order management system



Activity Diagram

Activity Diagram

- Activity diagram is UML behavior diagram which emphasis on the sequence and conditions of the flow
- It shows a sequence of actions or flow of control in a system.
- It is like to a flowchart or a flow diagram.
- It is frequently used in business process modeling. They can also describe the steps in a use case diagram.
- The modeled Activities are either sequential or concurrent.

Benefits

- It illustrates the logic of an algorithm.
- It describes the functions performed in use cases.
- Illustrate a business process or workflow between users and the system.
- It Simplifies and improves any process by descriptive complex use cases.
- Model software architecture elements, such as method, function, and operation.

Symbols and Notations

Activity

- Is used to illustrate a set of actions.
- It shows the non-interruptible action of objects.

Activity

Symbols and Notations

Action Flow

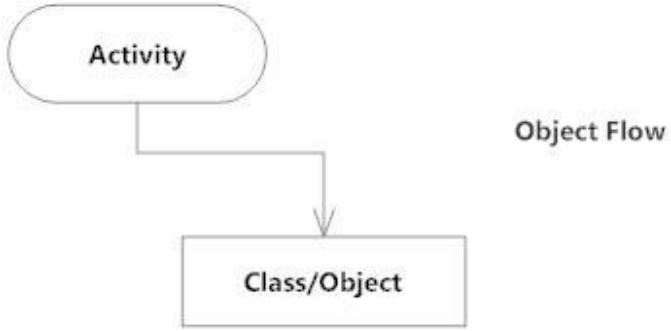
- It is also called edges and paths
- It shows switching from one action state to another. It is represented as an arrowed line.

Action Flow

Symbols and Notations

Object Flow

- Object flow denotes the making and modification of objects by activities.
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object.



Symbols and Notations

Decisions and Branching

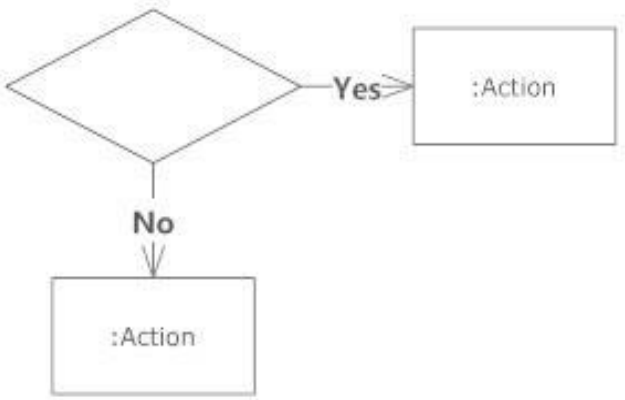
- A diamond represents a decision with alternate paths.
- When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.
- The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."

Decision Symbol

Symbols and Notations

Guards

- In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity.
- These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.



Guard Symbols

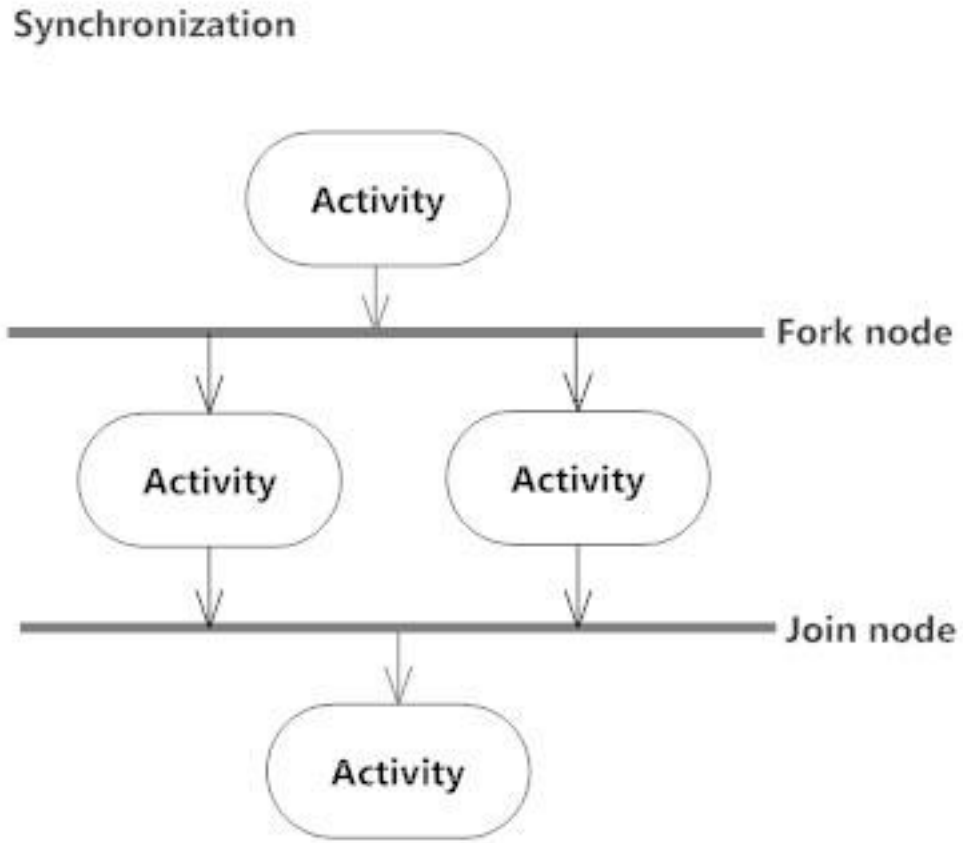
Symbols and Notations

Synchronization

- A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join node joins multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization.

Symbols and Notations

Synchronization



Symbols and Notations

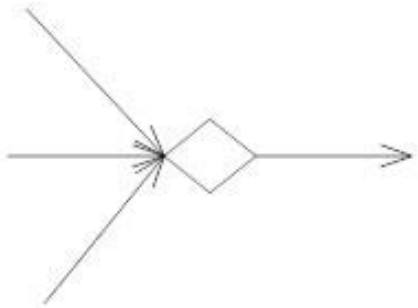
Time Event

- This refers to an event that stops the flow for a time; an hourglass depicts it.

Symbols and Notations

Merge Event

- A merge event brings together multiple flows that are not concurrent.



Merge

Final State or End Poi..

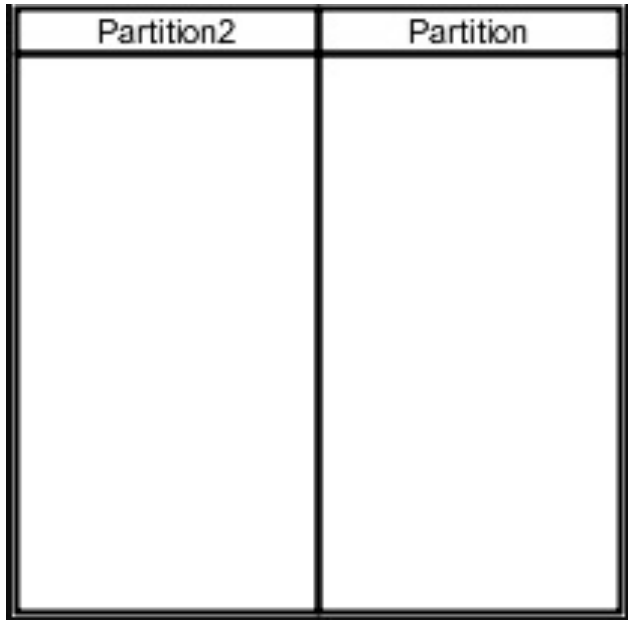
- An arrow pointing to a filled circle nested inside another circle represents the final action state.

End Point Symbol

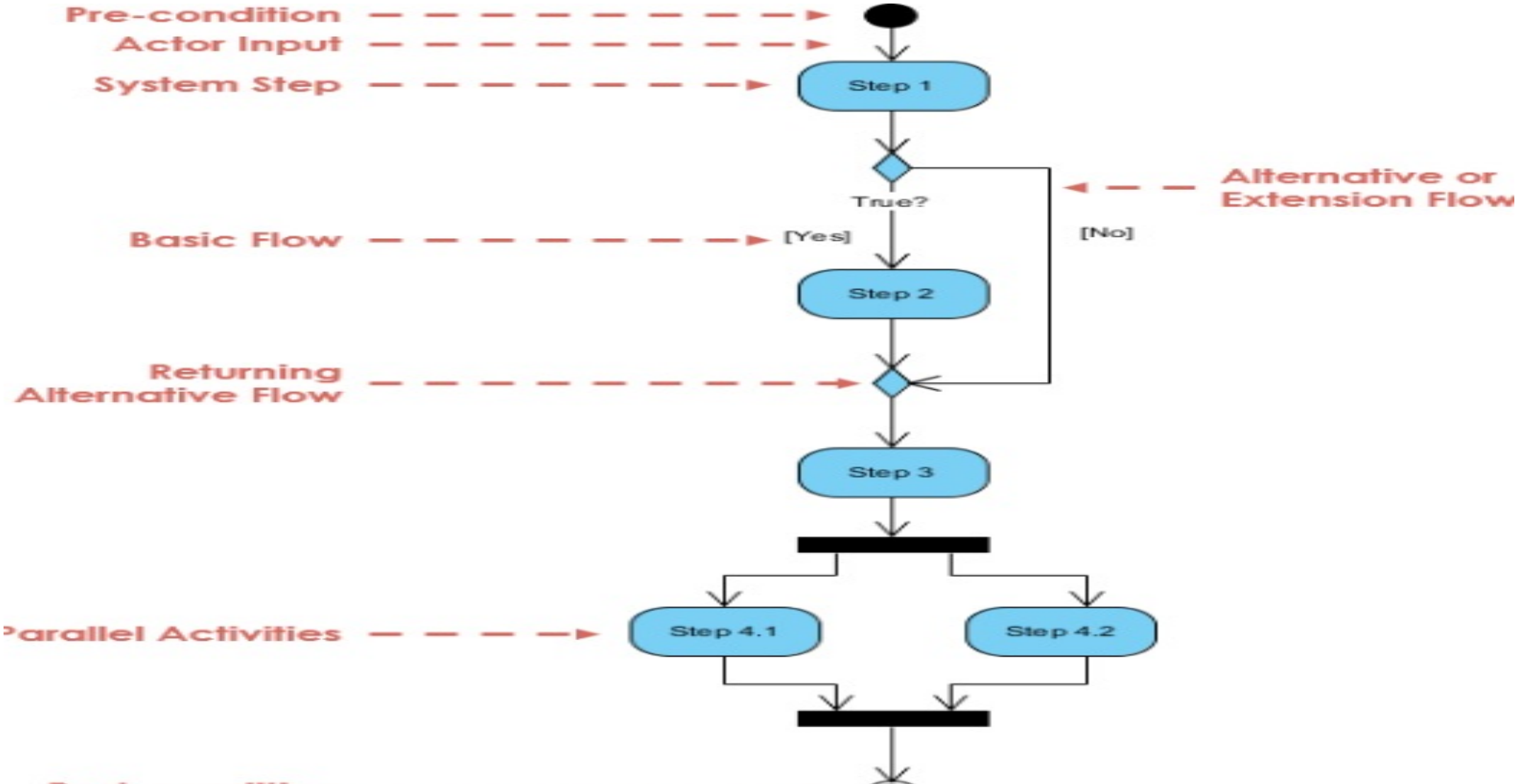
Symbols and Notations

Swimlane and Partition

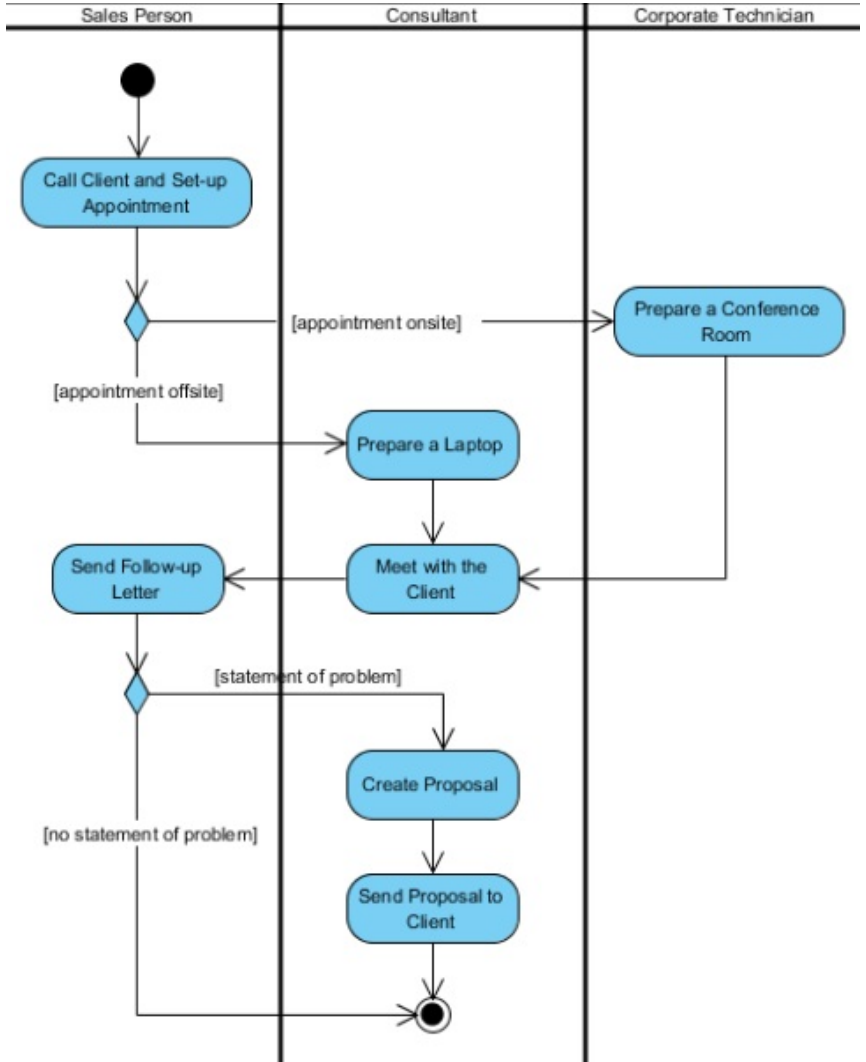
- A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread



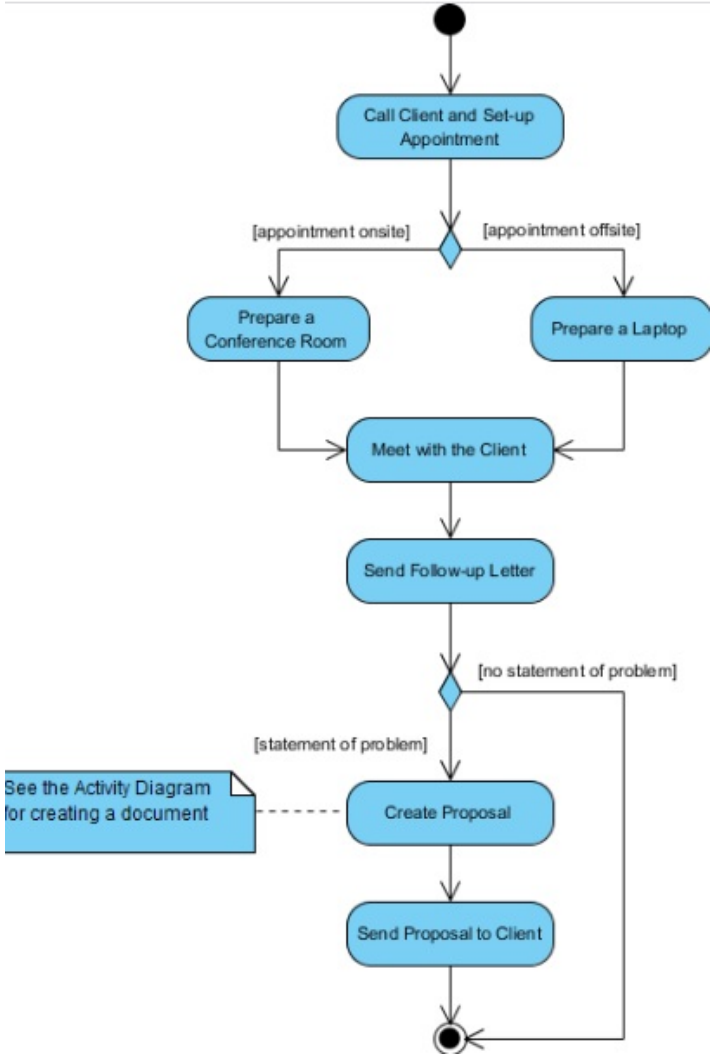
Activity Diagram



Activity Diagram with Swimlane



Activity Diagram without Swimlane



State Chart and Activity Diagram Scenarios

6.4 PROBLEM STATEMENT: MCA ADMISSION SYSTEM

MCA admission procedure as controlled by Directorate of Technical Education (DTE) is as follows:

1. DTE advertises the date of MCA entrance examination.
2. Student has to apply for the entrance examination.
3. Results are declared by DTE.
4. Student has to fill up the option form to select the college of his/her choice.
5. DTE displays the allotment list in the web site and intimation to all colleges.
6. Students should report the allotted colleges and complete the admission procedure.

1. Analysis of MCA Admission System

Drawing an activity diagram for the whole system we,

1. Find out swimlanes if any. To find swimlanes, see if we can span some activities over different organizational units/places.
2. Find out in which swimlane the admission process begins and where it ends. Those will become the initial and final states.
3. Then, identify activities occurring in each swimlane. Arrange activities in sequence flow spanning over all the swimlanes.
4. Identify conditional flow or parallel flow of activities. Parallel flow of activities must converge at a single point using join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Swimlanes identified for admission process are shown in Figure 6.9.

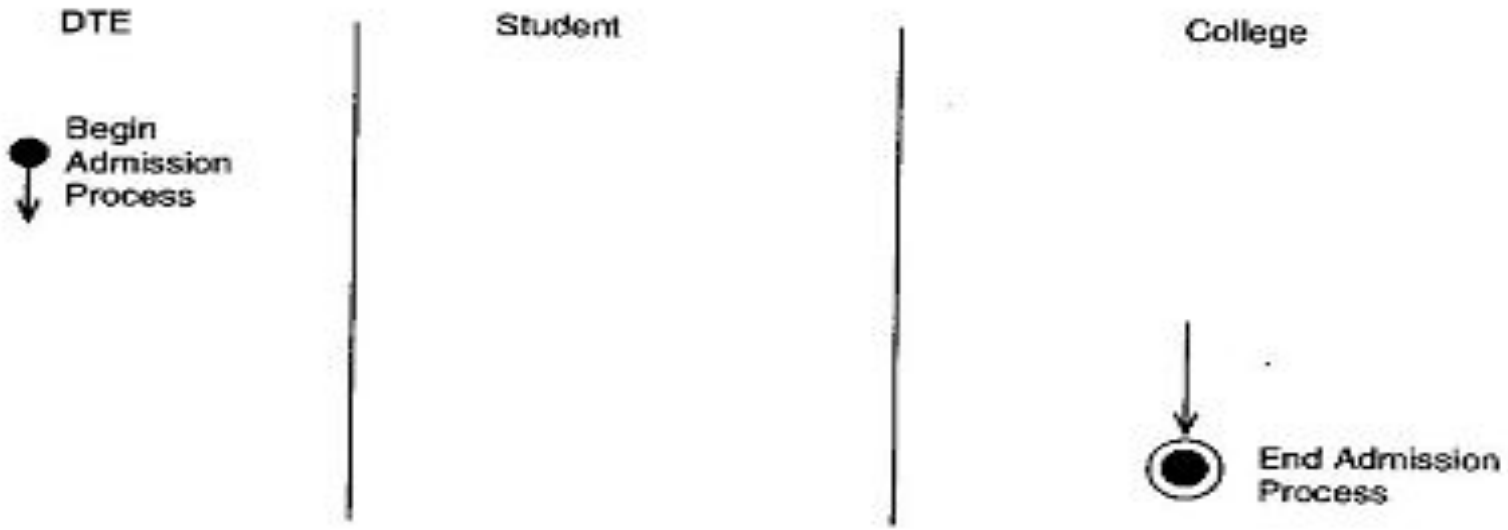


FIGURE 6.9 Swimlanes in admission process.

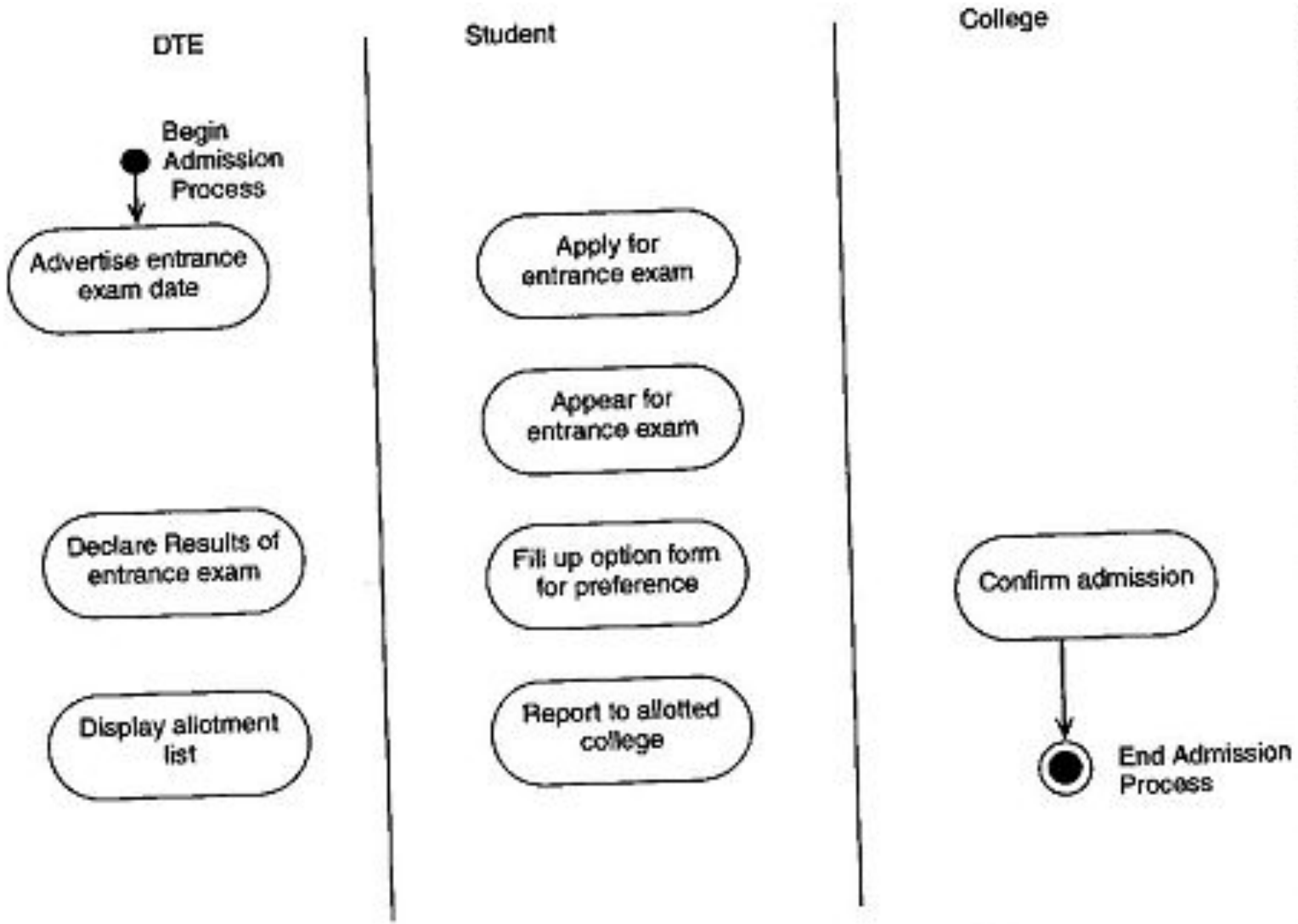


FIGURE 6.10 Activities in admission process.

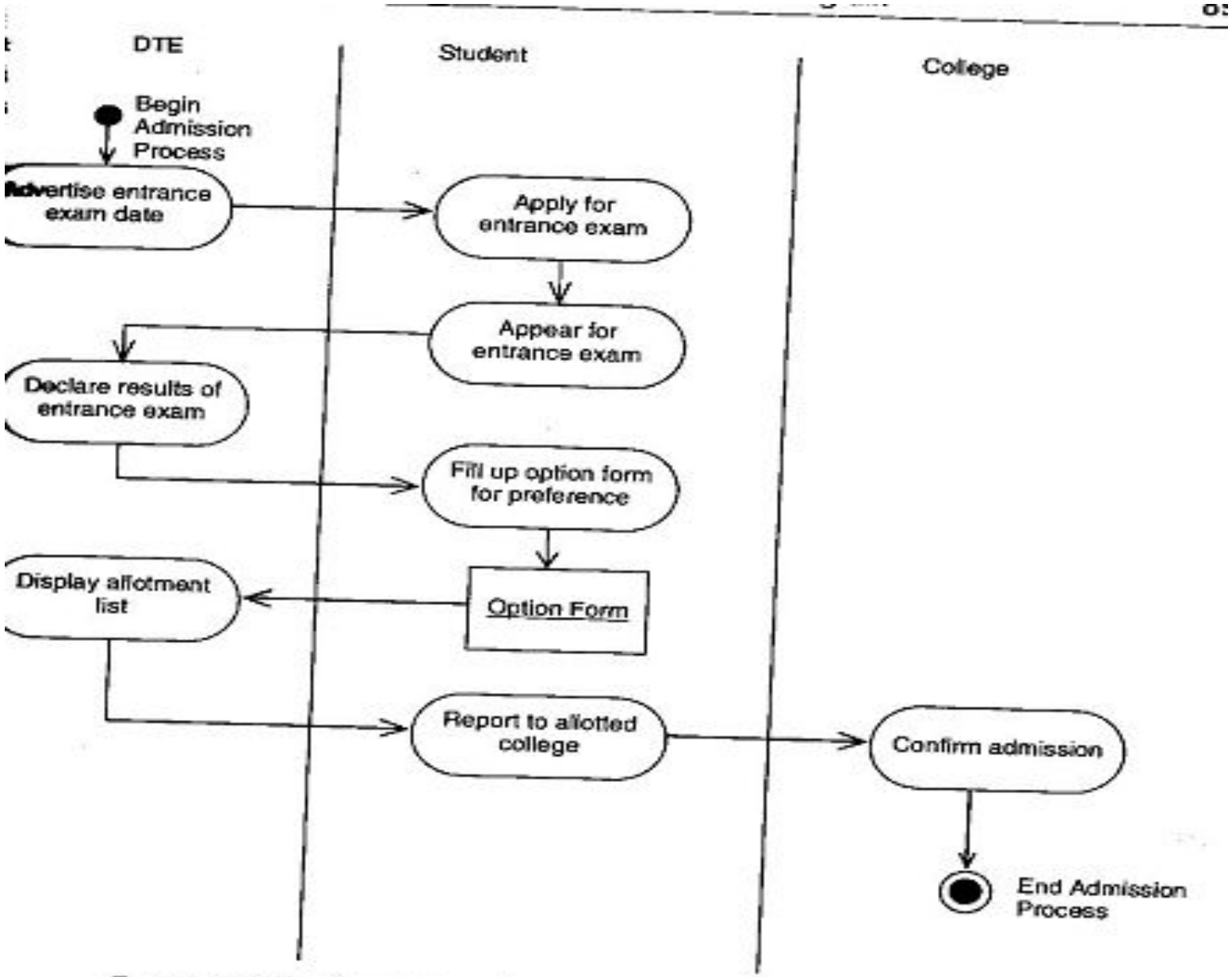


FIGURE 6.12 Complete activity diagram of admission process.

State Chart Diagram

For example, a simple state diagram for a Door object with states *Opened*, *Closed* and *Locked* is shown in Figure 6.15(b). All the three states of the door are simple states without any substates.

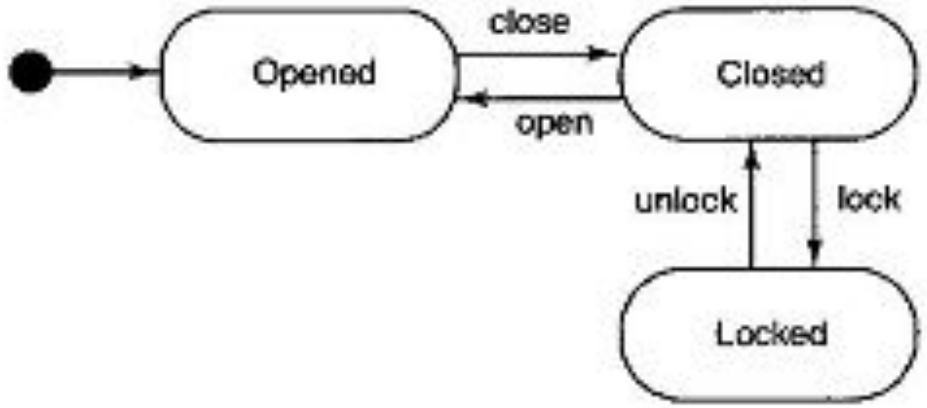


FIGURE 6.15(b) State diagram for a door showing simple states.

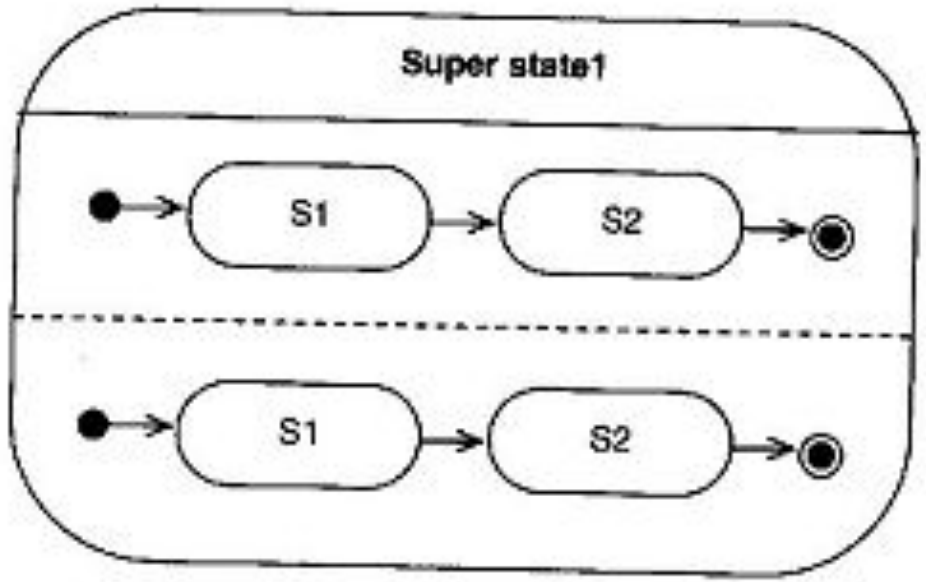


FIGURE 6.16(a) Representing super state concurrent composite state.

For example, Figure 6.16(b) shows a statechart diagram for a gas station where on arrival for filling gas, attendants can perform two tasks in parallel, filling gas as well as washing windshield. InService is the concurrent composite state having Filling Gas Tank and Washing Windshield substates in parallel.

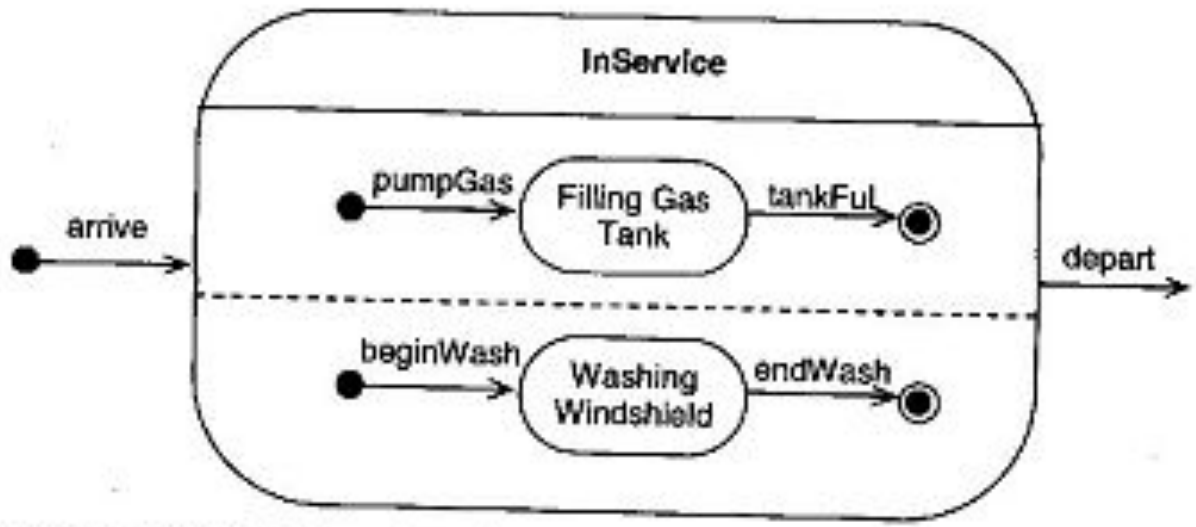


FIGURE 6.16(b) Example of concurrent composite state—InService.

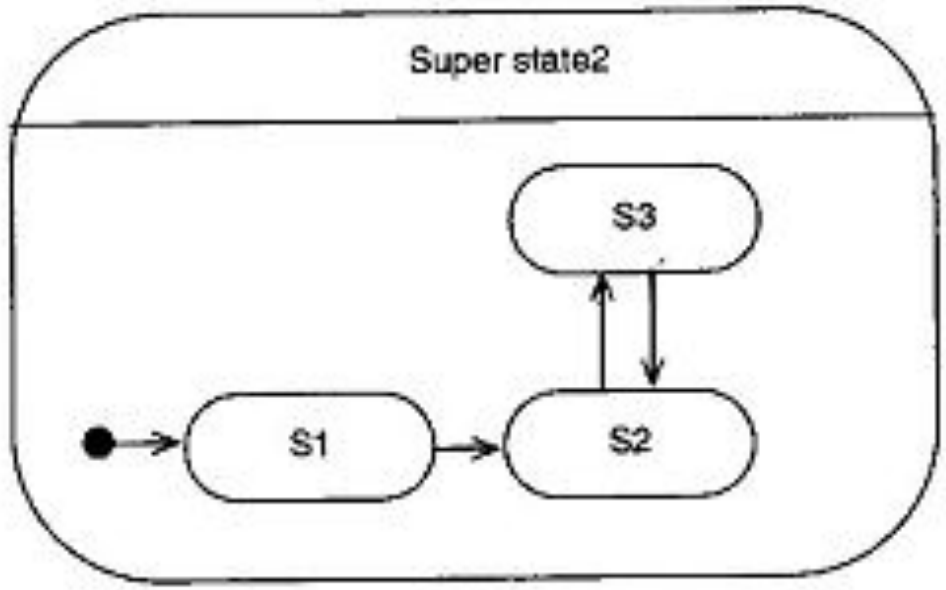


FIGURE 6.16(c) Representing superstate—sequential composite state.

6.8. PROBLEM STATEMENT: ADVERTISEMENT CAMPAIGN OF ABC PVT. LTD.

ABC Pvt. Ltd. company wants to start its advertisement campaign. The advertisement will be prepared. After the approval of the advertisement, the advertisement will be scheduled for publication. The advertisement will be published after scheduling is done.

6.8.1 Analysis of Advertisement Campaign of ABC Pvt. Ltd.

In this problem statement, the object for which a statechart diagram should be drawn is:

Advertisement campaign

Since the *Advertisement campaign* has fixed states from start till end, the statechart diagram will be of type one shot life cycle statechart diagram. Hence there will be one initial state and one or more final state.

Initial state: Figure 6.18(a) shows the initial state of the advertisement compain process.

FIGURE 6.18(a) Begin advertisement campaign process.

Final state: Figure 6.18(b) shows the final state of the compain process.

FIGURE 6.18(b) End advertisement campaign process.

Intermediate states: Figure 6.19(a) shows the intermediate states of the process.

FIGURE 6.19(a) Intermediate states—advertisement campaign.

Figure 6.19(b) shows all the transitions of the advertisement compain process.

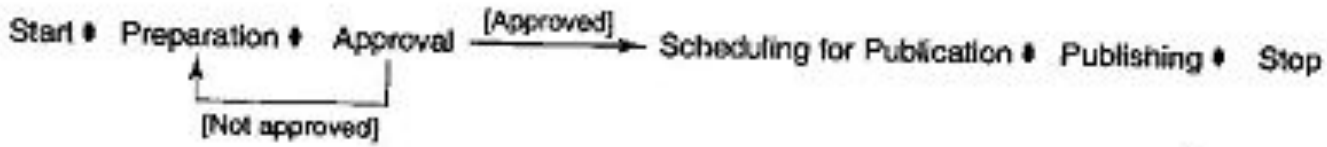


FIGURE 6.19(b) Transition—advertisement campaign: scheduling for publication and publishing.

The complete state transition diagram for the *advertisement campaign* object is shown in Figure 6.20.

FIGURE 6.20 State chart diagram—advertisement campaign.

Practice Questions- MCQ

In an Activity Diagram, organizing the activities into groups is called _____

A. forking

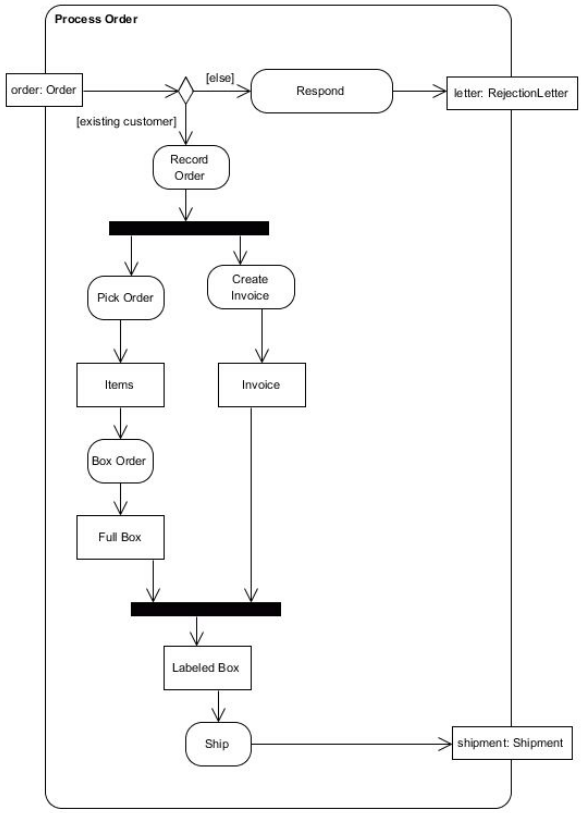
B. joining

C. swimlane

D. synchronization

Ans: C

Practice questions



1. Identify all of the activities in this diagram.
2. Identify all of the object/data nodes in this diagram.
3. Identify all of the actions in this diagram.
4. Identify all of the decision nodes in this diagram.
5. Identify all of the fork nodes in this diagram.
6. Identify all of the join nodes in this diagram.
7. Identify a control flow in this diagram.
8. Identify a data flow in this diagram.
9. Can "Pick Order" and "Create Invoice" occur at the same time?
10. Can "Record Order" and "Ship" occur at the same time?

Thank you