UNIT 3: INHERITANCE AND INTERFACE

PROF. ABHIJIT S. PANDE
YCCE, Nagpur

Inheritance

- C++ strongly supports the concept of reusability.
- The mechanism of deriving a new class from old class is called as inheritance.
- The old class is referred as base class or super class.
- The new class is referred as derived class or sub class.
- The derived class inherits some or all of the properties from the base class.
- Furthermore, the derived class can add new features of its own.
- A class can inherit properties from more than one class or from more than one level.
- In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format :

  class derived_class: accessSpecifier base_class
  {
  ...
  };

- Where *derived_class* is the name of the derived class and *base_class* is the name of the class on which it is based.
- The accessSpecifier may be public, protected or private.
- This access specifier describes the access level for the members that are inherited from the base class.
- The default access Specifier is private.

Advantages of Inheritance

1. Reusability

- Inheritance helps the code to be reused in many situations.
- The base class is defined and once it is compiled, it need not be reworked.
- 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.

2. Saves Time and Effort

- The above concept of reusability achieved by inheritance saves the programmer time and effort as the main code written can be reused.

3. Extendibility

- We can extend the already made classes by adding some new features.

4. Maintainability

- It is easy to debug a program when divided in parts. Inheritance provides an opportunity to capture the problem.

Access Specifier	**How Members of the Base Class Appear in the Derived Class**
Private	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
Protected	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
Public	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become public members of the derived class.

Types of inheritance

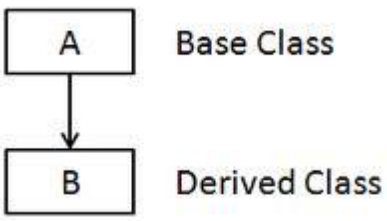
1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance

4. Multilevel inheritance

5. Hybrid inheritance

Single Inheritance

- In this type of inheritance one derived class inherits from only one base class.
- It is the simplest form of inheritance.
- A derived class can be created by deriving the properties of the base class and adding its own properties.
- Figure shows the single inheritance



class A
{
.....
.....
};
class B: access_specifier A
{
.....
.....
};

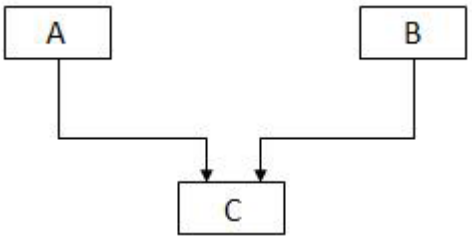
o Here class A is the base class and class B is the derived class.

o *access_specifier* may be private, public or protected. The default *access_specifier* is private.

Multiple Inheritance

- In this type of inheritance one class is derived from two or more base classes.
- Multiple inheritance allows you to combine features of several existing classes.

- A derived class can be created by deriving the properties of the base class and adding its own properties.
- Figure shows the multiple inheritance



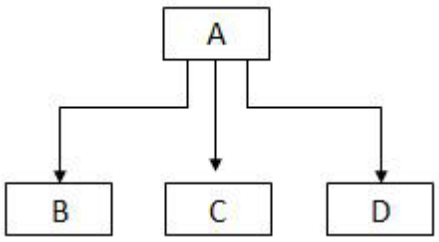

```
class A
{
.....
.....
};
class B
{
.....
.....
};
class C : access_specifier A , access_specifier B
{
.....
.....
};
```

o Here class A and class B are base classes and class C is the derived class.

o *access_specifier* may be private, public or protected. The default *access_specifier* is private.

Hierarchical Inheritance

- In this type of inheritance two or more classes are derived from the same base class.
- This form of inheritance follows one to many relationship.
- A derived class can be created by deriving the properties of the base class and adding its own properties.
- Figure shows hierarchical inheritance



class A

{

.....

.....

};

class B: access_specifier A

{

.....

.....

};

class C: access_specifier A

{

.....

.....

};

class D: access_specifier A

{

.....

.....

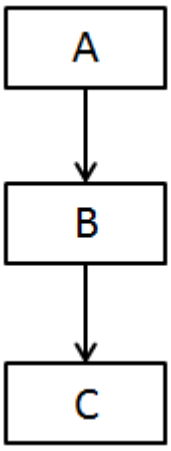
};

o Here class A is the base class. Class B, class C, class D are derived classes.

o *access_specifier* may be private, public or protected. The default *access_specifier* is private.

Multilevel Inheritance

- In this type of inheritance a class is derived from another derived class.
- Sometimes, a derived class itself, acts as a base class for another class.
- Figure shows multilevel inheritance



class A

{

.....

.....

};

class B: access_specifier A

{

.....

.....

};

class C: access_specifier A

{

.....

.....

};

class D: access_specifier B , access_specifier C

{

.....

.....

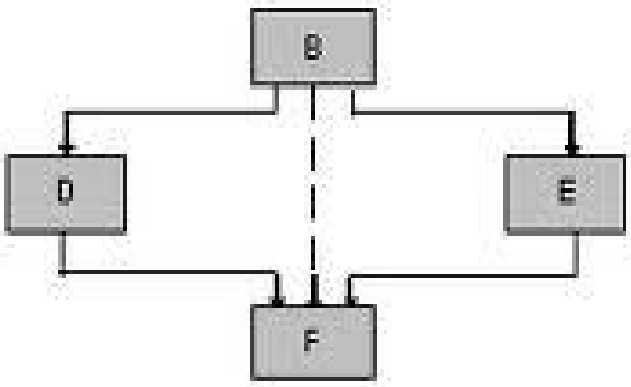
};

o Here class D is derived from classes B and C which forms multiple inheritance and Classes B and C are derived from class A which forms hierarchical inheritance.

o *access_specifier* may be private, public or protected. The default *access_specifier* is private.

Virtual base class

- An ambiguity can arise when several paths exist to a class from the same base class.
- This means that a child class could have duplicate sets of members inherited from a single base class.



- Class F has two direct base classes i.e class D and E which has a common base class i.e. class B.
- All the public, protected members of B are inherited into F twice, first via D and second via E.
- This means class F would have duplicate set of members inherited from class B. This introduces ambiguity and should be avoided.
- C++ solves this issue by introducing a virtual base class.
- We can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.
- When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

```
#include<iostream.h>
#include<conio.h>
class student
{
        int rno;
        public:
        void getnumber()
        {
                cout<<"Enter Roll No:";
```

```
                cin>>rno;
        }
        void putnumber()
        {
                cout<<"\n\n\tRoll No:"<<rno<<"\n";
        }
};
class test:virtual public student
{
        public:
        int part1,part2;
        void getmarks()
        {
                cout<<"Enter Marks\n";
                cout<<"Part1:";
                cin>>part1;
                cout<<"Part2:";
                cin>>part2;
        }
        void putmarks()
        {
                cout<<"\tMarks Obtained\n";
                cout<<"\n\tPart1:"<<part1;
                cout<<"\n\tPart2:"<<part2;
        }
};
class sports: virtual public student
{
        public:
        int score;
        void getscore()
        {
                cout<<"Enter Sports Score:";
                cin>>score;
        }
```

```
        void putscore()
        {
                cout<<"\n\tSports Score is:"<<score;
        }
};
class result:public test,public sports
{
        int total;
        public:
        void display()
        {
                total=part1+part2+score;
                putnumber();
                putmarks();
                putscore();
                cout<<"\n\tTotal Score:"<<total;
        }
};
void main()
{
        result obj;
        clrscr();
        obj.getnumber();
        obj.getmarks();
        obj.getscore();
        obj.display();
        getch();
}
```

Virtual Function

- In polymorphism we use the pointer to the base class to refer to all derived objects.
- But, a base pointer, even when it is made to contain address of derived class, always executes the function in the base class.

- To achieve polymorphism we use the virtual function.
- A virtual function is a member function that is declared within a base class and redefined by a derived class.
- To create virtual function, precede the function's declaration in the base class with the keyword virtual.
- When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.
- Base class pointer can point to derived class object.
- In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked.
- But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.
- **Syntax:**

```
class A
{
    .....
    public:
    virtual return_type function_name(arg_list)
    {
            .....
    }
};
```

Example

```
#include<iostream.h>
#include<conio.h>
class base
{
    public:
    virtual void show()
```

```
        {
                cout<<"\n Base class show:";
        }
        void display()
        {
                cout<<"\n Base class display:" ;
        }
};
class drive:public base
{
        public:
        void display()
        {
                cout<<"\n Drive class display:";
        }
        void show()
        {
                cout<<"\n Drive class show:";
        }
};
void main()
{
        clrscr();
        base obj1;
        base *p;
        cout<<"\n\t P points to base:\n" ;
        p=&obj1;
        p->display();
        p->show();
        cout<<"\n\n\t P points to drive:\n";
        drive obj2;
        p=&obj2;
        p->display();
        p->show();
        getch();
```

```
}
```

Output:

P points to Base

Base class display

Base class show

P points to Drive

Base class Display

Drive class Show

Rules for virtual function

1. The virtual functions must be the member of some class.
2. They are accessed by using object pointers.
3. A virtual function can be a friend of another class.
4. A virtual function in a base class must be defined, even though it may not be used.
5. The prototype of the base class version of a virtual function and all the derived class versions must be identical.
6. We cannot have virtual constructors, but we can have virtual destructors.
7. We cannot use a pointer to a derived class to access an object of the base type.
8. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class.
9. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure virtual function

- Pure virtual Functions are virtual functions with no definition.
- They start with **virtual** keyword and ends with = 0.
- The **"= 0"** portion of a pure virtual function is also known as the *pure specifier*, because it's what makes a pure virtual function "pure".
- A class containing pure virtual functions cannot be used to declare any objects of its own.
- Here is the syntax for a pure virtual function,

virtual *return_type function_name()=0;*

Example

```
#include<iostream.h>
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";
        }
};
int main()
{
```

```
    Base obj;
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

Output : Implementation of Virtual Function in Derived class

Abstract class

- Abstract Class is a class which contains at least 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.

Characteristics

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.

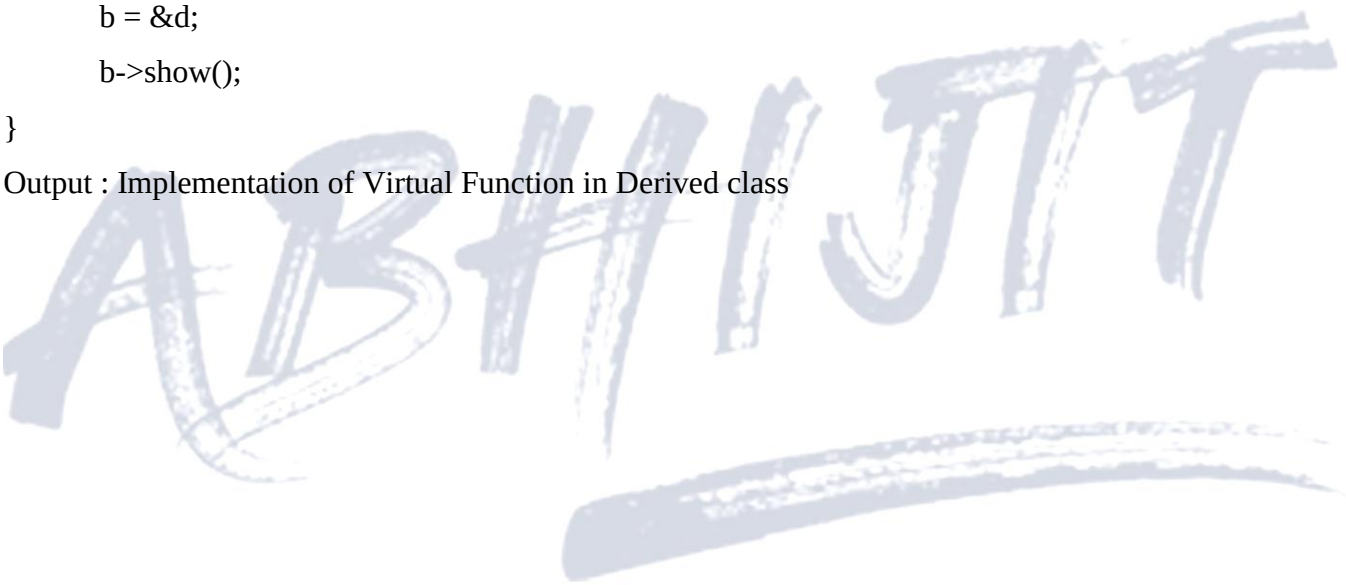
Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Example

```
class Base // abstract 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;
        Base *b;
        Derived d;
        b = &d;
        b->show();
}
```

Output : Implementation of Virtual Function in Derived class



Interface

- 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.
- It is used to achieve abstraction and multiple inheritance in Java.

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

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

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.

Syntax:

```
interface <interface_name>
{
  // declare constant fields
   // declare methods that abstract
  // by default.
}
```

Example

```
interface printable{
void print();
}
class A6 implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }

public static void main(String args[])
{
        A6 obj = new A6();
        obj.print();
```

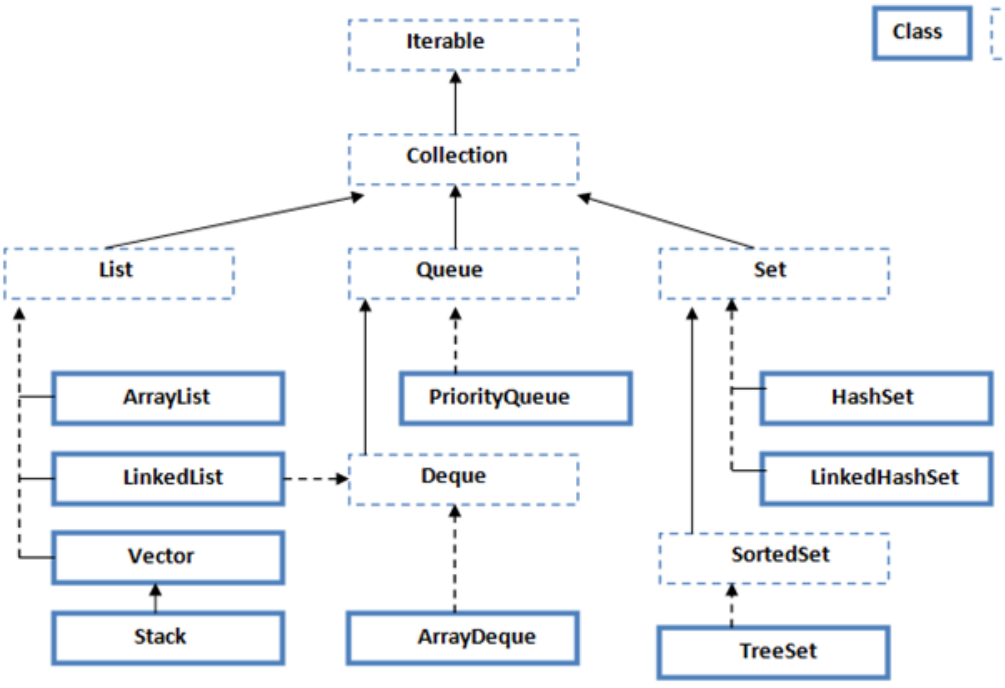
```
}

}
```

Collections in Java

1. Java Collection Framework
2. Hierarchy of Collection Framework
3. Collection interface
4. Iterator interface

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects.
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



List Interface

- List interface is the child interface of Collection interface.
- It inhibits a list type data structure in which we can store the ordered collection of objects.
- It can have duplicate values.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

ArrayList

- The ArrayList class implements the List interface.
- It uses a dynamic array to store the duplicate element of different data types.
- The ArrayList class maintains the insertion order and is non-synchronized.
- The elements stored in the ArrayList class can be randomly accessed.

LinkedList

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.
- In LinkedList, the manipulation is fast because no shifting is required.

Vector

- Vector uses a dynamic array to store the data elements.
- It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Stack

- The stack is the subclass of Vector.
- It implements the last-in-first-out data structure, i.e., Stack.
- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Queue Interface

- Queue interface maintains the first-in-first-out order.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed.
- There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Priority Queue

- The PriorityQueue class implements the Queue interface.
- It holds the elements or objects which are to be processed by their priorities.
- PriorityQueue doesn't allow null values to be stored in the queue.

Deque Interface

- Deque interface extends the Queue interface.
- In Deque, we can remove and add the elements from both the side.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

ArrayDeque

- ArrayDeque class implements the Deque interface.
- It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Set Interface

- Set Interface in Java is present in java.util package.
- It extends the Collection interface.
- It represents the unordered set of elements which doesn't allow us to store the duplicate items.
- We can store at most one null value in Set.
- Set is implemented by HashSet, LinkedHashSet, and TreeSet.

HashSet

- HashSet class implements Set Interface.
- It represents the collection that uses a hash table for storage.
- Hashing is used to store the elements in the HashSet. It contains unique items.

LinkedHashSet

- LinkedHashSet class represents the LinkedList implementation of Set Interface.
- It extends the HashSet class and implements Set interface.
- Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.
- The elements of the SortedSet are arranged in the increasing (ascending) order.
- The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.