Inheritance

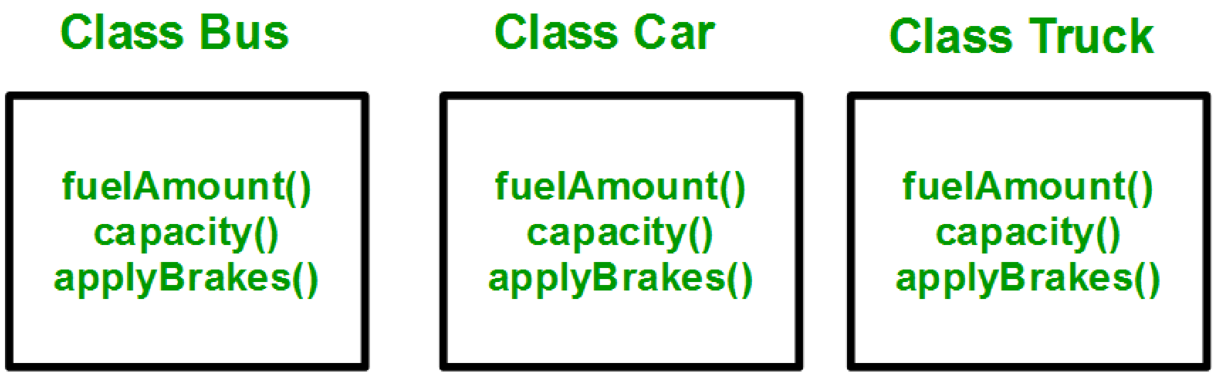
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Why and when to use inheritance?

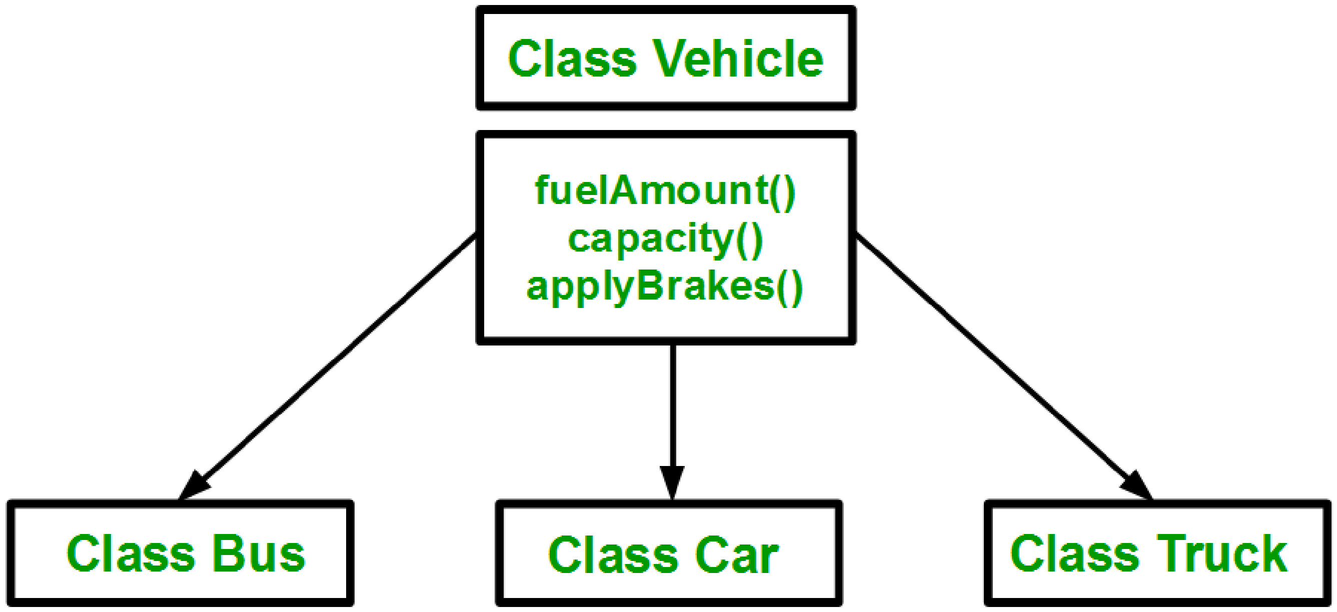
Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



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

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:The class whose properties are inherited by sub class is called Base Class or Super class.



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Modes of Inheritance

1. **Public mode**: 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.
2. **Protected mode**: 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.
3. **Private mode**: 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.

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)

```
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};


// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Polymorphism

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example of Function Overriding

```
class Base
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}
```

In this example, function **show()** is overridden in the derived class. Now let us study how these overridden functions are called in **main()** function.

Function Call Binding with class Objects

Connecting the function call to the function body is called **Binding**. When it is done before the program is run, its called **Early** Binding or **Static** Binding or **Compile-time** Binding.

```
class Base
{
 public:
 void show()
 {
  cout << "Base class\t";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}

int main()
```

```
{
 Base b;       //Base class object
 Derived d;     //Derived class object
 b.show();     //Early Binding Ocuurs
 d.show();
}
```

```
Output : Base class    Derived class
```

In the above example, we are calling the overriden function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

Function Call Binding using Base class Pointer

But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.

```
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
}
```

`Output : Base class`

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing **Base class's pointer**, set call to Base class's **show()** function, without knowing the actual object type.

Virtual Functions

Virtual Function is a function in base class, which is overriden 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.

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime**Binding.

Problem without Virtual Keyword

```
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 Ocuurs
}
```

`Output : Base class`

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```
class Base
{
 public:
 virtual 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();     //Late Binding Ocuurs
}
```

`Output : Derived class`

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

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.

```
#include
using namespace std;

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

class B: public A
{
private:
    virtual void show()
    {
        cout << "Derived class\n";
    }
};

int main()
{
    A *a;
    B b;
    a = &b;

    a -> show();
}
```

```
Output : Derived class
```

Abstract Class

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.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. 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.

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 f() = 0;
```

Example of Abstract Class

```
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.

Why can't we create Object of Abstract Class ?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

Pure Virtual definitions

- 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.

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

void Base :: show()         //Pure Virtual definition
{
 cout << "Pure Virtual definition\n";
}

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

int main()
{
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output :

```
Implementation of Virtual Function in Derived class
```

Example:

// virtpers.cpp // virtual functions with person class

#include <iostream>

using namespace std;   //

```
person class

{ protected: char name[40];

public: void getName()

{ cout << "   Enter name: "; cin >> name; }

void putName() { cout << "Name is: " << name << endl; }

virtual void getData() = 0;       //pure virtual func

virtual bool isOutstanding() = 0;  //pure virtual func };
////////////////////////////////////////////////////////////////

 class student : public person       //student class

{ private: float gpa;             //grade point average

 public: void getData()         //get student data from user

 { person::getName();

cout << "   Enter student's GPA: "; cin >> gpa; } bool isOutstanding() { return (gpa > 3.5) ? true : false; } };
//////////////////////////////////////////////////////////////// class professor : public person
//professor class { private: int numPubs;            //number of papers published public: void getData()
//get professor data from user

{ person::getName();

cout << "   Enter number of professor's publications: ";

cin >> numPubs; }

 bool isOutstanding()

 { return (numPubs > 100) ? true : false; } };




int main()

 { person* persPtr[100];     //array of pointers to persons

 int n = 0;               //number of persons on list char choice;

do

 {

 cout << "Enter student or professor (s/p): ";

 cin >> choice; if(choice=='s')                 //put new student
 persPtr[n] = new student;     //   in array
else                           //put new professor
 persPtr[n] = new professor;   //   in array
persPtr[n++]->getData();        //get data for person
 cout << "   Enter another (y/n)? ";  //do another person?
cin >> choice; }
 while( choice=='y' );         //cycle until not 'y'
for(int j=0; j<n; j++)              //print names of all
{                              //persons, and persPtr[j]->putName();          //say if outstanding if( persPtr[j]-
>isOutstanding() ) cout << "   This person is outstanding\n"; } return 0; }  //end main()
```