Objective:

- It will help you understand the benefits that we get through inheritance relationship.
- It will also help in comparing inheritance and composition but we shall look in detail about them in nest few lectures.

Task-1:

Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class *BasePlusCommissionEmployee* of the hierarchy discussed in 12.4 section of text-book-A to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for designing classes *CommissionEmployee* and *BasePlusCommissionEmployee*, as well as for object-oriented programs in general. Which approach is more natural? Why?

Task-2:

*(*Account *Inheritance Hierarchy)* Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class Account and derived classes SavingsAccount and CheckingAccount that inherit from class Account. Base class Account should include one data member of type double to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to 0.0. If not, the balance should be set to 0.0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function credit should add an amount to the current balance. Member function debit should withdraw money from the Account and ensure that the debit amount does not exceed the Account's balance. If it does, the balance should be left unchanged and the function should print the message "Debit amount exceeded account balance." Member function getBalance should return the current balance.

Derived class SavingsAccount should inherit the functionality of an Account, but also include a data member of type double indicating the interest rate (percentage) assigned to the Account. SavingsAccount's constructor should receive the initial balance, as well as an initial value for the SavingsAccount's interest rate. SavingsAccount should provide a public member function calculateInterest that returns a double indicating the amount of interest earned by an account. Member function calculateInterest should determine this amount by multiplying the interest rate by the account balance. [*Note:* SavingsAccount should inherit member functions credit and debit as is without redefining them.]
Derived class CheckingAccount should inherit from base class Account and include an additional data member of type double that represents the fee charged per transaction. CheckingAccount's constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class CheckingAccount should redefine member functions credit and debit so that they subtract the fee from the account balance whenever either transaction is performed successfully. CheckingAccount's versions of these functions should invoke the base-class Account version to perform the updates to an account balance. CheckingAccount's debit function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [*Hint:* Define Account's debit function so that it returns a bool indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the SavingsAccount object by first invoking its calculateInterest function, then passing the returned interest amount to the object's credit function.

Task-3:

Case Study: Chapter 15: Inheritance

The Automobile, Car, Truck, and SUV classes

Suppose we are developing a program that a car dealership can use to manage its inven- tory of used cars. The dealership s inventory includes three types of automobiles: cars, pickup trucks, and sport-utility vehicles (SUVs). Regardless of the type, the dealership keeps the following data about each automobile:

- Make
- Year model
- Mileage
- Price

Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics. For cars, the dealership keeps the following additional data:

- Number of doors (2 or 4)

For pickup trucks, the dealership keeps the following additional data:

- Drive type (two-wheel drive or four-wheel drive)

And, for SUVs, the dealership keeps the following additional data:

- Passenger capacity

In designing this program, one approach would be to write the following three classes:

- A Car class with attributes for the make, year model, mileage, price, and number of doors.
- A Truck class with attributes for the make, year model, mileage, price, and drive type.
- An SUV class with attributes for the make, year model, mileage, price, and passen- ger capacity.

This would be an inefficient approach, however, because all three classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later that we need to add more common attributes, we would have to modify all three classes.

A better approach would be to write an Automobile base class to hold all the general data about an automobile, and then write derived classes for each specific type of automobile. The following code shows the Automobile class.

Contents of Automobile.h

```
#ifndef AUTOMOBILE_H
#define AUTOMOBILE_H
class Automobile
{

private:
    CString make;
    int model;
```

```
    int mileage;
    double price;
public:
    Automobile();
    Automobile(CString autoMake, int autoModel, int autoMileage, double
autoPrice);
    CString getMake() const;
    int getModel() const;
    int getMileage() const;
    double getPrice() const;
};
#endif
```

Contents of Automobile.cpp

```
#include "Automobile.h"
Automobile::Automobile():make("")
{
    model = 0;
    mileage = 0;
    price = 0.0;
}
Automobile::Automobile(CString autoMake, int autoModel, int autoMileage, double
autoPrice):make(autoMake)
{
    model = autoModel;
    mileage = autoMileage;
    price = autoPrice;
}
CString Automobile::getMake() const
{
    return make;
}
int Automobile::getModel() const
{
    return model;
}
int Automobile::getMileage() const
{
    return mileage;
}
double Automobile::getPrice() const
{
    return price;
}
```

The Automobile class is a complete class that we can create objects from. If we wish, we can write a program that creates instances of the Automobile class. However, the Automobile class holds only general data about an automobile. It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs. To hold data about those specific types of automobiles we will write derived classes that inherit from the Automobile class. The following shows the code for the Car class.

Contents of Car.h

```
#ifndef CAR_H
#define CAR_H
#include "Automobile.h"

class Car: public Automobile
{
```

```
private:
    int doors;
public:
    Car();
    Car(CString carMake, int carModel, int carMileage,
        double carPrice, int carDoors);
    int getDoors();
};
#endif
```

Contents of Car.cpp

```
#include "Car.h"
Car::Car() : Automobile()
{
    doors=0;
}
Car::Car(CString carMake, int carModel, int carMileage,
    double carPrice, int carDoors) :
Automobile(carMake, carModel, carMileage, carPrice)
{
   doors = carDoors;
}
int Car::getDoors()
{
    return doors;
}
```

Notice that the 'Car' default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The Car class also has an overloaded constructor, which accepts arguments for the car's make, model, mileage, price, and number of doors. This parameterized constructor also calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments.

Now lets look at the Truck class, which also inherits from the Automobile class.

Contents of Truck.h

```
#ifndef TRUCK_H
#define TRUCK_H
#include "Automobile.h"
class Truck : public Automobile
{
private:
    CString driveType;
public:
    Truck();
     Truck(CString truckMake, int truckModel, int truckMileage, double
     truckPrice, CString truckDriveType);
    CString getDriveType();
};
#endif
```

Contents of Truck.cpp

```
#include "Truck.h"
```

```
Truck::Truck() : driverType(""),Automobile()
{
}
Truck::Truck(CString truckMake, int truckModel, int truckMileage, double
truckPrice, CString truckDriveType) : Automobile(truckMake, truckModel,
truckMileage, truckPrice),driveType(truckDriveType)
{
}
CString Truck::getDriveType()
{
    return driveType;
}
```

The Truck class defines a driveType attribute to hold a string describing the truck's drive type. The class has a default constructor that sets the driveType attribute to an empty string. Notice that the Truck's default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The Truck class also has an overloaded constructor, that accepts arguments for the truck's make, model, mileage, price, and drive type, which calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments.

Now lets look at the SUV class, which also inherits from the Automobile class.

Contents of SUV.h

```
#ifndef SUV_H
#define SUV_H
#include "Automobile.h"
class SUV : public Automobile
{
private:
    int passengers;
public:
    SUV();
    SUV(string SUVMake, int SUVModel, int SUVMileage,double SUVPrice, int
SUVPassengers);
    int getPassengers();
};
#endif
```

Contents of SUV.cpp

```
#include "SUV.h"
SUV::SUV() : Automobile()
{
    passengers = 0;
}
SUV::SUV(string SUVMake, int SUVModel, int SUVMileage,double SUVPrice, int
SUVPassengers) : Automobile(SUVMake, SUVModel, SUVMileage, SUVPrice)
{
    passengers = SUVPassengers;
}
int SUV::getPassengers()
{
    return passengers;
}
```

The SUV class defines a passenger's attribute to hold the number of passengers that the vehicle can

accommodate. The class has a default constructor that sets the passengers attribute to 0. Notice hat the SUV's default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The SUV class also has an overloaded constructor that accepts arguments for the SUV s make, model, mileage, price, and number of passengers.

```
#include <iostream>
#include "Car.h"
#include "Truck.h"
#include "SUV.h"
using namespace std;
int main()
{
    // Create a Car object for a used 2007 BMW with
    // 50,000 miles, priced at $15,000, with 4 doors.

    Car car("BMW", 2007, 50000, 15000.0, 4);

    // Create a Truck object for a used 2006 Toyota
    // pickup with 40,000 miles, priced at $12,000,
    // with 4-wheel drive.

    Truck truck("Toyota", 2006, 40000, 12000.0, "4WD");

    // Create an SUV object for a used 2005 Volvo
    // with 30,000 miles, priced at $18,000, with
    // 5 passenger capacity.

    SUV suv("Volvo", 2005, 30000, 18000.0, 5);
    // Display the automobiles we have in inventory.
    cout << "We have the following car in inventory:\n"
         << car.getModel() << " " << car.getMake()
         << " with " << car.getDoors() << " doors and "
         << car.getMileage() << " miles.\nPrice: $"
         << car.getPrice() << endl << endl;
    cout << "We have the following truck in inventory:\n"
        << truck.getModel() << " " << truck.getMake()
        << " with " << truck.getDriveType()
        << " drive type and " << truck.getMileage()
        << " miles.\nPrice: $" << truck.getPrice()
        << endl << endl;

    cout << "We have the following SUV in inventory:\n"
        << suv.getModel() << " " << suv.getMake()
        << " with " << suv.getMileage() << " miles and "
        << suv.getPassengers() << " passenger capacity.\n"
        << "Price: $" << suv.getPrice() << endl;

    return 0;
}
```

Program Output

We have the following car in inventory:
2007 BMW with 4 doors and 50000 miles.
Price: $15000.00

We have the following truck in inventory:
2006 Toyota with 4WD drive type and 40000 miles.
Price: $12000.00

We have the following SUV in inventory:
2005 Volvo with 30000 miles and 5 passenger capacity.
Price: $18000.00