CHAPTER 12 Inheritance

TOPICS

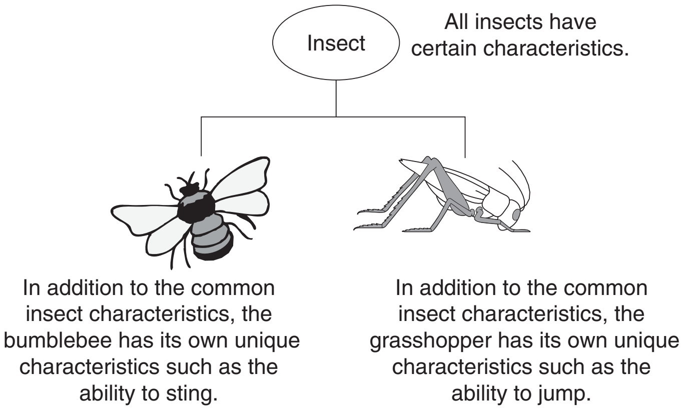
12.1 Introduction to Inheritance

CONCEPT: Inheritance allows a new class to extend an existing class. The new class inherits the members of the class it extends.

Generalization and Specialization

In the real world, you can find many objects that are specialized versions of other more general objects. For example, the term “insect” describes a general type of creature with various characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 12-1.

Figure 12-1 Bumblebees and grasshoppers are specialized versions of an insect


Insect
All insects have certain characteristics.
In addition to the common insect characteristics, the bumblebee has its own unique characteristics such as the ability to sting.
In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

Inheritance and the "Is a" Relationship

When one object is a specialized version of another object, there is an "is a" relationship between them. For example, a grasshopper is an insect. Here are a few other examples of the "is a" relationship:

- A poodle is a dog.
- A car is a vehicle.
- A flower is a plant.
- A rectangle is a shape.
- A football player is an athlete.

When an "is a" relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, inheritance is used to create an "is a" relationship among classes. This allows you to extend the capabilities of a class by creating another class that is a specialized version of it.

Inheritance involves a superclass and a subclass. The *superclass* is the general class and the *subclass* is the specialized class. You can think of the subclass as an extended version of the superclass. The subclass inherits attributes and methods from the superclass without any of them having to be rewritten. Furthermore, new attributes and methods may be added to the subclass, and that is what makes it a specialized version of the superclass.

NOTE: Superclasses are also called *base classes,* and subclasses are also called *derived classes*. Either set of terms is correct. For consistency, this text will use the terms superclass and subclass.

Let's look at an example of how inheritance can be used. Suppose we are developing a program that a car dealership can use to manage its inventory 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 data attributes for the make, year model, mileage, price, and the number of doors.
- A `Truck` class with data attributes for the make, year model, mileage, price, and the drive type.
- An `SUV` class with data attributes for the make, year model, mileage, price, and the passenger capacity.

This would be an inefficient approach, however, because all three of the 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` superclass to hold all the general data about an automobile and then write subclasses for each specific type of automobile. Program 12-1 shows the `Automobile` class's code, which appears in a module named `vehicles`.

Program 12-1 (Lines 1 through 44 of vehicles.py)

```
# The Automobile class holds general data
# about an automobile in inventory.

class Automobile:
    # The __init__method accepts arguments for the
    # make, model, mileage, and price. It initializes
    # the data attributes with these values.

    def __init__(self, make, model, mileage, price):
        self.__make = make
```

(program continues)

Program 12-1 *(continued)*

```
11          self.__model = model
12          self.__mileage = mileage
13          self.__price = price
14
15      # The following methods are mutators for the
16      # class's data attributes.
17
18      def set_make(self, make):
19          self.__make = make
20
21      def set_model(self, model):
22          self.__model = model
23
24      def set_mileage(self, mileage):
25          self.__mileage = mileage
26
27      def set_price(self, price):
28          self.__price = price
29
30      # The following methods are the accessors
31      # for the class's data attributes.
32
33      def get_make(self):
34          return self.__make
35
36      def get_model(self):
37          return self.__model
38
39      def get_mileage(self):
40          return self.__mileage
41
42      def get_price(self):
43          return self.__price
44
```

The `Automobile` class's `__init__` method accepts arguments for the vehicle's make, model, mileage, and price. It uses those values to initialize the following data attributes:

- `__make`
- `__model`
- `__mileage`
- `__price`

(Recall from Chapter 11 that a data attribute becomes hidden when its name begins with two underscores.) The methods that appear in lines 18 through 28 are mutators for each of the data attributes, and the methods in lines 33 through 43 are the accessors.

The `Automobile` class is a complete class that we can create objects from. If we wish, we can write a program that imports the `vehicle` module and 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 subclasses that inherit from the `Automobile` class. Program 12-2 shows the code for the `Car` class, which is also in the `vehicles` module.

Program 12-2 (Lines 45 through 72 of vehicles.py)

```
45  # The Car class represents a car. It is a subclass
46  # of the Automobile class.
47
48  class Car(Automobile):
49      # The __init__ method accepts arguments for the
50      # car's make, model, mileage, price, and doors.
51
52      def __init__(self, make, model, mileage, price, doors):
53          # Call the superclass's __init__ method and pass
54          # the required arguments. Note that we also have
55          # to pass self as an argument.
56          Automobile.__init__(self, make, model, mileage, price)
57
58          # Initialize the __doors attribute.
59          self.__doors = doors
60
61      # The set_doors method is the mutator for the
62      # __doors attribute.
63
64      def set_doors(self, doors):
65          self.__doors = doors
66
67      # The get_doors method is the accessor for the
68      # __doors attribute.
69
70      def get_doors(self):
71          return self.__doors
72
```

Take a closer look at the first line of the class declaration, in line 48:

```
class Car(Automobile):
```

This line indicates that we are defining a class named `Car`, and it inherits from the `Automobile` class. The `Car` class is the subclass and the `Automobile` class is the superclass. If we want to express the relationship between the `Car` class and the `Automobile` class, we can say that a `Car` is an `Automobile`. Because the `Car` class extends the `Automobile` class, it inherits all of the methods and data attributes of the `Automobile` class.

Look at the header for the `__init__` method in line 52:

```
def __init__(self, make, model, mileage, price, doors):
```

Notice that in addition to the required `self` parameter, the method has parameters named `make`, `model`, `mileage`, `price`, and `doors`. This makes sense because a `Car` object will have data attributes for the car's make, model, mileage, price, and number of doors. Some of these attributes are created by the `Automobile` class, however, so we need to call the `Automobile` class's `__init__` method and pass those values to it. That happens in line 56:

```
Automobile.__init__(self, make, model, mileage, price)
```

This statement calls the `Automobile` class's `__init__` method. Notice that the statement passes the `self` variable, as well as the `make`, `model`, `mileage`, and `price` variables as arguments. When that method executes, it initializes the `__make`, `__model`, `__mileage`, and `__price` data attributes. Then, in line 59, the `__doors` attribute is initialized with the value passed into the `doors` parameter:

```
self.__doors = doors
```

The `set_doors` method, in lines 64 through 65, is the mutator for the `__doors` attribute, and the `get_doors` method, in lines 70 through 71 is the accessor for the `__doors` attribute. Before going any further, let's demonstrate the `Car` class, as shown in Program 12-3.

Program 12-3 (car_demo.py)

```
# This program demonstrates the Car class.

import vehicles

def main():
    # Create an object from the Car class.
    # The car is a 2007 Audi with 12,500 miles, priced
    # at $21,500.00, and has 4 doors.
    used_car = vehicles.Car('Audi', 2007, 12500, 21500.00, 4)

    # Display the car's data.
    print('Make:', used_car.get_make())
    print('Model:', used_car.get_model())
    print('Mileage:', used_car.get_mileage())
    print('Price:', used_car.get_price())
    print('Number of doors:', used_car.get_doors())

# Call the main function.
main()
```

Program Output

```
Make: Audi
Model: 2007
```

```
Mileage: 12500
Price: 21500.0
Number of doors: 4
```

Line 3 imports the `vehicles` module, which contains the class definitions for the `Automobile` and `Car` classes. Line 9 creates an instance of the `Car` class, passing `'Audi'` as the car's make, 2007 as the car's model, 125,00 as the mileage, 21,500.00 as the car's price, and 4 as the number of doors. The resulting object is assigned to the `used_car` variable.

The statements in lines 12 through 15 calls the object's `get_make`, `get_model`, `get_mileage`, and `get_price` methods. Even though the `Car` class does not have any of these methods, it inherits them from the `Automobile` class. Line 16 calls the `get_doors` method, which is defined in the `Car` class.

Now let's look at the `Truck` class, which also inherits from the `Automobile` class. The code for the `Truck` class, which is also in the `vehicles` module, is shown in Program 12-4.

Program 12-4 (Lines 73 through 100 of vehicles.py)

```
# The Truck class represents a pickup truck. It is a
# subclass of the Automobile class.

class Truck(Automobile):
    # The __init__ method accepts arguments for the
    # Truck's make, model, mileage, price, and drive type.

    def __init__(self, make, model, mileage, price, drive_type):
        # Call the superclass's __init__ method and pass
        # the required arguments. Note that we also have
        # to pass self as an argument.
        Automobile.__init__(self, make, model, mileage, price)

        # Initialize the __drive_type attribute.
        self.__drive_type = drive_type

    # The set_drive_type method is the mutator for the
    # __drive_type attribute.

    def set_drive_type(self, drive_type):
        self.__drive = drive_type

    # The get_drive_type method is the accessor for the
    # __drive_type attribute.

    def get_drive_type(self):
        return self.__drive_type

```

The Truck class's __init__ method begins in line 80. Notice that it takes arguments for the truck's make, model, mileage, price, and drive type. Just as the Car class did, the Truck class calls the Automobile class's __init__ method (in line 84) passing the make, model, mileage, and price as arguments. Line 87 creates the __drive_type attribute, initializing it to the value of the drive_type parameter.

The set_drive_type method in lines 92 through 93 is the mutator for the __drive_type attribute, and the get_drive_type method in lines 98 through 99 is the accessor for the attribute.

Now let's look at the SUV class, which also inherits from the Automobile class. The code for the SUV class, which is also in the vehicles module, is shown in Program 12-5.

Program 12-5 (Lines 101 through 128 of vehicles.py)

```
101 # The SUV class represents a sport utility vehicle. It
102 # is a subclass of the Automobile class.
103
104 class SUV(Automobile):
105     # The __init__ method accepts arguments for the
106     # SUV's make, model, mileage, price, and passenger
107     # capacity.
108
109     def __init__(self, make, model, mileage, price, pass_cap):
110         # Call the superclass's __init__ method and pass
111         # the required arguments. Note that we also have
112         # to pass self as an argument.
113         Automobile.__init__(self, make, model, mileage, price)
114
115         # Initialize the __pass_cap attribute.
116         self.__pass_cap = pass_cap
117
118     # The set_pass_cap method is the mutator for the
119     # __pass_cap attribute.
120
121     def set_pass_cap(self, pass_cap):
122         self.__pass_cap = pass_cap
123
124     # The get_pass_cap method is the accessor for the
125     # __pass_cap attribute.
126
127     def get_pass_cap(self):
128         return self.__pass_cap
```

The SUV class's __init__ method begins in line 109. It takes arguments for the vehicle's make, model, mileage, price, and passenger capacity. Just as the Car and Truck classes did, the SUV class calls the Automobile class's __init__ method (in line 113) passing the make, model, mileage, and price as arguments. Line 116 creates the __pass_cap attribute, initializing it to the value of the pass_cap parameter.

The set_pass_cap method in lines 121 through 122 is the mutator for the __pass_cap attribute, and the get_pass_cap method in lines 127 through 128 is the accessor for the attribute.

Program 12-6 demonstrates each of the classes we have discussed so far. It creates a Car object, a Truck object, and an SUV object.

Program 12-6 (car_truck_suv_demo.py)

```
# This program creates a Car object, a Truck object,
# and an SUV object.

import vehicles

def main():
    # Create a Car object for a used 2001 BMW
    # with 70,000 miles, priced at $15,000, with
    # 4 doors.
    car = vehicles.Car('BMW', 2001, 70000, 15000.0, 4)

    # Create a Truck object for a used 2002
    # Toyota pickup with 40,000 miles, priced
    # at $12,000, with 4-wheel drive.
    truck = vehicles.Truck('Toyota', 2002, 40000, 12000.0, '4WD')

    # Create an SUV object for a used 2000
    # Volvo with 30,000 miles, priced
    # at $18,500, with 5 passenger capacity.
    suv = vehicles.SUV('Volvo', 2000, 30000, 18500.0, 5)

    print('USED CAR INVENTORY')
    print('===================')

    # Display the car's data.
    print('The following car is in inventory:')
    print('Make:', car.get_make())
    print('Model:', car.get_model())
    print('Mileage:', car.get_mileage())
    print('Price:', car.get_price())
    print('Number of doors:', car.get_doors())
    print()

    # Display the truck's data.
    print('The following pickup truck is in inventory.')
```

(program continues)

Program 12-6 *(continued)*

```
    print('Make:', truck.get_make())
    print('Model:', truck.get_model())
    print('Mileage:', truck.get_mileage())
    print('Price:', truck.get_price())
    print('Drive type:', truck.get_drive_type())
    print()

    # Display the SUV's data.
    print('The following SUV is in inventory.')
    print('Make:', suv.get_make())
    print('Model:', suv.get_model())
    print('Mileage:', suv.get_mileage())
    print('Price:', suv.get_price())
    print('Passenger Capacity:', suv.get_pass_cap())

# Call the main function.
main()
```

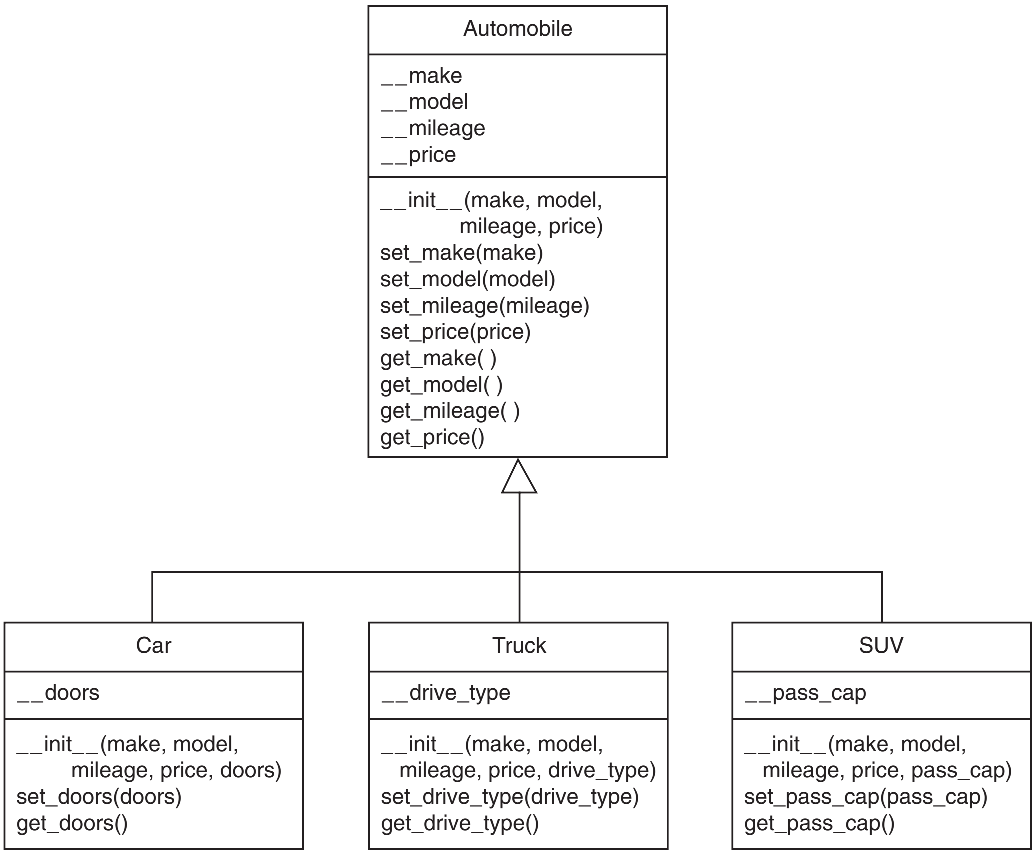
Program Output

```
USED CAR INVENTORY
==================
The following car is in inventory:
Make: BMW
Model: 2001
Mileage: 70000
Price: 15000.0
Number of doors: 4

The following pickup truck is in inventory.
Make: Toyota
Model: 2002
Mileage: 40000
Price: 12000.0
Drive type: 4WD

The following SUV is in inventory.
Make: Volvo
Model: 2000
Mileage: 30000
Price: 18500.0
Passenger Capacity: 5
```

Inheritance in UML Diagrams

You show inheritance in a UML diagram by drawing a line with an open arrowhead from the subclass to the superclass. (The arrowhead points to the superclass.) Figure 12-2 is a UML diagram showing the relationship between the `Automobile`, `Car`, `Truck`, and `SUV` classes.

Figure 12-2 UML diagram showing inheritance



In the Spotlight:
Using Inheritance

Bank Financial Systems, Inc. develops financial software for banks and credit unions. The company is developing a new object-oriented system that manages customer accounts. One of your tasks is to develop a class that represents a savings account. The data that must be held by an object of this class is:

- The account number
- The interest rate
- The account balance

You must also develop a class that represents a certificate of deposit (CD) account. The data that must be held by an object of this class is:

- The account number
- The interest rate
- The account balance
- The account maturity date

As you analyze these requirements, you realize that a CD account is really a specialized version of a savings account. The class that represents a CD will hold all of the same data as the class that represents a savings account, plus an extra attribute for the maturity date. You decide to design a `SavingsAccount` class to represent a savings account, and then design a subclass of `SavingsAccount` named `CD` to represent a CD account. You will store both of these classes in a module named `accounts`. Program 12-7 shows the code for the `SavingsAccount` class.

Program 12-7 (Lines 1 through 37 of accounts.py)

```
# The SavingsAccount class represents a
# savings account.

class SavingsAccount:

    # The __init__ method accepts arguments for the
    # account number, interest rate, and balance.

    def __init__(self, account_num, int_rate, bal):
        self.__account_num = account_num
        self.__interest_rate = int_rate
        self.__balance = bal

    # The following methods are mutators for the
    # data attributes.

    def set_account_num(self, account_num):
        self.__account_num = account_num

    def set_interest_rate(self, int_rate):
        self.__interest_rate = int_rate

    def set_balance(self, bal):
        self.__balance = bal

    # The following methods are accessors for the
    # data attributes.

    def get_account_num(self):
        return self.__account_num

    def get_interest_rate(self):
        return self.__interest_rate

    def get_balance(self):
        return self.__balance

```

The class's `__init__` method appears in lines 9 through 12. The `__init__` method accepts arguments for the account number, interest rate, and balance. These arguments are used to initialize data attributes named `__account_num`, `__interest_rate`, and `__balance`.

The `set_account_num`, `set_interest_rate`, and `set_balance` methods that appear in lines 17 through 24 are mutators for the data attributes. The `get_account_num`, `get_interest_rate`, and `get_balance` methods that appear in lines 29 through 36 are accessors.

The `CD` class is shown in the next part of Program 12-7.

Program 12-7 (Lines 38 through 65 of accounts.py)

```
38  # The CD account represents a certificate of
39  # deposit (CD) account. It is a subclass of
40  # the SavingsAccount class.
41
42  class CD(SavingsAccount):
43
44      # The init method accepts arguments for the
45      # account number, interest rate, balance, and
46      # maturity date.
47
48      def __init__(self, account_num, int_rate, bal, mat_date):
49          # Call the superclass __init__ method.
50          SavingsAccount.__init__(self, account_num, int_rate, bal)
51
52          # Initialize the __maturity_date attribute.
53          self.__maturity_date = mat_date
54
55      # The set_maturity_date is a mutator for the
56      # __maturity_date attribute.
57
58      def set_maturity_date(self, mat_date):
59          self.__maturity_date = mat_date
60
61      # The get_maturity_date method is an accessor
62      # for the __maturity_date attribute.
63
64      def get_maturity_date(self):
65          return self.__maturity_date
```

The `CD` class's `__init__` method appears in lines 48 through 53. It accepts arguments for the account number, interest rate, balance, and maturity date. Line 50 calls the `SavingsAccount` class's `__init__` method, passing the arguments for the account number, interest rate, and balance. After the `SavingsAccount` class's `__init__` method executes, the `__account_num`, `__interest_rate`, and `__balance` attributes will be created and initialized. Then the statement in line 53 creates the `__maturity_date` attribute.

The set_maturity_date method in lines 58 through 59 is the mutator for the __maturity_date attribute, and the get_maturity_date method in lines 64 through 65 is the accessor.

To test the classes, we use the code shown in Program 12-8. This program creates an instance of the SavingsAccount class to represent a savings account, and an instance of the CD account to represent a certificate of deposit account.

Program 12-8 (account_demo.py)

```
# This program creates an instance of the SavingsAccount
# class and an instance of the CD account.

import accounts

def main():
    # Get the account number, interest rate,
    # and account balance for a savings account.
    print('Enter the following data for a savings account.')
    acct_num = input('Account number: ')
    int_rate = float(input('Interest rate: '))
    balance = float(input('Balance: '))

    # Create a CD object.
    savings = accounts.SavingsAccount(acct_num, int_rate, \
                                      balance)

    # Get the account number, interest rate,
    # account balance, and maturity date for a CD.
    print('Enter the following data for a CD.')
    acct_num = input('Account number: ')
    int_rate = float(input('Interest rate: '))
    balance = float(input('Balance: '))
    maturity = input('Maturity date: ')

    # Create a CD object.
    cd = accounts.CD(acct_num, int_rate, balance, maturity)

    # Display the data entered.
    print('Here is the data you entered:')
    print()
    print('Savings Account')
    print('---------------')
    print('Account number:', savings.get_account_num())
    print('Interest rate:', savings.get_interest_rate())
    print('Balance: $', \
          format(savings.get_balance(), ',.2f'), \
          sep='')
```