CHAPTER 11

Classes and Object-Oriented Programming

TOPICS

11.1 Procedural and Object-Oriented Programming

CONCEPT: Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered on objects. Objects are created from abstract data types that encapsulate data and functions together.

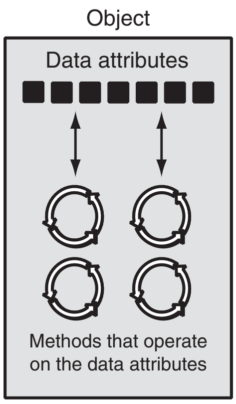
There are primarily two methods of programming in use today: procedural and object-oriented. The earliest programming languages were procedural, meaning a program was made of one or more procedures. You can think of a *procedure* simply as a function that performs a specific task such as gathering input from the user, performing calculations, reading or writing files, displaying output, and so on. The programs that you have written so far have been procedural in nature.

Typically, procedures operate on data items that are separate from the procedures. In a procedural program, the data items are commonly passed from one procedure to another. As you might imagine, the focus of procedural programming is on the creation of procedures that operate on the program's data. The separation of data and the code that operates on the data can lead to problems, however, as the program becomes larger and more complex.

For example, suppose you are part of a programming team that has written an extensive customer database program. The program was initially designed so that a customer's name, address, and phone number were referenced by three variables. Your job was to design several functions that accept those three variables as arguments and perform operations on them. The software has been operating successfully for some time, but your team has been asked to update it by adding several new features. During the revision process, the senior programmer informs you that the customer's name, address, and phone number will no longer be stored in variables. Instead, they will be stored in a list. This means that you will have to modify all of the functions that you have designed so that they accept and work with a list instead of the three variables. Making these extensive modifications not only is a great deal of work, but also opens the opportunity for errors to appear in your code.

Whereas procedural programming is centered on creating procedures (functions), *object-oriented programming* (*OOP*) is centered on creating objects. An *object* is a software entity that contains both data and procedures. The data contained in an object is known as the object's *data attributes*. An object's data attributes are simply variables that reference data. The procedures that an object performs are known as *methods*. An object's methods are functions that perform operations on the object's data attributes. The object is, conceptually, a self-contained unit that consists of data attributes and methods that operate on the data attributes. This is illustrated in Figure 11-1.

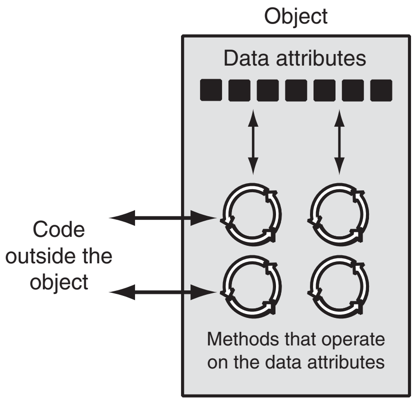
Figure 11-1 An object contains data attributes and methods



OOP addresses the problem of code and data separation through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data attributes from code that is outside the object. Only the object's methods may directly access and make changes to the object's data attributes.

An object typically hides its data, but allows outside code to access its methods. As shown in Figure 11-2, the object's methods provide programming statements outside the object with indirect access to the object's data attributes.

Figure 11-2 Code outside the object interacts with the object's methods



When an object's data attributes are hidden from outside code, and access to the data attributes is restricted to the object's methods, the data attributes are protected from accidental corruption. In addition, the code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's methods. When a programmer changes the structure of an object's internal data attributes, he or she also modifies the object's methods so that they may properly operate on the data. The way in which outside code interacts with the methods, however, does not change.

Object Reusability

In addition to solving the problems of code and data separation, the use of OOP has also been encouraged by the trend of *object reusability*. An object is not a stand-alone program, but is used by programs that need its services. For example, Sharon is a programmer who has developed a set of objects for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her objects are coded to perform all of the necessary 3D mathematical operations and handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's objects to perform the 3D rendering (for a small fee, of course!).

An Everyday Example of an Object

Imagine that your alarm clock is actually a software object. If it were, it would have the following data attributes:

- `current_second` (a value in the range of 0–59)
- `current_minute` (a value in the range of 0–59)
- `current_hour` (a value in the range of 1–12)
- `alarm_time` (a valid hour and minute)
- `alarm_is_set` (True or False)

As you can see, the data attributes are merely values that define the *state* that the alarm clock is currently in. You, the user of the alarm clock object, cannot directly manipulate these data attributes because they are *private*. To change a data attribute's value, you must use one of the object's methods. The following are some of the alarm clock object's methods:

- `set_time`
- `set_alarm_time`
- `set_alarm_on`
- `set_alarm_off`

Each method manipulates one or more of the data attributes. For example, the `set_time` method allows you to set the alarm clock's time. You activate the method by pressing a button on top of the clock. By using another button, you can activate the `set_alarm_time` method.

In addition, another button allows you to execute the `set_alarm_on` and `set_alarm_off` methods. Notice that all of these methods can be activated by you, who are outside the alarm clock. Methods that can be accessed by entities outside the object are known as *public methods*.

The alarm clock also has *private methods*, which are part of the object's private, internal workings. External entities (such as you, the user of the alarm clock) do not have direct access to the alarm clock's private methods. The object is designed to execute these methods automatically and hide the details from you. The following are the alarm clock object's private methods:

- `increment_current_second`
- `increment_current_minute`
- `increment_current_hour`
- `sound_alarm`

Every second the `increment_current_second` method executes. This changes the value of the `current_second` data attribute. If the `current_second` data attribute is set to 59 when this method executes, the method is programmed to reset `current_second` to 0, and then cause the `increment_current_minute` method to execute. This method adds 1 to the `current_minute` data attribute, unless it is set to 59. In that case, it resets `current_minute` to 0 and causes the `increment_current_hour` method to execute. The `increment_current_minute` method compares the new time to the `alarm_time`. If the two times match and the alarm is turned on, the `sound_alarm` method is executed.

Checkpoint

11.1 What is an object?

11.2 What is encapsulation?

11.3 Why is an object's internal data usually hidden from outside code?

11.4 What are public methods? What are private methods?

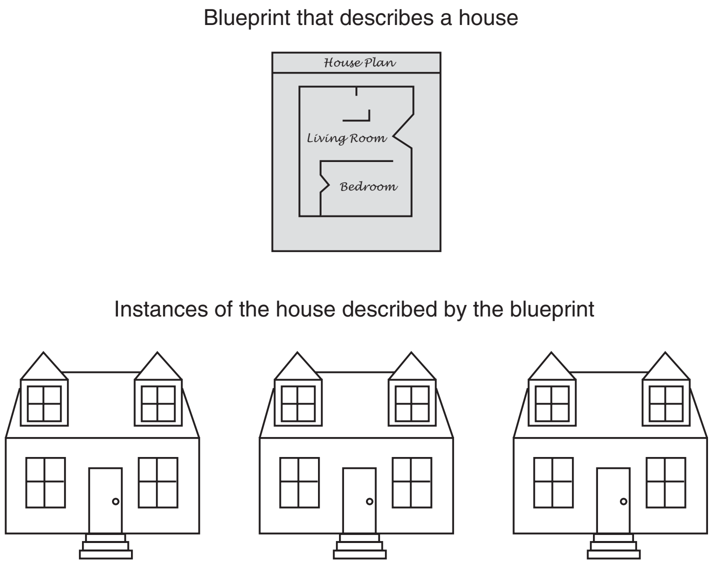
11.2 Classes

CONCEPT: A class is code that specifies the data attributes and methods for a particular type of object.

VideoNote
Classes and Objects

Now, let's discuss how objects are created in software. Before an object can be created, it must be designed by a programmer. The programmer determines the data attributes and methods that are necessary, and then creates a *class*. A class is code that specifies the data attributes and methods of a particular type of object. Think of a class as a "blueprint" that objects may be created from. It serves a similar purpose as the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an *instance* of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 11-3.

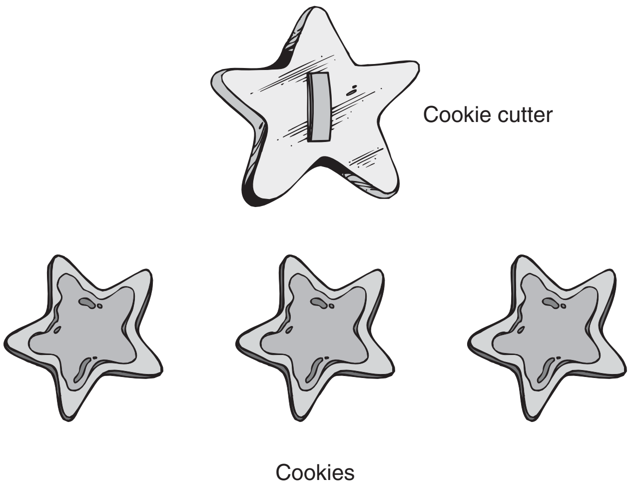
Figure 11-3 A blueprint and houses built from the blueprint



Another way of thinking about the difference between a class and an object is to think of the difference between a cookie cutter and a cookie. While a cookie cutter itself is not a cookie, it describes a cookie. The cookie cutter can be used to make several cookies, as shown in Figure 11-4. Think of a class as a cookie cutter and the objects created from the class as cookies.

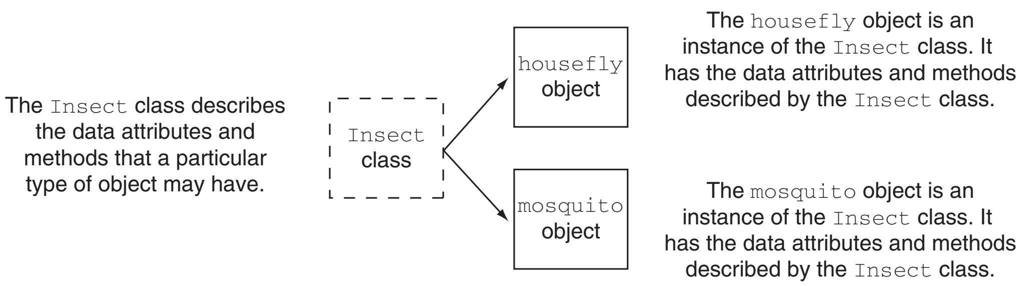
So, a class is a description of an object's characteristics. When the program is running, it can use the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

Figure 11-4 The cookie cutter metaphor



For example, Jessica is an entomologist (someone who studies insects) and she also enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named `Insect`, which specifies characteristics that are common to all types of insects. The `Insect` class is a specification that objects may be created from. Next, she writes programming statements that create an object named `housefly`, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about a housefly. It has the data attributes and methods specified by the `Insect` class. Then she writes programming statements that create an object named `mosquito`. The `mosquito` object is also an instance of the `Insect` class. It has its own area in memory, and stores data about a mosquito. Although the `housefly` and `mosquito` objects are separate entities in the computer's memory, they were both created from the `Insect` class. This means that each of the objects has the data attributes and methods described by the `Insect` class. This is illustrated in Figure 11-5.

Figure 11-5 The `housefly` and `mosquito` objects are instances of the `Insect` class



Class Definitions

To create a class, you write a *class definition*. A class definition is a set of statements that define a class's methods and data attributes. Let's look at a simple example. Suppose we are writing a program to simulate the tossing of a coin. In the program we need to repeatedly toss the coin and each time determine whether it landed heads up or tails up. Taking an object-oriented approach, we will write a class named `Coin` that can perform the behaviors of the coin.

Program 11-1 shows the class definition, which we will explain shortly. Note that this is not a complete program. We will add to it as we go along.

Program 11-1 (Coin class, not a complete program)

```
1  import random
2
3  # The Coin class simulates a coin that can
4  # be flipped.
5
6  class Coin:
7
8      # The __init__ method initializes the
9      # sideup data attribute with 'Heads'.
10
11     def __init__(self):
12         self.sideup = 'Heads'
13
14     # The toss method generates a random number
15     # in the range of 0 through 1. If the number
16     # is 0, then sideup is set to 'Heads'.
17     # Otherwise, sideup is set to 'Tails'.
18
19     def toss(self):
20         if random.randint(0, 1) == 0:
21             self.sideup = 'Heads'
22         else:
23             self.sideup = 'Tails'
24
25     # The get_sideup method returns the value
26     # referenced by sideup.
27
28     def get_sideup(self):
29         return self.sideup
```

In line 1 we import the `random` module. This is necessary because we use the `randint` function to generate a random number. Line 6 is the beginning of the class definition. It begins with the keyword `class`, followed by the class name, which is `Coin`, followed by a colon.

The same rules that apply to variable names also apply to class names. However, notice that we started the class name, `Coin`, with an uppercase letter. This is not a requirement, but it is a widely used convention among programmers. This helps to easily distinguish class names from variable names when reading code.

The `Coin` class has three methods:

- The `__init__` method appears in lines 11 through 12.
- The `toss` method appears in lines 19 through 23.
- The `get_sideup` method appears in lines 28 through 29.

Except for the fact that they appear inside a class, notice that these method definitions look like any other function definition in Python. They start with a header line, which is followed by an indented block of statements.

Take a closer look at the header for each of the method definitions (lines 11, 19, and 28) and notice that each method has a parameter variable named `self`:

Line 11:	`def __init__(self):`
Line 19:	`def toss(self):`
Line 28:	`def get_sideup(self):`

The `self` parameter[1] is required in every method of a class. Recall from our earlier discussion on object-oriented programming that a method operates on a specific object's data attributes. When a method executes, it must have a way of knowing which object's data attributes it is supposed to operate on. That's where the `self` parameter comes in. When a method is called, Python makes the `self` parameter reference the specific object that the method is supposed to operate on.

Let's look at each of the methods. The first method, which is named `__init__`, is defined in lines 11 through 12:

```
def __init__(self):
    self.sideup = 'Heads'
```

Most Python classes have a special method named `__init__`, which is automatically executed when an instance of the class is created in memory. The `__init__` method is commonly known as an *initializer method* because it initializes the object's data attributes. (The name of the method starts with two underscore characters, followed by the word `init`, followed by two more underscore characters.)

Immediately after an object is created in memory, the `__init__` method executes, and the `self` parameter is automatically assigned the object that was just created. Inside the method, the statement in line 12 executes:

```
self.sideup = 'Heads'
```

This statement assigns the string `'Heads'` to the `sideup` data attribute belonging to the object that was just created. As a result of this `__init__` method, each object that we create from the `Coin` class will initially have a `sideup` attribute that is set to `'Heads'`.

NOTE: The `__init__` method is usually the first method inside a class definition.

[1] The parameter must be present in a method. You are not required to name it `self`, but this is strongly recommended to conform with standard practice.

The `toss` method appears in lines 19 through 23:

```
def toss(self):
    if random.randint(0, 1) == 0:
        self.sideup = 'Heads'
    else:
        self.sideup = 'Tails'
```

This method also has the required `self` parameter variable. When the `toss` method is called, `self` will automatically reference the object that the method is to operate on.

The `toss` method simulates the tossing of the coin. When the method is called, the `if` statement in line 20 calls the `random.randint` function to get a random integer in the range of 0 through 1. If the number is 0, then the statement in line 21 assigns `'Heads'` to `self.sideup`. Otherwise, the statement in line 23 assigns `'Tails'` to `self.sideup`.

The `get_sideup` method appears in lines 28 through 29:

```
def get_sideup(self):
    return self.sideup
```

Once again, the method has the required `self` parameter variable. This method simply returns the value of `self.sideup`. We call this method any time we want to know which side of the coin is facing up.

To demonstrate the `Coin` class, we need to write a complete program that uses it to create an object. Program 11-2 shows an example. The `Coin` class definition appears in lines 6 through 29. The program has a `main` function, which appears in lines 32 through 44.

Program 11-2 (coin_demo1.py)

```
1   import random
2
3   # The Coin class simulates a coin that can
4   # be flipped.
5
6   class Coin:
7
8       # The __init__ method initializes the
9       # sideup data attribute with 'Heads'.
10
11      def __init__(self):
12          self.sideup = 'Heads'
13
14      # The toss method generates a random number
15      # in the range of 0 through 1. If the number
16      # is 0, then sideup is set to 'Heads'.
17      # Otherwise, sideup is set to 'Tails'.
18
19      def toss(self):
20          if random.randint(0, 1) == 0:
```

(program continues)

Program 11-2 *(continued)*

```
21                self.sideup = 'Heads'
22            else:
23                self.sideup = 'Tails'
24
25        # The get_sideup method returns the value
26        # referenced by sideup.
27
28        def get_sideup(self):
29            return self.sideup
30
31    # The main function.
32    def main():
33        # Create an object from the Coin class.
34        my_coin = Coin()
35
36        # Display the side of the coin that is facing up.
37        print('This side is up:', my_coin.get_sideup())
38
39        # Toss the coin.
40        print('I am tossing the coin...')
41        my_coin.toss()
42
43        # Display the side of the coin that is facing up.
44        print('This side is up:', my_coin.get_sideup())
45
46    # Call the main function.
47    main()
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Tails
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Tails
```

Take a closer look at the statement in line 34:

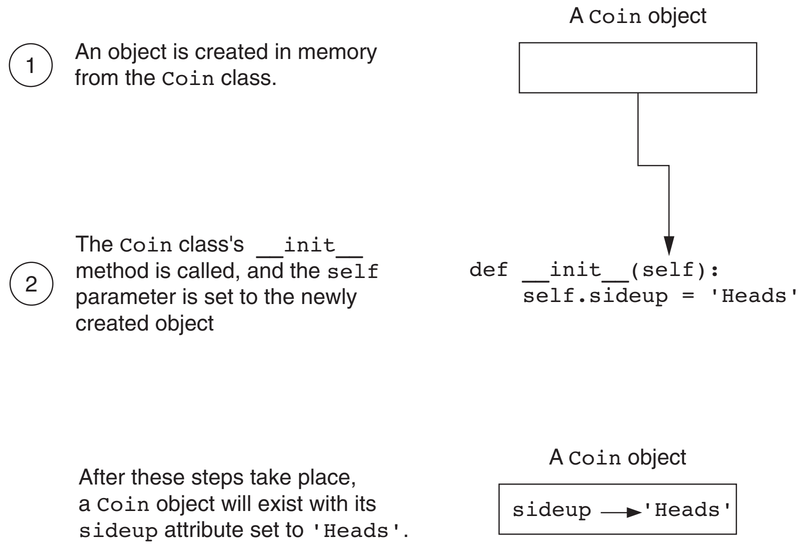
```
my_coin = Coin()
```

The expression `Coin()` that appears on the right side of the = operator causes two things to happen:

1. An object is created in memory from the `Coin` class.
2. The `Coin` class's `__init__` method is executed, and the `self` parameter is automatically set to the object that was just created. As a result, that object's `sideup` attribute is assigned the string `'Heads'`.

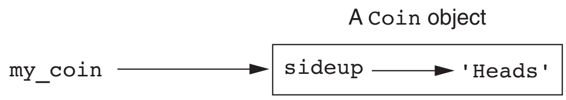
Figure 11-6 illustrates these steps.

Figure 11-6 Actions caused by the `Coin()` expression



After this, the = operator assigns the `Coin` object that was just created to the `my_coin` variable. Figure 11-7 shows that after the statement in line 12 executes, the `my_coin` variable will reference a `Coin` object, and that object's `sideup` attribute will be assigned the string `'Heads'`.

Figure 11-7 The `my_coin` variable references a `Coin` object



The next statement to execute is line 37:

```
print('This side is up:', my_coin.get_sideup())
```

This statement prints a message indicating the side of the coin that is facing up. Notice that the following expression appears in the statement:

```
my_coin.get_sideup()
```

This expression uses the object referenced by `my_coin` to call the `get_sideup` method. When the method executes, the `self` parameter will reference the `my_coin` object. As a result, the method returns the string `'Heads'`.

Notice that we did not have to pass an argument to the `sideup` method, despite the fact that it has the `self` parameter variable. When a method is called, Python automatically passes a reference to the calling object into the method's first parameter. As a result, the `self` parameter will automatically reference the object that the method is to operate on.

Lines 40 and 41 are the next statements to execute:

```
print('I am tossing the coin...')
my_coin.toss()
```

The statement in line 41 uses the object referenced by `my_coin` to call the `toss` method. When the method executes, the `self` parameter will reference the `my_coin` object. The method will randomly generate a number and use that number to change the value of the object's `sideup` attribute.

Line 44 executes next. This statement calls `my_coin.get_sideup()` to display the side of the coin that is facing up.

Hiding Attributes

Earlier in this chapter we mentioned that an object's data attributes should be private, so that only the object's methods can directly access them. This protects the object's data attributes from accidental corruption. However, in the `Coin` class that was shown in the previous example, the `sideup` attribute is not private. It can be directly accessed by statements that are not in a `Coin` class method. Program 11-3 shows an example. Note that lines 1 through 30 are not shown to conserve space. Those lines contain the `Coin` class, and they are the same as lines 1 through 30 in Program 11-2.

Program 11-3 (coin_demo2.py)

Lines 1 through 30 are omitted. These lines are the same as lines 1 through 30 in Program 11-2.

```
31 # The main function.
32 def main():
33     # Create an object from the Coin class.
34     my_coin = Coin()
35
36     # Display the side of the coin that is facing up.
37     print('This side is up:', my_coin.get_sideup())
38
39     # Toss the coin.
40     print('I am tossing the coin...')
41     my_coin.toss()
42
43     # But now I'm going to cheat! I'm going to
44     # directly change the value of the object's
45     # sideup attribute to 'Heads'.
46     my_coin.sideup = 'Heads'
47
48     # Display the side of the coin that is facing up.
```

```
49      print('This side is up:', my_coin.get_sideup())
50
51  # Call the main function.
52  main()
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Line 34 creates a `Coin` object in memory and assigns it to the `my_coin` variable. The statement in line 37 displays the side of the coin that is facing up, and then line 41 calls the object's `toss` method. Then the statement in line 46 directly assigns the string `'Heads'` to the object's `sideup` attribute:

```
my_coin.sideup = 'Heads'
```

Regardless of the outcome of the `toss` method, this statement will change the `my_coin` object's `sideup` attribute to `'Heads'`. As you can see from the three sample runs of the program, the coin always lands heads up!

If we truly want to simulate a coin that is being tossed, then we don't want code outside the class to be able to change the result of the `toss` method. To prevent this from happening, we need to make the `sideup` attribute private. In Python you can hide an attribute by starting its name with two underscore characters. If we change the name of the `sideup` attribute to `__sideup`, then code outside the `Coin` class will not be able to access it. Program 11-4 shows a new version of the `Coin` class, with this change made.

Program 11-4 (coin_demo3.py)

```
import random

# The Coin class simulates a coin that can
# be flipped.

class Coin:

    # The __init__ method initializes the
```

(program continues)

Program 11-4 *(continued)*

```
    # __sideup data attribute with 'Heads'.

    def __init__(self):
        self.__sideup = 'Heads'

    # The toss method generates a random number
    # in the range of 0 through 1. If the number
    # is 0, then sideup is set to 'Heads'.
    # Otherwise, sideup is set to 'Tails'.

    def toss(self):
        if random.randint(0, 1) == 0:
            self.__sideup = 'Heads'
        else:
            self.__sideup = 'Tails'

    # The get_sideup method returns the value
    # referenced by sideup.

    def get_sideup(self):
        return self.__sideup

# The main function.
def main():
    # Create an object from the Coin class.
    my_coin = Coin()

    # Display the side of the coin that is facing up.
    print('This side is up:', my_coin.get_sideup())

    # Toss the coin.
    print('I am going to toss the coin ten times:')
    for count in range(10):
        my_coin.toss()
        print(my_coin.get_sideup())

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

Program Output

```
This side is up: Heads
I am going to toss the coin ten times:
Tails
Heads
Heads
```

```
Tails
Tails
Tails
Tails
Tails
Heads
Heads
```

Storing Classes in Modules

The programs you have seen so far in this chapter have the `Coin` class definition in the same file as the programming statements that use the `Coin` class. This approach works fine with small programs that use only one or two classes. As programs use more classes, however, the need to organize those classes becomes greater.

Programmers commonly organize their class definitions by storing them in modules. Then the modules can be imported into any programs that need to use the classes they contain. For example, suppose we decide to store the `Coin` class in a module named `coin`. Program 11-5 shows the contents of the `coin.py` file. Then, when we need to use the `Coin` class in a program, we can import the `coin` module. This is demonstrated in Program 11-6.

Program 11-5 (coin.py)

```
import random

# The Coin class simulates a coin that can
# be flipped.

class Coin:

    # The __init__ method initializes the
    # __sideup data attribute with 'Heads'.

    def __init__(self):
        self.__sideup = 'Heads'

    # The toss method generates a random number
    # in the range of 0 through 1. If the number
    # is 0, then sideup is set to 'Heads'.
    # Otherwise, sideup is set to 'Tails'.

    def toss(self):
        if random.randint(0, 1) == 0:
            self.__sideup = 'Heads'
        else:
            self.__sideup = 'Tails'
```

(program continues)

Program 11-5 *(continued)*

```

    # The get_sideup method returns the value
    # referenced by sideup.

    def get_sideup(self):
        return self.__sideup
```

Program 11-6 (coin_demo4.py)

```
# This program imports the coin module and
# creates an instance of the Coin class.

import coin

def main():
    # Create an object from the Coin class.
    my_coin = coin.Coin()

    # Display the side of the coin that is facing up.
    print('This side is up:', my_coin.get_sideup())

    # Toss the coin.
    print('I am going to toss the coin ten times:')
    for count in range(10):
        my_coin.toss()
        print(my_coin.get_sideup())

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

Program Output

```
This side is up: Heads
I am going to toss the coin ten times:
Tails
Tails
Heads
Tails
Heads
Heads
Tails
Heads
Tails
Tails
```

Line 4 imports the `coin` module. Notice that in line 8 we had to qualify the name of the `Coin` class by prefixing it with the name of the module, followed by a dot:

```
my_coin = coin.Coin()
```

The `BankAccount` Class

Let's look at another example. Program 11-7 shows a `BankAccount` class, stored in a module named `bankaccount`. Objects that are created from this class will simulate bank accounts, allowing us to have a starting balance, make deposits, make withdrawals, and get the current balance.

Program 11-7 (bankaccount.py)

```
1  # The BankAccount class simulates a bank account.
2
3  class BankAccount:
4
5      # The __init__ method accepts an argument for
6      # the account's balance. It is assigned to
7      # the __balance attribute.
8
9      def __init__(self, bal):
10         self.__balance = bal
11
12     # The deposit method makes a deposit into the
13     # account.
14
15     def deposit(self, amount):
16         self.__balance += amount
17
18     # The withdraw method withdraws an amount
19     # from the account.
20
21     def withdraw(self, amount):
22         if self.__balance >= amount:
23             self.__balance -= amount
24         else:
25             print('Error: Insufficient funds')
26
27     # The get_balance method returns the
28     # account balance.
29
30     def get_balance(self):
31         return self.__balance
```

Notice that the `__init__` method has two parameter variables: `self` and `bal`. The `bal` parameter will accept the account's starting balance as an argument. In line 10 the `bal` parameter amount is assigned to the object's `__balance` attribute.

The `deposit` method is in lines 15 through 16. This method has two parameter variables: `self` and `amount`. When the method is called, the amount that is to be deposited into the account is passed into the `amount` parameter. The value of the parameter is then added to the `__balance` attribute in line 16.

The `withdraw` method is in lines 21 through 25. This method has two parameter variables: `self` and `amount`. When the method is called, the amount that is to be withdrawn from the account is passed into the `amount` parameter. The `if` statement that begins in line 22 determines whether there is enough in the account balance to make the withdrawal. If so, `amount` is subtracted from `__balance` in line 23. Otherwise line 25 displays the message `'Error: Insufficient funds'`.

The `get_balance` method is in lines 30 through 31. This method returns the value of the `__balance` attribute.

Program 11-8 demonstrates how to use the class.

Program 11-8 (account_test.py)

```
# This program demonstrates the BankAccount class.

import bankaccount

def main():
    # Get the starting balance.
    start_bal = float(input('Enter your starting balance: '))

    # Create a BankAccount object.
    savings = bankaccount.BankAccount(start_bal)

    # Deposit the user's paycheck.
    pay = float(input('How much were you paid this week? '))
    print('I will deposit that into your account.')
    savings.deposit(pay)

    # Display the balance.
    print('Your account balance is $', \
          format(savings.get_balance(), ',.2f'),
          sep='')

    # Get the amount to withdraw.
    cash = float(input('How much would you like to withdraw? '))
    print('I will withdraw that from your account.')
    savings.withdraw(cash)

    # Display the balance.
    print('Your account balance is $', \
          format(savings.get_balance(), ',.2f'),
          sep='')
```

```
31
32   # Call the main function.
33   main()
```

Program Output (with input shown in bold)

```
Enter your starting balance: 1000.00 [Enter]
How much were you paid this week? 500.00 [Enter]
I will deposit that into your account.
Your account balance is $1,500.00
How much would you like to withdraw? 1200.00 [Enter]
I will withdraw that from your account.
Your account balance is $300.00
```

Program Output (with input shown in bold)

```
Enter your starting balance: 1000.00 [Enter]
How much were you paid this week? 500.00 [Enter]
I will deposit that into your account.
Your account balance is $1,500.00
How much would you like to withdraw? 2000.00 [Enter]
I will withdraw that from your account.
Error: Insufficient funds
Your account balance is $1,500.00
```

Line 7 gets the starting account balance from the user and assigns it to the `start_bal` variable. Line 10 creates an instance of the `BankAccount` class and assigns it to the `savings` variable. Take a closer look at the statement:

```
savings = bankaccount.BankAccount(start_bal)
```

Notice that the `start_bal` variable is listed inside the parentheses. This causes the `start_bal` variable to be passed as an argument to the `__init__` method. In the `__init__` method, it will be passed into the `bal` parameter.

Line 13 gets the amount of the user's pay and assigns it to the `pay` variable. In line 15 the `savings.deposit` method is called, passing the `pay` variable as an argument. In the `deposit` method, it will be passed into the `amount` parameter.

The statement in lines 18 through 20 displays the account balance. It displays the value returned from the `savings.get_balance` method.

Line 23 gets the amount that the user wants to withdraw and assigns it to the `cash` variable. In line 25 the `savings.withdraw` method is called, passing the `cash` variable as an argument. In the `withdraw` method, it will be passed into the `amount` parameter. The statement in lines 28 through 30 displays the ending account balance.

The `__str__` method

Quite often we need to display a message that indicates an object's state. An object's *state* is simply the values of the object's attributes at any given moment. For example, recall that the `BankAccount` class has one data attribute: `__balance`. At any given moment, a `BankAccount` object's `__balance` attribute will reference some value. The value of the `__balance` attribute represents the object's state at that moment. The following might be an example of code that displays a `BankAccount` object's state:

```
account = bankaccount.BankAccount(1500.0)
print('The balance is $', format(savings.get_balance(), ',.2f'), sep='')
```

The first statement creates a `BankAccount` object, passing the value 1500.0 to the `__init__` method. After this statement executes, the `account` variable will reference the `BankAccount` object. The second line displays a string showing the value of the object's `__balance` attribute. The output of this statement will look like this:

```
The balance is $1,500.00
```

Displaying an object's state is a common task. It is so common that many programmers equip their classes with a method that returns a string containing the object's state. In Python, you give this method the special name `__str__`. Program 11-9 shows the `BankAccount` class with a `__str__` method added to it. The `__str__` method appears in lines 36 through 37. It returns a string indicating the account balance.

Program 11-9 (bankaccount2.py)

```
1  # The BankAccount class simulates a bank account.
2
3  class BankAccount:
4
5      # The __init__ method accepts an argument for
6      # the account's balance. It is assigned to
7      # the __balance attribute.
8
9      def __init__(self, bal):
10         self.__balance = bal
11
12     # The deposit method makes a deposit into the
13     # account.
14
15     def deposit(self, amount):
16         self.__balance += amount
17
18     # The withdraw method withdraws an amount
19     # from the account.
20
21     def withdraw(self, amount):
22         if self.__balance >= amount:
23             self.__balance -= amount
24         else:
25             print('Error: Insufficient funds')
26
27     # The get_balance method returns the
28     # account balance.
29
30     def get_balance(self):
```

```
        return self.__balance

    # The __str__ method returns a string
    # indicating the object's state.

    def __str__(self):
        return 'The balance is $' + format(self.__balance, ',.2f')
```

You do not directly call the `__str__` method. Instead, it is automatically called when you pass an object's as an argument to the `print` function. Program 11-10 shows an example.

Program 11-10 (account_test2.py)

```
# This program demonstrates the BankAccount class
# with the __str__ method added to it.

import bankaccount2

def main():
    # Get the starting balance.
    start_bal = float(input('Enter your starting balance: '))

    # Create a BankAccount object.
    savings = bankaccount2.BankAccount(start_bal)

    # Deposit the user's paycheck.
    pay = float(input('How much were you paid this week? '))
    print('I will deposit that into your account.')
    savings.deposit(pay)

    # Display the balance.
    print(savings)

    # Get the amount to withdraw.
    cash = float(input('How much would you like to withdraw? '))
    print('I will withdraw that from your account.')
    savings.withdraw(cash)

    # Display the balance.
    print(savings)

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

Program Output (with input shown in bold)

```
Enter your starting balance: 1000.00 Enter
How much were you paid this week? 500.00 Enter
I will deposit that into your account.
```

(program output continues)

Program Output *(continued)*

```
The account balance is $1,500.00
How much would you like to withdraw? 1200.00 Enter
I will withdraw that from your account.
The account balance is $300.00
```

The name of the object, `savings`, is passed to the `print` function in lines 19 and 27. This causes the `BankAccount` class's `__str__` method to be called. The string that is returned from the `__str__` method is then displayed.

The `__str__` method is also called automatically when an object is passed as an argument to the built-in `str` function. Here is an example:

```
account = bankaccount2.BankAccount(1500.0)
message = str(account)
print(message)
```

In the second statement, the `account` object is passed as an argument to the `str` function. This causes the `BankAccount` class's `__str__` method to be called. The string that is returned is assigned to the `message` variable and then displayed by the `print` function in the third line.

Checkpoint

11.5 You hear someone make the following comment: "A blueprint is a design for a house. A carpenter can use the blueprint to build the house. If the carpenter wishes, he or she can build several identical houses from the same blueprint." Think of this as a metaphor for classes and objects. Does the blueprint represent a class, or does it represent an object?

11.6 In this chapter, we use the metaphor of a cookie cutter and cookies that are made from the cookie cutter to describe classes and objects. In this metaphor, are objects the cookie cutter, or the cookies?

11.7 What is the purpose of the `__init__` method? When does it execute?

11.8 What is the purpose of the `self` parameter in a method?

11.9 In a Python class, how do you hide an attribute from code outside the class?

11.10 What is the purpose of the `__str__` method?

11.11 How do you call the `__str__` method?

11.3 Working with Instances

CONCEPT: Each instance of a class has its own set of data attributes.

When a method uses the `self` parameter to create an attribute, the attribute belongs to the specific object that `self` references. We call these attributes *instance attributes*, because they belong to a specific instance of the class.

It is possible to create many instances of the same class in a program. Each instance will then have its own set of attributes. For example, look at Program 11-11. This program creates three instances of the `Coin` class. Each instance has its own `__sideup` attribute.

Program 11-11 (coin_demo5.py)

```
# This program imports the simulation module and
# creates three instances of the Coin class.

import coin

def main():
    # Create three objects from the Coin class.
    coin1 = coin.Coin()
    coin2 = coin.Coin()
    coin3 = coin.Coin()

    # Display the side of each coin that is facing up.
    print('I have three coins with these sides up:')
    print(coin1.get_sideup())
    print(coin2.get_sideup())
    print(coin3.get_sideup())
    print()

    # Toss the coin.
    print('I am tossing all three coins...')
    print()
    coin1.toss()
    coin2.toss()
    coin3.toss()

    # Display the side of each coin that is facing up.
    print('Now here are the sides that are up:')
    print(coin1.get_sideup())
    print(coin2.get_sideup())
    print(coin3.get_sideup())
    print()

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

Program Output

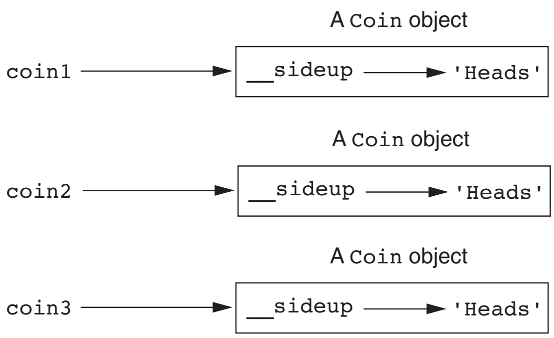
```
I have three coins with these sides up:
Heads
Heads
Heads

I am tossing all three coins...

Now here are the sides that are up:
Tails
Tails
Heads
```

In lines 8 through 10, the following statements create three objects, each an instance of the `Coin` class:

```
coin1 = coin.Coin()
coin2 = coin.Coin()
coin3 = coin.Coin()
```

Figure 11-8 illustrates how the `coin1`, `coin2`, and `coin3` variables reference the three objects after these statements execute. Notice that each object has its own `__sideup` attribute. Lines 14 through 16 display the values returned from each object's `get_sideup` method.

Figure 11-8 The `coin1`, `coin2`, and `coin3` variables reference three `Coin` objects

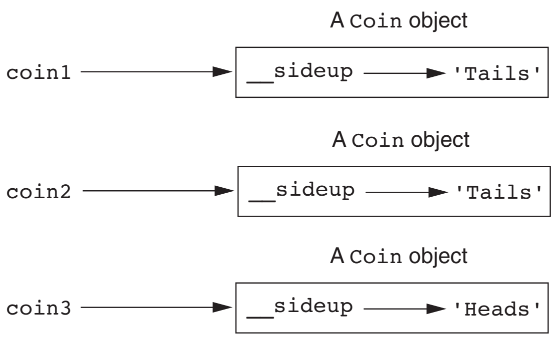


Then, the statements in lines 22 through 24 call each object's `toss` method:

```
coin1.toss()
coin2.toss()
coin3.toss()
```

Figure 11-9 shows how these statements changed each object's `__sideup` attribute in the program's sample run.

Figure 11-9 The objects after the `toss` method



In the Spotlight:

Creating the `CellPhone` Class

Wireless Solutions, Inc. is a business that sells cell phones and wireless service. You are a programmer in the company's IT department, and your team is designing a program to manage all of the cell phones that are in inventory. You have been asked to design a class that represents a cell phone. The data that should be kept as attributes in the class are as follows:

- The name of the phone's manufacturer will be assigned to the `__manufact` attribute.
- The phone's model number will be assigned to the `__model` attribute.
- The phone's retail price will be assigned to the `__retail_price` attribute.

The class will also have the following methods:

- An `__init__` method that accepts arguments for the manufacturer, model number, and retail price.
- A `set_manufact` method that accepts an argument for the manufacturer. This method will allow us to change the value of the `__manufact` attribute after the object has been created, if necessary.
- A `set_model` method that accepts an argument for the model. This method will allow us to change the value of the `__model` attribute after the object has been created, if necessary.
- A `set_retail_price` method that accepts an argument for the retail price. This method will allow us to change the value of the `__retail_price` attribute after the object has been created, if necessary.
- A `get_manufact` method that returns the phone's manufacturer.
- A `get_model` method that returns the phone's model number.
- A `get_retail_price` method that returns the phone's retail price.

Program 11-12 shows the class definition. The class is stored in a module named `cellphone`.

Program 11-12 (cellphone.py)

```
# The CellPhone class holds data about a cell phone.

class CellPhone:

    # The __init__ method initializes the attributes.

    def __init__(self, manufact, model, price):
        self.__manufact = manufact
        self.__model = model
        self.__retail_price = price

    # The set_manufact method accepts an argument for
    # the phone's manufacturer.

    def set_manufact(self, manufact):
        self.__manufact = manufact
```

(program continues)

Program 11-12 *(continued)*

```

    # The set_model method accepts an argument for
    # the phone's model number.

    def set_model(self, model):
        self.__model = model

    # The set_retail_price method accepts an argument
    # for the phone's retail price.

    def set_retail_price(self, price):
        self.__retail_price = price

    # The get_manufact method returns the
    # phone's manufacturer.

    def get_manufact(self):
        return self.__manufact

    # The get_model method returns the
    # phone's model number.

    def get_model(self):
        return self.__model

    # The get_retail_price method returns the
    # phone's retail price.

    def get_retail_price(self):
        return self.__retail_price
```

The `CellPhone` class will be imported into several programs that your team is developing. To test the class, you write the code in Program 11-13. This is a simple program that prompts the user for the phone's manufacturer, model number, and retail price. An instance of the `CellPhone` class is created and the data is assigned to its attributes.

Program 11-13 (cell_phone_test.py)

```
# This program tests the CellPhone class.

import cellphone

def main():
    # Get the phone data.
    man = input('Enter the manufacturer: ')
```

```
    mod = input('Enter the model number: ')
    retail = float(input('Enter the retail price: '))

    # Create an instance of the CellPhone class.
    phone = cellphone.CellPhone(man, mod, retail)

    # Display the data that was entered.
    print('Here is the data that you entered:')
    print('Manufacturer:', phone.get_manufact())
    print('Model Number:', phone.get_model())
    print('Retail Price: $', format(phone.get_retail_price(), ',.2f'), sep='')

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

Program Output (with input shown in bold)

```
Enter the manufacturer: Acme Electronics [Enter]
Enter the model number: M1000 [Enter]
Enter the retail price: 199.99 [Enter]
Here is the data that you entered:
Manufacturer: Acme Electronics
Model Number: M1000
Retail Price: $199.99
```

Accessor and Mutator Methods

As mentioned earlier, it is a common practice to make all of a class's data attributes private and to provide public methods for accessing and changing those attributes. This ensures that the object owning those attributes is in control of all the changes being made to them.

A method that returns a value from a class's attribute but does not change it is known as an *accessor method*. Accessor methods provide a safe way for code outside the class to retrieve the values of attributes, without exposing the attributes in a way that they could be changed by the code outside the method. In the `CellPhone` class that you saw in Program 11-12 (in the previous *In the Spotlight* section), the `get_manufact`, `get_model`, and `get_retail_price` methods are accessor methods.

A method that stores a value in a data attribute or changes the value of a data attribute in some other way is known as a *mutator method*. Mutator methods can control the way that a class's data attributes are modified. When code outside the class needs to change the value of an object's data attribute, it typically calls a mutator and passes the new value as an argument. If necessary, the mutator can validate the value before it assigns it to the data attribute. In Program 11-12, the `set_manufact`, `set_model`, and `set_retail_price` methods are mutator methods.

NOTE: Mutator methods are sometimes called "setters" and accessor methods are sometimes called "getters."

In the Spotlight:
Storing Objects in a List

The `CellPhone` class that you created in the previous *In the Spotlight* section will be used in a variety of programs. Many of these programs will store `CellPhone` objects in lists. To test the ability to store `CellPhone` objects in a list, you write the code in Program 11-14. This program gets the data for five phones from the user, creates five `CellPhone` objects holding that data, and stores those objects in a list. It then iterates over the list displaying the attributes of each object.

Program 11-14 (cell_phone_list.py)

```
# This program creates five CellPhone objects and
# stores them in a list.

import cellphone

def main():
    # Get a list of CellPhone objects.
    phones = make_list()

    # Display the data in the list.
    print('Here is the data you entered:')
    display_list(phones)

# The make_list function gets data from the user
# for five phones. The function returns a list
# of CellPhone objects containing the data.

def make_list():
    # Create an empty list.
    phone_list = []

    # Add five CellPhone objects to the list.
    print('Enter data for five phones.')
    for count in range(1, 6):
        # Get the phone data.
        print('Phone number ' + str(count) + ':')
        man = input('Enter the manufacturer: ')
        mod = input('Enter the model number: ')
        retail = float(input('Enter the retail price: '))
        print()

        # Create a new CellPhone object in memory and
        # assign it to the phone variable.
        phone = cellPhone.CellPhone(man, mod, retail)

        # Add the object to the list.
```

```
        phone_list.append(phone)

    # Return the list.
    return phone_list

# The display_list function accepts a list containing
# CellPhone objects as an argument and displays the
# data stored in each object.

def display_list(phone_list):
    for item in phone_list:
        print(item.get_manufact())
        print(item.get_model())
        print(item.get_retail_price())
        print()

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

Program Output (with input shown in bold)

```
Enter data for five phones.

Phone number 1:
Enter the manufacturer: Acme Electronics [Enter]
Enter the model number: M1000 [Enter]
Enter the retail price: 199.99 [Enter]

Phone number 2:
Enter the manufacturer: Atlantic Communications [Enter]
Enter the model number: S2 [Enter]
Enter the retail price: 149.99 [Enter]

Phone number 3:
Enter the manufacturer: Wavelength Electronics [Enter]
Enter the model number: N477 [Enter]
Enter the retail price: 249.99 [Enter]

Phone number 4:
Enter the manufacturer: Edison Wireless [Enter]
Enter the model number: SLX88 [Enter]
Enter the retail price: 169.99 [Enter]

Phone number 5:
Enter the manufacturer: Sonic Systems [Enter]
Enter the model number: X99 [Enter]
Enter the retail price: 299.99 [Enter]

Here is the data you entered:
Acme Electronics
M1000
199.99
```

(program output continues)

Program Output *(continued)*

```
Atlantic Communications
S2
149.99

Wavelength Electronics
N477
249.99

Edison Wireless
SLX88
169.99

Sonic Systems
X99
299.99
```

The `make_list` function appears in lines 18 through 40. In line 20 an empty list named `phone_list` is created. The `for` loop, which begins in line 24, iterates five times. Each time the loop iterates, it gets the data for a cell phone from the user (lines 27 through 29), it creates an instance of the `CellPhone` class that is initialized with the data (line 34), and it appends the object to the `phone_list` list (line 37). Line 40 returns the list.

The `display_list` function in lines 46 through 51 accepts a list of `CellPhone` objects as an argument. The `for` loop that begins in line 47 iterates over the objects in the list and displays the values of each object's attributes.

Passing Objects as Arguments

When you are developing applications that work with objects, you often need to write functions and methods that accept objects as arguments. For example, the following code shows a function named `show_coin_status` that accepts a `Coin` object as an argument:

```
def show_coin_status(coin_obj):
    print('This side of the coin is up:', coin_obj.get_sideup())
```

The following code sample shows how we might create a `Coin` object and then pass it as an argument to the `show_coin_status` function:

```
my_coin = coin.Coin()
show_coin_status(my_coin)
```

When you pass a object as an argument, the thing that is passed into the parameter variable is a reference to the object. As a result, the function or method that receives the object as an argument has access to the actual object. For example, look at the following `flip` method:

```
def flip(coin_obj):
    coin_obj.toss()
```

This method accepts a `Coin` object as an argument, and it calls the object's `toss` method. Program 11-15 demonstrates the method.

Program 11-15 (coin_argument.py)

```
1 # This program passes a Coin object as
2 # an argument to a function.
3 import coin
4
5 # main function
6 def main():
7     # Create a Coin object.
8     my_coin = coin.Coin()
9
10    # This will display 'Heads'.
11    print(my_coin.get_sideup())
12
13    # Pass the object to the flip function.
14    flip(my_coin)
15
16    # This might display 'Heads', or it might
17    # display 'Tails'.
18    print(my_coin.get_sideup())
19
20 # The flip function flips a coin.
21 def flip(coin_obj):
22     coin_obj.toss()
23
24 # Call the main function.
25 main()
```

Program Output

```
Heads
Tails
```

Program Output

```
Heads
Heads
```

Program Output

```
Heads
Tails
```

The statement in line 8 creates a `Coin` object, referenced by the variable `my_coin`. Line 11 displays the value of the `my_coin` object's `__sideup` attribute. Because the object's `__init__` method set the `__sideup` attribute to `'Heads'`, we know that line 11 will display the string `'Heads'`. Line 14 calls the `flip` function, passing the `my_coin` object as an argument. Inside the `flip` function, the `my_coin` object's `toss` method is called. Then, line 18 displays the value of the `my_coin` object's `__sideup` attribute again. This time, we cannot predict whether `'Heads'` or `'Tails'` will be displayed, because the `my_coin` object's `toss` method has been called.

In the Spotlight:
Pickling Your Own Objects

Recall from Chapter 10 that the `pickle` module provides functions for serializing objects. Serializing an object means converting it to a stream of bytes that can be saved to a file for later retrieval. The `pickle` module's `dump` function serializes (pickles) an object and writes it to a file, and the load function retrieves an object from a file and deserializes (unpickles) it.

In Chapter 10 you saw examples in which dictionary objects were pickled and unpickled. You can also pickle and unpickle objects of your own classes. Program 11-16 shows an example that pickles three `CellPhone` objects and saves them to a file. Program 11-17 retrieves those objects from the file and unpickles them.

Program 11-16 (pickle_cellphone.py)

```
# This program pickles CellPhone objects.
import pickle
import cellphone

# Constant for the filename.
FILENAME = 'cellphones.dat'

def main():
    # Initialize a variable to control the loop.
    again = 'y'

    # Open a file.
    output_file = open(FILENAME, 'wb')

    # Get data from the user.
    while again.lower() == 'y':
        # Get cell phone data.
        man = input('Enter the manufacturer: ')
        mod = input('Enter the model number: ')
        retail = float(input('Enter the retail price: '))

        # Create a CellPhone object.
        phone = cellphone.CellPhone(man, mod, retail)

        # Pickle the object and write it to the file.
        pickle.dump(phone, output_file)

        # Get more cell phone data?
        again = input('Enter more phone data? (y/n): ')

    # Close the file.
    output_file.close()
    print('The data was written to', FILENAME)
```

```

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

Program Output (with input shown in bold)

Enter the manufacturer: **ACME Electronics** Enter
Enter the model number: **M1000** Enter
Enter the retail price: **199.99** Enter
Enter more phone data? (y/n): **y** Enter
Enter the manufacturer: **Sonic Systems** Enter
Enter the model number: **X99** Enter
Enter the retail price: **299.99** Enter
Enter more phone data? (y/n): **n** Enter
The data was written to cellphones.dat

Program 11-17 (unpickle_cellphone.py)

```
# This program unpickles CellPhone objects.
import pickle
import cellphone

# Constant for the filename.
FILENAME = 'cellphones.dat'

def main():
    end_of_file = False    # To indicate end of file

    # Open the file.
    input_file = open(FILENAME, 'rb')

    # Read to the end of the file.
    while not end_of_file:
        try:
            # Unpickle the next object.
            phone = pickle.load(input_file)

            # Display the cell phone data.
            display_data(phone)
        except EOFError:
            # Set the flag to indicate the end
            # of the file has been reached.
            end_of_file = True

    # Close the file.
    input_file.close()

```

(program continues)

Program 11-17 *(continued)*

```
# The display_data function displays the data
# from the CellPhone object passed as an argument.
def display_data(phone):
    print('Manufacturer:', phone.get_manufact())
    print('Model Number:', phone.get_model())
    print('Retail Price: $', \
          format(phone.get_retail_price(), ',.2f'), \
          sep='')
    print()

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

Program Output

```
Manufacturer: ACME Electronics
Model Number: M1000
Retail Price: $199.99

Manufacturer: Sonic Systems
Model Number: X99
Retail Price: $299.99
```

In the Spotlight:

Storing Objects in a Dictionary

Recall from Chapter 10 that dictionaries are objects that store elements as key-value pairs. Each element in a dictionary has a key and a value. If you want to retrieve a specific value from the dictionary, you do so by specifying its key. In Chapter 10 you saw examples that stored values such as strings, integers, floating-point numbers, lists, and tuples in dictionaries. Dictionaries are also useful for storing objects that you create from your own classes.

Let's look at an example. Suppose you want to create a program that keeps contact information, such as names, phone numbers, and email addresses. You could start by writing a class such as the `Contact` class, shown in Program 11-18. An instance of the `Contact` class keeps the following data:

- A person's name is stored in the `__name` attribute.
- A person's phone number is stored in the `__phone` attribute.
- A person's email address is stored in the `__email` attribute.

The class has the following methods:

- An `__init__` method that accepts arguments for a person's name, phone number, and email address
- A `set_name` method that sets the `__name` attribute

- A `set_phone` method that sets the `__phone` attribute
- A `set_email` method that sets the `__email` attribute
- A `get_name` method that returns the `__name` attribute
- A `get_phone` method that returns the `__phone` attribute
- A `get_email` method that returns the `__email` attribute
- A `__str__` method that returns the object's state as a string

Program 11-18 (contact.py)

```
# The Contact class holds contact information.

class Contact:
    # The __init__ method initializes the attributes.
    def __init__(self, name, phone, email):
        self.__name = name
        self.__phone = phone
        self.__email = email

    # The set_name method sets the name attribute.
    def set_name(self, name):
        self.__name = name

    # The set_phone method sets the phone attribute.
    def set_phone(self, phone):
        self.__phone = phone

    # The set_email method sets the email attribute.
    def set_email(self, email):
        self.__email = email

    # The get_name method returns the name attribute.
    def get_name(self):
        return self.__name

    # The get_phone method returns the phone attribute.
    def get_phone(self):
        return self.__phone

    # The get_email method returns the email attribute.
    def get_email(self):
        return self.__email

    # The __str__ method returns the object's state
    # as a string.
    def __str__(self):
        return "Name: " + self.__name + \
               "\nPhone: " + self.__phone + \
               "\nEmail: " + self.__email
```

Next, you could write a program that keeps `Contact` objects in a dictionary. Each time the program creates a `Contact` object holding a specific person's data, that object would be stored as a value in the dictionary, using the person's name as the key. Then, any time you need to retrieve a specific person's data, you would use that person's name as a key to retrieve the `Contact` object from the dictionary.

Program 11-19 shows an example. The program displays a menu that allows the user to perform any of the following operations:

- Look up a contact in the dictionary
- Add a new contact to the dictionary
- Change an existing contact in the dictionary
- Delete a contact from the dictionary
- Quit the program

Additionally, the program automatically pickles the dictionary and saves it to a file when the user quits the program. When the program starts, it automatically retrieves and unpickles the dictionary from the file. (Recall from Chapter 10 that pickling an object saves it to a file, and unpickling an object retrieves it from a file.) If the file does not exist, the program starts with an empty dictionary.

The program is divided into eight functions: `main`, `load_contacts`, `get_menu_choice`, `look_up`, `add`, `change`, `delete`, and `save_contacts`. Rather than presenting the entire program at once, let's first examine the beginning part, which includes the `import` statements, global constants, and the `main` function:

Program 11-19 (contact_manager.py: `main` function)

```
# This program manages contacts.
import contact
import pickle

# Global constants for menu choices
LOOK_UP = 1
ADD = 2
CHANGE = 3
DELETE = 4
QUIT = 5

# Global constant for the filename
FILENAME = 'contacts.dat'

# main function
def main():
    # Load the existing contact dictionary and
    # assign it to mycontacts.
    mycontacts = load_contacts()

    # Initialize a variable for the user's choice.
```

```
22     choice = 0
23
24     # Process menu selections until the user
25     # wants to quit the program.
26     while choice != QUIT:
27         # Get the user's menu choice.
28         choice = get_menu_choice()
29
30         # Process the choice.
31         if choice == LOOK_UP:
32             look_up(mycontacts)
33         elif choice == ADD:
34             add(mycontacts)
35         elif choice == CHANGE:
36             change(mycontacts)
37         elif choice == DELETE:
38             delete(mycontacts)
39
40     # Save the mycontacts dictionary to a file.
41     save_contacts(mycontacts)
42
```

Line 2 imports the `contact` module, which contains the `Contact` class. Line 3 imports the `pickle` module. The global constants that are initialized in lines 6 through 10 are used to test the user's menu selection. The `FILENAME` constant that is initialized in line 13 holds the name of the file that will contain the pickled copy of the dictionary, which is `contacts.dat`.

Inside the `main` function, line 19 calls the `load_contacts` function. Keep in mind that if the program has been run before and names were added to the dictionary, those names have been saved to the `contacts.dat` file. The `load_contacts` function opens the file, gets the dictionary from it, and returns a reference to the dictionary. If the program has not been run before, the `contacts.dat` file does not exist. In that case, the `load_contacts` function creates an empty dictionary and returns a reference to it. So, after the statement in line 19 executes, the `mycontacts` variable references a dictionary. If the program has been run before, `mycontacts` references a dictionary containing `Contact` objects. If this is the first time the program has run, `mycontacts` references an empty dictionary.

Line 22 initializes the `choice` variable with the value 0. This variable will hold the user's menu selection.

The `while` loop that begins in line 26 repeats until the user chooses to quit the program. Inside the loop, line 28 calls the `get_menu_choice` function. The `get_menu_choice` function displays the following menu:

1. Look up a contact
2. Add a new contact
3. Change an existing contact
4. Delete a contact
5. Quit the program

The user's selection is returned from the `get_menu_choice` function and is assigned to the `choice` variable.

The `if-elif` statement in lines 31 through 38 processes the user's menu choice. If the user selects item 1, line 32 calls the `look_up` function. If the user selects item 2, line 34 calls the `add` function. If the user selects item 3, line 36 calls the `change` function. If the user selects item 4, line 38 calls the `delete` function.

When the user selects item 5 from the menu, the `while` loop stops repeating, and the statement in line 41 executes. This statement calls the `save_contacts` function, passing `mycontacts` as an argument. The `save_contacts` function saves the `mycontacts` dictionary to the `contacts.dat` file.

The `load_contacts` function is next.

Program 11-19 (contact_manager.py: load_contacts function)

```
43 def load_contacts():
44     try:
45         # Open the contacts.dat file.
46         input_file = open(FILENAME, 'rb')
47
48         # Unpickle the dictionary.
49         contact_dct = pickle.load(input_file)
50
51         # Close the phone_inventory.dat file.
52         input_file.close()
53     except IOError:
54         # Could not open the file, so create
55         # an empty dictionary.
56         contact_dct = {}
57
58     # Return the dictionary.
59     return contact_dct
60
```

Inside the try suite, line 46 attempts to open the `contacts.dat` file. If the file is successfully opened, line 49 loads the dictionary object from it, unpickles it, and assigns it to the `contact_dct` variable. Line 52 closes the file.

If the `contacts.dat` file does not exist (this will be the case the first time the program runs), the statement in line 46 raises an `IOError` exception. That causes the program to jump to the `except` clause in line 53. Then, the statement in line 56 creates an empty dictionary and assigns it to the `contact_dct` variable.

The statement in line 59 returns the `contact_dct` variable.

The `get_menu_choice` function is next.

Program 11-19 (contact_manager.py: `get_menu_choice` function)

```
61 # The get_menu_choice function displays the menu
62 # and gets a validated choice from the user.
63 def get_menu_choice():
64     print()
65     print('Menu')
66     print('---------------------------')
67     print('1. Look up a contact')
68     print('2. Add a new contact')
69     print('3. Change an existing contact')
70     print('4. Delete a contact')
71     print('5. Quit the program')
72     print()
73 
74     # Get the user's choice.
75     choice = int(input('Enter your choice: '))
76 
77     # Validate the choice.
78     while choice < LOOK_UP or choice > QUIT:
79         choice = int(input('Enter a valid choice: '))
80 
81     # return the user's choice.
82     return choice
83 
```

The statements in lines 64 through 72 display the menu on the screen. Line 75 prompts the user to enter his or her choice. The input is converted to an `int` and assigned to the `choice` variable. The `while` loop in lines 78 through 79 validates the user's input and, if necessary, prompts the user to reenter his or her choice. Once a valid choice is entered, it is returned from the function in line 82.

The `look_up` function is next.

Program 11-19 (contact_manager.py: `look_up` function)

```
# The look_up function looks up an item in the
# specified dictionary.
def look_up(mycontacts):
    # Get a name to look up.
    name = input('Enter a name: ')

    # Look it up in the dictionary.
    print(mycontacts.get(name, 'That name is not found.'))

```

The purpose of the `look_up` function is to allow the user to look up a specified contact. It accepts the `mycontacts` dictionary as an argument. Line 88 prompts the user to enter a name, and line 91 passes that name as an argument to the dictionary's `get` function. One of the following actions will happen as a result of line 91:

- If the specified name is found as a key in the dictionary, the `get` method returns a reference to the `Contact` object that is associated with that name. The `Contact` object is then passed as an argument to the `print` function. The `print` function displays the string that is returned from the `Contact` object's `__str__` method.
- If the specified name is not found as a key in the dictionary, the `get` method returns the string `'That name is not found.'`, which is displayed by the `print` function.

The `add` function is next.

Program 11-19 (contact_manager.py: `add` function)

```
 93 # The add function adds a new entry into the
 94 # specified dictionary.
 95 def add(mycontacts):
 96     # Get the contact info.
 97     name = input('Name: ')
 98     phone = input('Phone: ')
 99     email = input('Email: ')
100
101     # Create a Contact object named entry.
102     entry = contact.Contact(name, phone, email)
103
104     # If the name does not exist in the dictionary,
105     # add it as a key with the entry object as the
106     # associated value.
107     if name not in mycontacts:
108         mycontacts[name] = entry
109         print('The entry has been added.')
110     else:
111         print('That name already exists.')
112
```

The purpose of the `add` function is to allow the user to add a new contact to the dictionary. It accepts the `mycontacts` dictionary as an argument. Lines 97 through 99 prompt the user to enter a name, a phone number, and an email address. Line 102 creates a new `Contact` object, initialized with the data entered by the user.

The `if` statement in line 107 determines whether the name is already in the dictionary. If not, line 108 adds the newly created `Contact` object to the dictionary, and line 109 prints a message indicating that the new data is added. Otherwise, a message indicating that the entry already exists is printed in line 111.

The `change` function is next.

Program 11-19 (contact_manager.py: `change` function)

```
113 # The change function changes an existing
114 # entry in the specified dictionary.
115 def change(mycontacts):
116     # Get a name to look up.
117     name = input('Enter a name: ')
118 
119     if name in mycontacts:
120         # Get a new phone number.
121         phone = input('Enter the new phone number: ')
122 
123         # Get a new email address.
124         email = input('Enter the new email address: ')
125 
126         # Create a contact object named entry.
127         entry = contact.Contact(name, phone, email)
128 
129         # Update the entry.
130         mycontacts[name] = entry
131         print('Information updated.')
132     else:
133         print('That name is not found.')
134 
```

The purpose of the `change` function is to allow the user to change an existing contact in the dictionary. It accepts the `mycontacts` dictionary as an argument. Line 117 gets a name from the user. The `if` statement in line 119 determines whether the name is in the dictionary. If so, line 121 gets the new phone number, and line 124 gets the new email address. Line 127 creates a new `Contact` object initialized with the existing name and the new phone number and email address. Line 130 stores the new `Contact` object in the dictionary, using the existing name as the key.

If the specified name is not in the dictionary, line 133 prints a message indicating so.

The `delete` function is next.

Program 11-19 (contact_manager.py: `delete` function)

```
# The delete function deletes an entry from the
# specified dictionary.
def delete(mycontacts):
    # Get a name to look up.
    name = input('Enter a name: ')

    # If the name is found, delete the entry.
    if name in mycontacts:
```

(program continues)

Program 11-19 *(continued)*

```
143         del mycontacts[name]
144         print('Entry deleted.')
145     else:
146         print('That name is not found.')
147
```

The purpose of the `delete` function is to allow the user to delete an existing contact from the dictionary. It accepts the `mycontacts` dictionary as an argument. Line 139 gets a name from the user. The `if` statement in line 142 determines whether the name is in the dictionary. If so, line 143 deletes it, and line 144 prints a message indicating that the entry was deleted. If the name is not in the dictionary, line 146 prints a message indicating so.

The `save_contacts` function is next.

Program 11-19 (contact_manager.py: `save_contacts` function)

```
148 # The save_contacts funtion pickles the specified
149 # object and saves it to the contacts file.
150 def save_contacts(mycontacts):
151     # Open the file for writing.
152     output_file = open(FILENAME, 'wb')
153
154     # Pickle the dictionary and save it.
155     pickle.dump(mycontacts, output_file)
156
157     # Close the file.
158     output_file.close()
159
160 # Call the main function.
161 main()
```

The `save_contacts` function is called just before the program stops running. It accepts the `mycontacts` dictionary as an argument. Line 152 opens the `contacts.dat` file for writing. Line 155 pickles the `mycontacts` dictionary and saves it to the file. Line 158 closes the file.

The following program output shows two sessions with the program. The sample output does not demonstrate everything the program can do, but it does demonstrate how contacts are saved when the program ends and then loaded when the program runs again.

Program Output (with input shown in bold)

```
Menu
---------------------------
1. Look up a contact
2. Add a new contact
```

```
3. Change an existing contact
4. Delete a contact
5. Quit the program

Enter your choice: 2 Enter
Name: Matt Goldstein Enter
Phone: 617-555-1234 Enter
Email: matt@fakecompany.com Enter
The entry has been added.

Menu
---------------------------
1. Look up a contact
2. Add a new contact
3. Change an existing contact
4. Delete a contact
5. Quit the program

Enter your choice: 2 Enter
Name: Jorge Ruiz Enter
Phone: 919-555-1212 Enter
Email: jorge@myschool.edu Enter
The entry has been added.

Menu
---------------------------
1. Look up a contact
2. Add a new contact
3. Change an existing contact
4. Delete a contact
5. Quit the program

Enter your choice: 5 Enter
```

Program Output (with input shown in bold)

```
Menu
---------------------------
1. Look up a contact
2. Add a new contact
3. Change an existing contact
4. Delete a contact
5. Quit the program

Enter your choice: 1 Enter
Enter a name: Matt Goldstein Enter
Name: Matt Goldstein
Phone: 617-555-1234
Email: matt@fakecompany.com
```

(program output continues)

Program Output *(continued)*

```
Menu
---------------------------
1. Look up a contact
2. Add a new contact
3. Change an existing contact
4. Delete a contact
5. Quit the program

Enter your choice: 1 Enter
Enter a name: Jorge Ruiz Enter
Name: Jorge Ruiz
Phone: 919-555-1212
Email: jorge@myschool.edu

Menu
---------------------------
1. Look up a contact
2. Add a new contact
3. Change an existing contact
4. Delete a contact
5. Quit the program

Enter your choice: 5 Enter
```

Checkpoint

11.12 What is an instance attribute?

11.13 A program creates 10 instances of the `Coin` class. How many `__sideup` attributes exist in memory?

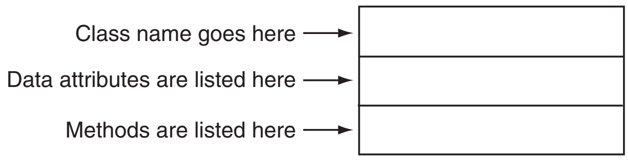
11.14 What is an accessor method? What is a mutator method?

11.4 Techniques for Designing Classes

The Unified Modeling Language

When designing a class, it is often helpful to draw a UML diagram. UML stands for Unified Modeling Language. It provides a set of standard diagrams for graphically depicting object-oriented systems. Figure 11-10 shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's data attributes. The bottom section holds a list of the class's methods.

Figure 11-10 General layout of a UML diagram for a class



Following this layout, Figure 11-11 and 11-12 show UML diagrams for the `Coin` class and the `CellPhone` class that you saw previously in this chapter. Notice that we did not show the `self` parameter in any of the methods, since it is understood that the `self` parameter is required.

Figure 11-11 UML diagram for the `Coin` class

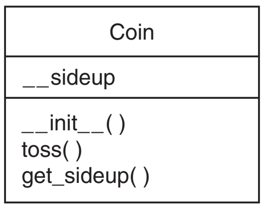
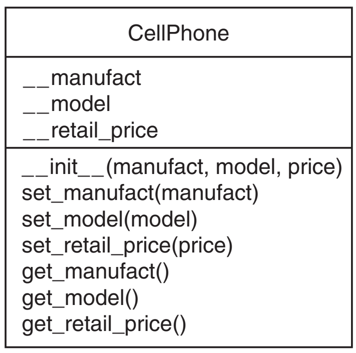


Figure 11-12 UML diagram for the `CellPhone` class



Finding the Classes in a Problem

When developing an object-oriented program, one of your first tasks is to identify the classes that you will need to create. Typically, your goal is to identify the different types of real-world objects that are present in the problem, and then create classes for those types of objects within your application.

Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps.

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

Writing a Description of the Problem Domain

The *problem domain* is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are writing a program that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

> Joe's Automotive Shop services foreign cars and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects such as vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer, teacher, student, etc.
- The results of a business event, such as a customer order, or in this case a service quote
- Recordkeeping items, such as customer histories and payroll records

Identify All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.) Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.

> **Joe's Automotive Shop** services **foreign cars**, and specializes in servicing **cars** made by **Mercedes**, **Porsche**, and **BMW**. When a **customer** brings a **car** to the **shop**, the **manager** gets the **customer**'s **name**, **address**, and **telephone number**. The **manager** then determines the **make**, **model**, and **year** of the **car**, and gives the **customer** a **service quote**. The **service quote** shows the **estimated parts charges**, **estimated labor charges**, **sales tax**, and **total estimated charges**.

Notice that some of the nouns are repeated. The following list shows all of the nouns without duplicating any of them.

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
foreign cars
Joe's Automotive Shop
make
manager
Mercedes
model
name

Porsche
sales tax,
service quote
shop
telephone number
total estimated charges
year

Refining the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

1. **Some of the nouns really mean the same thing.**

In this example, the following sets of nouns refer to the same thing:

- **car**, **cars,** and **foreign cars**
  These all refer to the general concept of a car.
- **Joe's Automotive Shop** and **shop**
  Both of these refer to the company "Joe's Automotive Shop."

We can settle on a single class for each of these. In this example we will arbitrarily eliminate **foreign cars** from the list, and use the word **cars**. Likewise we will eliminate **Joe's Automotive Shop** from the list and use the word **shop**. The updated list of potential classes is:

---

address
BMW
car
~~cars~~
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
manager
Mercedes
model
name
Porsche
sales tax
service quote

Because **car, cars,** and **foreign cars** mean the same thing in this problem, we have eliminated **cars** and **foreign cars**. Also, because **Joe's Automotive Shop** and **shop** mean the same thing, we have eliminated **Joe's Automotive Shop**.

(continued)

shop
telephone number
total estimated charges
year

2. **Some nouns might represent items that we do not need to be concerned with in order to solve the problem.**

A quick review of the problem description reminds us of what our application should do: print a service quote. In this example we can eliminate two unnecessary classes from the list:

- We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the **manager** because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

The updated list of potential classes at this point is:

address
BMW
car
~~cars~~
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
~~manager~~
Mercedes
model
name
Porsche
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

Our problem description does not direct us to process any information about the **shop**, or any information about the **manager**, so we have eliminated those from the list.

3. **Some of the nouns might represent objects, not classes.**

We can eliminate **Mercedes**, **Porsche**, and **BMW** as classes because, in this example, they all represent specific cars, and can be considered instances of a **car** class. At this point the updated list of potential classes is:

address
~~BMW~~
car
~~cars~~
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
~~manager~~
make
~~Mercedes~~
model
name
~~Porsche~~
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

We have eliminated **Mercedes, Porsche,** and **BMW** because they are all instances of a **cars** class. That means that these nouns identify objects, not classes.

NOTE: Some object-oriented designers take note of whether a noun is plural or singular. Sometimes a plural noun will indicate a class and a singular noun will indicate an object.

4. **Some of the nouns might represent simple values that can be assigned to a variable and do not require a class.**

Remember, a class contains data attributes and methods. Data attributes are related items that are stored in an object of the class, and define the object's state. Methods are actions or behaviors that can be performed by an object of the class. If a noun represents a type of item that would not have any identifiable data attributes or methods, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have data attributes and methods, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a simple variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: **address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges,** and **year**. These are all simple string or numeric values that can be stored in variables. Here is the updated list of potential classes:

~~Address~~
~~BMW~~
car
~~cars~~
customer
~~estimated labor charges~~
~~estimated parts charges~~
~~foreign cars~~
~~Joe's Automotive Shop~~
~~make~~
~~manager~~
~~Mercedes~~
~~model~~
~~name~~
~~Porsche~~
~~sales tax~~
service quote
~~shop~~
~~telephone number~~
~~total estimated charges~~
~~year~~

We have eliminated **address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges,** and **year** as classes because they represent simple values that can be stored in variables.

As you can see from the list, we have eliminated everything except **car, customer,** and **service quote**. This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a `Car` class, a `Customer` class, and a `ServiceQuote` class.

Identifying a Class's Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's *responsibilities* are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

When you have identified the things that a class is responsible for knowing, then you have identified the class's data attributes. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its methods.

It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things that a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so further consideration is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

The `Customer` Class

In the context of our problem domain, what must the `Customer` class know? The description directly mentions the following items, which are all data attributes of a customer:

- the customer's name
- the customer's address
- the customer's telephone number

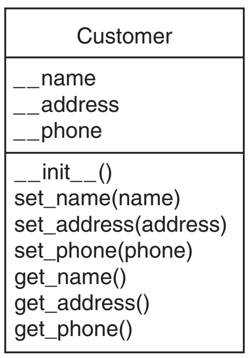
These are all values that can be represented as strings and stored as data attributes. The `Customer` class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a `Customer` class might know the customer's email address. This particular problem domain does not mention that the customer's email address is used for any purpose, so we should not include it as a responsibility.

Now let's identify the class's methods. In the context of our problem domain, what must the `Customer` class do? The only obvious actions are:

- initialize an object of the `Customer` class
- set and return the customer's name
- set and return the customer's address
- set and return the customer's telephone number

From this list we can see that the `Customer` class will have an `__init__` method, as well as accessors and mutators for the data attributes. Figure 11-13 shows a UML diagram for the `Customer` class. The Python code for the class is shown in Program 11-20.

Figure 11-13 UML diagram for the `Customer` class



Program 11-20 (customer.py)

```
# Customer class
class Customer:
    def __init__(self, name, address, phone):
        self.__name = name
        self.__address = address
        self.__phone = phone

    def set_name(self, name):
        self.__name = name

    def set_address(self, address):
        self.__address = address

    def set_phone(self, phone):
        self.__phone = phone

    def get_name(self):
        return self.__name

    def get_address(self):
        return self.__address

    def get_phone(self):
        return self.__phone
```

The `Car` Class

In the context of our problem domain, what must an object of the `Car` class know? The following items are all data attributes of a car, and are mentioned in the problem domain:

- the car's make
- the car's model
- the car's year

Now let's identify the class's methods. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of methods that we will find in most classes (an `__init__` method, accessors, and mutators). Specifically, the actions are:

- initialize an object of the `Car` class
- set and get the car's make
- set and get the car's model
- set and get the car's year

Figure 11-14 shows a UML diagram for the `Car` class at this point. The Python code for the class is shown in Program 11-21.

Figure 11-14 UML diagram for the `Car` class

Car
__make __model __year
__init__( ) set_make(make) set_model(make) set_year(y) get_make( ) get_model( ) get_year( )

Program 11-21 (car.py)

```
# Car class
class Car:
    def __init__(self, make, model, year):
        self.__make = make
        self.__model = model
        self.__year = year

    def set_make(self, make):
        self.__make = make

    def set_model(self, model):
        self.__model = model

    def set_year(self, year):
        self.__year = year

    def get_make(self):
        return self.__make

    def get_model(self):
        return self.__model

    def get_year(self):
        return self.__year
```

The `ServiceQuote` Class

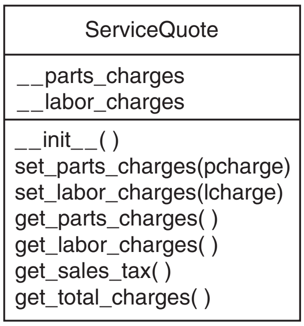
In the context of our problem domain, what must an object of the `ServiceQuote` class know? The problem domain mentions the following items:

- the estimated parts charges
- the estimated labor charges

- the sales tax
- the total estimated charges

The methods that we will need for this class are an `__init__` method and the accessors and mutators for the estimated parts charges and estimated labor charges attributes. In addition, the class will need methods that calculate and return the sales tax and the total estimated charges. Figure 11-15 shows a UML diagram for the `ServiceQuote` class. Program 11-22 shows an example of the class in Python code.

Figure 11-15 UML diagram for the ServiceQuote class



Program 11-22 (servicequote.py)

```
# Constant for the sales tax rate
TAX_RATE = 0.05

# ServiceQuote class
class ServiceQuote:
    def __init__(self, pcharge, lcharge):
        self.__parts_charges = pcharge
        self.__labor_charges = lcharge

    def set_parts_charges(self, pcharge):
        self.__parts_charges = pcharge

    def set_labor_charges(self, lcharge):
        self.__labor_charges = lcharge

    def get_parts_charges(self):
        return self.__parts_charges

    def get_labor_charges(self):
        return self.__labor_charges

    def get_sales_tax(self):
        return __parts_charges * TAX_RATE

```

```
    def get_total_charges(self):
        return __parts_charges + __labor_charges + \
               (__parts_charges * TAX_RATE)
```

This Is only the Beginning

You should look at the process that we have discussed in this section merely as a starting point. It's important to realize that designing an object-oriented application is an iterative process. It may take you several attempts to identify all of the classes that you will need and determine all of their responsibilities. As the design process unfolds, you will gain a deeper understanding of the problem, and consequently you will see ways to improve the design.

Checkpoint

11.15 The typical UML diagram for a class has three sections. What appears in these three sections?

11.16 What is a problem domain?

11.17 When designing an object-oriented application, who should write a description of the problem domain?

11.18 How do you identify the potential classes in a problem domain description?

11.19 What are a class's responsibilities?

11.20 What two questions should you ask to determine a class's responsibilities?

11.21 Will all of a classes actions always be directly mentioned in the problem domain description?

Review Questions

Multiple Choice

1. The ______________ programming practice is centered on creating functions that are separate from the data that they work on.
   a. modular
   b. procedural
   c. functional
   d. object-oriented
2. The ______________ programming practice is centered on creating objects.
   a. object-centric
   b. objective
   c. procedural
   d. object-oriented

3. A(n) ______________ is a component of a class that references data.
   a. method
   b. instance
   c. data attribute
   d. module

4. An object is a(n) ______________.
   a. blueprint
   b. cookie cutter
   c. variable
   d. instance

5. By doing this you can hide a class's attribute from code outside the class.
   a. avoid using the `self` parameter to create the attribute
   b. begin the attribute's name with two underscores
   c. begin the name of the attribute with `private__`
   d. begin the name of the attribute with the `@` symbol

6. A(n) ______________ method gets the value of a data attribute but does not change it.
   a. retriever
   b. constructor
   c. mutator
   d. accessor

7. A(n) ______________ method stores a value in a data attribute or changes its value in some other way.
   a. modifier
   b. constructor
   c. mutator
   d. accessor

8. The ______________ method is automatically called when an object is created.
   a. `__init__`
   b. `init`
   c. `__str__`
   d. `__object__`

9. If a class has a method named `__str__`, which of these is a way to call the method?
   a. you call it like any other method: *`object`*`.__str__()`
   b. by passing an instance of the class to the built in `str` function
   c. the method is automatically called when the object is created
   d. by passing an instance of the class to the built-in `state` function

10. A set of standard diagrams for graphically depicting object-oriented systems is provided by ______________.
    a. the Unified Modeling Language
    b. flowcharts

c. pseudocode
d. the Object Hierarchy System

11. In one approach to identifying the classes in a problem, the programmer identifies the _____________ in a description of the problem domain.
    a. verbs
    b. adjectives
    c. adverbs
    d. nouns

12. In one approach to identifying a class's data attributes and methods, the programmer identifies the class's _____________.
    a. responsibilities
    b. name
    c. synonyms
    d. nouns

True or False

1. The practice of procedural programming is centered on the creation of objects.
2. Object reusability has been a factor in the increased use of object-oriented programming.
3. It is a common practice in object-oriented programming to make all of a class's data attributes accessible to statements outside the class.
4. A class method does not have to have a `self` parameter.
5. Starting an attribute name with two underscores will hide the attribute from code outside the class.
6. You cannot directly call the `__str__` method.
7. One way to find the classes needed for an object-oriented program is to identify all of the verbs in a description of the problem domain.

Short Answer

1. What is encapsulation?
2. Why should an object's data attributes be hidden from code outside the class?
3. What is the difference between a class and an instance of a class?
4. The following statement calls an object's method. What is the name of the method? What is the name of the variable that references the object?

   ```
   wallet.get_dollar()
   ```
5. When the `__init__` method executes, what does the `self` parameter reference?
6. In a Python class, how do you hide an attribute from code outside the class?
7. How do you call the `__str__` method?

Algorithm Workbench

1. Suppose `my_car` is the name of a variable that references an object, and `go` is the name of a method. Write a statement that uses the `my_car` variable to call the `go` method. (You do not have to pass any arguments to the `go` method.)

2. Write a class definition named `Book`. The `Book` class should have data attributes for a book's title, the author's name, and the publisher's name. The class should also have the following:
   a. An `__init__` method for the class. The method should accept an argument for each of the data attributes.
   b. Accessor and mutator methods for each data attribute.
   c. An `__str__` method that returns a string indicating the state of the object.
3. Look at the following description of a problem domain:

   > The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on the account. Each account has an interest rate.

   Assume that you are writing a program that will calculate the amount of interest earned for a bank account.

   a. Identify the potential classes in this problem domain.
   b. Refine the list to include only the necessary class or classes for this problem.
   c. Identify the responsibilities of the class or classes.

Programming Exercises

1. `Pet` Class

VideoNote
The `Pet` class

Write a class named `Pet`, which should have the following data attributes:

- `__name` (for the name of a pet)
- `__animal_type` (for the type of animal that a pet is. Example values are 'Dog', 'Cat', and 'Bird')
- `__age` (for the pet's age)

The `Pet` class should have an `__init__` method that creates these attributes. It should also have the following methods:

- `set_name`
  This method assigns a value to the `__name` field.
- `set_animal_type`
  This method assigns a value to the `__animal_type` field.
- `set_age`
  This method assigns a value to the `__age` field.
- `get_name`
  This method returns the value of the `name` field.
- `get_type`
  This method returns the value of the `type` field.
- `get_age`
  This method returns the value of the `age` field.

Once you have written the class, write a program that creates an object of the class and prompts the user to enter the name, type, and age of his or her pet. This data should be stored as the object's attributes. Use the object's accessor methods to retrieve the pet's name, type, and age and display this data on the screen.

2. `Car` Class

Write a class named `Car` that has the following data attributes:

- `__year_model` (for the car's year model)
- `__make` (for the make of the car)
- `__speed` (for the car's current speed)

The `Car` class should have an `__init__` method that accept the car's year model and make as arguments. These values should be assigned to the object's `__year_model` and `__make` data attributes. It should also assign 0 to the `__speed` data attribute.

The class should also have the following methods:

- `accelerate`
  The `accelerate` method should add 5 to the speed data attribute each time it is called.
- `brake`
  The `brake` method should subtract 5 from the speed data attribute each time it is called.
- `get_speed`
  The `get_speed` method should return the current speed.

Next, design a program that creates a `Car` object, and then calls the `accelerate` method five times. After each call to the `accelerate` method, get the current speed of the car and display it. Then call the `brake` method five times. After each call to the `brake` method, get the current speed of the car and display it.

3. Personal `Information` Class

Design a class that holds the following personal data: name, address, age, and phone number. Write appropriate accessor and mutator methods. Also, write a program that creates three instances of the class. One instance should hold your information, and the other two should hold your friends' or family members' information.

4. `Employee` Class

Write a class named `Employee` that holds the following data about an employee in attributes: name, ID number, department, and job title.

Once you have written the class, write a program that creates three `Employee` objects to hold the following data:

Name	ID Number	Department	Job Title
Susan Meyers	47899	Accounting	Vice President
Mark Jones	39119	IT	Programmer
Joy Rogers	81774	Manufacturing	Engineer

The program should store this data in the three objects and then display the data for each employee on the screen.

5. `RetailItem` Class

Write a class named `RetailItem` that holds data about an item in a retail store. The class should store the following data in attributes: item description, units in inventory, and price.

Once you have written the class, write a program that creates three `RetailItem` objects and stores the following data in them:

	Description	Units in Inventory	Price
Item #1	Jacket	12	59.95
Item #2	Designer Jeans	40	34.95
Item #3	Shirt	20	24.95

6. Employee Management System

This exercise assumes that you have created the `Employee` class for Programming Exercise 4. Create a program that stores `Employee` objects in a dictionary. Use the employee ID number as the key. The program should present a menu that lets the user perform the following actions:

- Look up an employee in the dictionary
- Add a new employee to the dictionary
- Change an existing employee's name, department, and job title in the dictionary
- Delete an employee from the dictionary
- Quit the program

When the program ends, it should pickle the dictionary and save it to a file. Each time the program starts, it should try to load the pickled dictionary from the file. If the file does not exist, the program should start with an empty dictionary.

7. Cash Register

This exercise assumes that you have created the `RetailItem` class for Programming Exercise 5. Create a `CashRegister` class that can be used with the `RetailItem` class. The `CashRegister` class should be able to internally keep a list of `RetailItem` objects. The class should have the following methods:

- A method named `purchase_item` that accepts a `RetailItem` object as an argument. Each time the `purchase_item` method is called, the `RetailItem` object that is passed as an argument should be added to the list.
- A method named `get_total` that returns the total price of all the `RetailItem` objects stored in the `CashRegister` object's internal list.
- A method named `show_items` that displays data about the `RetailItem` objects stored in the `CashRegister` object's internal list.
- A method named clear that should clear the `CashRegister` object's internal list.

Demonstrate the `CashRegister` class in a program that allows the user to select several items for purchase. When the user is ready to check out, the program should display a list of all the items he or she has selected for purchase, as well as the total price.

8. Trivia Game

In this programming exercise you will create a simple trivia game for two players. The program will work like this:

- Starting with player 1, each player gets a turn at answering 5 trivia questions. (There should be a total of 10 questions.) When a question is displayed, 4 possible answers are also displayed. Only one of the answers is correct, and if the player selects the correct answer, he or she earns a point.
- After answers have been selected for all the questions, the program displays the number of points earned by each player and declares the player with the highest number of points the winner.

To create this program, write a `Question` class to hold the data for a trivia question. The `Question` class should have attributes for the following data:

- A trivia question
- Possible answer 1
- Possible answer 2
- Possible answer 3
- Possible answer 4
- The number of the correct answer (1, 2, 3, or 4)

The `Question` class also should have an appropriate `__init__` method, accessors, and mutators.

The program should have a list or a dictionary containing 10 `Question` objects, one for each trivia question. Make up your own trivia questions on the subject or subjects of your choice for the objects.

CHAPTER

12 Inheritance

TOPICS

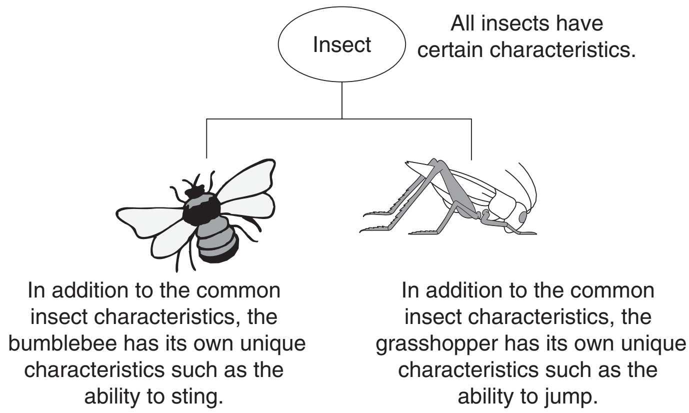
12.1 Introduction to Inheritance

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



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)

```
 73 # The Truck class represents a pickup truck. It is a
 74 # subclass of the Automobile class.
 75
 76 class Truck(Automobile):
 77     # The __init__ method accepts arguments for the
 78     # Truck's make, model, mileage, price, and drive type.
 79
 80     def __init__(self, make, model, mileage, price, drive_type):
 81         # Call the superclass's __init__ method and pass
 82         # the required arguments. Note that we also have
 83         # to pass self as an argument.
 84         Automobile.__init__(self, make, model, mileage, price)
 85
 86         # Initialize the __drive_type attribute.
 87         self.__drive_type = drive_type
 88
 89     # The set_drive_type method is the mutator for the
 90     # __drive_type attribute.
 91
 92     def set_drive_type(self, drive_type):
 93         self.__drive = drive_type
 94
 95     # The get_drive_type method is the accessor for the
 96     # __drive_type attribute.
 97
 98     def get_drive_type(self):
 99         return self.__drive_type
100
```

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()
```

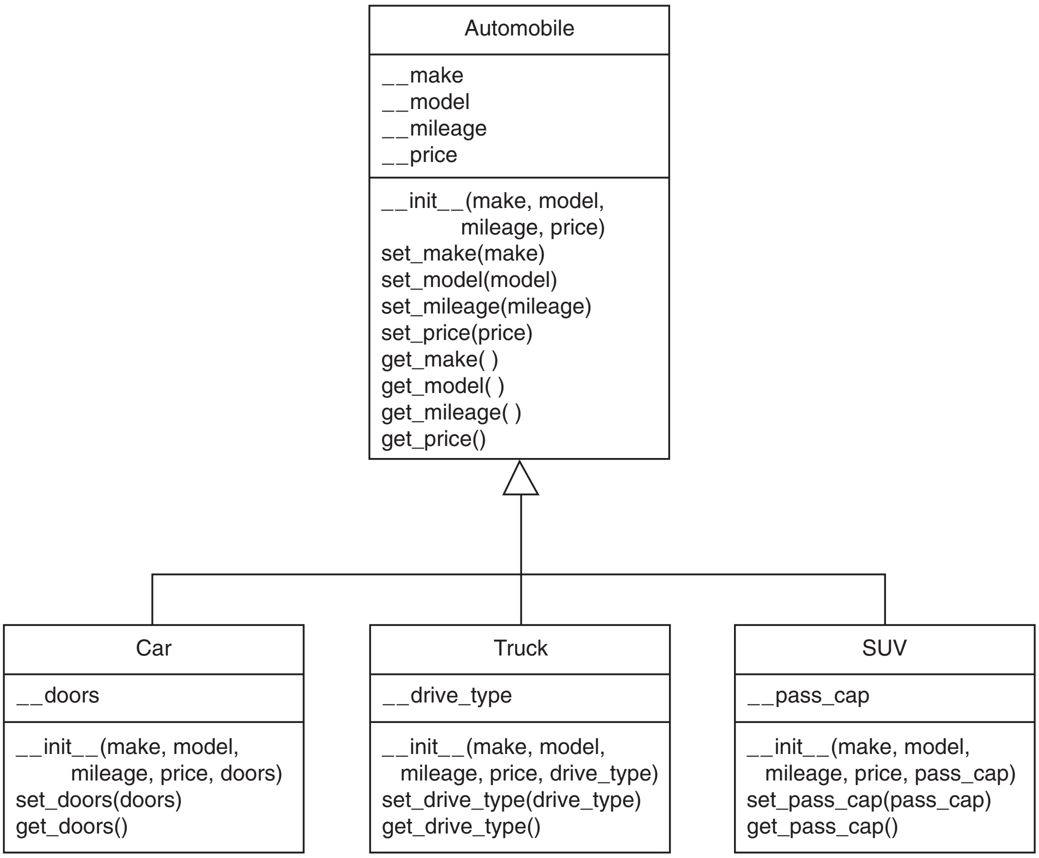
Program Output

```
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='')
```

```
    print()
    print('CD')
    print('---------------')
    print('Account number:', cd.get_account_num())
    print('Interest rate:', cd.get_interest_rate())
    print('Balance: $', \
          format(cd.get_balance(), ',.2f'), \
          sep='')
    print('Maturity date:', cd.get_maturity_date())

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

Program Output (with input shown in bold)

```
Enter the following data for a savings account.
Account number: 1234SA [Enter]
Interest rate: 3.5 [Enter]
Balance: 1000.00 [Enter]
Enter the following data for a CD.
Account number: 2345CD [Enter]
Interest rate: 5.6 [Enter]
Balance: 2500.00 [Enter]
Maturity date: 12/12/2014 [Enter]
Here is the data you entered:

Savings Account
---------------
Account number: 1234SA
Interest rate: 3.5
Balance: $1,000.00

CD
---------------
Account number: 2345CD
Interest rate: 5.6
Balance: $2,500.00
Maturity date: 12/12/2014
```

Checkpoint

12.1 In this section we discussed superclasses and subclasses. Which is the general class and which is the specialized class?

12.2 What does it mean to say there is an “is a” relationship between two objects?

12.3 What does a subclass inherit from its superclass?

12.4 Look at the following code, which is the first line of a class definition. What is the name of the superclass? What is the name of the subclass?

```
class Canary(Bird):
```

12.2 Polymorphism

CONCEPT: Polymorphism allows subclasses to have methods with the same names as methods in their superclasses. It gives the ability for a program to call the correct method depending on the type of object that is used to call it.

The term *polymorphism* refers to an object's ability to take different forms. It is a powerful feature of object-oriented programming. In this section, we will look at two essential ingredients of polymorphic behavior:

1. The ability to define a method in a superclass, and then define a method with the same name in a subclass. When a subclass method has the same name as a superclass method, it is often said that the subclass method *overrides* the superclass method.
2. The ability to call the correct version of an overridden method, depending on the type of object that is used to call it. If a subclass object is used to call an overridden method, then the subclass's version of the method is the one that will execute. If a superclass object is used to call an overridden method, then the superclass's version of the method is the one that will execute.

Actually, you've already seen method overriding at work. Each subclass that we have examined in this chapter has a method named `__init__` that overrides the superclass's `__init__` method. When an instance of the subclass is created, it is the subclass's `__init__` method that automatically gets called.

Method overriding works for other class methods too. Perhaps the best way to describe polymorphism is to demonstrate it, so let's look at a simple example. Program 12-9 shows the code for a class named `Mammal`, which is in a module named `animals`.

Program 12-9 (Lines 1 through 22 of animals.py)

```
# The Mammal class represents a generic mammal.

class Mammal:

    # The __init__ method accepts an argument for
    # the mammal's species.

    def __init__(self, species):
        self.__species = species

    # The show_species method displays a message
    # indicating the mammal's species.

    def show_species(self):
        print('I am a', self.__species)

    # The make_sound method is the mammal's
    # way of making a generic sound.
```

```

    def make_sound(self):
        print('Grrrrr')

```

The Mammal class has three methods: __init__, show_species and make_sound. Here is an example of code that creates an instance of the class and calls the uses these methods:

```
import animals
mammal = animals.Mammal('regular mammal')
mammal.show_species()
mammal.make_sound()
```

This code will display the following:

```
I am a regular mammal
Grrrrr
```

The next part of Program 12-9 shows the Dog class. The Dog class, which is also in the animals module, is a subclass of the Mammal class.

Program 12-9 (Lines 23 through 38 of animals.py)

```
# The Dog class is a subclass of the Mammal class.

class Dog(Mammal):

    # The __init__ method calls the superclass's
    # __init__ method passing 'Dog' as the species.

    def __init__(self):
        Mammal.__init__(self, 'Dog')

    # The make_sound method overrides the superclass's
    # make_sound method.

    def make_sound(self):
        print('Woof! Woof!')

```

Even though the Dog class inherits the __init__ and make_sound methods that are in the Mammal class, those methods are not adequate for the Dog class. So, the Dog class has its own __init__ and make_sound methods, which perform actions that are more appropriate for a dog. We say that the __init__ and make_sound methods in the Dog class override the __init__ and make_sound methods in the Mammal class. Here is an example of code that creates an instance of the Dog class and calls the methods:

```
import animals
dog = animals.Dog()
```

```
dog.show_species()
dog.make_sound()
```

This code will display the following:

```
I am a Dog
Woof! Woof!
```

When we use a `Dog` object to call the `show_species` and `make_sound` methods, the versions of these methods that are in the `Dog` class are the ones that execute. Next, look at Program 12-10, which shows the `Cat` class. The `Cat` class, which is also in the `animals` module, is another subclass of the `Mammal` class.

Program 12-9 (Lines 39 through 53 of animals.py)

```
# The Cat class is a subclass of the Mammal class.

class Cat(Mammal):

    # The __init__ method calls the superclass's
    # __init__ method passing 'Cat' as the species.

    def __init__(self):
        Mammal.__init__(self, 'Cat')

    # The make_sound method overrides the superclass's
    # make_sound method.

    def make_sound(self):
        print('Meow')
```

The `Cat` class also overrides the `Mammal` class's `__init__` and `make_sound` methods. Here is an example of code that creates an instance of the `Cat` class and calls these methods:

```
import animals
cat = animals.Cat()
cat.show_species()
cat.make_sound()
```

This code will display the following:

```
I am a Cat
Meow
```

When we use a `Cat` object to call the `show_species` and `make_sound` methods, the versions of these methods that are in the `Cat` class are the ones that execute.

The `isinstance` Function

Polymorphism gives us a great deal of flexibility when designing programs. For example, look at the following function:

```
def show_mammal_info(creature):
    creature.show_species()
    creature.make_sound()
```

We can pass any object as an argument to this function, and as long as it has a `show_species` method and a `make_sound` method, the function will call those methods. In essence, we can pass any object that "is a" `Mammal` (or a subclass of `Mammal`) to the function. Program 12-10 demonstrates.

Program 12-10 (polymorphism_demo.py)

```
# This program demonstrates polymorphism.

import animals

def main():
    # Create a Mammal object, a Dog object, and
    # a Cat object.
    mammal = animals.Mammal('regular animal')
    dog = animals.Dog()
    cat = animals.Cat()

    # Display information about each one.
    print('Here are some animals and')
    print('the sounds they make.')
    print('--------------------------')
    show_mammal_info(mammal)
    print()
    show_mammal_info(dog)
    print()
    show_mammal_info(cat)

# The show_mammal_info function accepts an object
# as an argument, and calls its show_species
# and make_sound methods.

def show_mammal_info(creature):
    creature.show_species()
    creature.make_sound()

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

(program output continues)

Program Output *(continued)*

```
Here are some animals and
the sounds they make.
--------------------------
I am a regular animal
Grrrrr

I am a Dog
Woof! Woof!

I am a Cat
Meow
```

But what happens if we pass an object that is not a `Mammal`, and not of a subclass of `Mammal` to the function? For example, what will happen when Program 12-11 runs?

Program 12-11 (wrong_type.py)

```
1  def main():
2      # Pass a string to show_mammal_info...
3      show_mammal_info('I am a string')
4
5  # The show_mammal_info function accepts an object
6  # as an argument, and calls its show_species
7  # and make_sound methods.
8
9  def show_mammal_info(creature):
10     creature.show_species()
11     creature.make_sound()
12
13 # Call the main function.
14 main()
```

In line 3 we call the `show_mammal_info` function passing a string as an argument. When the interpreter attempts to execute line 10, however, an `AttributeError` exception will be raised because strings do not have a method named `show_species`.

We can prevent this exception from occurring by using the built-in function `isinstance`. You can use the `isinstance` function to determine whether an object is an instance of a specific class, or a subclass of that class. Here is the general format of the function call:

```
isinstance(object, ClassName)
```

In the general format, *`object`* is a reference to an object and *`ClassName`* is the name of a class. If the object referenced by *`object`* is an instance of *`ClassName`* or is an instance of a subclass of *`ClassName`*, the function returns true. Otherwise it returns false. Program 12-12 shows how we can use it in the `show_mammal_info` function.

Program 12-12 (polymorphism_demo2.py)

```
# This program demonstrates polymorphism.

import animals

def main():
    # Create an Mammal object, a Dog object, and
    # a Cat object.
    mammal = animals.Mammal('regular animal')
    dog = animals.Dog()
    cat = animals.Cat()

    # Display information about each one.
    print('Here are some animals and')
    print('the sounds they make.')
    print('--------------------------')
    show_mammal_info(mammal)
    print()
    show_mammal_info(dog)
    print()
    show_mammal_info(cat)
    print()
    show_mammal_info('I am a string')

# The show_mammal_info function accepts an object
# as an argument, and calls its show_species
# and make_sound methods.

def show_mammal_info(creature):
    if isinstance(creature, animals.Mammal):
        creature.show_species()
        creature.make_sound()
    else:
        print('That is not a Mammal!')

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

Program Output

```
Here are some animals and
the sounds they make.
--------------------------
I am a regular animal
Grrrrr
```

(program output continues)

Program Output *(continued)*

```
I am a Dog
Woof! Woof!

I am a Cat
Meow

That is not a Mammal!
```

In lines 16, 18, and 20 we call the `show_mammal_info` function, passing references to a `Mammal` object, a `Dog` object, and a `Cat` object. In line 22, however, we call the function and pass a string as an argument. Inside the `show_mammal_info` function, the `if` statement in line 29 calls the `isinstance` function to determine whether the argument is an instance of `Mammal` (or a subclass). If it is not, an error message is displayed.

Checkpoint

12.5 Look at the following class definitions:

```
class Vegetable:
    def __init__(self, vegtype):
        self.__vegtype = vegtype

    def message(self):
        print("I'm a vegetable.")

class Potato(Vegetable):
    def __init__(self):
        Vegetable.__init__(self, 'potato')

    def message(self):
        print("I'm a potato.")
```

Given these class definitions, what will the following statements display?

```
v = Vegetable('veggie')
p = Potato()
v.message()
p.message()
```

Review Questions

Multiple Choice

1. In an inheritance relationship, the __________ is the general class.
   a. subclass
   b. superclass
   c. slave class
   d. child class

2. In an inheritance relationship, the __________ is the specialized class.
   a. superclass
   b. master class
   c. subclass
   d. parent class
3. Suppose a program uses two classes: `Airplane` and `JumboJet`. Which of these would most likely be the subclass?
   a. `Airplane`
   b. `JumboJet`
   c. Both
   d. Neither
4. This characteristic of object-oriented programming allows the correct version of an overridden method to be called when an instance of a subclass is used to call it.
   a. polymorphism
   b. inheritance
   c. generalization
   d. specialization
5. You can use this to determine whether an object is an instance of a class.
   a. The `in` operator
   b. The `is_object_of` function
   c. The `isinstance` function
   d. The error messages that are displayed when a program crashes

True or False

1. Polymorphism allows you to write methods in a subclass that have the same name as methods in the superclass.
2. It is not possible to call a superclass's `__init__` method from a subclass's `__init__` method.
3. A subclass can have a method with the same name as a method in the superclass.
4. Only the `__init__` method can be overridden.
5. You cannot use the `isinstance` function to determine whether an object is an instance of a subclass of a class.

Short Answer

1. What does a subclass inherit from its superclass?
2. Look at the following class definition. What is the name of the superclass? What is the name of the subclass?

   ```
   class Tiger(Felis):
   ```
3. What is an overridden method?

Algorithm Workbench

1. Write the first line of the definition for a `Poodle` class. The class should extend the `Dog` class.

2. Look at the following class definitions:

```
class Plant:
    def __init__(self, plant_type):
        self.__plant_type = plant_type

    def message(self):
        print("I'm a plant.")

class Tree(Plant):
    def __init__(self):
        Plant.__init__(self, 'tree')

    def message(self):
        print("I'm a tree.")
```

Given these class definitions, what will the following statements display?

```
p = Plant('sapling')
t = Tree()
p.message()
t.message()
```

3. Look at the following class definition:

```
class Beverage:
    def __init__(self, bev_name):
        self.__bev_name = bev_name
```

Write the code for a class named `Cola` that is a subclass of the `Beverage` class. The `Cola` class's `__init__` method should call the `Beverage` class's `__init__` method, passing 'cola' as an argument.

Programming Exercises

1. `Employee` and `ProductionWorker` Classes

Write an `Employee` class that keeps data attributes for the following pieces of information:

- Employee name
- Employee number

Next, write a class named `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data attributes for the following information:

- Shift number (an integer, such as 1, 2, or 3)
- Hourly pay rate

The workday is divided into two shifts: day and night. The shift attribute will hold an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write the appropriate accessor and mutator methods for each class.

Once you have written the classes, write a program that creates an object of the `ProductionWorker` class and prompts the user to enter data for each of the object's data attributes. Store the data in the object and then use the object's accessor methods to retrieve it and display it on the screen.

2. `ShiftSupervisor` Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Write a `ShiftSupervisor` class that is a subclass of the `Employee` class you created in Programming Exercise 1. The `ShiftSupervisor` class should keep a data attribute for the annual salary and a data attribute for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

VideoNote
The `Person` and `Customer` Classes

3. `Person` and `Customer` Classes

Write a class named `Person` with data attributes for a person's name, address, and telephone number. Next, write a class named `Customer` that is a subclass of the `Person` class. The `Customer` class should have a data attribute for a customer number and a Boolean data attribute indicating whether the customer wishes to be on a mailing list. Demonstrate an instance of the `Customer` class in a simple program.