

---

*In Chapter 6, we learned what objects are and how they are used. We found that all values in Python are objects. Classes in object-oriented programming define objects. Thus, a class is a “cookie cutter” for creating any number of objects of that type. As with functions, there are predefined classes in Python, as well as the ability of programmers to define their own. In this chapter, we see how to define and use classes in Python.*

---

### OBJECTIVES

After reading this chapter and completing the exercises, you will be able to

- ◆ Explain the fundamental concepts of object-oriented programming
- ◆ Explain the concept of a class
- ◆ Define encapsulation, inheritance, and polymorphism
- ◆ Explain the use of subclasses as a subtype
- ◆ Explain the purpose of UML
- ◆ Explain the relationships of a UML class diagram
- ◆ Write simple UML class diagrams
- ◆ Define and use classes in Python
- ◆ Explain and use special methods in Python
- ◆ Effectively use inheritance and polymorphism in Python

### CHAPTER CONTENTS

#### Motivation

#### Fundamental Concepts

##### **10.1** What Is Object-Oriented Programming?

##### **10.2** Encapsulation

##### **10.3** Inheritance

##### **10.4** Polymorphism

##### **10.5** Object-Oriented Design Using UML

#### Computational Problem Solving

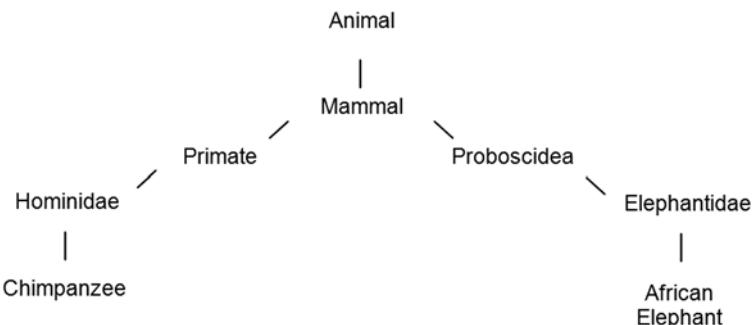
##### **10.6** Vehicle Rental Agency Program

## MOTIVATION

Classification can be described as the act of grouping entities into various categories that have something in common. In Biology, organisms are placed in a taxonomy based on their individual traits, for example. Libraries group books using various classification systems based on subject matter.

Most classification systems contain subcategories (and subcategories of subcategories, etc.), resulting in a hierarchy of types as shown in Figure 10.1. For example, chimpanzees are species in the Hominidae family, which is in the order Primate, which is in the mammal class, which is in the Animal Kingdom. The African elephant, on the other hand, is a species in the Elephantidae family, which is in the order Proboscidea, which is in the same class as chimpanzees, the Mammal class.

By this taxonomy, therefore, chimpanzees and African elephants have the same traits identified in the Mammal class, as well as traits of the Animal Kingdom. Their differences, on the other hand, are identified in the Primate and Proboscidea orders, as well as the traits within the Hominidae and Elephantidae families, and the chimpanzee and African elephant species. In this chapter, we see how the organization of class type and subtypes are utilized in object-oriented programming.



**FIGURE 10-1** Taxonomy of Chimpanzees and African Elephants

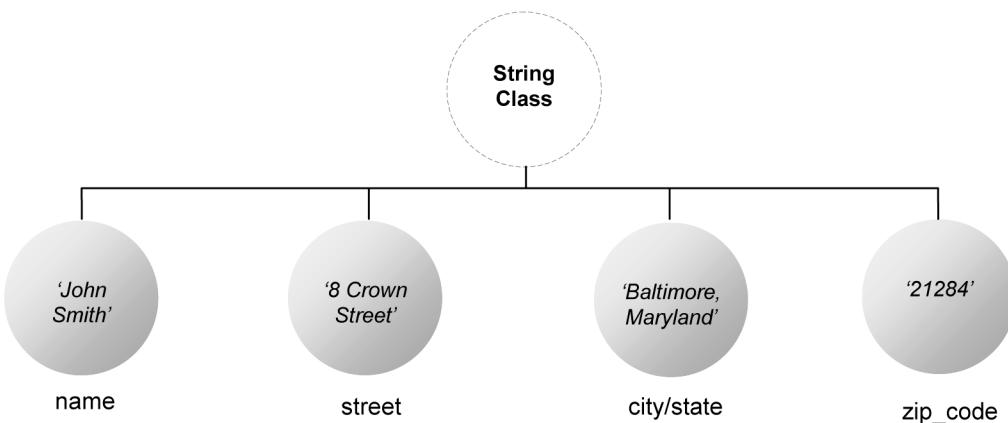
## FUNDAMENTAL CONCEPTS

### 10.1 What Is Object-Oriented Programming?

We have discussed and have been using objects in our programs. The use of objects in itself, however, does not constitute object-oriented programming. For that, we first need to introduce the concept of a class, which we discuss next.

### 10.1.1 What Is a Class?

A **class** specifies the set of instance variables and methods that are “bundled together” for defining a type of object. A class, therefore, is a “cookie cutter” that can be used to make as many object instances of that type object as needed. For example, strings in Python are object instances of the built-in `String` class, depicted in Figure 10-2.



**FIGURE 10-2** Object Instances of String Class

One method of the `String` class is `isdigit`. Thus, every string object has this method. The specific object whose `isdigit` method is called determines the specific string that it is applied to,

<code>name.isdigit() → False</code>	<code>city_state.isdigit() → False</code>
<code>address.isdigit() → False</code>	<code>zip_code.isdigit() → True</code>

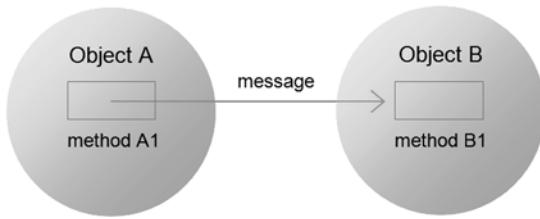
We next look at the three fundamental features of object-oriented programming.

A **class** specifies the set of instance variables and methods that are “bundled together” for defining a type of object.

### 10.1.2 Three Fundamental Features of Object-Oriented Programming

Object-oriented programming languages, such as Python, provide three fundamental features that support object-oriented programming—*encapsulation*, *inheritance*, and *polymorphism*. These support a paradigm shift in software development from the focus on variables and passing of variables to functions in procedural programming, to the focus on objects and *message passing* between them.

Message passing occurs when a method of one object calls a method of another, as depicted in Figure 10-3. For example, if Object B were a list and B1 a sorting method, then a call to B1 is a message (or request) for it to become sorted. A message to one object can result in the propagation of messages among many other objects in order to accomplish a request. In the next sections we discuss these three features of object-oriented programming, as well as introduce a means of specifying an object-oriented design using UML.



**FIGURE 10-3** Message Passing in Object-Oriented Programming

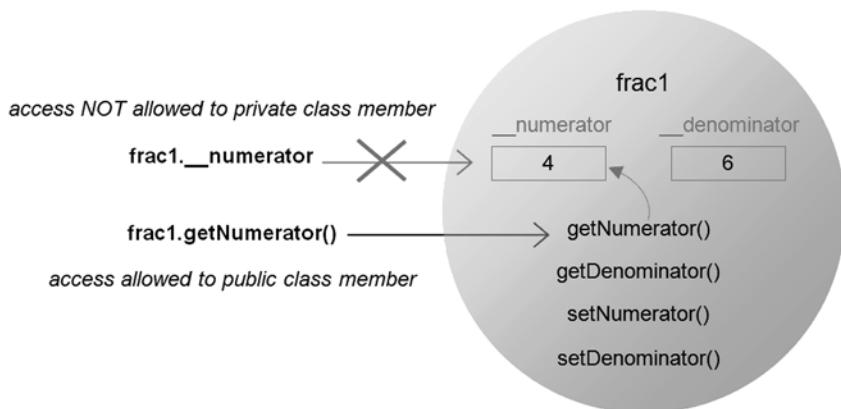
Three fundamental features supporting the design of object-oriented programs are referred to as *encapsulation*, *inheritance*, and *polymorphism*.

## 10.2 Encapsulation

In this section we develop a Fraction class for demonstrating the notion of encapsulation. (Note that the Fraction class developed here is unrelated to the Fraction class of the Python Standard Library.)

### 10.2.1 What Is Encapsulation?

**Encapsulation** is a means of bundling together instance variables and methods to form a given type (class). Selected members of a class can be made inaccessible (“hidden”) from its clients, referred to as *information hiding*. Information hiding is a form of abstraction. This is an important capability that object-oriented programming languages provide. As an example, we give a depiction of a Fraction object in Figure 10-4.



**FIGURE 10-4** Fraction Object Access

The Fraction object shows private instance variables `__numerator` and `__denominator`, and four public methods. Private members of a class begin with two underscore characters, and cannot

be directly accessed. For example, trying to access the instance variables of Fraction object `frac1` is invalid,

```
frac1.__numerator = 4      NOT ALLOWED
frac1.__denominator = 6    NOT ALLOWED
```

Public members of a class, on the other hand, are directly accessible. For example, the following are valid method calls,

<code>frac1.getNumerator()</code>	ALLOWED
<code>frac1.getDenominator()</code>	ALLOWED
<code>frac1.setNumerator(4)</code>	ALLOWED
<code>frac1.setDenominator(6)</code>	ALLOWED

These methods are referred to as **getters** and **setters** since their purpose is to get (return) and set (assign) private instance variables of a class. Restricting access to instance variables via getter and setter methods allows the methods to control what values are assigned (such as not allowing an assignment of 0 to the denominator), and how they are represented when retrieved. Thus, the instance variables of a class are generally made private, and the methods of the class generally made public.

**Encapsulation** is a means of bundling together instance variables and methods to form a given type, as well as a way of restricting access to certain class members.

## 10.2.2 Defining Classes in Python

In this section, we develop a Python Fraction class.

### Defining a Fraction Class

The first stage in the development of a Fraction class is given in Figure 10-5.

The `class` keyword is used to define classes, much as `def` is used for defining functions. All lines following the class declaration line are indented. Instance variables are initialized in the `__init__` special method. (We discuss special methods in the following.) Being private, instance variables `__numerator` and `__denominator` are not meant to be directly accessed.

```
>>> frac1.__numerator
AttributeError: 'Fraction' object has no attribute '__numerator'
```

In actuality, however, private members *are* accessible if written as follows:

```
>>> frac1._Fraction__numerator
```

```

class Fraction(object):
    """Special Methods

    def __init__(self, numerator, denominator):
        """Inits Fraction with values numerator and denominator."""

        self.__numerator = numerator
        self.__denominator = denominator
        self.reduce()

    def getNumerator(self):
        """Returns the numerator of a Fraction."""

        return self.__numerator

    def getDenominator(self):
        """Returns the denominator of a Fraction."""

        return self.__denominator

    def setNumerator(self, value):
        """Sets the numerator of a Fraction to the provided value."""

        self.__numerator = value

    def setDenominator(self, value):
        """Sets the denominator of a Fraction to the provided value.

        Raises a ValueError exception if a value of zero provided.
        """

        if value == 0:
            raise ValueError('Divide by Zero Error')

        self.__denominator = value

```

**FIGURE 10-5** Initial Fraction Class

To understand this, all private class members are automatically renamed to begin with a single underscore character followed by the class name. Such renaming of identifiers is called *name mangling*. Unless the variable or method is accessed with its complete (mangled) name, it will not be found. Name mangling prevents unintentional access of private members of a class, while still allowing access when needed.

The methods of a class, as we have seen, are essentially functions meant to operate on the instance variables of the class. In Python, functions serving as a method must have an extra first parameter, by convention named `self`. This parameter contains a reference to the object instance to which the method belongs. When a method accesses any other member of the same class, the member name must be preceded by '`self.`' (`self.__numerator`). Getter and setter methods are also defined. Note that `setDenominator` raises an exception when passed a value of 0 to ensure that `Fraction` objects cannot be set to an invalid value. We further discuss special methods in Python next.

### LET'S TRY IT

Enter and execute the following Python class. Then enter the given instructions within the Python shell and observe the results.

```
class SomeClass(object):
    def __init__(self):
        self.__n = 0
        self.n2 = 0
>>> obj = SomeClass()
>>> obj.__n
???
>>> obj._SomeClass__n
???
>>> obj.n2
???
```

The `class` keyword is used to define a class in Python. Class members beginning with two underscore characters are intended to be private members of a class. This is effectively accomplished in Python by the use of *name mangling*.

## Special Methods in Python

Method names that begin and end with two underscore characters are called *special methods* in Python. Special methods are automatically called. For example, the `__init__` method of the `Fraction` class developed is automatically called whenever a new `Fraction` object is created,

```
frac1 = Fraction(1,2)  - creates new fraction with value 1/2
frac2 = Fraction(6,8)  - creates new fraction with value 6/8
```

The values in parentheses are arguments to the `__init__` method to initialize a new `Fraction` object to a specific value. Note that although there are three parameters defined (`self`, `numerator`, `denominator`), the first is always implied. Therefore, only the remaining arguments (`numerator` and `denominator`) are explicitly provided when creating a `Fraction` object.

Two other special methods of Python are `__str__` and `__repr__`. These methods are used for representing the value of an object as a string. The `__str__` method is called when an object is displayed using `print` (and when the `str` conversion function is used.) The `__repr__` function is called when the value of an object is displayed in the Python shell (when interactively using Python). This is demonstrated below.

```
class DemoStrRepr():
    def __repr__(self):
        return '__repr__ called'
    def __str__(self):
        return '__str__ called'
>>> s = DemoStrRepr()
>>> print(s)
__str__ called
>>> s
__repr__ called
```

The difference in these special methods is that `__str__` is for producing a string representation of an object's value that is most readable (for humans), and `__repr__` is for producing a string representation that Python can evaluate. If special method `__str__` is not implemented,

then special method `__repr__` is used in its place. An implementation of `__repr__` for the `Fraction` class is given below.

```
def __repr__(self):
    return str(self.__numerator) + '/' + str(self.__denominator)
```

This, therefore, will display `Fraction` values as we normally write them:

```
>>> frac1 = Fraction(3,4)      >>> print('Value of frac1 is', frac1)
>>> frac1                      Value of frac1 is 3/4
3/4
```

We give special method `__repr__` of the `Fraction` class (to also serve as the implementation of special method `__str__`) in Figure 10-6.

```
class Fraction(object):

    def __repr__(self):
        """Returns Fraction value as x/y."""
        return str(self.__numerator) + '/' + str(self.__denominator)
```

#### Additional Special Methods

**FIGURE 10-6** Additional Special Method `repr`

We will look at more Python special methods in the next section.

#### LET'S TRY IT

Enter and save the following class definition in a Python file and execute it. Then enter the given instructions in the Python shell and observe the results.

```
class XYcoord(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def __repr__(self):
        return '(' + str(self.__x) + ', ' \
               + str(self.__y) + ')'
```

>>> coord = XYcoord(5, 2)	>>> print(coord)
???	???
>>> str(coord)	>>> coord
???	???

**Special methods** in Python have names that begin and end with two underscore characters, and are automatically called in Python.

### Adding Arithmetic Operators to the `Fraction` Class

Special methods used to provide arithmetic operators for class types are shown in Figure 10-7.

Operator	Example Use	Special Method
- (negation)	-frac1	__neg__
+ (addition)	frac1 + frac2	__add__
- (subtraction)	frac1 - frac2	__sub__
* (multiplication)	frac1 * frac2	__mul__

**FIGURE 10-7** Arithmetic Operator Special Methods

The expression `frac1 + frac2`, for example, evaluates to the returned value of the `__add__` method in the class, where the left operand (`frac1`) is the object on which the method call is made: `frac1.__add__(frac2)`. Arithmetic methods for the `Fraction` class are given in Figure 10-8.

```

Arithmetic Operator Special Methods

def __neg__(self):
    """Returns a new Fraction equal to the negation of self."""
    return Fraction(-self.__numerator, self.__denominator)

def __add__(self, rfraction):
    """Returns a new reduced Fraction equal to self + rfraction."""
    numer = self.__numerator * rfraction.getDenominator() + \
            rfraction.getNumerator() * self.__denominator
    denom = self.__denominator * rfraction.getDenominator()
    resultFrac = Fraction(numer, denom)
    return resultFrac.reduce()

def __sub__(self, rfraction):
    """Returns a new reduced Fraction equal to self - rfraction."""
    return self + (-rfraction)

def __mul__(self, rfraction):
    """Returns a new reduced Fraction equal to self * rfraction."""
    numer = self.__numerator * rfraction.getNumerator()
    denom = self.__denominator * rfraction.getDenominator()
    resultFrac = Fraction(numer, denom)
    resultFrac.reduce()
    return resultFrac

```

**FIGURE 10-8** Arithmetic Operators of the Fraction Class

Special method `__add__` is implemented to add the numerator and denominator of each fraction based on a common denominator as shown below,

$$\begin{aligned}
 2/4 + 1/5 &\rightarrow (2 * 5) / (4 * 5) + (1 * 4) / (5 * 4) \\
 &\rightarrow 10/20 + 4/20 \\
 &\rightarrow 14/20 \\
 &\rightarrow 7/10 \quad (\text{after call to method reduce})
 \end{aligned}$$

Note that a `reduce` method of the class (not given) is called on the resulting fraction to return the result in simplest form. Thus, rather than returning  $14/20$ , the value  $7/10$  is returned.

The `__sub__` special method has a very simple and elegant implementation. It simply adds the first fraction to the negation of the second fraction, relying on the implementation of the `__add__` and `__neg__` methods. Finally, the `__mul__` special method multiplies the numerators of each of the fractions, as well as the denominators of each, building a new `Fraction` object from the results, and reducing the new fraction to simplest form by call to method `reduce`. (We omit the special method for the division operator here.)

Just as with the other arithmetic operators, a new (negated) `Fraction` object is returned by the `__neg__` method, rather than the original fraction object being altered. To negate a fraction, reassignment would be used instead,

```
frac1 = -frac1
```

### LET'S TRY IT

Enter and save the following class definition in a Python file, and execute. Then enter the given instructions in the Python shell and observe the results.

```
class XYcoord(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
    def __repr__(self):
        return '(' + str(self.__x) + ', ' +
               str(self.__y) + ')'
    def __add__(self, rCoord):
        new_x = self.__x + rCoord.__x
        new_y = self.__y + rCoord.__y
        return XYCoord(new_x, new_y)
>>> coord_1 = XYcoord(4,2)
>>> coord_2 = XYCoord(6,10)
>>> coord_1 + coord_2
???
>>> coord = coord_1 + coord_2
>>> print(coord)
???
```

## Adding Relational Operators to the Fraction Class

We have yet to add the capability of applying relational operators to `Fraction` objects. The set of relational operator special methods that can be implemented is shown in Figure 10-9.

Operator	Example Use	Special Method
< (less than)	frac1 < frac2	<code>__lt__</code>
<= (less than or equal to)	frac1 <= frac2	<code>__le__</code>
== (equal to)	frac1 == frac2	<code>__eq__</code>
!= (not equal to)	frac1 != frac2	<code>__ne__</code>
> (greater than)	frac1 > frac2	<code>__gt__</code>
>= (greater than or equal to)	frac1 >= frac2	<code>__ge__</code>

FIGURE 10-9 Relational Operator Special Methods

The special methods for providing the relational operators of the `Fraction` class are given in Figure 10-10.

```
Relational Operators
```

```

def __eq__(self, rfraction):
    """Returns True if self arithmetically equal to rfraction.
    Otherwise, returns False.
    """
    temp_frac1 = self.copy()
    temp_frac2 = rfraction.copy()
    temp_frac1.reduce()
    temp_frac2.reduce()

    return temp_frac1.getNumerator() == temp_frac2.getNumerator() and \
           temp_frac1.getDenominator() == temp_frac2.getDenominator()

def __neq__(self, rfraction):
    """Returns True if Fraction not arithmetically equal to rfraction.
    Otherwise, returns False.
    """
    return not self.__eq__(rfraction)

def __lt__(self, rfraction):
    """Returns True if self less than rfraction."""

    if self.getDenominator() == rfraction.getDenominator():
        return self.getNumerator() < rfraction.getNumerator()
    else:
        temp_frac1 = self.copy()
        temp_frac2 = rfraction.copy()

        saved_denom = temp_frac1.getDenominator()
        temp_frac1._adjust(temp_frac2.getDenominator())
        temp_frac2._adjust(saved_denom)

        return temp_frac1.getNumerator() < temp_frac2.getNumerator()

def __le__(self, rfraction):
    """Returns True if self less than or equal to rfraction."""

    return not (rfraction < self)

def __gt__(self, rfraction):
    """Returns True if self greater than rfraction."""

    return not(self <= rfraction)

def __ge__(self, rfraction):
    """Returns True if self greater than or equal to rfraction."""

    return not(self < rfraction)

```

**FIGURE 10-10** Relational Operators of the Fraction Class

Each of the special methods for the relational operators is implemented. For example, `frac1 < frac2` is determined by call to method `__lt__` on the first object, `frac1`, with the second object, `frac2`, passed as an argument,

```
frac1.__lt__(frac2)
```

In order to compare two fractions, they must have common denominators. Therefore, method `__lt__` first checks if the denominators are equal. If so, then the result of `self.__getNumerator() < rfraction.__getNumerator()` is returned. Otherwise, since we do not want the two fractions to be altered as a result of the comparison, a copy of each is made, assigned to `temp_frac1` and `temp_frac2`. In order to convert them common denominators, the numerator and denominator of each is multiplied by the denominator of the other. This is accomplished by call to private method `__adjust`. Then, the Boolean result `temp_frac1.__getNumerator() < temp_frac2.__getNumerator()` is returned.

Most other relational operators are grounded in the implementation of the less than special method, `__lt__`. The implementation of special method `__le__` (less than or equal to) is based on the fact that `a <= b` is the same as not (`b < a`). Special method `__neq__` (not equal to) is simply implemented as not (`a = b`). Finally, special method `__gt__` (greater than) is implemented as not `a <= b`, and special method `__ge__` (greater than or equal to) is implemented as not (`a < b`). Finally, the implementation of methods `copy`, `reduce`, and `__adjust` are left as an exercise at the end of the chapter.

The `Fraction` type developed represents the power of abstraction, as implemented through the use of encapsulation. The `Fraction` class contains two integer instance variables and an associated set of methods. That is what it “really” is. Through the use of information hiding, however, the client is provided an abstract view, which, for all intents and purposes, is a `Fraction` type. The `Fraction` class can be defined within its own Python file, and thus serve as a module that can be easily imported and used by any program.

There exist special methods for the arithmetic and relational operators in Python that can be implemented to determine how these operators are evaluated for a given object type.

### 10.2.3 Let's Apply It—A Recipe Conversion Program

The following Python program (Figure 10-12) will convert the measured amount of ingredients of recipes, based on a provided conversion factor, to vary the number of servings. The program utilizes the following programming features:

- programmer-defined class

Example execution of the program is given in Figure 10-11.

The program makes use of the `fraction` class module developed in section 10.2.2, imported on **line 3**. Since there is only one item to be imported (the class), the choice of using `import Fraction` vs. `from Fraction import *` only affects whether `Fraction` objects are created as `frac1 = Fraction.Fraction(1, 2)` for the first form of import, or `frac1 = Fraction(1, 2)` for the second

#### Chocolate Chip Cookies Recipe

4 1/2 cups all-purpose flour
2 teaspoons baking soda
2 cups butter, softened
1 1/2 cups packed brown sugar
1/2 cup white sugar
2 packages instant vanilla pudding mix
4 eggs
2 teaspoons vanilla extract
4 cups semisweet chocolate chips
2 cups chopped walnuts

```
This program will convert a given recipe to a different
quantity based on a specified conversion factor. Enter a
factor of 1/2 to halve, 2 to double, 3 to triple, etc.

Enter file name: ChocChipCookies.txt
File Open Error

Enter file name: ChocChipCookies.txt
Enter the conversion factor: 2
Converted recipe in file: conv_ChocChipCookies.txt
```

<pre>Chocolate Chip Cookies Recipe 9 cups all-purpose flour 4 teaspoons baking soda 4 cups butter, softened 3 cups packed brown sugar 1 cup white sugar 4 packages instant vanilla pudding mix 8 eggs 4 teaspoons vanilla extract 8 cups semisweet chocolate chips 4 cups chopped walnuts</pre>
---

```
Enter file name: ChocChipCookies.txt
Enter the conversion factor: 1/2
Converted recipe in file: conv_ChocChipCookies.txt
```

<pre>Chocolate Chip Cookies Recipe 9/4 cups all-purpose flour 1 teaspoons baking soda 1 cups butter, softened 3/4 cups packed brown sugar 1/4 cup white sugar 1 packages instant vanilla pudding mix 2 eggs 1 teaspoons vanilla extract 2 cups semisweet chocolate chips 1 cups chopped walnuts</pre>
---

**FIGURE 10-11** Execution of the Recipe Conversion Program

form. We therefore choose the `from-import` form of import. Note that the methods of a class are called the same way regardless of the form of import used to import the class.

The main section of the program is in **lines 106–143**. The program welcome is provided on **lines 109–111**. The rest of the code is encompassed within a try block for catching any `IOError` exceptions. There is one instance of an `IOError` exception that is raised by the program (in addition to those raised by the Python standard functions) in function `getFile` (**lines 5–28**).

The `getFile` function prompts for a file name to open, returning both the file name and associated file object as a tuple. If, after three attempts the file fails to open successfully, an `IOError` exception is raised containing the error message 'Exceeded number of open file attempts' (**line 26**). Thus, a try block is used to catch each `IOError` exception raised by the `open` function. When such an exception is caught, variable `num_attempts` is incremented (**line 22**) in the corresponding exception handler (**lines 21–23**). When an input file is successfully opened, the loop terminates and `file_name` and `input_file` are returned as a tuple.

Back in the main module of the program, the user is prompted for the conversion factor (**line 119**). Since all calculations in the program are executed as `Fraction` types, the conversion factor is scanned (read) by call to function `scanAsFraction` (**lines 45–84**). If a single integer value (read as a string) is entered, for example '2', then `scanAsFraction` returns the `Fraction` value  $2/1$ . If a single fraction value is entered, such as '2/4', then the `Fraction` value (in reduced form)  $1/2$  is returned. If an integer and fraction are entered, such as '1 1/2', then a `Fraction` equal to the sum of both is returned,  $3/2$ .

Function `scanAsFraction` returns, as a `Fraction` value, the total value of the initial part of the parameter string passed.

4 1/2 cups all-purpose flour

```

1 # Recipe Conversion Program
2
3 from fraction import *
4
5 def getFile():
6
7     """Returns as a tuple the file name entered by the user and the
8         open file object. If the file exceeds three attempts of opening
9         successfully, an IOError exception is raised.
10    """
11
12     file_name = input('Enter file name: ')
13     input_file_opened = False
14     num_attempts = 1
15
16     while not input_file_opened and num_attempts < 3:
17         try:
18             input_file = open(file_name, 'r')
19             input_file_opened = True
20         except IOError:
21             print('File Open Error\n')
22             num_attempts = num_attempts + 1
23             file_name = input('Enter file name: ')
24
25     if num_attempts == 3:
26         raise IOError('Exceeded number of file open attempts')
27
28     return (file_name, input_file)
29
30 def removeMeasure(line):
31
32     """Returns provided line with any initial digits and fractions
33         (and any surrounding blanks) removed.
34     """
35
36     k = 0
37     blank_char = ' '
38
39     while k < len(line) and (line[k].isdigit() or \
40                             line[k] in ('/', blank_char)):
41         k = k + 1
42
43     return line[k:len(line)]
44
45 def scanAsFraction(line):
46
47     """Scans all digits, including fractions, and returns as a
48         Fraction object. For example, '1/2' would return as Fraction
49         value 1/2, '2' would return as Fraction 2/1, and '2 1/2' would
50         return as Fraction value 3/2.
51     """
52

```

**FIGURE 10-12** Recipe Conversion Program (*Continued*)

```

53     completed_scan = False
54     value_as_frac = Fraction(0,1)
55
56     while not completed_scan:
57         k = 0
58         while k < len(line) and line[k].isdigit():
59             k = k + 1
60
61         numerator = int(line[0:k])
62
63         if k < len(line) and line[k] == '/':
64             k = k + 1
65             start = k
66             while k < len(line) and line[k].isdigit():
67                 k = k + 1
68
69             denominator = int(line[start:k])
70         else:
71             denominator = 1
72
73         value_as_frac = value_as_frac + Fraction(numerator,
74                                         denominator)
75
76         if k == len(line):
77             completed_scan = True
78         else:
79             line = line[k:len(line)].strip()
80
81         if not line[0].isdigit():
82             completed_scan = True
83
84     return value_as_frac
85
86 def convertLine(line, factor):
87
88     """If line begins with a digit, returns line with the value
89     multiplied by factor. Otherwise, returns line unaltered
90     (e.g., for a factor of 2, '1/4 cup' returns as '1/2 cup'.)
91     """
92
93     if line[0].isdigit():
94         blank_char = ' '
95         frac_meas = scanAsFraction(line) * factor
96
97         if frac_meas.getDenominator() == 1:
98             frac_meas = frac_meas.getNumerator()
99
100        conv_line = str(frac_meas) + blank_char + removeMeasure(line)
101    else:
102        conv_line = line
103
104    return conv_line
105

```

**FIGURE 10-12** Recipe Conversion Program (*Continued*)

```

106 # ---- main
107
108 # display welcome
109 print('This program will convert a given recipe to a different')
110 print('quantity based on a specified conversion factor. Enter a')
111 print('factor of 1/2 to halve, 2 to double, 3 to triple, etc.\n')
112
113 try:
114
115     # get file name and open file
116     file_name, input_file = getFile()
117
118     # get conversion factor
119     conv_factor = input('Enter the conversion factor: ')
120     conv_factor = scanAsFraction(conv_factor)
121
122     # open output file named 'conv_' + file_name
123     output_file_name = 'conv_' + file_name
124     output_file = open(output_file_name, 'w')
125
126     # convert recipe
127     empty_str = ''
128     recipe_line = input_file.readline()
129
130     while recipe_line != empty_str:
131         recipe_line = convertLine(recipe_line, conv_factor)
132         output_file.write(recipe_line)
133         recipe_line = input_file.readline()
134
135     # close files
136     input_file.close()
137     output_file.close()
138
139     # display completion message to user
140     print('Converted recipe in file: ', output_file_name)
141
142 except IOError as err_mesg: # catch file open error
143     print(err_mesg)

```

**FIGURE 10-12** Recipe Conversion Program

First, variable completed\_scan is initialized to False (**line 53**) and set to True when the initial digit (and '/') characters have been scanned. Then variable value\_as\_frac (**line 54**) is initialized to the Fraction value 0/1, so that it can be used to accumulate values (**line 73**) when both an integer and a fractional value are found, as shown. Within the while loop, index variable k is initialized to 0 (**line 57**). Following that, k is incremented to scan past each character in line that is a digit character (as long as k is less than the length of the line). This places k at the index location of the first non-digit. Then, the range of characters scanned so far, line [0:k], is converted to an integer type and assigned to numerator. The next character is then scanned for the '/' character (indicating that a fraction notation exists), as long as the end of line has not been reached (that is, k < len(line)). If the slash character is found, then there is a denominator to go with the numerator value just scanned. Thus, the following characters are scanned until a non-digit is found (**lines 66–67**).

The current location of k is first stored in variable start (**line 65**) so that the beginning of the new substring of digits can be scanned. At that point, denominator is set to the integer value

of the digits from index `start` to  $k-1$  (`line[start:k]`). Then, on **line 73**, variable `value_as_frac` is set to the current value of `value_as_frac` plus the newly created `Fraction` object with the current values of `numerator` and `denominator`. If a slash character is not found, then `denominator` is set to one (**line 71**) so that the integer value scanned, for example 2, is returned as  $2/1$ .

On **line 76** a check is made to determine if the end of the line has been reached. (This would not normally be true of a recipe line, but would be True when scanning a conversion value.) If the end has been reached, then `completed_scan` is set to True, which terminates the main while loop, causing the final return statement to return the scanned value. If, however, the end of the line has not been reached, the next character is checked to see if it is a digit. If it is, then the while loop at **line 56** iterates again to scan the following value, which is expected to be a fractional value. If the next character is not a digit, then `completed_scan` is set to True (**line 82**), causing the function to return the (non-fractional) value scanned.

In **lines 123–124** of the main module, `output_file_name` is assigned to the name of the input file name and prepended with the string '`conv_`', and file object `output_file` is created (where the converted recipe is written). Following that, the process of converting each line of the recipe file begins. First, `empty_str` is initialized and the first line of the recipe file is read as `recipe_line` (**lines 127–128**). The while loop at **line 130** then converts the current recipe line by a call to `convertLine`. Function `convertLine` (**lines 86–104**) first checks that the first character of the line is a digit. If not, then the line is assigned unaltered to `conv_line` on **line 102** (since there is no initial numerical value to convert). If a digit is found, then `blank_char` is initialized (**line 94**), and `frac_meas` is set to the `Fraction` value returned by `scanAsFunction` (for the given conversion factor). If the denominator of `frac_meas` is found to be 1 (e.g.,  $2/1$ ), then `frac_meas` is set to the value of the numerator (2), otherwise it is left as is. Variable `line` is set to the remaining part of the line by call to `removeMeasure` (**line 100**), and `conv_line` is set to the string representation of `frac_meas` concatenated with a blank and the remaining part of the original line, and returned. Finally, function `removeMeasure` (**lines 30–43**) scans past the initial digit, blank, and slash characters, returning the remaining part of the line.

## Self-Test Questions

1. Encapsulation bundles together instance variables and methods of a class, and allows certain members of the class to be made inaccessible by use of keyword `private` in Python. (TRUE/FALSE)
2. Private members of a class are not meant to be directly accessed by methods of other classes. (TRUE/FALSE)
3. Methods of a class that provide access to private members of the class are called \_\_\_\_\_ and \_\_\_\_\_.
4. Which of the following are special methods in Python?
  - (a) Getter and setters
  - (b) Method names that begin and end with two underscore characters
  - (c) Methods that are part of any Python built-in type

5. Which of the following is a special method for defining an operator in a class?
- `__init__`
  - `__add__`
  - `__add__`

ANSWERS: 1. False, 2. True, 3. getters/setters, 4. (b), 5. (c)

## 10.3 Inheritance

The true capabilities of object-oriented programming present themselves when inheritance of classes is employed. In this section, we explore the use of inheritance in Python.

### 10.3.1 What Is Inheritance?

**Inheritance**, in object-oriented programming, is the ability of a class to inherit members of another class as part of its own definition. The inheriting class is called a **subclass** (also “derived class” or “child class”), and the class inherited from is called the **superclass** (also “base class” or “parent class”). Superclasses may themselves inherit from other classes, resulting in a hierarchy of classes as shown in Figure 10-13 (inherited class members are in gray).

Class A is the superclass of all the classes in the figure. Thus, subclasses B and E each inherit variable var1 and method method1 from Class A. In addition, Class B defines variable var2 and method method2, and Class E defines method5. Since Class C is a subclass of Class B, it inherits everything in Class A and Class B, adding var3 and method3 in its own definition. And since Class D is also a subclass of Class B, it inherits everything in Class A and Class B, also defining method4.

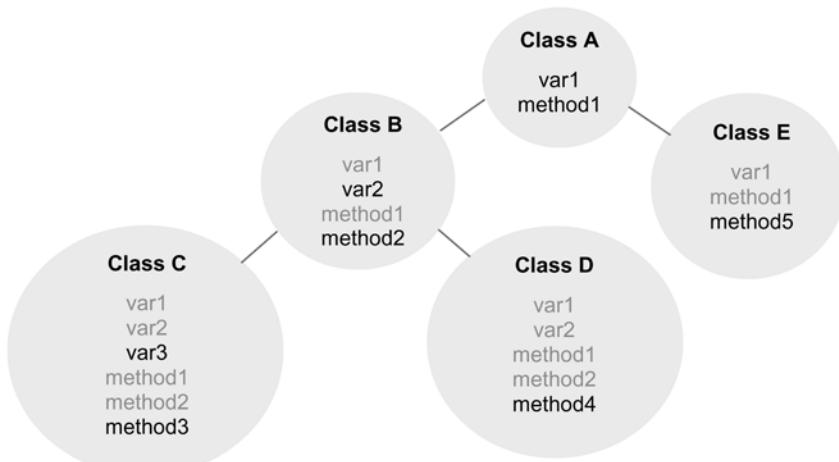
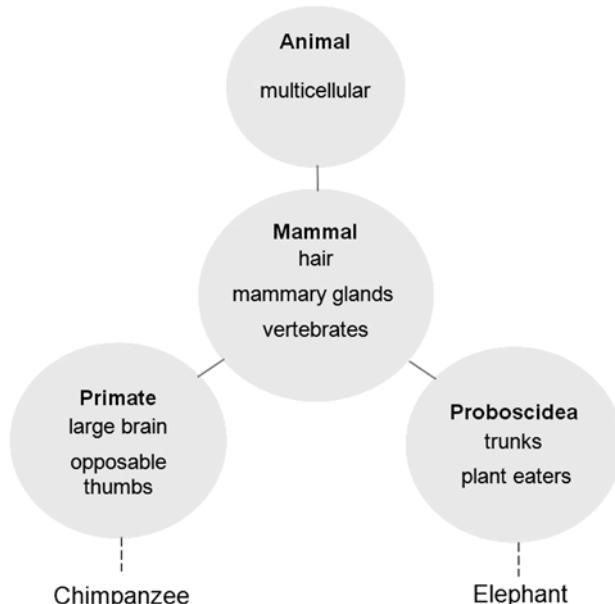


FIGURE 10-13 Class Hierarchy

**Inheritance** in object-oriented programming is the ability of a **subclass** (also “derived class” or “child class”) to inherit members of a **superclass** (also “base class” or “parent class”) as part of its own definition.

### 10.3.2 Subtypes

A **subtype** is something that can be substituted for and behave as its parent type (and its parent’s parent type, etc.). For example, consider the characteristic features within the Animal Kingdom in Figure 10-14.



**FIGURE 10-14** Hierarchy of Characteristics of Animals

Mammals are a type of animal. Therefore, they have the characteristic of being multicellular. In addition, they have the characteristics of having hair, mammary glands, and vertebrae. Primates (which include chimpanzees) are a type of mammal (and thus a type of animal). Therefore, they have the characteristics of mammals, as well as having large brains and opposable thumbs. Proboscidea (which includes elephants) are a type of mammal (and a type of animal) and thus have the characteristics of mammals, as well as having some form of trunk and being plant eaters. Thus, each classification describes a subtype of the classifications from which it derives. For example, consider the following simple story.

*Andy was very interested in animals. He had many books about them, and went to see animals whenever he had the chance.*

Because chimpanzees are a type of animal, an alternate version of the story can be generated by substituting “chimpanzee” for “animal,”

*Andy was very interested in chimpanzees. He had many books about them, and went to see chimpanzees whenever he had the chance.*

Now consider the following story,

*Andy was very interested in chimpanzees. He had many books about them, and loved to watch the chimpanzees swing from tree to tree.*

In this case, if we substitute another animal, “elephant,” for “chimpanzee,” we get the following story,

*Andy was very interested in elephants. He had many books about them, and loved to watch the elephants swing from tree to tree.*

Because an elephant is not a chimpanzee, this version of the story does not make sense. Note, however, that since an elephant is a type of animal, “elephant” can be substituted in the original story just as chimpanzee was and make sense. We next look at how to create subclasses in Python.

A **subtype** is something that can be substituted for and behave as its parent type (and *its* parent type, etc.).

### 10.3.3 Defining Subclasses in Python

We now look at how to employ the object-oriented programming feature of inheritance in Python. We give an example of an “exploded” string class as a subclass of the built-in string class, and look at whether an exploded string can be substituted as a string type.

#### Class Names of the Built-In Types in Python

Recall that all values in Python are objects. Thus, there exists a class definition for each of the built-in types. To determine the type (class name) of a particular value (object) in Python, the built-in function `type` can be used.

```
>>> type(12)           >>> type(12.4)          type('')
<class 'int'>        <class 'float'>       <class 'str'>
>>> type([])          >>> type(())          type({})
<class 'list'>        <class 'tuple'>       <class 'dict'>
```

The resulting expression `<class classname>` gives the associated class name for any value. (We use some empty values here, such as the empty string, but the type of any value can be

determined this way.) A detailed description of a built-in class can be displayed by use of the `help` function,

```
>>> help(int)           >>> help(str)
```

Partial display of the `str` built-in type is given in Figure 10-15.

```
>>> help(str)
Help on class str in module builtins:

class str(object)
| str(string[, encoding[, errors]]) -> str
|
| Create a new string object from the given encoded string.
| encoding defaults to the current default string encoding.
| errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.
|
| Methods defined here:
|
| __add__(...)
|     x.__add__(y) <==> x+y
|
| __contains__(...)
|     x.__contains__(y) <==> y in x
|
| __eq__(...)
|     x.__eq__(y) <==> x==y
|
| .
|
| find(...)
|     S.find(sub[, start[, end]]) -> int
|
|     Return the lowest index in S where substring sub is found,
|     such that sub is contained within S[start:end]. Optional
|     arguments start and end are interpreted as in slice notation.
|
|     Return -1 on failure.
|
| .
```

**FIGURE 10-15** Description of Built-in String Class

For programmer-defined classes, such as the `Fraction` class developed earlier, we use the name of the class (with any required arguments) for creating objects of that type,

```
frac1 = Fraction(1,2)
```

For built-in types such as the `str` type, object instances are generally created using a more convenient syntax. For example, to create a new string value (object), we simply put quotes around the desired characters of the string,

```
name = 'John Smith'
```

For creating a new list, the list elements are surrounded by square brackets,

```
nums = [10, 20, 30, 40]
```

We have seen a similar means of creating tuples, dictionaries, and sets. Knowing the class names of the built-in types, we can alternatively create object instances of each as follows,

>>> int(1)	>>> int('1')	>>> int(1.2)
1	1	1
>>> list([1, 2, 3])	>>> list((1,2,3))	>>> list('123')
[1, 2, 3]	[1, 2, 3]	[1, 2, 3]

When using the class name of a built-in type to create a new object instance, arguments of various types can be used to initialize its value. For example, an integer can be created from a provided string or float value; a list can be created from a provided tuple or string. Finally, we note that the built-in class names in Python contain only lowercase letters. Programmer-defined classes, such as `Fraction` however, are by convention named with a beginning uppercase letter.

### LET'S TRY IT

Enter the following in the Python shell and observe the results.

>>> type(1)	>>> type([])	>>> help(int)
???	???	???
>>> type(1.5)	>>> type([1,2,3])	>>> help(float)
???	???	???
>>> type('')	>>> type(())	>>> help(list)
???	???	???
>>> type('Hi')	>>> type((1,2,3))	>>> help(tuple)
???	???	

Built-in function `type` can be used to determine the type (class name) of any value in Python. Built-in function `help` can be used to get the class description of a built-in type.

### Defining an Exploded String Type

Given the built-in string class, we can easily create a new string type that is identical to the string class, and in addition provide the option of being “exploded.” By an exploded string is meant a string with spaces (blank characters) between all characters, for example, '`H e l l o`'. We call this new class `ExplodedStr`, and give its definition in Figure 10.16.

When defining a subclass, the name of the class is followed by the name of the parent class within parentheses,

```
class ExplodedStr(str):
```

```
# Exploded String Class

class ExplodedStr(str):

    def __init__(self, value = ''):

        # call to init of str class
        str.__init__(value)

    def explode(self):

        # empty str returned unaltered
        if len(self) == 0:
            return self
        else:
            # create exploded string
            empty_str = ''
            blank_char = ' '
            temp_str = empty_str

            for k in range(0, len(self) - 1):
                temp_str = temp_str + self[k] + blank_char

            # append last char without following blank
            temp_str = temp_str + self[len(self)- 1]

            # return exploded str by joining all chars in list
            return temp_str
```

**FIGURE 10-16** ExplodedStr Type as a Subclass of the Built-in str Class

Our new exploded string type can be used as shown below.

```
>>> title = ExplodedStr('My Favorite Movies')
>>> print(title=explode())
M y   F a v o r i t e   M o v i e s
```

Let's see how this works. The ExplodedStr class does not define any instance variables of its own. Thus, its `__init__` method simply calls the `__init__` method of the built-in `str` class to pass the value that the string is to be initialized to. If an initial value is not provided, the method has a default argument assigned to the empty string,

```
def __init__(self, value = ''):

    # call to init of str class
    str.__init__(value)
```

Here `str.__init__(value)` is used to call the `str` class's `__init__` method.

Method `explode` returns the exploded version of the string. First, a check is made to see if the string is the empty string. If so, then the reference `self` is returned, thus returning its unaltered value. Otherwise, a new string is created (`temp_str`), equal to the original string referenced by `self`, with a blank appended after every character except the last. A for loop is used for this construction,

```
for k in range(0, len(self) - 1):
    temp_str = temp_str + self[k] + blank_char
```

Since the `range` function is called with parameters 0 and `len(self) - 1`, all except the last character of the original string is appended by this loop (since the last character should not have a blank character appended after it). Finally, the newly constructed exploded string (`temp_str`) is returned.

We can place the `ExplodedStr` class in its own module called `exploded_str` and import it when needed. Testing this new string type, we see that it behaves as desired.

```
>>> reg_str = 'Hello'                      defining a regular string
>>> ex_str = ExplodedStr('Hello')          defining an exploded string
>>> reg_str                                value of regular string
'Hello'
>>> ex_str                                 value of exploded string
'Hello'
>>> ex_str.explode()                      call to explode method
H e l l o
>>> reg_str == ex_str                     comparing the strings
True
>>>
```

### LET'S TRY IT

For the following underlined string `uStr` class definition, enter and save it in a Python file, and execute. Then enter the associated instructions within the Python shell and observe the results.

```
class uStr(str):
    def __init__(self, u_str):
        str.__init__(u_str)

    def underline(self):
        return str.__str__(self) + '\n' + \
            format('', '-<' + str(len(self)))
```

```
>>> reg_str = 'Hello'
>>> u_str = uStr('Hello')
>>> reg_str
???
>>> u_str
???
>>> u_str.underline()
???
>>> reg_str == u_str
???
```

We see that exploded strings can be used as a regular string or in exploded form. Thus, it is only the added behavior of being able to be exploded that is different, not its value. As a result, the `ExplodedStr` subclass serves as a subtype of the built-in string class. We discuss the related issue of polymorphism next.

When defining a subclass, the name of the class is followed by the name of the parent class within parentheses.

### 10.3.4 Let's Apply It—A Mixed Fraction Class

The following `MixedFraction` class is implemented as a subclass (subtype) of the `Fraction` class developed earlier. The program utilizes the following programming features:

► Inheritance of classes

The `Fraction` class that we developed only represents values as common fractions, that is, with just a numerator and denominator. Thus, the value one and a half is represented as  $3/2$ . Mixed (compound) fractions denote values with a separate whole value and (proper) fraction— $3/2$  is represented as  $1\frac{1}{2}$ . In certain applications, such as in the recipe conversion program developed earlier, the latter representation is preferable. An example of this is given below for a conversion of the chocolate chip cookie recipe. Example testing of the `Fraction` class is shown in Figure 10-17.

<pre>&gt;&gt;&gt; frac1 = Fraction(5, 4) &gt;&gt;&gt; frac1 5/4 &gt;&gt;&gt; frac2 = Fraction(11, 4) &gt;&gt;&gt; frac2 11/4 &gt;&gt;&gt; -frac1 -5/4 &gt;&gt;&gt; frac1 + frac2 4/1 &gt;&gt;&gt; frac1 - frac2 -3/2 &gt;&gt;&gt; frac1 * frac2 55/16 &gt;&gt;&gt;</pre>	<pre>&gt;&gt;&gt; frac1 = MixedFraction(5, 4) &gt;&gt;&gt; frac1 1 1/4 &gt;&gt;&gt; frac2 = MixedFraction(11, 4) &gt;&gt;&gt; frac2 2 3/4 &gt;&gt;&gt; -frac1 -1 1/4 &gt;&gt;&gt; frac1 + frac2 4 &gt;&gt;&gt; frac1 - frac2 -1 1/2 &gt;&gt;&gt; frac1 * frac2 3 7/16 &gt;&gt;&gt;</pre>
--	--

**Use of Fraction Type**

**Use of MixedFraction Type**

**FIGURE 10-17** Interactive Testing of the `MixedFraction` Class

The implementation of the `MixedFraction` class is given in Figure 10-18. The `Fraction` class is imported on **line 3** using the `from-import` form of import. This class provides the (special) methods for performing arithmetic and relational operations on fractions. Therefore, the operations

```

1 # MixedFraction Class
2
3 from fraction import *
4
5 class MixedFraction(Fraction):
6
7     def __init__(self, *args):
8         if len(args) == 2:
9             self.__whole_num = 0
10            Fraction.__init__(self, args[0], args[1])
11        elif len(args) == 3:
12            self.__whole_num = args[0]
13            Fraction.__init__(self, args[1], args[2])
14        else:
15            raise TypeError('MixedFraction takes 2 or 3 arguments ' + \
16                            '(' + str(len(args)) + ' given)')
17
18    def __str__(self):
19        empty_str = ''
20        blank = ' '
21
22        displayFrac = Fraction.copy(self)
23        displayFrac.reduce()
24
25        whole_num = 0
26        numer = displayFrac.getNumerator()
27        denom = displayFrac.getDenominator()
28
29        if numer == 0:
30            return '0'
31
32        if denom == 1:
33            return str(numer)
34
35        if numer < 0:
36            numer = abs(numer)
37            sign = '-'
38        else:
39            sign = empty_str
40
41        if abs(numer) > abs(denom):
42            whole_num = abs(numer) // abs(denom)
43            numer = abs(numer) % abs(denom)
44
45        if whole_num == 0:
46            return sign + str(numer) + '/' + str(denom)
47        else:
48            return sign + str(whole_num) + blank + \
49                            str(numer) + '/' + str(denom)
50
51    def __repr__(self):
52        return self.__str__()
53

```

## Special Methods

**FIGURE 10-18** MixedFraction Class (*Continued*)

of addition, subtraction, multiplication, and comparison on mixed fractions can be accomplished by the inherited methods of the `Fraction` class.

The difference between the `Fraction` class and the `MixedFraction` class is in how fraction values are displayed. For example, a `Fraction` object with the value  $3/2$  is displayed by the

```

54
55
56     def getWholeNum(self):
57         return self.getNumerator() // self.getDenominator()
58
59
60     def setWholeNum(self, value):
61         self.setNumerator(self.getNumerator() + \
62                           value * self.getDenominator())
63
64     def set(self, whole_num, numer, denom):
65         Fraction.set(self, numer + whole_num * denom, denom)
66
67
68                                         Getter and Setter Methods
69     def __neg__(self):
70         return MixedFraction(-Fraction.getNumerator(self),
71                               Fraction.getDenominator(self))
72
73     def __sub__(self, rfraction):
74         tempFrac = Fraction.__sub__(self, rfraction)
75
76         return self.__createMixedFraction(tempFrac)
77
78     def __add__(self, rfraction):
79         tempFrac = Fraction.__add__(self, rfraction)
80
81         return self.__createMixedFraction(tempFrac)
82
83
84     def __mul__(self, rfraction):
85         tempFrac = Fraction.__mul__(self, rfraction)
86
87         return self.__createMixedFraction(tempFrac)
88
89     def __createMixedFraction(self, frac):           Private Methods
90         numer = frac.getNumerator()
91         denom = frac.getDenominator()
92
93         return MixedFraction(numer, denom)
94
95

```

**FIGURE 10-18** MixedFraction Class

MixedFraction class as  $1 \frac{1}{2}$ . For each object instance, however, the same values are stored—as an integer numerator and integer denominator value, and thus can be operated on the same way.

How fractions are displayed is determined by the implementation of the `__str__` special method. In the `Fraction` class, the numerator and denominator values are simply concatenated with a ‘/’ between them. In the `MixedFraction` class, however, the fraction value is reconstructed in three parts, a whole number part (possibly 0), and a proper fraction part in which the numerator is less than the denominator. How the fraction is displayed depends on these three values. If, for example, the whole number part is 0, then only a proper fraction is displayed,

`0 + 1/2` displayed as  $1/2$

If the denominator is 1, then only the numerator is displayed,

$4/1$  displayed as  $4$

Otherwise, both the whole number part and the associated proper fraction part are displayed,

$5/4$  displayed as  $1\frac{1}{4}$

The implementation of special method `__str__` in the `MixedFraction` class is on **lines 18–49**. Variables `empty_str` and `blank` are initialized on **lines 19–20**. These are used in the construction of the mixed-fraction string. On **line 22** a copy is made of the object's value by use of the `copy` method of the `Fraction` class. This is so that the value of the fraction can be reduced to its simplest form, in preparation for the conversion of the fraction value to a whole number and proper fraction part. The reduction is performed by the `reduce()` method inherited from the `Fraction` class.

Once the (temporary) fraction object is reduced to simplest terms, it can be determined whether there is a whole number part to be displayed as part of the fraction value. This depends on whether the numerator is greater than the denominator. Thus, on **line 25**, variable `whole_num` is initialized to 0. The numerator and denominator values of the temporary `displayFrac` object are set to variables `numer` and `denom`, respectively (**lines 26–27**). What remains is to determine how the fraction value should be displayed based on these three values. On **line 29**, if variable `numer` is 0, then the fraction value is also 0; therefore '`0`' is returned (**line 30**). If the denominator in variable `denom` is 1, then the fraction value can be displayed as a single integer value (**lines 32–33**).

Next, special method `__str__` handles the proper display of negative fractions. For example,  $-4/3$  is stored as negative integer value 4 and positive integer value 3. When this value is displayed in mixed fraction form, the negative sign should appear as follows,

$-1\frac{1}{3}$

Thus, on **line 35**, a check is made to determine if the numerator value in `numer` is negative. If so, then `numer` is set to its absolute value (**line 36**). This prevents a negative sign with the numerator from being displayed. Because a negative sign will need to appear with the whole number part of the displayed value, variable `sign` is set the dash character (**line 37**). If, on the other hand, the numerator is found to be nonnegative, then variable `sign` is set to the empty string (**line 39**).

In addition to checking the sign of the numerator, a check is made to see if the numerator is greater than the denominator (**line 41**). If so, then there needs to be a separate whole number value displayed along with the (proper) fraction value. Thus, if the numerator is found to be greater, then variable `whole_num` is set to the result of the integer division of the absolute value of the numerator by the absolute value of the denominator (**line 42**). Absolute values are used because the sign of the displayed result has already been determined (stored in variable `sign`). The numerator is then set to the remaining fractional part by use of the modulo operator (**line 43**). Thus, for  $4/3$ , `whole_num` is set to  $1$  ( $4 // 3$ ), and `numer` is set to  $1$  ( $4 \% 3$ ). Finally, if the whole number part of the value is 0, then a string containing only the proper fraction is returned (**line 46**). If not 0, then a string of the form ' $1\frac{1}{3}$ ' is constructed and returned (**lines 48–49**).

The remaining part of the `MixedFraction` class provided the needed getters and setters for mixed fractions (**lines 56–65**). It also provides a set of arithmetic operators and arithmetic methods to replace those inherited from the `Fraction` class (**lines 69–87**). Finally, private method `__createMixedFraction` is defined as a private supporting method called by the arithmetic operators in the class.

## Self-Test Questions

1. A class is made a subclass of another class by the use of \_\_\_\_\_.
2. Which of the following contains terms that mean the same thing?
  - (a) Parent class, derived class
  - (b) Parent class, base class
  - (c) Subclass, base class
3. All subclasses are a subtype in object-oriented programming. (TRUE/FALSE)
4. When defining a subclass in Python that is meant to serve as a subtype, the `subtype` Python keyword is used. (TRUE/FALSE)
5. Built-in function `type` can be used to,
  - (a) Determine the type of only the built-in types in Python
  - (b) Determine the type of only programmer-defined types (classes)
  - (c) Determine the type of all types

ANSWERS: 1. inheritance, 2. (b), 3. False, 4. False, 5. (c)

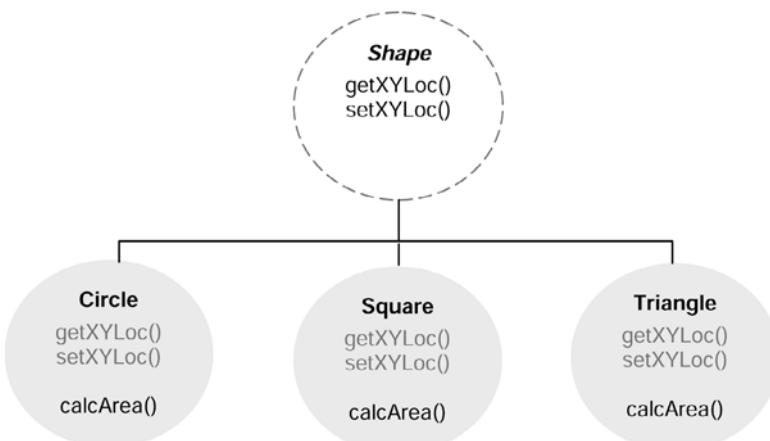
## 10.4 Polymorphism

Polymorphism is a powerful feature of object-oriented programming languages. It allows for the implementation of elegant software that is well designed and easily modified. We explore the use of polymorphism in this section.

### 10.4.1 What Is Polymorphism?

The word *polymorphism* derives from Greek meaning “something that takes many forms.” In object-oriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type. For example, consider the `Shape` class and its subclasses given in Figure 10-19.

All `Shape` objects have an `x`, `y` coordinate (with corresponding getter and setter methods). In addition, `Shape` objects can also calculate their areas. How a shape’s area is computed, however,



**FIGURE 10-19** Polymorphic Shape Class

depends on what shape it is. Thus, it is not possible to define a `calcArea` method in the `Shape` class that serves the purposes of all types of shapes. On the other hand, we want all shape types to have a `calcArea` method. Therefore, we add an *unimplemented* version of the `calcArea` method to the `Shape` class as shown in Figure 10-20.

```
class Shape:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getXYLoc(self):
        return (self.__x, self.__y)

    def setXYLoc(self, x, y):
        self.__x = x
        self.__y = y

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")
```

**FIGURE 10-20** Abstract Shape Class

Subclasses of the `Shape` class must implement the `calcArea` method, otherwise a `NotImplementedError` exception is raised. A class in which one or more methods are unimplemented (or only implemented to raise an exception) is called an **abstract class**. Figure 10-21 gives `Circle`, `Square`, and `Triangle` subclasses of the `Shape` class.

```
class Circle(Shape):
    def __init__(self, x, y, r):
        Shape.__init__(self, x, y)
        self.__radius = r

    def calcArea(self):
        return math.pi * self.__radius ** 2

class Square(Shape):
    def __init__(self, x, y, s):
        Shape.__init__(self, x, y)
        self.__side = s

    def calcArea(self):
        return self.__side ** 2

class Triangle(Shape):
    def __init__(self, x, y, s):
        Shape.__init__(self, x, y)
        self.__side = s

    def calcArea(self):
        return (self.__side ** 2) * math.sqrt(3) / 4.0
```

**FIGURE 10-21** Subclasses `Circle`, `Square`, and `Triangle`

Each of the subclasses contains an `__init__` method, in which the first two arguments provide the `x, y` location of the shape (within a graphics window) and the third argument indicates its size. Each first calls the `__init__` method of the `Shape` class with arguments `x,y` to set its location, since the `x,y` values are maintained by the `Shape` class. Note that methods `getXYLoc` and `setXYLoc` are not defined in the subclasses, as they are inherited from the `Shape` class.

The size of each shape is handled differently, however. In the `Circle` class size is stored as the radius, and in the `Square` and `Triangle` classes it is stored as the length of each side. (The `Triangle` class represents only equilateral triangles, those in which each side is of the same length.) Given these classes, we can now see how polymorphism works in Python.

Suppose that there was a list of `Shape` objects for which the total area of all shapes combined was to be calculated,

```
shapes_list = (circle1, circle2, square1, triangle1, triangle2)
```

Because each implements the methods of the `Shape` class, they are all of a common general type, and therefore can be treated in the same way,

```
total_area = 0
for shape in shapes_list:
    total_area = total_area + shape.calcArea()
```

### LET'S TRY IT

For the following class definitions, enter and save them in a single Python file, and execute. Then enter the associated instructions within the Python shell and observe the results.

```
class Bird(object):
    def __init__(self, w):
        print('__init__ of Bird Class called')
        self.__weight = w

    def getWeight(self):
        return str(self.__weight) + 'ounces'

    def getColor(self):
        raise NotImplementedError( \
            'Method color not implemented')

class BlueJay(Bird):
    def __init__(self, w):
        Bird.__init__(self, w)

    def getColor(self):
        return 'Blue'

class Cardinal(Bird):
    def __init__(self, w):
        Bird.__init__(self, w)

    def getColor(self):
        return 'Red'

class BlackBird(Bird):
    def __init__(self, w):
        Bird.__init__(self, w)

    def getColor(self):
        return 'Black'

>>> b1 = BlueJay(1)
???
>>> b2 = Cardinal(1.4)
???
>>> b3 = BlackBird(3.5)
???
>>> b1.getWeight()
???
>>> b2.getWeight()
???
>>> b3.getWeight()
???
>>> b1.getColor()
???
>>> b2.getColor()
???
>>> b3.getColor()
???
>>> b4 = Bird(2.0)
>>> b.getColor()
```

In Python, it is not because `Circle`, `Square`, and `Triangle` are subclasses of the `Shape` class that allows them to be treated in a similar way. It is because the classes are *subtypes* of a common parent class. (Actually, in Python, any set of classes with a common set of methods, even if not subclasses of a common type, can be treated similarly. This kind of typing is called **duck typing**—that is, “if it looks like a duck and quacks like a duck, then it’s a duck.”)

In object-oriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type.

### 10.4.2 The Use of Polymorphism

To fully appreciate the benefits of polymorphism, let’s consider the development of a program for manipulating geometric shapes. We consider a graphical environment in which classes `Circle`, `Square`, and `Triangle` do not have a common set of methods, and therefore cannot be treated polymorphically.

We assume that we have a graphics program in which the user selects the geometric shape that they want (stored in variable `selected_shape`) for which the appropriate object type is created:

```
if selected_shape == 1:
    cir = Circle(0, 0, 1)
elif selected_shape == 2:
    sqr = Square(0, 0, 1)
elif selected_shape == 3:
    tri = Triangle(0, 0, 1)
```

Next, the geometric object is displayed. Since each object has its own set of methods, the appropriate method must be called. This is determined by use of another `if` statement:

```
if selected_shape == 1:
    cir.drawCircle()
elif selected_shape == 2:
    sqr.drawSquare()
elif selected_shape == 3:
    tri.drawTriangle()
```

The user may then request that the area of the graphic object be displayed. Since each of the `Circle`, `Square`, and `Triangle` classes have a different method for calculating their area, then an `if` statement must again be used:

```
if selected_shape == 1:
    area = cir.calcCircleArea()
elif selected_shape == 2:
    area = sqr.calcSquareArea()
elif selected_shape == 3:
    area = tri.calcTriangleArea()
```

And when the graphic object is repositioned, an `if` statement is again needed:

```
if selected_shape == 1:
    cir.moveCircle(x, y)
elif selected_shape == 2:
    sqr.moveSquare(x, y)
elif selected_shape == 3:
    tri.moveTriangle(x, y)
```

The design of this program becomes rather tedious and inelegant. If statements abound throughout the program. Let's now look at how the same program can be written with the use of polymorphism. First, we give a more complete `Shape` class in Figure 10-22.

```
class Shape(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getXYLoc(self):
        return (self.__x, self.__y)

    def setXYLoc(self, x, y):
        self.__x = x
        self.__y = y

    def draw(self):
        raise NotImplementedError("Method draw not implemented")

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")

    def resize(self, amt):
        raise NotImplementedError("Method resize not implemented")
```

**FIGURE 10-22** Methods of `Circle`, `Square`, and `Triangle` Classes Following Duck Typing

As before, the unimplemented methods of the class raise a `NotImplementedError` exception. In this case, each of the `Circle`, `Square`, and `Triangle` classes are defined as subclasses of the `Shape` class. Therefore, each is required to have methods for the unimplemented methods of the `Shape` class in order to be completely defined.

We reconsider the implementation of a graphics program for manipulating geometric shapes. We give side-by-side listing of nonpolymorphic vs. polymorphic code in Figure 10-23.

One benefit of polymorphism, as apparent from the example, is that the program code is much more straightforward and concise. Without polymorphism, there is constant use of `if` statements that clutters up the code. Another benefit is that it allows the programmer to think at a more abstract level (“this section of code needs to draw *some kind of geometric shape*”) without having to keep in mind specific entities such as circles, squares, and triangles (“this section of code needs to draw *either a circle, square, or triangle*”).

The most significant benefit of polymorphism is that programs are much easier to maintain and update. Suppose, for example, that our program had to be updated to also handle rectangles.

Non-Polyomophic Code	Polyomophic Code
<pre> # Create Appropriate Object  if selected_shape == 1:     cir = Circle(0, 0, 1) elif selected_shape == 2:     sqr = Square(0, 0, 1) elif selected_shape == 3:     tri = Triangle(0, 0, 1)  # draw if selected_shape == 1:     cir.drawCircle() elif selected_shape == 2:     sqr.drawSquare() elif selected_shape == 3:     tri.drawTriangle()  # calc area if selected_shape == 1:     area = cir.calcCircleArea() elif selected_shape == 2:     area = sqr.calcSquareArea() elif selected_shape == 3:     area = tri.calcTriangleArea()  # resize if selected_shape == 1:     cir.resizeCircle(percentage) elif selected_shape == 2:     sqr.resizeSquare(percentage) elif selected_shape == 3:     tri.resizeTriangle(percentage)  # reposition if selected_shape == 1:     cir.setXY(x, y) elif selected_shape == 2:     sqr.setPosition(x, y) elif selected_shape == 3:     tri.moveTo(x, y) </pre>	<pre> # Create Appropriate Object  if selected_shape == 1:     fig = Circle(0, 0, 1) elif selected_shape == 2:     fig = Square(0, 0, 1) elif selected_shape == 3:     fig = Triangle(0, 0, 1)  # draw fig.draw()  # calc area area = fig.calcArea()  # resize fig.resize(selected_percentage)  # reposition fig.setXYLoc(x, y) </pre>

For the call to method `draw()`, the method defined in the specific subclass is the actual method called.

For the call to method `calcArea()`, the method defined in the specific subclass is the actual method called.

For the call to method `resize()`, the method defined in the specific subclass is the actual method called.

For the call to method `setXYLoc()`, there is no method defined in any of the subclasses. Therefore, for each particular shape, the method of the `Shape` class is the method called.

**FIGURE 10-23** Nonpolyomophic vs. Polyomophic Code

How much of the program would need to be changed? Without polymorphism, every if statement for the selection of method calls would need to be updated. With polymorphism, however, only the first if statement that creates the appropriate type object would need to be changed to include a `Rectangle` type. The rest of the code would remain the same, as shown in Figure 10-24.

In the end, selection needs to occur somewhere—either within the program (in the nonpolyomophic approach), or by the programming language (with the use of polymorphism).

Polymorphism allows the programmer to think at a more abstract level during program development, and supports the development of programs that are easier to maintain and update.

```
# Create Appropriate Object

if selected_shape == 1:
    fig = Circle(0, 0, 1)
elif selected_shape == 2:
    fig = Square(0, 0, 1)
elif selected_shape == 3:
    fig = Triangle(0, 0, 1)
elif selected_shape == 4:
    fig = Rectangle(0, 0, 1)
```

**FIGURE 10-24** Updated Polymorphic Code for Incorporating a New Rectangle Type

### Self-Test Questions

1. The term *polymorphism* in object-oriented programming refers to
  - (a) The ability to treat various type objects in a similar way
  - (b) The ability to change an object from one type to another type
  - (c) The ability to have multiple objects of the same type treated as one
2. The use of duck typing in Python results in
  - (a) More restriction on the type values that can be passed to a given method
  - (b) Less restriction on the type values that can be passed to a given method
3. The *biggest* reason for the use of polymorphism in a program is
  - (a) There is less program code to write
  - (b) The program will result in a more elegant design, and thus will be easier to maintain and update
  - (c) It allows the programmer to think at a more abstract level

ANSWERS: 1. (a), 2. (b), 3. (b)

## 10.5 Object-Oriented Design Using UML

We have mainly focused on object-oriented *programming* (OOP). The first step in the development of any object-oriented program, however, is the development of an appropriate **object-oriented design (OOD)**. Next we discuss a specification language for denoting an object-oriented design, referred to as the *Unified Modeling Language*.

### 10.5.1 What Is UML?

The **Unified Modeling Language (UML)** is a standardized design specification (modeling) language for specifying an object-oriented design. The term “Unified” comes from the fact that the language is a unification of three earlier object-oriented design modeling languages.

UML is a language-independent, *graphical* specification language. It contains numerous types of graphical diagrams for expressing various aspects of an object-oriented design. One of the most widely used graphical diagrams is called a “class diagram.” A class diagram specifies the

classes and their relationships of a given object-oriented design. We look at class diagrams in UML next.

**UML** (“Unified Modeling Language”) is a standardized language-independent, graphical modeling language for specifying an object-oriented design.

### 10.5.2 UML Class Diagrams

In UML, **class diagrams** are used to express the *static* aspects of a design, such as the instance variables and methods of individual classes, their visibility (i.e., public or private), and various relationships between classes. Other UML diagrams, called **interaction diagrams**, are used to represent the sequence of method calls between objects during program execution (the *dynamic* aspect of a design). We omit discussion of interaction diagrams and look at the UML notation for denoting the classes of an object-oriented design.

**Class diagrams** in UML are used to express the static aspects of an object-oriented design. Other diagrams, called **interaction diagrams**, are used to represent the sequence of method calls between objects during program execution.

### The Representation of Classes

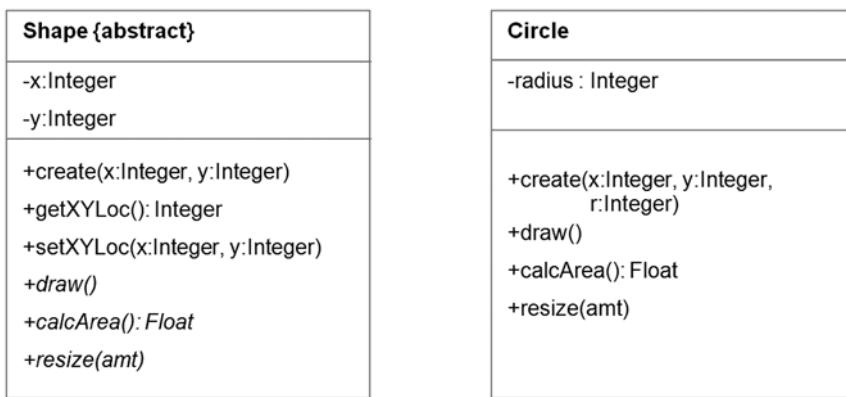
A class is denoted in UML in three parts: a class name, a set of class attributes (instance variables), and a set of methods, as given below.

<b>Class Name</b>
instance variables
methods

We give the UML specification for the abstract `Shape` class and the (concrete) `Circle` class in Figure 10-25. The names of unimplemented (abstract) methods are denoted in italics.

Initialization methods like `__init__` in Python are named `create()` in UML. The types of attributes (instance variables) and the return type of methods is indicated by `:<type name>`, for any given type (for example, `:Integer`). The `+` and `-` symbols are used to specify if a given member of a class has either public (+) or private (-) access.

A class is denoted in UML in three parts: a class name, a set of class attributes (instance variables), and a set of methods.



**FIGURE 10-25** UML Class Diagrams for Shape and Circle Classes

### Denoting Associations between Classes

Associations are the most common relationship in UML class diagrams. An **association** between two classes indicates that the methods of one class make calls to methods of the other, as shown in Figure 10-26.



**FIGURE 10-26** Association in UML

In this example, it is assumed that the classes are part of a graphical design package, in which the **GraphicsWindow** class creates and can manipulate a set of **Shape** objects. (We have left out the details of the **GraphicsWindow** and **Shape** classes in this diagram.)

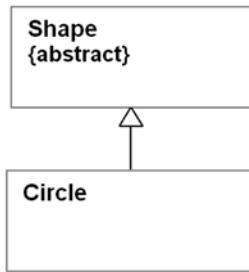
The numbers above the association ends are referred to as **multiplicity**. The multiplicity of 1 at the **GraphicsWindow** end of the association and 0..\* at the **Shape** end indicates that one **GraphicsWindow** object may be associated with any number of (zero or more) **Shape** objects. The “creates” label at the **GraphicsWindow** association end is referred to as a **role name**. Role names are used to describe an association between two classes.

Finally, the arrow denotes **navigability**. It indicates the direction of method calls made. In this example, it shows that a **GraphicsWindow** object makes method calls (sends messages) to **Shape** objects, and not the other way around.

An **association** between two classes, denoted by a connecting solid line (and a possible arrow-head) indicates that methods of one class call methods of the other.

### Denoting Subclass Relationships

Subclasses are indicated in UML by use of a solid line with a closed arrow head from a subclass to its superclass, as shown in Figure 10-27.



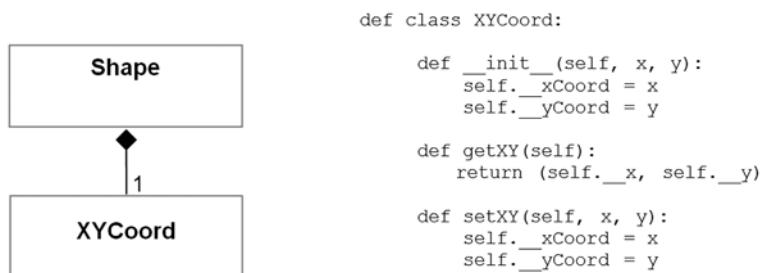
**FIGURE 10-27** UML  
Subclass Notation

This diagram indicates that the `Circle` class is a subclass of the abstract `Shape` class. (We again have left out the details of both the `Shape` and `Circle` classes.) Note that multiplicity is not used in subclass relationships.

Subclass relationships in UML are indicated by use of a solid line with a closed arrow head from a subclass to a superclass.

### Denoting Composition vs. Aggregation

**Composition** indicates a “part-of” relationship between classes (Figure 10-28). The containing object is viewed as “owning” the contained object, of which the contained object is an integral part. For example, the `Shape` class is comprised of two integers for holding the `x,y` location of any `Shape` type. Because these two integer values are used together, we could develop an `XYCoord` class, in which the `Shape` class contains an instance. This is an example of the use of composition.



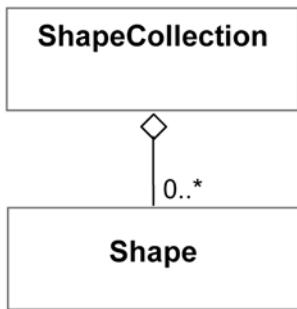
**FIGURE 10-28** Composition of Classes in UML

When denoting composition, a filled diamond head is used at the end of the line connected to the containing class (the `Shape` class). With composition, it is implied that the containing class (`Shape`) makes calls to the member class (`XYCoord`), and thus composition also connotes the relationship of association.

We note that, since all values in Python are objects, *every* class with instance variables involves the use of composition, including the `Shape` class. However, the reason to indicate a non-built-in type as composition is that it provides a place to specify the details of the type.

Instance variables of a built-in type can simply be included in the class as primitive types as was done for the `x` and `y` instances variables of the `Shape` class above.

**Aggregation**, in contrast to composition, is not a part-of relationship. It is used to denote a class that groups together, or aggregates, a set of objects that exists independently of the aggregating class. An example of aggregation is given in Figure 10-29.



```

def class ShapeCollection:
    def __init__(self):
        self.__collection = []

    def add(self, shape):
        self.__collection.append(shape)

    def remove(self, shape):
        self.__collection.remove(shape)
  
```

FIGURE 10-29 Aggregation of Classes in UML

Aggregation is denoted by an unfilled diamond head. Here, the `ShapeCollection` class contains references to an arbitrary number of `Shape` objects. This might be used, for example, when a graphics window allows the user to select a group of `Shape` objects on the screen and change an attribute of each, such as their size, all at once.

**Composition** is a “part of” relationship between classes denoted by a filled diamond head in UML. **Aggregation** is a “grouping” relationship, denoted by an unfilled diamond head.

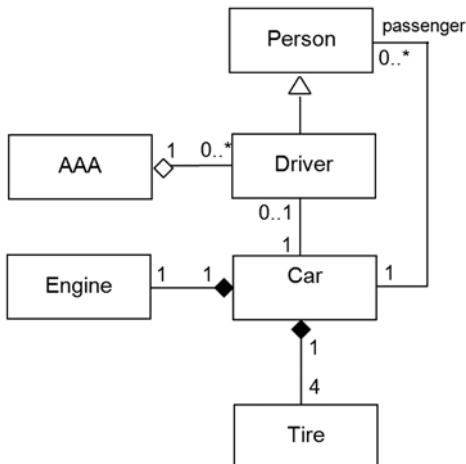
## An Example Class Diagram

Although UML is a specification language for modeling object-oriented software, it can be used to specify any set of entities and their relationships. Modeling everyday concepts and entities can be instructive in understanding UML. Figure 10-30 shows a UML class diagram modeling the concept of a car.

For every car, there is one engine, an integral part of a car. Thus, a composition relationship is denoted between `Car` and `Engine`, with a multiplicity of `1` on each end. As with engines, tires are an integral part of a car, so this is also indicated by composition, with a multiplicity of four tires for each passenger car.

A car is still a car with or without a driver. Therefore, a relationship of composition is not appropriate here. There is simply an association between `Car` and `Driver` with multiplicity of `0..1` on the driver end of the association. Since a `Driver` is a `Person`, a subclass relationship is denoted between the two.

There is an association between `Car` and `Person` with a multiplicity of `0..*`. Since the association denoted is not so apparent (owner? passenger?), we add the role name `Passenger` to the `Person` end of the association to be more explicit. Finally, any number of `Drivers` may belong to AAA (Automobile Association of America), denoted by the use of aggregation.



**FIGURE 10-30** Passenger Car UML Class Diagram

---

### Self-Test Questions

1. Which of the following is true of UML?
  - UML is a specification language for designing Python programs
  - UML is a specification language that can be used for designing programs in various programming languages
  
2. In UML, class diagrams are used to express the \_\_\_\_\_ aspects of a design, and \_\_\_\_\_ are used to denote the dynamic aspects
  
3. In UML, an association between two classes indicates that
  - The two classes have a common superclass
  - Objects of each of the two class types are created at the same time
  - Methods of one of the classes make calls to methods of the other
  
4. Multiplicity in UML indicates
  - How many objects of a given class type exist
  - How many objects of one given class there are in relation to another
  - How many subclasses of a given class there may be
  
5. Composition in UML indicates,
  - A “part of” relationship
  - A grouping of objects
  
6. Aggregation in UML indicates,
  - A “part of” relationship
  - A grouping of objects

ANSWERS: 1. (b), 2. static, interaction diagrams, 3. (c), 4. (b), 5. (a), 6. (b)

## COMPUTATIONAL PROBLEM SOLVING

---

### 10.6 Vehicle Rental Agency Program

In this section, we design, implement, and test a program that will serve the needs of a vehicle rental agency.



#### 10.6.1 The Problem

The problem is to develop an object-oriented design and implementation of a program capable of maintaining reservations for a vehicle rental agency. The agency rents out three types of vehicles—cars, vans, and moving trucks. The program should allow users to check for available vehicles, request rental charges by vehicle type, get the cost of renting a particular type vehicle for a specified period of time, and make/cancel reservations.

#### 10.6.2 Problem Analysis

The program needs an appropriate set of objects for the vehicle rental agency domain. An obvious class to include is a **Vehicle** class. It can be implemented to maintain information common to all vehicle types. Subclasses of the **Vehicle** class can maintain information specific to each subtype.

For example, all vehicles have a miles-per-gallon rating and a vehicle identification number (VIN). Thus, this information can be maintained in the **Vehicle** class. However, there are different make and model cars (with either two or four doors, that hold a specific number of passengers); different make and model vans (able to hold a specific number of passengers); and moving trucks of various lengths, each providing a certain amount of cargo space. Therefore, the **Vehicle** class is made a superclass of classes **Car**, **Van**, and **Truck**, in which each subclass contains information (instance variables and/or methods) specific to that vehicle type.

For each type vehicle, there is a rental charge based on daily, weekly, and weekend rental rates. There is also a mileage charge and some number of free miles (on select vehicles), plus the cost of optional insurance. Because these costs are associated with particular types, but cost is not inherently *part* of a vehicle's attribute, we include a separate **VehicleCost** class.

Finally, we incorporate a **Reservation** class that maintains the information for each reservation made. This will include the customer name, address, credit card number, and the VIN of the vehicle rented.

#### 10.6.3 Program Design

##### Meeting the Program Requirements

The general requirements for this program are for users to be able to check for the availability of vehicles of a certain type (cars, vans, or trucks); request rental charges by vehicle type; determine the rental cost for a particular vehicle and rental period; and make and cancel reservations. The specific requirements of this program are given in Figure 10-31.

The program must maintain a group of specific model vehicles for the following vehicle categories: cars, vans, and (moving) trucks with the following characteristics:

Cars: make/model, miles-per-gallon, num of passengers, num of doors, VIN  
 Vans: make/model, miles-per-gallon, number of passengers, VIN  
 Trucks: miles-per-gallon, length, number of rooms, VIN

The program must be able to display the specific vehicles available for rent by vehicle type.

The program must display the cost associated with a given type vehicle including daily, weekend and weekly rate, insurance cost, mileage charge, and number of free miles. It must also allow the user to determine the cost of a particular vehicle, for a given period of time, an estimated number of miles, and the cost of optional insurance.

The program must be able to allow a particular vehicle to be reserved and cancelled.

**FIGURE 10-31** Program Requirements for the Vehicle Rental Agency Program

The specific rental costs for each vehicle type are given in Figure 10-32.

	Daily Rate	Weekly Rate	Weekend Rate	Free Miles Per Day	Per Mile Charge
Car	\$24.99	\$180.00	\$45.00	100	0.15
Van	\$35.00	\$220.00	\$55.00	0	0.20
Truck	\$34.95	\$425.00	\$110.00	25	0.25

**FIGURE 10-32** Rental Costs by Vehicle Type

The specific vehicles in stock at the rental agency are shown in Figure 10-33.

## Data Description

All the data is stored as string types, converted to a numeric type when needed in a computation (such as the cost of daily insurance).

## Algorithmic Approach

The algorithmic methods of the program will consist of simple search (for finding and retrieving the requested vehicle information by the user), updating of information (for marking vehicles as reserved or unreserved), and direct calculation (for calculating the total cost of a rental).

Make/Model	Mileage	Num Passengers	Num Doors	Vehicle #
<u>CARS</u>				
Chevrolet Camaro	30 mpg	4	2	WG8JM5492DY
Chevrolet Camaro	30 mpg	4	2	KH4GM4564GD
Ford Fusion	34 mpg	5	4	AB4FG5689GM
Ford Fusion Hybrid	35 mpg	5	4	GH2KL4278TK
Ford Fusion Hybrid	32 mpg	5	4	KU4EG3245RW
Chevrolet Impala	36 mpg	6	4	QD4PK7394JI
Chevrolet Impala	30 mpg	6	4	RK3BM4256YH
<u>Vans</u>				
Chrysler Town&Country	25 mpg	7		DK3KG8312UE
Chrysler Town&Country	25 mpg	7		VM9RE2645TD
Chrysler Town&Country	25 mpg	7		WK8BF4287DX
Dodge Caravan	25 mpg	7		QK3FL4278ME
Dodge Caravan	25 mpg	7		KY8EW2053XT
Ford Expedition	20 mpg	8		JK2RT8364HY
Ford Expedition	20 mpg	8		KH4ME4216XW
<u>Trucks</u>				
Ten-Foot	12 mpg	1 bedroom		EJ5KU2435BC
Ten-Foot	12 mpg	1 bedroom		KF8JP7293EK
Seventeen-Foot	10 mpg	2 bedrooms		KG4DM5472RK
Seventeen-Foot	10 mpg	2 bedrooms		PR8JH4893WQ
Twenty-Four-Foot	8 mpg	4 bedrooms		EP2WR3182QB
Twenty-Four-Foot	8 mpg	4 bedrooms		TY3GH4290EK
Twenty-Four-Foot	8 mpg	4 bedrooms		KU9FL4235RH

**FIGURE 10-33** Specific Vehicles of the Vehicle Rental Agency

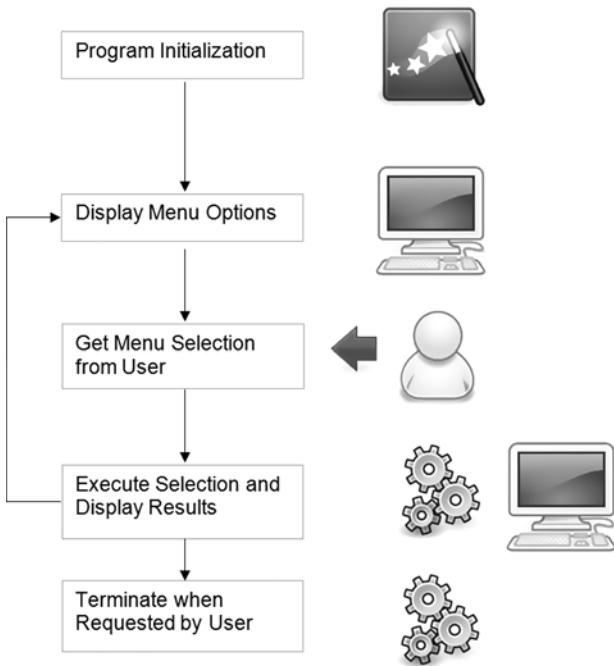
## Overall Program Steps

The overall steps in this program design are given in Figure 10-34.

## UML Class Diagram

We give a UML class diagram for the program in Figure 10-35. In addition to the “domain objects” that we have decided on in our analysis, we add a text-based user interface, provided by the `RentalAgencyUI` class.

Three classes store the information in the system—`Vehicle` (and its subclasses), `VehicleCost`, and `Reservation`. For each of these classes there is a corresponding aggregator class—`Vehicles`, `VehicleCosts`, and `Reservations`—that maintains a collection of the corresponding object type. Each aggregator class has methods for maintaining its collection of objects (for example, `addVehicle` in the `Vehicles` class and `addVehicleCost` in the `VehicleCosts` class).



**FIGURE 10-34** Overall Design of the Vehicle Rental Agency Program

The **Vehicle** class has three subclasses—**Car**, **Van**, and **Truck**. It is responsible for maintaining a vehicle’s type, its VIN, and its reservation status. Method `getDescription` is provided in the **Vehicle** class to return the information common to all vehicles: miles per gallon, and a VIN. Each subclass builds on this inherited method to include the specific information for that vehicle type. The **Car** class stores the maximum number of passengers and number of doors, the **Van** class stores the maximum number of passengers, and the **Truck** class stores its length and the number of rooms of storage it can hold.

The **VehicleCost** class does not have any subclasses. Its `__init__` method is passed six arguments: the daily/weekly/weekend rates, the number of free miles, the per mile charge, and the daily insurance rate to initialize the object with. The `getVehicleCost` method of the **VehicleCosts** aggregating class returns the cost of a specified vehicle type as a single string for display. The **Reservation** and corresponding **Reservations** aggregator class are designed in a similar manner.

Finally, a **SystemInterface** class provides all the methods that any user interface would need for interacting with the system. Such a set of methods is referred to as an API—*Application Programming Interface*. Thus, the **SystemInterface** object is created first. It then reads all the vehicle rental agency data from text files **VehiclesStock.txt** and **RentalCost.txt** and populates the corresponding objects. Then, an instance of the **RentalAgencyUI** is created and initialized with a reference to the system interface. The only public method of the

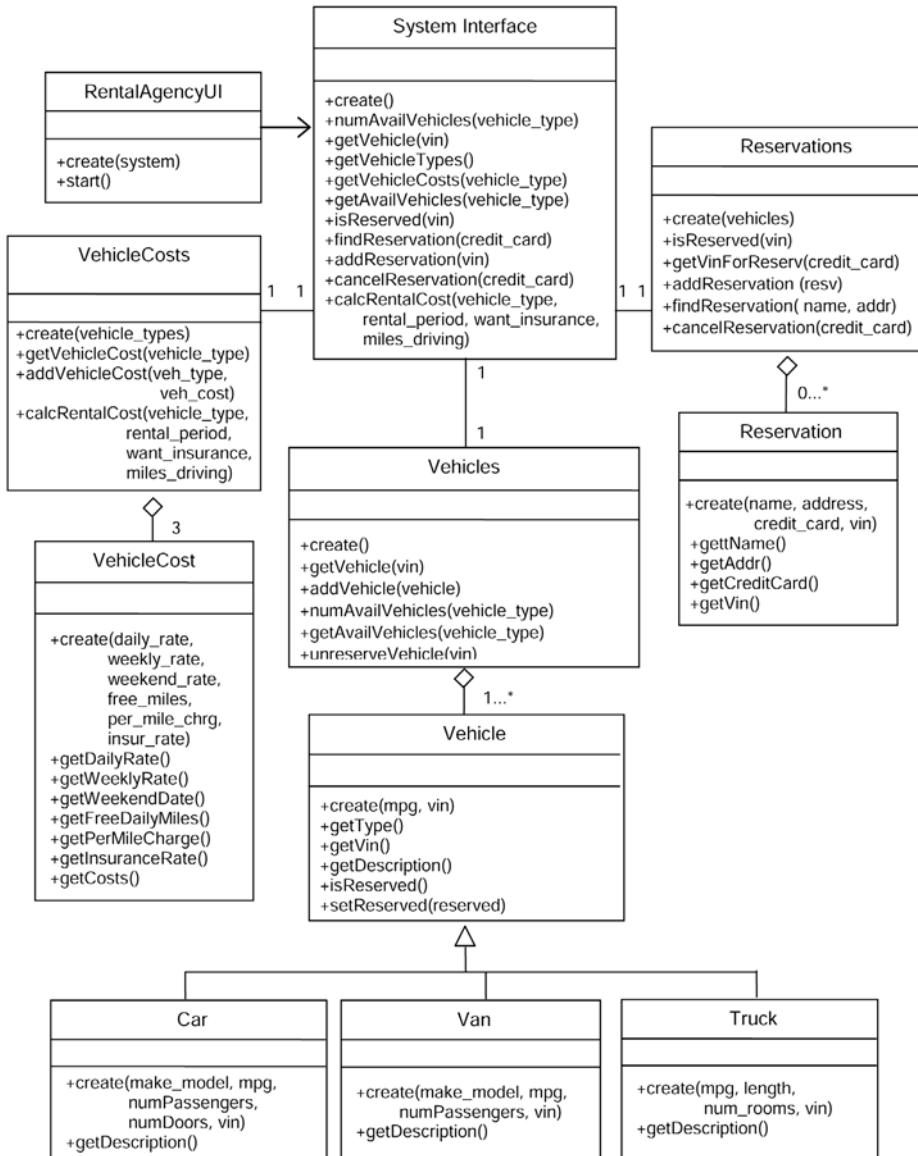


FIGURE 10-35 Class Diagram for Vehicle Rental Agency Program

**RentalAgencyUI** class, `start()`, is called to start the console interaction. The main menu for the program is given in Figure 10-36.

Example use of the system is shown in Figure 10-37. (For the sake of space, the main menu is not repeatedly shown before each selection as in the actual program execution.)

```
<<< MAIN MENU >>>
1 - Display vehicle types
2 - Check rental costs
3 - Check available vehicles
4 - Get cost of specific rental
5 - Make a reservation
6 - Cancel a reservation
7 - Quit

Enter:
```

**FIGURE 10-36** Text-Based (Console) Interface for the Vehicle Rental Agency Program

#### Display Vehicle Types

```
*****
* Welcome to the Friendly Vehicle Rental Agency *
*****  

<<< MAIN MENU >>>
1 - Display vehicle types
2 - Check rental costs
3 - Check available vehicles
4 - Get cost of specific rental
5 - Make a reservation
6 - Cancel a reservation
7 - Quit

Enter: 1
----- Types of Vehicles Available for Rent -----
1 - Car
2 - Van
3 - Truck
-----
```

#### Display Rental Fees for a Given Type Vehicle

```
Enter: 2
Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
----- Rental Charges for Cars -----
      Free   Per Mile   Daily
Daily  Weekly  Weekend Miles Charge Insurance
24.99 180.00  45.00   100   .15    14.99
-----
```

#### Check for Available Vehicles

```
Enter: 3
Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
----- Available Cars -----
Chevrolet Camaro    passengers: 4  doors: 2    mpg: 30  vin: WG8JM5492DY
Chevrolet Camaro    passengers: 4  doors: 2    mpg: 30  vin: KH4GM4564GD
Ford Fusion         passengers: 5  doors: 4    mpg: 34  vin: AB4FG5689GM
Ford Fusion Hybrid  passengers: 5  doors: 4    mpg: 36  vin: GH2KL4278TK
Ford Fusion Hybrid  passengers: 5  doors: 4    mpg: 36  vin: KU4EG3245RW
Chevrolet Impala    passengers: 6  doors: 4    mpg: 30  vin: QD4PK7394JL
-----
```

#### Get the Cost of a Particular Rental

```
Enter: 4
Enter type of vehicle
1 - Car
```

**FIGURE 10-37** Example Program Execution (*Continued*)

```

2 - Van
3 - Truck
Enter: 1
Enter the rental period:
1 - Daily  2 - Weekly  3 - Weekend

Enter: 1
How many days do you need the vehicle? 3
Would you like the insurance? (y/n): n
Number of miles expect to drive?: 240

----- ESTIMATED Car RENTAL COST -----
* You have opted out of the daily insurance *

Daily rental for 3 days would be $ 74.97

Your cost with an estimated mileage of 240 would be 95.97
which includes 100 free miles and a charge of 0.15 per mile
-----

```

### Reserve a Particular Vehicle

```

Enter: 5

Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
----- Available Cars -----
1-Chevrolet Camero    passengers: 4   doors: 2   mpg: 30   vin: W68JM5492DY
2-Chevrolet Camero    passengers: 4   doors: 2   mpg: 30   vin: KH4GM4564GD
3-Ford Fusion         passengers: 5   doors: 4   mpg: 34   vin: AB4FG5689GM
4-Ford Fusion Hybrid  passengers: 5   doors: 4   mpg: 36   vin: GH2KL4278TK
5-Ford Fusion Hybrid  passengers: 5   doors: 4   mpg: 36   vin: KU4EG3245RW
6-Chevrolet Impala    passengers: 6   doors: 4   mpg: 30   vin: QD4PK7394JL
7-Chevrolet Impala    passengers: 6   doors: 4   mpg: 30   vin: RK3BM4256YH

Enter number of vehicle to reserve: 4
Ford Fusion Hybrid    passengers: 5   doors: 4   mpg: 36   vin: GH2KL4278TK

Enter first and last name: John Smith
Enter address: 123 Main Street, Sometown 12345
Enter credit card number: 204881605
* Reservation Made *

```

### Cancel a Reservation

```

Enter: 6
Please enter your credit card number: 204881605

RESERVATION INFORMATION
Name: John Smith 123 Main Street, Sometown 21252
Vehicle: Chevrolet Camaro    passengers: 4   doors: 2   mpg: 30   vin: KH4GM4564GD

Confirm Cancellation (y/n): v
* Reservation Cancelled *

```

```

Enter: 7
Thank you for using the Friendly Rental Agency

```

**FIGURE 10-37** Example Program Execution

## 10.6.4 Program Implementation and Testing

From the UML specification, we implement and test the program. We start with the implementation of the `Vehicle` class, given in Figure 10-38. The classes will be placed in their own file, and imported into the program.

### Development and Testing of the Vehicle, Car, Van, and Truck Classes

In the `Vehicle` class, the `__init__` method (lines 10–15) defines the attributes common to each subclass—`mpg`, `vin`, and `reserved`. There is only one setter method, `setReserved`, since the other values are defined when the object is created. Other methods include getters `getType` (line 17), `getVin` (line 22), `getDescription` (line 27), and Boolean `isReserved` (line 35).

```

1  """This module provides a Vehicle class."""
2
3  class Vehicle:
4      """The Vehicle class holds the mpg, vin and reserved flag of a vehicle.
5
6          Contains attributes common to all vehicles: mpg, vin, and reserved
7          (Boolean). Provides polymorphic behavior for method getDescription.
8      """
9
10     def __init__(self, mpg, vin):
11         """Initializes a Vehicle object with mpg and vin."""
12
13         self.__mpg = mpg
14         self.__vin = vin
15         self.__reserved = False
16
17     def getType(self):
18         """Returns the type of vehicle (car, van, truck.)"""
19
20         return type(self).__name__
21
22     def getVin(self):
23         """Returns the vin of vehicle."""
24
25         return self.__vin
26
27     def getDescription(self):
28         """Returns general description of car, not specific to type."""
29
30         descript = 'mpg:' + format(self.__mpg, '>3') + ' ' + \
31                     'vin:' + format(self.__vin, '>12')
32
33         return descript
34
35     def isReserved(self):
36         """Returns True if vehicle is reserved, otherwise returns False."""
37
38         return self.__reserved
39
40     def setReserved(self, reserved):
41         """Sets reserved flag of vehicle to provided Boolean value."""
42
43         self.__reserved = reserved

```

**FIGURE 10-38** Vehicle Class

The `getType` methods returns the specific type of a `Vehicle` object (`Car`, `Van`, or `Truck`) by use of `type(self).__name__` which returns its own type.

We can execute the file and interactively and unit test the class in the Python shell.

```

>>> v = Vehicle('32', 'ABC123')      — create instance
>>> v.getType()                    — get object type
'Vehicle'
>>> v.getVin()                      — get vehicle identification number
'ABC123'

```

We test the `setReserved` and `isReserved` methods similarly.

An implementation of the `Car` subclass is given in Figure 10-39.

```

1  """This module provides a Car class, a subtype of the Vehicle class."""
2
3  from vehicle import Vehicle
4
5  class Car(Vehicle):
6      """This class is a subtype of the Vehicle class.
7
8          Contains additional attributes of make and model, num of passengers, and
9          num of doors. Supports polymorphic behavior of method getDescription.
10         """
11
12     def __init__(self, make_model, mpg, num_passengers, num_doors, vin):
13         """Initialized with provided parameter values."""
14
15         super().__init__(mpg, vin)
16
17         self.__make_model = make_model
18         self.__num_passengers = num_passengers
19         self.__num_doors = num_doors
20
21     def getDescription(self):
22         """Returns description of car as a formatted string."""
23
24         spacing = ' '
25         descript = format(self.__make_model, '<18') + spacing + \
26                     'passenger: ' + self.__num_passengers + spacing + \
27                     'doors: ' + format(self.__num_doors, '<2') + spacing + \
28                     Vehicle.getDescription(self)
29
30
31     return descript

```

**FIGURE 10-39** Car Subclass of the Vehicle Class

The `Car` class is defined as a subclass of the `Vehicle` class, imported on line 3. The methods inherited from the `Vehicle` class have already been tested. We therefore test method `getDescription` (lines 21–30) by executing the `Car` class file and performing the following.

```

>>> v = Car('Ford Fusion', '34', '5', '4', 'AB4FG5689GM')
>>> v.getDescription()
'Ford Fusion  passengers: 5  doors: 4  mpg: 34  vin: AB4FG5689GM'

```

These results are correct. Figure 10-40 and Figure 10-41 show similar implementations of the `Van` and `Truck` classes.

The `Van` and `Truck` classes differ from the `Car` class in the particular instance variables they contain, and the information returned by method `getDescription`. These differences support the polymorphic behavior of `Vehicle` types.

## Development and Testing of the Vehicles Class

Whereas the `Vehicle` class represents the information of a single vehicle, the `Vehicles` class maintains a complete collection of `Vehicle` types. Figure 10-42 illustrates an implementation of the `Vehicles` class, maintaining the cost, availability, and reservation status of all vehicles of the rental agency.

```

1  """This module provides a Van class, a subtype of the Vehicle class."""
2
3  from vehicle import Vehicle
4
5  class Van(Vehicle):
6
7      """This class is a subtype of the Vehicle class.
8
9          Contains additional attributes of make and mode, and num of passengers.
10         Supports polymorphic behavior of method getDescription.
11     """
12     def __init__(self, make_model, mpg, num_passengers, vin):
13         """ Initializes with make-model, mpg, num_passengers, vin."""
14
15         super().__init__(mpg, vin)
16
17         self.__make_model = make_model
18         self.__num_passengers = num_passengers
19
20     def getDescription(self):
21         """Returns complete description of van"""
22
23         spacing = ' '
24         descript = format(self.__make_model, '<22') + spacing + \
25             'passengers:' + format(self.__num_passengers, '>2') + \
26             spacing + Vehicle.getDescription(self)
27
28     return descript

```

**FIGURE 10-40** Van Subclass of the Vehicle Class

```

1  """This module provides a Truck class. a subtype of the Vehicle class."""
2
3  from vehicle import Vehicle
4
5  class Truck(Vehicle):
6
7      """This class is a subtype of the Vehicle class.
8
9          Contains additional attributes length and num of rooms storage capacity.
10         Supports polymorphic behavior of method getDescription.
11     """
12     def __init__(self, mpg, length, num_rooms, vin):
13         """ Initializes with mpg, length, num_rooms, vin."""
14
15         super().__init__(mpg, vin)
16
17         self.__length = length
18         self.__num_rooms = num_rooms
19
20     def getDescription(self):
21         """Returns complete description of truck."""
22
23         spacing = ' '
24         descript = 'length(feet):' + format(self.__length, '>3') + spacing + \
25             'rooms:' + format(self.__num_rooms, '>2') + spacing + \
26             Vehicle.getDescription(self)
27
28     return descript

```

**FIGURE 10-41** Truck Subclass of the Vehicle Class

```

1  """Vehicles class module. Raises InvalidFormatException and InvalidVinError."""
2
3  from vehicle import Vehicle
4  from car import Car
5  from van import Van
6  from truck import Truck
7
8  class InvalidVinError(Exception):
9      """Exception indicating that a provided vin was not found."""
10     pass
11
12
13 class Vehicles:
14     """This class maintains a collection of Vehicle objects."""
15
16     def __init__(self):
17         """Initializes empty list of vehicles."""
18
19         self.__vehicles = []
20
21     def getVehicle(self, vin):
22         """Returns Vehicle for provided vin. Raises InvalidVinError."""
23
24         for vehicle in self.__vehicles:
25             if vehicle.getVin() == vin:
26                 return vehicle
27
28         raise InvalidVinError
29
30     def addVehicle(self, vehicle):
31         """Adds new vehicle to list of vehicles."""
32
33         self.__vehicles.append(vehicle)
34
35     def numAvailVehicles(self, vehicle_type):
36         """Returns number of available vehicles of vehicle_type."""
37
38         return len(self.getAvailVehicles(vehicle_type))
39
40     def getAvailVehicles(self, vehicle_type):
41         """Returns a list of unreserved Vehicles objects of vehicle_type."""
42
43         return [veh for veh in self.__vehicles \
44                 if veh.getType() == vehicle_type and not veh.isReserved()]
45
46     def unreserveVehicle(self, vin):
47         """Sets reservation status of vehicle with vin to unreserved."""
48
49         k = 0
50         found = False
51
52         while not found:
53             if self.__vehicles[k].getVin() == vin:
54                 self.__vehicles[k].setReserved(False)
55                 found = True

```

**FIGURE 10-42** Vehicles Class

The Vehicles class imports classes Vehicle, Car, Van, and Truck (**lines 3–6**). An InvalidVinError exception class is defined (**lines 8–10**), raised when method getVehicle is called for a nonexistent VIN. Method addVehicle (**lines 30–33**) adds a new vehicle to the collection. This is called when the Vehicles object is initially populated at the start of the program (in the SystemInterface class). Method getAvailVehicles (**lines 40–44**) returns a list of vehicles that are not currently reserved.

We can easily test these classes from the Python shell as given below.

**First, the needed classes are imported:**

```
>>> from vehicle import Vehicle
>>> from car import Car
>>> from van import Van
>>> from truck import Truck
>>> from vehicles import Vehicles
```

**Then, an instance of each type Vehicle is created:**

```
>>> veh_1 = Car('Ford Fusion', '34', '5', '4', 'FG1000')
>>> veh_2 = Van('Dodge Caravan', '25', '7', 'TF1000')
>>> veh_3 = Truck('12', '10', '1', 'HG1000')
```

**A Vehicles instance is created, and each Vehicle instance is added to the collection:**

```
>>> v = Vehicles()
>>> v.addVehicle(veh_1)
>>> v.addVehicle(veh_2)
>>> v.addVehicle(veh_3)
```

**Finally, the description of each vehicle is obtained:**

```
>>> v.getAvailVehicles('Car')[0].getDescription()
'Ford Fusion           passengers: 5   doors: 4   mpg: 34   vin:   FG1000'
>>> v.getAvailVehicles('Van')[0].getDescription()
'Dodge Caravan         passengers: 7   mpg: 25   vin:       TF1000'
>>> v.getAvailVehicles('Truck')[0].getDescription()
'length(feet): 10   rooms: 1   mpg: 12   vin:       HG1000'
>>>
```

Method getAvailVehicles returns a list of available cars for the provided vehicle type. Since this returns a list of vehicles, the first vehicle in the list is selected (at index [0]), and the getDescription method is called to display the full description of the vehicle. We see that we get the correct results.

### Implementation of the VehicleCost and VehicleCosts Classes

The VehicleCosts class is responsible for maintaining and providing the costs for each vehicle type. First, the VehicleCost class is given in Figure 10-43.

```
1  """This module provides a VehicleCost class."""
2
3  class VehicleCost:
4      """This class provides the methods for maintaining rental costs."""
5
6      def __init__(self, daily_rate, weekly_rate, weekend_rate,
7                   free_miles, per_mile_chrg, insur_rate):
8          """Initialzes rental rates, num free miles /mileage chrg/insur rate."""
9
10         self.__daily_rate = daily_rate
11         self.__weekly_rate = weekly_rate
12         self.__weekend_rate = weekend_rate
13         self.__free_miles = free_miles
14         self.__per_mile_chrg = per_mile_chrg
15         self.__insur_rate = insur_rate
16
17     def getDailyRate(self):
18         """Returns daily rental rate of vehicle."""
19
20         return float(self.__daily_rate)
21
22     def getWeeklyRate(self):
23         """Returns weekly rental rate of vehicle."""
24
25         return float(self.__weekly_rate)
26
27     def getWeekendRate(self):
28         """Returns weekend rental rate of vehicle."""
29
30         return float(self.__weekend_rate)
31
32     def getFreeMiles(self):
33         """Returns number of free miles for vehicle rental."""
34
35         return int(self.__free_miles)
36
37     def getPerMileCharge(self):
38         """Returns per mile charge for vehicle rental."""
39
40         return float(self.__per_mile_chrg)
41
42     def getInsuranceRate(self):
43         """Returns daily insurance rate for vehicle."""
44
45         return float(self.__insur_rate)
46
47     def getCosts(self):
48         """Returns a list containing all costs for vehicle."""
49
50         return [self.__daily_rate, self.__weekly_rate,
51                 self.__weekend_rate, self.__free_miles,
52                 self.__per_mile_chrg, self.__insur_rate]
```

**FIGURE 10-43** VehicleCost Class

The `VehicleCost` class, like the `Vehicle` class, stores information accessed by the provided getter methods. The information to be stored is passed to the `__init__` method (**lines 6–15**). The `getCosts` method (**lines 47–52**) returns all the individual cost components as a list. This method is called when the vehicle costs need to be displayed, or the cost of a particular vehicle rental needs to be determined.

We can easily test the `VehicleCost` class from the Python shell as given below.

**First, the `VehicleCost` class is imported:**

```
>>> from vehicleCost import VehicleCost
```

**Then, an instance of each type `VehicleCost` is created:**

```
>>> vc = VehicleCost('24.99', '180.00', '45.00', '100', '.15', '14.99')
```

**Finally, each getter method is tested:**

```
>>> vc.getDailyRate()
24.99
>>> vc.getWeeklyRate()
180.0
>>> vc.getWeekendRate()
45.0
>>> vc.getFreeMiles()
100
>>> vc.getPerMileCharge()
0.15
>>> vc.getInsuranceRate()
14.99
>>> vc.getCosts()
['24.99', '180.00', '45.00', '100', '.15', '14.99']
>>>
```

We see that we get the correct results.

In Figure 10-44, we give the `VehicleCosts` class that maintains the collection of `VehicleCost` objects.

The `vehicleCosts` class, as with the `Vehicles` class, maintains a collection of `VehicleCost` objects. The `VehicleCost` class is imported on **line 3**. On **lines 6–8** three symbolic constants are defined, `DAILY_RENTAL`, `WEEKLY_RENTAL`, and `WEEKEND_RENTAL`. The use of these constants (equal to integer values 1, 2, and 3, respectively) make the program more readable than if the corresponding integer values were used.

The `VehicleCosts` class is defined on **lines 10–89**. The vehicle costs for the three types of vehicles (cars, vans, and trucks) are stored in a dictionary using the vehicle types as key values. The `__init__` method (**lines 13–16**), therefore, initializes instance variable `vehicle_costs` to an empty dictionary. The vehicle costs are individually added to the collection of costs by method `addVehicleCost` (**lines 23–26**), called from the `SystemInterface` class when populating the rental costs from file. Method `getVehicleCost` (**lines 18–21**) returns the costs as a list of individual costs. Finally, method `calcRentalCost` (**lines 28–89**) calculates the cost of a particular rental. We test the `VehicleCosts` class from the Python shell.

```

1 """This module provides a VehicleCosts class"""
2
3 from vehicleCost import VehicleCost
4
5 # symbol constants
6 DAILY_RENTAL = 1
7 WEEKLY_RENTAL = 2
8 WEEKEND_RENTAL = 3
9
10 class VehicleCosts:
11     """This class provides the methods for maintaining rental costs."""
12
13     def __init__(self):
14         """Initializes vehicle costs to empty."""
15
16         self.__vehicle_costs = dict()
17
18     def getVehicleCost(self, vehicle_type):
19         """Returns VehicleCost object for the specified vehicle type."""
20
21         return self.__vehicle_costs[vehicle_type]
22
23     def addVehicleCost(self, veh_type, veh_cost):
24         """Adds a vehicle cost object to dictionary with keyword veh_type."""
25
26         self.__vehicle_costs[veh_type] = veh_cost
27
28     def calcRentalCost(self, vehicle_type, rental_period,
29                         want_insurance, miles_driving):
30         """Returns estimate of rental cost for provided parameter values.
31
32             Returns dictionary with key values: {'base_charges', 'insur_rate',
33             'num_free_miles', 'per_mile_charge', 'estimated_mileage_charges'}
34         """
35
36         # get vehicle cost
37         vehicle_cost = self.getVehicleCost(vehicle_type)
38
39         # calc rental charges
40         rental_time = rental_period[1]
41
42         if rental_period[0] == DAILY_RENTAL:
43             rental_rate = vehicle_cost.getDailyRate()
44             rental_period_value = DAILY_RENTAL
45             rental_period_str = 'daily rental'
46             rental_days = rental_time
47         elif rental_period[0] == WEEKLY_RENTAL:
48             rental_rate = vehicle_cost.getWeeklyRate()
49             rental_period_value = WEEKLY_RENTAL
50             rental_period_str = 'weekly rental'
51             rental_days = rental_time * 7
52         elif rental_period[0] == WEEKEND_RENTAL:
53             rental_rate = vehicle_cost.getWeekendRate()
54             rental_period_value = WEEKEND_RENTAL
55             rental_period_str = 'weekend rental'
56             rental_days = 2
57

```

**FIGURE 10-44** VehicleCosts Class (*Continued*)

```

58         elif rental_period[0] == WEEKEND_RENTAL:
59
60             rental_rate = vehicle_cost.getWeekendRate()
61             rental_period_value = WEEKEND_RENTAL
62             rental_period_str = 'weekend rental'
63             rental_days = 2
64
65             # get free miles, per mile charge and insurance rate
66             num_free_miles = vehicle_cost.getFreeMiles()
67             per_mile_charge = vehicle_cost.getPerMileCharge()
68             insurance_rate = vehicle_cost.getInsuranceRate()
69
70             # calc rental charge for selected rental period
71             if not want_insurance:
72                 insurance_rate = 0
73
74             # calc base rental charges
75             base_rental_charges = rental_days * rental_rate + \
76                                 rental_days * insurance_rate
77
78             miles_charged = miles_driving - num_free_miles
79
80             if miles_charged < 0:
81                 miles_charged = 0
82
83             estimated_mileage_charges = miles_charged * per_mile_charge
84
85             return {'base_charges' : base_rental_charges,
86                     'insur_rate' : insurance_rate,
87                     'num_free_miles' : num_free_miles,
88                     'per_mile_charge' : per_mile_charge,
89                     'estimated_mileage_charges' : estimated_mileage_charges}

```

**FIGURE 10-44** VehicleCosts Class**First, the VehicleCosts class is imported:**

```
>>> from vehicleCosts import VehicleCosts
```

**Then, instances of type VehicleCost are created:**

```
>>> vc_1 = VehicleCost('24.99', '180.00', '45.00', '100', '.15', '14.99')
>>> vc_2 = VehicleCost('35.00', '220.00', '55.00', '0', '.20', '14.99')
>>> vc_3 = VehicleCost('55.00', '425.00', '110.00', '25', '.25', '24.99')
```

**Then a VehicleCosts instance is created, and each VehicleCost instance is added to the collection:**

```
>>> vc.addVehicleCost('Car', vc_1)
>>> vc.addVehicleCost('Van', vc_2)
>>> vc.addVehicleCost('Truck', vc_3)
```

**Then, each getter method is tested:**

```
>>> vc.getVehicleCost('Car').getCosts()
['24.99', '180.00', '45.00', '100', '.15', '14.99']
>>> vc.getVehicleCost('Van').getCosts()
['35.00', '220.00', '55.00', '0', '.20', '14.99']
>>> vc.getVehicleCost('Truck').getCosts()
['55.00', '425.00', '110.00', '25', '.25', '24.99']
```

Finally, method `calcRentalCost` is tested:

```
>>> vc.calcRentalCost('Car', (1, 3), False, 150)
{'insur_rate': 0, 'base_charges': 74.97, 'num_free_miles': 100,
'per_mile_charge': 0.15, 'estimated_mileage_charges': 7.5}
```

In the call to `calcRentalCost`, recall that the first parameter is the vehicle type, the second is the rental period (1 for DAILY\_RENTAL) and the number of days (3), the third indicates if the insurance is desired, and the fourth is the expected number of miles to be driven.

We see that we get the correct results. We next give the implementation of the `Reservation` and `Reservations` classes.

## Implementation of the Reservation and Reservations Classes

The remaining aggregating class in the program is the `Reservations` class. We first give the `Reservation` class in Figure 10-45.

The `Reservation` class maintains the information for a given reservation. The `__init__` method (lines 6–14) is provided a name, address, credit card number, and VIN when a new `Reservation` object is created, with a getter methods provided for each of these four values. We omit the testing of this class and next give the implementation of the corresponding `Reservations` class in Figure 10-46.

```
1  """This module contains a Reservation class for storing details of a rental"""
2
3  class Reservation:
4      """This class provides the methods for maintaining reservations info."""
5
6      def __init__(self, name, address, credit_card, vin):
7          """Initializes a Reservation object with name, address, credit_card
8              and vin.
9              """
10
11         self.__name = name
12         self.__address = address
13         self.__credit_card = credit_card
14         self.__vin = vin
15
16     def getName(self):
17         """Returns first and last name for reservation."""
18
19         return self.__name
20
21     def getAddress(self):
22         """Returns address for reservation."""
23
24         return self.__address
25
26     def getCreditCard(self):
27         """Returns credit card on reservation."""
28
29         return self.__credit_card
30
31     def getVin(self):
32         """Returns vehicle identification number on reservation."""
33
34         return self.__vin
```

**FIGURE 10-45** Reservation Class

```

1  """This module provides a Reservations class."""
2
3  from reservation import Reservation
4
5  class Reservations:
6      """This class provides the methods for maintaining rental reservations."""
7
8      def __init__(self):
9          """Initializes empty collection of reservations."""
10
11         self.__reservations = dict()
12
13     def isReserved(self, vin):
14         """Returns True if reservation for vin, else returns False."""
15
16         return vin in self.__reservations
17
18     def getVinForReserv(self, credit_card):
19         """Returns vin of vehicles reserved with credit_card."""
20
21         return self.__reservations[credit_card].getVin()
22
23     def addReservation(self, resv):
24         """Adds new reservation."""
25
26         self.__reservations[resv.getCreditCard()] = resv
27
28     def findReservation(self, credit_card):
29         """Returns True if reservation for credit_card, else returns False."""
30
31         return credit_card in self.__reservations
32
33     def cancelReservation(self, credit_card):
34         """Deletes reservation matching provided credit card number."""
35
36         del(self.__reservations[credit_card])

```

**FIGURE 10-46** Reservations Class

The `Reservation` class is imported on **line 3**. The `__init__` method initializes an empty dictionary for storing the reservations, with credit card numbers serving as the key values. The remaining methods of the class include method `isReserved` (**lines 13–16**), `getVinForReserv` (**lines 18–21**), `addReservation` (**lines 23–26**), `findReservation` (**lines 28–31**), and `CancelReservation` (**lines 33–36**). We omit the testing of the `Reservations` class. We finally look at the implementation of the `SystemInterface` and `RentalAgencyUI` classes.

### **Implementation of the SystemInterface and RentalAgencyUI Classes**

An implementation of the `SystemInterface` class is given in Figure 10-47.

The docstring for the module specifies the formatting of the files storing the vehicle and vehicle rental cost information. There is one exception raised by the module to inform the user interface when a file error has occurred. This could be due to either a file not being found, or an improperly formatted file.

The import statements on **lines 28–30** import most of the classes in the system. Symbolic constants `VEHICLE_TYPES`, `VEHICLE_FILENAME`, and `VEHICLE_COSTS_FILENAME` are defined (**lines 33–35**). Defining these constants makes the program more readable and modifiable.

Exception class `InvalidFileFormatError` is defined (**lines 39–49**) and used within the `SystemInterface` class. Having this exception type allows typical I/O errors (such as a file not found error) to be reported as errors specific to the file formatting requirements of the program. The exception class defines special method `__str__` to enable information to be displayed specific to the error (that is, which file header was expected and not found, in which file).

The `__init__` method (**lines 55–77**) first creates instances of the three aggregator classes of the system—`Vehicles`, `VehicleCosts`, and `Reservations`. Each is initially empty when created. The rest of the `__init__` method consists of a try block that attempts to open and read the files defined by `VEHICLES_FILENAME` and `VEHICLE_COSTS_FILENAME`. If opened successfully,

```

1  """This module provides a SystemInterface class for the Vehicle Rental Program.
2
3  Vehicles File Format
4  -----
5  The format for the vehicles file contains comma-separated values with the
6  indicated header lines for cars, vans, and trucks.
7
8      #CARS#
9      make-model, mpg, num-passengers, vin
10     .
11      #VANS#
12      make-model, mpg, num passengers, vin
13     .
14      #TRUCKS#
15      mpg, length, num rooms, vin
16     .
17
18  Vehicle Cost File Format
19  -----
20
21  The format for the rental costs file includes a header line, followed
22  by three lines of comma-separated values for cars, vans and trucks.
23
24      daily, weekly, weekend, free miles, mileage charge, insurance
25
26  The following exceptions are raised: IOError.
27 """
28 from vehicles import Vehicles, Car, Van, Truck
29 from vehicleCosts import VehicleCost, VehicleCosts
30 from reservations import Reservations
31
32 # symbolic constants
33 VEHICLE_TYPES = ('Car', 'Van', 'Truck')
34 VEHICLES_FILENAME = 'VehiclesStock.txt'
35 VEHICLE_COSTS_FILENAME = 'RentalCost.txt'
36
37 # exception class
38
39 class InvalidFileFormatError(Exception):
40     """Exception indicating invalid file in file_name."""
41
42     def __init__(self, header, file_name):
43         self.__header = header
44         self.__file_name = file_name
45
46     def __str__(self):
47
48         return 'FILE FORMAT ERROR: File header ' + self.__header + \
49             ' expected in file ' + self.__file_name
50

```

**FIGURE 10-47** SystemInterface Class (*Continued*)

```
1 class SystemInterface:
2     """This class provides the system interface of the vehicle rental
3         system.
4     """
5     def __init__(self):
6         """Populates vehicles and rental costs from file. Raises IOError."""
7
8         self.__vehicles = Vehicles()
9         self.__vehicle_costs = VehicleCosts()
10        self.__reservations = Reservations()
11        self.__vehicle_info_file = None
12
13    try:
14        self.__vehicle_info_file = open(VEHICLES_FILENAME, 'r')
15        self.__rental_cost_file = open(VEHICLE_COSTS_FILENAME, 'r')
16
17        self.__populateVehicles(self.__vehicle_info_file)
18        self.__populateCosts(self.__rental_cost_file)
19    except InvalidFileFormatError as e:
20        print(e)
21        raise IOError
22    except IOError:
23        if self.__vehicle_info_file == None:
24            print('FILE NOT FOUND:', VEHICLES_FILENAME)
25        else:
26            print('FILE NOT FOUND:', VEHICLE_COSTS_FILENAME)
27        raise IOError
28
29    def numAvailVehicles(self, vehicle_type):
30        """Returns the number of available vehicles. Returns 0 if no
31            no vehicles available.
32        """
33
34        return self.__vehicles.numAvailVehicles(vehicle_type)
35
36    def getVehicle(self, vin):
37        """Returns Vehicle type for given vin."""
38
39        return self.__vehicles.getVehicle(vin)
40
41    def getVehicleTypes(self):
42        """Returns all vehicle types as a tuple of strings."""
43
44        return VEHICLE_TYPES
45
46    def getVehicleCosts(self, vehicle_type):
47        """Returns vehicle costs for provided vehicle type as a list.
48
49            List of form [daily rate, weekly rate, weekend rate,
50                          num free miles, per mile charge, insur rate]
51        """
52
53        return self.__vehicle_costs.getVehicleCost(vehicle_type).getCosts()
```

**FIGURE 10-47** SystemInterface Class (*Continued*)

then each is read to populate the corresponding object. The exception handler catches two types of exceptions, `InvalidFormatError` and `IOERROR`, as mentioned.

A set of getter methods is provided ([lines 86–103](#)) for retrieving vehicle, vehicle type, and vehicle cost information. Method `getAvailVehicles` returns the list of vehicles that are not currently reserved. (Note that the list returned is constructed by the use of a list comprehension.) The list of unreserved vehicles is of a specified type. Four additional methods are provided ([lines 113–132](#)) for maintaining reservation information (`isReserved`, `findReservation`,

```

107     def getAvailVehicles(self, vehicle_type):
108         """Returns a list of descriptions of unreserved vehicles."""
109
110         avail_vehicles = self.__vehicles.getAvailVehicles(vehicle_type)
111         return [veh for veh in avail_vehicles]
112
113     def isReserved(self, vin):
114
115         return self.__reservations.isReserved(vin)
116
117     def findReservation(self, credit_card):
118
119         return self.__reservations.findReservation(credit_card)
120
121     def addReservation(self, resv):
122         """Creates reservation and marks vehicles as reserved."""
123
124         self.__reservations.addReservation(resv)
125
126     def cancelReservation(self, credit_card):
127         """Cancels reservation made with provided credit card."""
128
129         vin = self.__reservations.getVinForReserv(credit_card)
130
131         self.__vehicles.unreserveVehicle(vin)
132         self.__reservations.cancelReservation(credit_card)
133
134     def calcRentalCost(self, vehicle_type, rental_period,
135                         want_insurance, miles_driving):
136         """Returns estimate of rental cost for provided parameter values.
137
138             Returns dictionary with key values: {'base_charges', 'insur_rate',
139             'num_free_miles', 'per_mile_charge', 'estimated_mileage_charges'}
140         """
141
142         return self.__vehicle_costs.calcRentalCost(vehicle_type, rental_period,
143                                         want_insurance, miles_driving)
144
145     # ----- Private Methods
146
147     def __populateVehicles(self, vehicle_file):
148         """Gets vehicles from vehicle_file. Raises InvalidFormatError."""
149
150         empty_str = ''
151
152         # init vehicle string file headers
153         vehicle_file_headers = ('#CARS#', '#VANS#', '#TRUCKS#')
154         vehicle_type_index = 0
155
156         # read first line of file (#CARS# expected)
157         vehicle_str = vehicle_file.readline()
158         vehicle_info = vehicle_str.rstrip().split(',')
159         file_header_found = vehicle_info[0]
160         expected_header = vehicle_file_headers[0]
161

```

**FIGURE 10-47** SystemInterface Class (*Continued*)

`addReservation`, and `cancelReservation`). Finally, method `calcRentalCost` (**lines 134–143**) returns the calculated rental charges for a given vehicle type, rental period, insurance option, and expected miles driven.

The remainder of the class consists of private supporting methods. Private method `populateVehicles` reads the information given by `VEHICLES_FILENAME` and populates the `Vehicles` instance. Private method `populateCosts` reads the information given by `VEHICLE_COSTS_FILENAME` and populates the `VehicleCosts` instance.

```

164     if file_header_found != expected_header:
165         raise InvalidFileFormatError(expected_header, VEHICLES_FILENAME)
166     else:
167         # read next line of file after #CARS# header line
168         vehicle_str = vehicle_file.readline()
169
170     while vehicle_str != empty_str:
171
172         # convert comma-separated string into list of strings
173         vehicle_info = vehicle_str.rstrip().split(',')
174
175         if vehicle_info[0][0] == '#':
176             vehicle_type_index = vehicle_type_index + 1
177             file_header_found = vehicle_info[0]
178             expected_header = vehicle_headers[vehicle_type_index]
179
180             if file_header_found != expected_header:
181                 raise InvalidFileFormatError(expected_header,
182                                              VEHICLES_FILENAME)
183             else:
184
185                 # create new vehicle object of the proper type
186                 if file_header_found == '#CARS#':
187                     vehicle = Car(*vehicle_info)
188                 elif file_header_found == '#VANS#':
189                     vehicle = Van(*vehicle_info)
190                 elif file_header_found == '#TRUCKS#':
191                     vehicle = Truck(*vehicle_info)
192
193                 # add new vehicle to vehicles list
194                 self.__vehicles.addVehicle(vehicle)
195
196             # read next line of vehicle information
197             vehicle_str = vehicle_file.readline()
198
199     def __populateCosts(self, cost_file):
200         """Populates RentalCost objects from provided file object."""
201
202         # skip file header / read first line of file
203         cost_file.readline()
204         cost_str = cost_file.readline()
205
206         for veh_type in VEHICLE_TYPES:
207             # strip off newline (last) character and split into list
208             cost_info = cost_str.rstrip().split(',')
209
210             for cost_item in cost_info:
211                 cost_item = cost_item.strip()
212
213             # add Vehicle Type/Rental Cost key/value to dictionary
214             self.__vehicle_costs.addVehicleCost(veh_type,
215                                                 VehicleCost(*cost_info))
216
217             # read next line of vehicle costs
218             cost_str = cost_file.readline()

```

**FIGURE 10-47** SystemInterface Class

We next look at the implementation of the final class, the `RentalAgencyUI`, given in Figure 10-48.

Any user interface for the rental agency system must access it through the system interface. Thus, the `__init__` method (**lines 14–16**) of the `RentalAgencyUI` class is passed a reference to the system interface to store (in a private variable) providing it access to the system. The only

```

1 """
2     This module provides class RentalAgencyUI, a console user interface.
3
4     Method start begins execution of the interface. Raises IOError exception.
5 """
6 from systemInterface import VEHICLE_TYPES
7 from vehicles import InvalidVinError
8 from vehicleCosts import DAILY_RENTAL, WEEKLY_RENTAL, WEEKEND_RENTAL
9 from reservations import Reservation
10
11 class RentalAgencyUI:
12     """This class provides a console interface for the rental agency system."""
13
14     def __init__(self, sys):
15         """Stores the provided reference to the vehicle rental system."""
16         self.__sys = sys
17
18     def start(self):
19         """Begins the command loop."""
20
21         self.__displayWelcomeScreen()
22         self.__displayMenu()
23
24         selection = self.__getSelection(7)
25
26         while selection != 7:
27             self.__executeCmd(selection)
28             self.__displayMenu()
29             selection = self.__getSelection(7)
30
31         print('Thank you for using the Friendly Rental Agency')
32
33
34     # Private Methods
35
36     def __displayWelcomeScreen(self):
37         """PRIVATE: Displays welcome message and general instructions."""
38
39         print('*' * 40)
40         print(' * Welcome to the Friendly Vehicle Rental Agency *')
41         print('*' * 40)
42
43     def __displayMenu(self):
44         """PRIVATE: Displays a list of menu options on the screen."""
45
46         print('\n<< MAIN MENU >>')
47         print('1 - Display vehicle types')
48         print('2 - Check rental costs')
49         print('3 - Check available vehicles')
50         print('4 - Get cost of specific rental')
51         print('5 - Make a reservation')
52         print('6 - Cancel a reservation')
53         print('7 - Quit\n')
54

```

**FIGURE 10-48** RentalAgencyUI Class (*Continued*)

other public method of the class is method `start` (lines 18–31). This method begins the command loop when called—that is, the repeated action of displaying the main menu, getting the user's selection, and executing the selected command. The loop continues until a value of 7 (to quit the program) is entered. The rest of the methods of the class are private methods in support of the execution of commands in the `start` method.

```

55     def __getSelection(self, num_selections, prompt='Enter: '):
56         """PRIVATE: Returns user-entered value in range 1-num_selections."""
57
58         valid_input = False
59
60         selection = input(prompt)
61
62         while not valid_input:
63             try:
64                 selection = int(selection)
65
66                 if selection < 1 or selection > num_selections:
67                     print('* Invalid Entry*\n')
68                     selection = input(prompt)
69                 else:
70                     valid_input = True
71             except ValueError:
72                 selection = input(prompt)
73
74         return selection
75
76     def __displayDivLine(self, title = ""):
77         """PRIVATE: Displays line of dashes, with optional title."""
78
79         if len(title) != 0:
80             title = ' ' + title + ' '
81         print(title.center(70,'-'))
82
83     def __executeCmd(self, selection):
84         """PRIVATE: Executes command for provided menu selection."""
85
86         if selection == 1:
87             self.__CMD_DisplayVehicleTypes()
88         elif selection == 2:
89             self.__CMD_DisplayVehicleCosts()
90         elif selection == 3:
91             self.__CMD_PromptAndDisplayAvailVehicles()
92         elif selection == 4:
93             self.__CMD_DisplaySpecificRentalCost()
94         elif selection == 5:
95             self.__CMD_MakeReservation()
96         elif selection == 6:
97             self.__CMD_CancelReservation()
98
99     def __CMD_DisplayVehicleTypes(self):
100        """PRIVATE: Displays vehicle types (Cars, Vans, Trucks)."""
101
102        self.__displayDivLine('Types of Vehicles Available for Rent')
103        self.__displayVehicleTypes()
104        self.__displayDivLine()
105

```

**FIGURE 10-48** RentalAgencyUI Class (*Continued*)

The methods provided by the system interface correspond to the user options given in the main menu in the user interface (as shown in Figure 10-49).

Private method `displayWelcomeScreen` (lines 36–41) provides the welcome message of the program. Private method `displayMenu` (lines 43–53) is repeatedly called to redisplay the users' options before each next command. Private method `getSelection` is called whenever the user is to enter a number within a provided range, such as in the main menu. Such selection occurs in other places in the program as well. Therefore, the method is designed to be passed an argument indicating the upper limit of the range of selections. (For example, for the selection of

```

106 def __CMD_DisplayVehicleCosts(self):
107     """PRIVATE: Displays rental costs for Cars, Vans, Trucks."""
108
109     empty_str = ''
110     blank_char = ' '
111
112     # get vehicle type from user
113     self.__displayVehicleTypes()
114     vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
115     vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
116
117     # set column headings
118     row1_colheadings = blank_char * 24 + format('Free', '7') + \
119                         format('Per Mile', '10') + format('Insurance', '17')
120
121     row2_colheadings = format('Daily', '7') + format('Weekly', '8') + \
122                         format('Weekend', '9') + format('Miles', '7') + \
123                         format('Charge', '10') + format('(per day)', '17')
124
125     # get vehicle costs
126     costs = self.__sys.getVehicleCosts(vehicle_type)
127
128     # build costs line to display
129     costs_str = empty_str
130
131     field_widths = ('7', '8', '9', '7', '10', '17')
132     for k in range(0, len(costs)):
133         costs_str = costs_str + format(costs[k], '<' + field_widths[k])
134
135     # display headings and costs line
136     self.__displayDivLine('Rental Charges for ' + vehicle_type + 's')
137     print()
138     print(row1_colheadings + '\n' + row2_colheadings + '\n' + costs_str)
139     self.__displayDivLine()
140
141 def __CMD_DisplayAvailVehicles(self, vehicle_type, numbered=False):
142     """PRIVATE: Displays unreserved vehicles of selected type.
143         When default argument is True, lists vehicles with sequential
144         numbers at the start of the line of each vehicle description.
145     """
146
147     avail_vehicles_list = self.__sys.getAvailVehicles(vehicle_type)
148     self.__displayDivLine('Available ' + vehicle_type + 's')
149
150     if avail_vehicles_list == []:
151         print('* No Vehicles Available of this Type *')
152     elif numbered:
153         k = 1
154         for veh in avail_vehicles_list:
155             print(str(k) + '-' + veh.getDescription())
156             k = k + 1
157     else:
158         for veh in avail_vehicles_list:
159             print(veh.getDescription())
160

```

**FIGURE 10-48** RentalAgencyUI Class (*Continued*)

items from the main menu, `getSelection` would be passed the value 7.) Two types of errors are checked by the method. One is if a value is outside of the allowable range. The other checks for invalid type of input (such as entering a letter instead of a number). The method requires that an appropriate value be entered by the user before returning.

```

161     def __CMD_PromptAndDisplayAvailVehicles(self):
162         """PRIVATE: Prompts user for vehicle type, and displays all
163             available vehicles of that type.
164         """
165
166         self.__displayVehicleTypes()
167         vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
168         vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
169         self.__CMD_DisplayAvailVehicles(vehicle_type)
170
171     def __CMD_DisplaySpecificRentalCost(self):
172         """PRIVATE: Prompts user for selections and displays rental cost."""
173
174         # get vehicle type and rental period from user
175         self.__displayVehicleTypes()
176         vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
177         vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
178
179         # assign tuple, e.g. ('DAILY_RENTAL', '4')
180         rental_period = self.__getRentalPeriod()
181
182         # prompt user for optional insurance
183         want_insurance = input('Would you like the insurance? (y/n): ')
184
185         while want_insurance not in ('y', 'Y', 'n', 'N'):
186             want_insurance = input('Would you like the insurance? (y/n): ')
187
188         # convert to Boolean value
189         want_insurance = want_insurance in ('y', 'Y')
190
191         # prompt user for num of miles expected to drive
192         print('\nNumber of miles expect to drive?')
193         num_miles = self.__getSelection(1000, prompt = 'Miles: ')
194
195         # calc base rental cost
196         rental_cost = self.__sys.calcRentalCost(vehicle_type, rental_period,
197                                                 want_insurance, num_miles)
198
199         # display estimated rental cost
200         print()
201         self.__displayDivLine('ESTIMATED ' + vehicle_type + ' RENTAL COST')
202
203         if want_insurance:
204             print('Insurance rate of', rental_cost['insur_rate'], 'per day')
205         else:
206             print('* You have opted out of insurance coverage *')
207
208         if rental_period[0] == DAILY_RENTAL:
209             print('\nDaily rental for', rental_period[1],
210                  'days would be $', format(rental_cost['base_charges'], '.2f'))
211         elif rental_period[0] == WEEKLY_RENTAL:
212             print('\nWeekly rental for', rental_period[1],
213                  'weeks would be $', format(rental_cost['base_charges'], '.2f'))
214         elif rental_period[0] == WEEKEND_RENTAL:
215             print('\nWeekend rental is $', rental_cost['base_charges'])
216
216         est_total_cost = rental_cost['base_charges'] + \
217                         rental_cost['estimated_mileage_charges']
218
219         print('\nYour cost with an estimated mileage of',
220              num_miles, 'miles would be', format(est_total_cost, '.2f'))
221
222         print('which includes', rental_cost['num_free_miles'],
223              'free miles and a charge of',
224              format(rental_cost['per_mile_charge'], '.2f'), 'per mile')
225
226         self.__displayDivLine()

```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

```

227     def __CMD_MakeReservation(self):
228         """PRIVATE: Prompts user for vehicle vin and reserves vehicle."""
229
230         # get vehicle type
231         self.__displayVehicleTypes()
232         vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
233         vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
234
235         if self.__sys.numAvailVehicles(vehicle_type) == 0:
236             print('Sorry - No available', \
237                  VEHICLE_TYPES[vehicle_type_num - 1] + 's', 'at the moment')
238         else:
239             self.__CMD_DisplayAvailVehicles(vehicle_type, numbered=True)
240             avail_vehicles = self.__sys.getAvailVehicles(vehicle_type)
241
242             valid_input = False
243
244             while not valid_input:
245                 selected = input('\nEnter number of vehicle to reserve: ')
246
247                 if not selected.isdigit():
248                     print('Please enter the number preceding the vehicle')
249                 elif int(selected) < 1 or \
250                     int(selected) > self.__sys.numAvailVehicles(vehicle_type):
251                     print('\nINVALID SELECTION - Please re-enter: ')
252                 else:
253                     valid_input = True
254
255             vin = avail_vehicles[int(selected) - 1].getVin()
256             vehicle = self.__sys.getVehicle(vin)
257             print(vehicle.getDescription())
258
259             vehicle.setReserved(True)
260
261             name = input('\nEnter first and last name: ')
262             addr = input('Enter address: ')
263             credit_card = input('Enter credit card number: ')
264
265             reserv = Reservation(name, addr, credit_card, vin)
266             self.__sys.addReservation(reserv)
267             print('* Reservation Made *')
268
269     def __CMD_CancelReservation(self):
270         """PRIVATE: Prompts user for credit card and cancels reservation."""
271
272         credit_card = input('Please enter your credit card number: ')
273
274         if self.__sys.findReservation(credit_card):
275             print('Calling cancelReservation of sys')  # ****
276             self.__sys.cancelReservation(credit_card)
277             print('** RESERVATION CANCELLED **')
278         else:
279             print('* Reservation not Found - Invalid Credit Card Entered *')
280

```

**FIGURE 10-48** RentalAgencyUI Class (*Continued*)

Method `displayDivLine` is called to display a row of dashes. It has a default parameter `title`. Thus, the method may be called with or without a supplied argument. If an argument is not supplied, then a complete row of dashes is displayed. If a title is displayed, it is centered within the displayed line of dashes as given below:

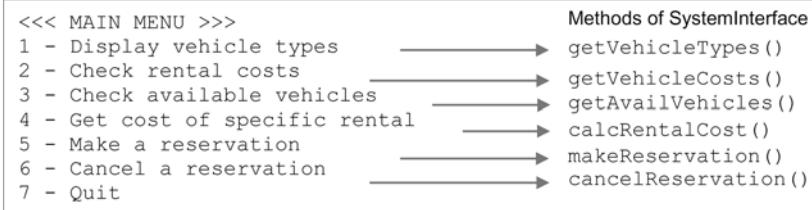
-----Types of Vehicles Available for Rent-----

```

281     def __displayVehicleTypes(self):
282         """PRIVATE: Displays the numbered selection of vehicle types."""
283
284         print('\nEnter type of vehicle')
285         vehicle_types = self.__sys.getVehicleTypes()
286
287         k = 1
288
289         for veh_type in vehicle_types:
290             print(k, '-', veh_type)
291             k = k + 1
292
293         print()
294
295     def __getRentalPeriod(self):
296         """PRIVATE: Prompts user for desired rental period.
297
298             Returns tuple of the form (x, n), where x is one of DAILY_RENTAL,
299             WEEKLY_RENTAL or WEEKEND_RENTAL, and n is the number of those units,
300             e.g. (DAILY_RENTAL, 3) three days of a daily rental.
301         """
302
303         print('\nEnter the rental period:')
304         print(DAILY_RENTAL, '- Daily ', WEEKLY_RENTAL, '- Weekly ',
305               WEEKEND_RENTAL, '- Weekend\n')
306
307         valid_input = False
308
309         while not valid_input:
310             try:
311                 selection = int(input('Enter: '))
312
313                 while selection not in (DAILY_RENTAL, WEEKLY_RENTAL,
314                                         WEEKEND_RENTAL):
315
316                     print('* Incorrect entry. Please Re-enter *\n')
317                     selection = int(input('Enter: '))
318
319                 if selection == DAILY_RENTAL:
320
321                     print('How many days do you need the vehicle?', end = ' ')
322                     num_days = self.__getSelection(14, prompt = ' ')
323
324                     if num_days > 6:
325                         print("Why don't you consider a weekly rental?\n")
326
327                     return (DAILY_RENTAL, num_days)
328
329                 elif selection == WEEKLY_RENTAL:
330                     print('How many weeks do you need the vehicle? (1-3): ')
331                     num_weeks = self.__getSelection(3, prompt = 'Weeks: ')
332
333                     return (selection, num_weeks)
334
335                 elif selection == WEEKEND_RENTAL:
336                     return (selection, 1)
337
338             except ValueError:
339                 print('* Invalid Input - Number Expected *\n')

```

**FIGURE 10-48** RentalAgencyUI Class



**FIGURE 10-49** Correspondence of Menu Selections and Methods of System Interface

Method `executeCmd` (**lines 83–97**) is called to execute each command of the main menu by selection number. Rather than place the code for each command within this method, a specific command method is called for each command. Private method `CMD_DisplayVehicleTypes` (**lines 99–104**) displays the three types of vehicles (Cars, Vans, and Trucks). It in turn makes use of private method `displayVehicleTypes` (**lines 281–293**). Within this method, `getVehicleTypes` of the `SystemInterface` is called to retrieve the types, since that is where they are defined.

Private method `CMD_DisplayVehicleCosts` (**lines 106–139**) displays the rental costs for a specific vehicle type:

```
Enter vehicle type: 1
-----
----- Following are the Rental Charges for Cars -----
      Free    Per Mile   Daily
Daily  Weekly  Weekend Miles Charge   Insurance
24.99 180.00  45.00    100   .15     14.99
-----
```

It displays a selection of vehicle types (**line 113**) for the user to choose from, constructs two rows of column headings (**lines 118–123**), and gets the vehicle costs for the selected vehicle type by call to method `getVehicleCosts` of the system interface (**line 126**). The rest of the method displays the information using formatted strings to align under the column headings.

Private method `CMD_DisplayAvailVehicles` displays all vehicles of a given vehicle type that are not currently reserved:

```
Enter vehicle type: 1
----- Available Cars -----
Chevrolet Camaro    passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
Chevrolet Camaro    passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
Ford Fusion         passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
Ford Fusion Hybrid  passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
Ford Fusion Hybrid  passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
Chevrolet Impala    passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394J1
Chevrolet Impala    passengers: 6   doors: 4    mpg: 30   vin: RK3BM4256YH
```

It displays a selection of vehicle types (**lines 158–159**) for the user to select from, and gets the available vehicles for the selected type from the system interface by call to method

`getAvailVehicles` of the system interface (**line 147**). The rest of the method simply displays, row by row, the information for each vehicle.

Private method `CMD_PromptAndDisplayAvailVehicles` (**lines 161–169**) prompts the user for a vehicle type, and then displays the information of vehicles of that type not currently reserved. Private method `CMD_DisplaySpecificRentalCost` (**lines 171–226**) displays the costs of a particular vehicle rental based on the vehicle type, the time period rented, whether insurance is opted for, and the estimated number of miles driven:

```
----- ESTIMATED Car RENTAL COST -----
* You have opted out of the daily insurance *

Your cost of a daily rental would be 74.97

Your cost with an estimated mileage of 240 would be 95.97
which includes 100 free miles and a charge of 0.15 per mile
-----
```

First, it displays a selection of vehicle types for the user to choose from (**line 176**). It then requests the desired rental period by a call to private method `getRentalPeriod` (**line 180**). A tuple is returned containing constant value `DAILY_RENTAL`, `WEEKLY_RENTAL`, or `WEEKEND_RENTAL`, and the number of days or weeks rental period (with weekends defaulting to 1). The user is then asked if they want the optional insurance (**line 186**) and the number of miles expected to drive (**line 193**). The estimated rental cost is then calculated by a call to method `calcRentalCost` of the system interface. The rest of the method has to do with formatting and displaying the output.

Private method `makeReservation` allows the user to reserve a vehicle of a given type (**line 227**). The method first checks if there are vehicles available (**line 235**). It then calls `CMD_DisplayAvailVehicles` (**line 239**) with the default argument set to True. This causes the vehicle descriptions to be listed as a number list so that the desired vehicle can be selected. On **lines 242–253**, the vehicle number selected is read, checking for any invalid selections. On **line 255**, the VIN is retrieved for the selected vehicle. The VIN is needed by method `getVehicle` (of the `SystemInterface` class) to retrieve a given `Vehicle` object. Having the object, it then calls method `getDescription` (**line 257**) to display the description of the vehicle to the user. It also sets the reservation status to True by a call to `setReserved` (**line 259**).

The user's name, address, and credit card number are requested (**lines 261–263**) to make the reservation. It does this by creating a new `Reservation` object constructed with the provided information (**line 265**) and adds it to the collection of reservations (**line 266**). Confirmation of the reservation is displayed (**line 267**). Private method `CMD_CancelReservation` (**line 269–279**) cancels reservations by the credit card number used for making the reservation. Finally, private methods `displayVehicleTypes` and `getRentalPeriod`, supporting methods of the command methods, are given on **lines 281–337**.

## Putting it All Together

Figure 10-50 shows the main module of the Vehicle Rental Agency Program that puts everything together. It first creates the vehicle rental agency system (with a system interface) and then creates a user interface containing a reference to the system, providing the user with a natural means of accessing the system.

```

1 # Rental Agency Program
2
3 """ This program performs the tasks of a vehicle rental agency, including the
4     display of vehicle information and the ability to make/cancel reservations.
5 """
6 from systemInterface import SystemInterface
7 from rentalAgencyUI import RentalAgencyUI
8
9 try:
10     # create system with populated data from file
11     sys = SystemInterface()
12
13     # associate user interface with system
14     ui = RentalAgencyUI(sys)
15
16     # start the user interface
17     ui.start()
18
19 except IOError:
20     print('** PROGRAM TERMINATION (IO Error) **')

```

**FIGURE 10-50** Rental Agency Program Main Module

The main module for the program imports the `SystemInterface` class (**line 6**) and the `RentalAgencyUI` class (**line 7**). First, an instance of the `SystemInterface` class is created (**line 11**). The `__init__` method of the class is implemented to open the vehicle information file, as well as the vehicle cost file. The information in these files is used to populate the vehicle and vehicle cost objects in the system. If any file errors occur during this process, an `IOError` exception is raised by the system interface and caught (**line 19**) to terminate the program.

Once the files are successfully opened and read, an instance of the `RentalAgencyUI` class is created (**line 14**). It is passed the needed reference to the system interface. Once the user interface is created, the `start` method of the user interface is called to begin the command loop and receive and execute commands from the user.

## CHAPTER SUMMARY

### General Topics

- Classes
- Encapsulation, Inheritance, and Polymorphism
- Public vs. Private Class Members
- Getters and Setters
- Name Mangling
- Superclass (Base Class/Parent Class)
- Subclass (Derived Class/Child Class)
- Subclass vs. Subtype
- UML (Unified Modeling Language)
- Class Diagrams in UML
- Association, Multiplicity, and Role Names in UML
- Composition vs. Aggregation

### Python-Specific Programming Topics

- Defining Classes in Python
- Denoting Public and Private Class Members in Python
- The Use of `self` in Python
- Special Method `__init__` in Python
- Arithmetic and Relational Special Methods in Python
- Inheritance and Polymorphism in Python
- Duck Typing in Python

## CHAPTER EXERCISES

### Section 10.1

1. What are the two kinds of entities “bundled” in a class?
2. What kind of entity can there be any number of instances created for a given class?
3. What are the three fundamental features that object-oriented programming languages have in support of object-oriented programming?

### Section 10.2

4. What else does encapsulation provide other than the ability to bundle together instance variables and methods?
5. Describe what it means for a member of a class to be private.
6. Explain the purpose of getters and setters.
7. Explain what the special identifier `self` is used for in Python.
8. Explain the use of name mangling in Python.
9. Explain when special methods `__str__` and `__repr__` are each used in Python.
10. Give an implementation of special method `__str__` for a `Range` class (representing a range of integers) that contains integer instance variables, `__start` and `__end`, so that the value of `Range` objects are displayed as follows: '10 . . . 16', when output with `print`.
11. Give an implementation of special method `__lt__` for the `Range` class of exercise 10 so that `range1 < range2` evaluates to True if all the values in `range1` are less than all the values in `range2`, and returns `False` otherwise.

### Section 10.3

12. Give the one-line class definition header for a class named `MySubclass` that is a subclass of the class `MySuperclass`.
13. Explain when a subclass can serve as a subtype.
14. For an object `obj`, show how in Python the type of the object may be determined from within a program or the Python shell.
15. Show how in the Python shell information about one of the built-in types of Python can be displayed.

### Section 10.4

16. Explain the concept of polymorphism in object-oriented programming.
17. Explain the advantages of using polymorphism in program design.
18. What is meant by “duck typing” in Python?

### Section 10.5

19. What does the name UML stand for?
20. What are the two types of diagrams in UML mentioned in the chapter, and what aspects of a program design does each represent?
21. Give a class diagram for the `XYCoord` class in the Let’s Try It box of section 10.2.2.
22. Give a class diagram that includes the partial description of the built-in `str` type in Figure 10-15, and the `ExplodedStr` subclass in Figure 10-16.
23. Give a class diagram for the `Fraction` class developed in section 10.2. Include the `MixedFraction` subclass in the diagram.

## PYTHON PROGRAMMING EXERCISES

**P1.** Give a UML class diagram for a library. Include as entities tangible objects (such as books), persons (such as borrowers and librarians), and status (such as whether a book is checked out or not). Use multiplicity, navigation, and role names where appropriate.

**P2.** Design and implement a `Money` class that stores monetary values in dollars and cents. Special method `__init__` should have the following function header,

```
def __init__(self, dollars, cents)
```

Include special method `__repr__` (`__str__`) for displaying values in dollars and cents: \$ 0.45, \$ 1.00, \$ 1.25. Also include special method `__add__`, and three getter methods that each provide the monetary value in another currency. Choose any three currencies to convert to.

**P3.** Implement a class named `AvgList` as a subclass of the built-in list class in Python, able to compute the average of a list of numeric values. If the list contains any nonnumeric types, a `ValueError` exception should be raised.

**P4.** Design and implement a `FootMeasure` class that stores a linear measurement of feet and inches. Your class should have the following function header for special method `__init__`,

```
def __init__(self, feet=0, inches=0)
```

Thus, the class should be able to create a `FootMeasure` object in various ways by use of optional keyword arguments,

```
meas = FootMeasure()
meas = FootMeasure(feet=5)
meas = FootMeasure(feet=5, inches=8)
meas = FootMeasure(inches=68)
```

Implement special method `__repr__` in the class so that measurements are displayed as follows,

```
5 ft.           NOT    5 ft. 0 in.
      5 ft. 8 in.  NOT    68 in.
```

When the measurement is 0, it should be displayed as, 0 ft. 0. ins. Include special method `add()` for adding `FootMeasure` values. Also include all the special methods for implementing the relational operators.

**P5.** Develop an abstract class named `Temperature` that stores a single temperature. The class should have the following function header for special method `__init__`,

```
def __init__(self, temperature)
```

The abstract class should contain the following methods:

`__str__` — should return a string of the form “75 degrees Fahrenheit”

`aboveFreezing()` — returns True if temperature above the freezing point

`convertToFahren` — returns a new `Temperature` object converted to degrees Fahrenheit

`convertToCelsius` — returns a new `Temperature` object converted to degrees Celsius

`convertToKelvin` — returns a new `Temperature` object converted to degrees Kelvin

Develop the subclasses Fahrenheit, Celsius and Kelvin to appropriately implement each of the methods in the abstract Temperature class. (Note that when a meaningless conversion method is applied, for example, `temp1.convertToFahrenheit()` where `temp1` is an object of type `Fahrenheit`, then a copy of the `Temperature` object should be returned.)

Demonstrate the correctness of your classes by doing the following:

- ◆ Create a list of `Temperature` objects of a mix of `Temperature` types
- ◆ Print out the value of each temperature in the list, and add “above freezing” if the temperature is above freezing (for the specific temperature scale).
- ◆ Create a new list of temperatures containing each temperature of the original list converted to a common temperature scale (Fahrenheit, Celsius, or Kelvin).
- ◆ For each temperature object in the new list, print out its temperature value, and if it is above the freezing point.

## PROGRAM MODIFICATION PROBLEMS

### M1. Fraction Class—Adding a Supporting Reduce Method

Complete the `Fraction` class in section 10.2.2 by implementing the missing methods using the specifications below,

```
def copy(self):
    """Creates a copy of a given Fraction."""

def reduce(self):
    """Reduces self to simplest terms. Also removes the signs
    if both numerator and denominator are negative."""

def __adjust(self, factor):
    """Multiplies numerator and denominator by factor."""
```

### M2. Vehicle Rental Agency Program—View Reservations

Modify the Vehicle Rental Agency program in section 10.6 to add the capability of viewing all current reservations.

### M3. Division Operator for Fraction Class

Modify the `Fraction` class developed in section 10.2.2 to include a division operator by implementing special method `__truediv__`.

### M4. Screen Display in Recipe Conversion Program

Modify the Recipe Conversion Program in section 10.2.3 so that the original recipe ingredient measurements and the converted measurements are both displayed on the screen (in addition to having the converted recipe written to a file).

### M5. Adjusted Measurements in the Recipe Conversion Program

Modify the Recipe Conversion Program in section 10.2.3 so that the units of measure for teaspoons and tablespoons in a converted recipe are displayed more appropriately. For example, if a recipe that calls for 2 teaspoons of baking soda is tripled, 6 teaspoons of baking soda are currently displayed in the converted recipe. Since there are 3 teaspoons to a tablespoon, the converted measurement would be more appropriately displayed as 2 tablespoons. Similarly, since there are 16 tablespoons in a cup,

any number of tablespoons over 16 would be more appropriately displayed as one cup, plus some number of tablespoons.

**M6. Revised Estimated Rental Cost in Vehicle Rental Agency Program**

The rental cost for a particular type vehicle computed in the Vehicle Rental Agency Program is based on the number of days (or weeks) rented, the cost of optional insurance, and the estimated number of miles that the vehicle is expected to be driven. However, the estimate does not include the cost of gas that the customer must also pay. Since the miles per gallon (mpg) is available for all vehicles, modify the program so that this additional expense is added to the calculation of the estimated cost. The cost that is displayed should identify this additional expense. Use an instance variable in the `VehicleCosts` class to store the current price of gas per gallon.

**M7. Additional Vehicle Type in Vehicle Rental Agency Program**

Modify the Vehicle Rental Agency Program to include a fourth SUV vehicle type, in addition to cars, vans, and trucks. The attributes stored for an SUV object should include make and model, miles per gallon (mpg), number of passengers, whether or not it has automatic sliding doors, and the vehicle identification number (VIN). Make *all* changes in the program to incorporate this new vehicle type.

**M8. Polymorphic Behavior of Vehicle Cost in Vehicle Rental Agency Program**

Modify the Vehicle Rental Agency Program to include three subclasses of the `VehicleCost` class, `CarCost`, `VanCost`, and `TruckCost` so that `VehicleCost` can behave polymorphically. Make all changes to the program to incorporate these new vehicle cost types.

## PROGRAM DEVELOPMENT PROBLEMS

**D1. RGB/Hexadecimal Color Code Manipulation Program**

Design an abstract class named `ColorCode`, and two subclass named `RGBColorCode` and `HexColorCode`. RGB color codes (as discussed in Chapter 9) are of the form (125, 80, 210), indicating the amount of red, green, and blue, respectively, for a given color. Each color value is on a scale of 0–255. Hexadecimal color codes are of the form F4F060, in which the first two hexadecimal digits indicate the amount of red, the middle two digits the amount of green, and the last two digits the amount of blue. Each pair of hexadecimal digits represent values in the range 0–255. Thus, these color encodings are just different representations of the same range of colors. Include in abstract class `ColorCode` methods for displaying a color code, for adding and reducing the amount of red, green, or blue by a given percentage, and for producing the complement of a given color. The complement is determined by subtracting each color value from 255. Develop a program in which the user can enter a series of colors in both RGB and hexadecimal form, and have them altered in the ways given above.

**D2. Morse Code Translation Program**

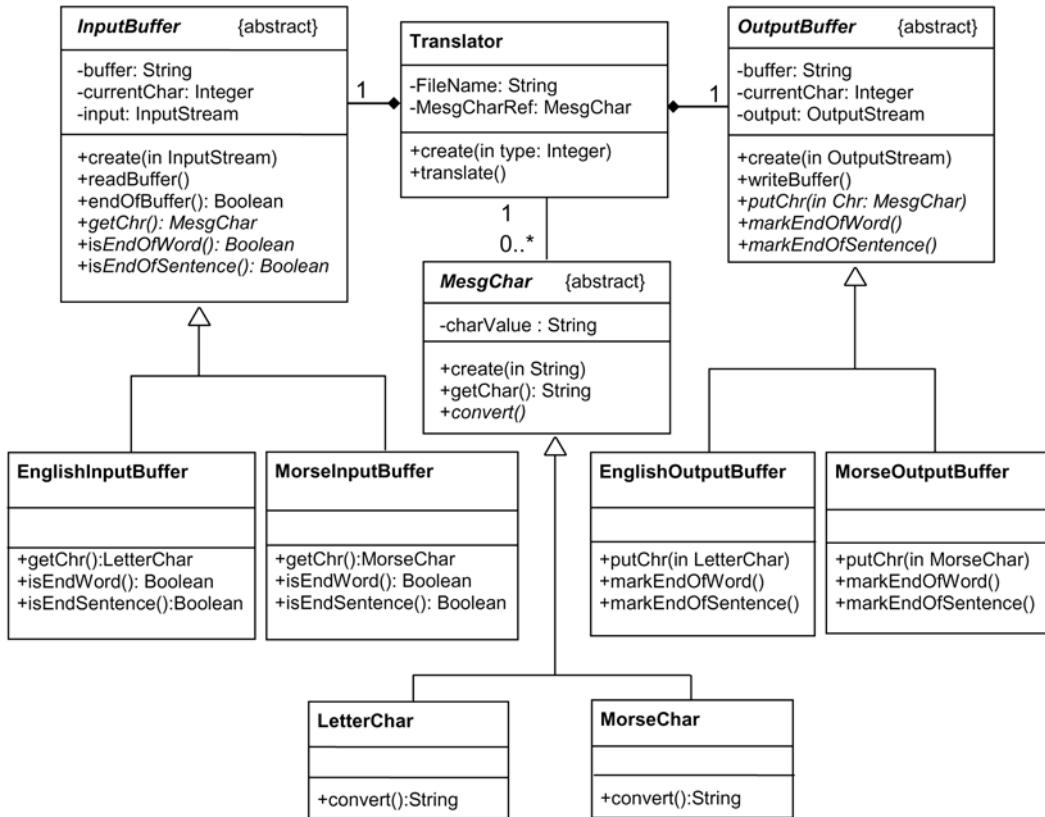
Implement a Morse code translator program from the UML design given below for translating English message files into Morse code, and Morse code files into English. The requirements for this program are as follows:

- ◆ English message files will be stored as text files, one sentence per line.
- ◆ Morse code message files will be stored as text files, one encoded character per line, where end of words are indicated by a blank line, and end of sentences by two blank lines.
- ◆ English message files will contain file extension `.eng`, and Morse code files file extension `.mor`.

The assumptions for this program are as follows:

- ◆ Messages will only contain lowercase letters, the digits 0–9, periods, commas, and question marks.

Following is the UML diagram for the program.



The `Translator` class coordinates the translation of messages (English to Morse code, and Morse code to English). It contains two class members through composition: `InputBuffer` and `OutputBuffer`. Thus the `Translator` class is responsible for their construction.

The `InputBuffer` contains the current message line read from the file. If the file contains an English message, then an `EnglishInputBuffer` is used; if the files contains a Morse-coded message, then a `MorseInputBuffer` is used. As the `Translator` reads the current `MesgChar` from the `InputBuffer`, it requests the `MesgChar` to translate itself, sending it to the `OutputBuffer`. When `isEndOfWord` or `isEndOfSentence` is found true, it calls the corresponding `MarkEndOfWord` or `MarkEndOfSentence` method of the `OuputBuffer`. If the output is in English, a space character for each end of word and a period for each end of sentence is added to the buffer; if the output is in Morse code, then a blank line for each end of word, and two blank lines for each end of sentence is added to the buffer. This is repeated until the end of file has been reached.

Note that what a given `MsgChar` is depends on the type of message file reading. If the message being read is an English message, then a `MsgChar` is a single character (letter, digit, etc.). If, however, the message being read is a Morse-code message, then a `MsgChar` is a `MorseChar` (i.e., a string containing up to six dot and dash characters). Thus, when getting and putting `MsgChars` of a given `InputBuffer` and `OutputBuffer`, the number of characters actually retrieved or placed in the buffer depends on what type of buffer is involved. The Morse Code table for this assignment is given below.

#### Morse Code Table

A	.-	N	-.	1	-----	.	.-.-.-
B	-...	O	---	2	.----	,	--...--
C	-.-.	P	.---.	3	....-	?	..---..
D	-..	Q	-.-.	4	....-	(	-.-..
E	.	R	.-.	5	.....	)	-.-.-.
F	...-.	S	...	6	.....	-	-.....-
G	--.	T	-	7	---..	"	.-.-.-.
H	....	U	...-	8	---..	-	..---..
I	..	V	...-	9	---..	^	.----.
J	----	W	.--	0	-----		
K	-.-	X	-..-				
L	-...-	Y	-.--				
M	--	Z	-..-				