2.5 Object-Oriented Programming

Object-oriented programming (OOP) is a method for organizing programs that brings together many of the ideas introduced in this chapter. Like the functions in data abstraction, classes create abstraction barriers between the use and implementation of data. Like dispatch dictionaries, objects respond to behavioral requests. Like mutable data structures, objects have local state that is not directly accessible from the global environment. The Python object system provides convenient syntax to promote the use of these techniques for organizing programs. Much of this syntax is shared among other object-oriented programming languages.

The object system offers more than just convenience. It enables a new metaphor for designing programs in which several independent agents interact within the computer. Each object bundles together local state and behavior in a way that abstracts the complexity of both. Objects communicate with each other, and useful results are computed as a consequence of their interaction. Not only do objects pass messages, they also share behavior among other objects of the same type and inherit characteristics from related types.

The paradigm of object-oriented programming has its own vocabulary that supports the object metaphor. We have seen that an object is a data value that has methods and attributes, accessible via dot notation. Every object also has a type, called its *class*. To create new types of data, we implement new classes.

2.5.1 Objects and Classes

A class serves as a template for all objects whose type is that class. Every object is an instance of some particular class. The objects we have used so far all have built-in classes, but new user-defined classes can be created as well. A class definition specifies the attributes and methods shared among objects of that class. We will introduce the class statement by revisiting the example of a bank account.

When introducing local state, we saw that bank accounts are naturally modeled as mutable values that have a `balance`. A bank account object should have a `withdraw` method that updates the account balance and returns the requested amount, if it is available. To complete the abstraction: a bank account should be able to return its current `balance`, return the name of the account `holder`, and an amount for `deposit`.

An `Account` class allows us to create multiple instances of bank accounts. The act of creating a new object instance is known as *instantiating* the class. The syntax in Python for instantiating a class is identical to the syntax of calling a function. In this case, we call `Account` with the argument `'Kirk'`, the account holder's name.

```
>>> a = Account('Kirk')
```

An *attribute* of an object is a name-value pair associated with the object, which is accessible via dot notation. The attributes specific to a particular object, as opposed to all objects of a class, are called *instance attributes*. Each `Account` has its own balance and account holder name, which are examples of instance attributes. In the broader programming community, instance attributes may also be called *fields*, *properties*, or *instance variables*.

```
>>> a.holder
'Kirk'
>>> a.balance
0
```

Functions that operate on the object or perform object-specific computations are called methods. The return values and side effects of a method can depend upon and change other attributes of the object. For example, `deposit` is a method of our `Account` object `a`. It takes one argument, the amount to deposit, changes the `balance` attribute of the object, and returns the resulting balance.

```
>>> a.deposit(15)
15
```

We say that methods are *invoked* on a particular object. As a result of invoking the `withdraw` method, either the withdrawal is approved and the amount is deducted, or the request is declined and the method returns an error message.

```
>>> a.withdraw(10)  # The withdraw method returns the balance after withdrawal
5
>>> a.balance       # The balance attribute has changed
5
>>> a.withdraw(10)
'Insufficient funds'
```

As illustrated above, the behavior of a method can depend upon the changing attributes of the object. Two calls to `withdraw` with the same argument return different results.

2.5.2 Defining Classes

User-defined classes are created by `class` statements, which consist of a single clause. A class statement defines the class name, then includes a suite of statements to define the attributes of the class:

```
class <name>:
    <suite>
```

When a class statement is executed, a new class is created and bound to `<name>` in the first frame of the current environment. The suite is then executed. Any names bound within the `<suite>` of a `class` statement, through `def` or assignment statements, create or modify attributes of the class.

Classes are typically organized around manipulating instance attributes, which are the name-value pairs associated with each instance of that class. The class specifies the instance attributes of its objects by defining a method for initializing new objects. For example, part of initializing an object of the `Account` class is to assign it a starting balance of 0.

The `<suite>` of a `class` statement contains `def` statements that define new methods for objects of that class. The method that initializes objects has a special name in Python, `__init__` (two underscores on each side of the word "init"), and is called the *constructor* for the class.

```
>>> class Account:
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
```

The `__init__` method for `Account` has two formal parameters. The first one, `self`, is bound to the newly created `Account` object. The second parameter, `account_holder`, is bound to the argument passed to the class when it is called to be instantiated.

The constructor binds the instance attribute name `balance` to 0. It also binds the attribute name `holder` to the value of the name `account_holder`. The formal parameter `account_holder` is a local name in the `__init__` method. On the other hand, the name `holder` that is bound via the final assignment statement persists, because it is stored as an attribute of `self` using dot notation.

Having defined the `Account` class, we can instantiate it.

```
>>> a = Account('Kirk')
```

This "call" to the `Account` class creates a new object that is an instance of `Account`, then calls the constructor function `__init__` with two arguments: the newly created object and the string `'Kirk'`. By convention, we use the parameter name `self` for the first argument of a constructor, because it is bound to the object being instantiated. This convention is adopted in virtually all Python code.

Now, we can access the object's `balance` and `holder` using dot notation.

```
>>> a.balance
0
>>> a.holder
'Kirk'
```

Identity. Each new account instance has its own balance attribute, the value of which is independent of other objects of the same class.

```
>>> b = Account('Spock')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
```

To enforce this separation, every object that is an instance of a user-defined class has a unique identity. Object identity is compared using the `is` and `is not` operators.

```
>>> a is a
True
>>> a is not b
True
```

Despite being constructed from identical calls, the objects bound to `a` and `b` are not the same. As usual, binding an object to a new name using assignment does not create a new object.

```
>>> c = a
>>> c is a
True
```

New objects that have user-defined classes are only created when a class (such as `Account`) is instantiated with call expression syntax.

Methods. Object methods are also defined by a `def` statement in the suite of a `class` statement. Below, `deposit` and `withdraw` are both defined as methods on objects of the `Account` class.

```
>>> class Account:
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

While method definitions do not differ from function definitions in how they are declared, method definitions do have a different effect when executed. The function value that is created by a `def` statement within a `class` statement is bound to the declared name, but bound locally within the class as an attribute. That value is invoked as a method using dot notation from an instance of the class.

Each method definition again includes a special first parameter `self`, which is bound to the object on which the method is invoked. For example, let us say that `deposit` is invoked on a particular `Account` object and passed a single argument value: the amount deposited. The object itself is bound to `self`, while the argument is bound to `amount`. All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

To invoke these methods, we again use dot notation, as illustrated below.

```
>>> spock_account = Account('Spock')
>>> spock_account.deposit(100)
100
>>> spock_account.withdraw(90)
10
>>> spock_account.withdraw(90)
'Insufficient funds'
>>> spock_account.holder
'Spock'
```

When a method is invoked via dot notation, the object itself (bound to `spock_account`, in this case) plays a dual role. First, it determines what the name `withdraw` means; `withdraw` is not a name in the environment, but instead a name that is local to the `Account` class. Second, it is bound to the first parameter `self` when the `withdraw` method is invoked.

2.5.3 Message Passing and Dot Expressions

Methods, which are defined in classes, and instance attributes, which are typically assigned in constructors, are the fundamental elements of object-oriented programming. These two concepts replicate much of the behavior of a dispatch dictionary in a message passing implementation of a data value. Objects take messages using dot notation, but instead of those messages being arbitrary string-valued keys, they are names local to a class. Objects also have named local state values (the instance attributes), but that state can be accessed and manipulated using dot notation, without having to employ `nonlocal` statements in the implementation.

The central idea in message passing was that data values should have behavior by responding to messages that are relevant to the abstract type they represent. Dot notation is a syntactic feature of Python that formalizes the message passing metaphor. The advantage of using a language with a built-in object system is that message passing can interact seamlessly with other language features, such as assignment statements. We do not require different messages to "get" or "set" the value associated with a local attribute name; the language syntax allows us to use the message name directly.

Dot expressions. The code fragment `spock_account.deposit` is called a *dot expression*. A dot expression consists of an expression, a dot, and a name:

```
<expression> . <name>
```

The `<expression>` can be any valid Python expression, but the `<name>` must be a simple name (not an expression that evaluates to a name). A dot expression evaluates to the value of the attribute with the given `<name>`, for the object that is the value of the `<expression>`.

The built-in function `getattr` also returns an attribute for an object by name. It is the function equivalent of dot notation. Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch dictionary.

```
>>> getattr(spock_account, 'balance')
10
```

We can also test whether an object has a named attribute with `hasattr`.

```
>>> hasattr(spock_account, 'deposit')
True
```

The attributes of an object include all of its instance attributes, along with all of the attributes (including methods) defined in its class. Methods are attributes of the class that require special handling.

Methods and functions. When a method is invoked on an object, that object is implicitly passed as the first argument to the method. That is, the object that is the value of the `<expression>` to the left of the dot is passed automatically as the first argument to the method named on the right side of the dot expression. As a result, the object is bound to the parameter `self`.

To achieve automatic `self` binding, Python distinguishes between *functions*, which we have been creating since the beginning of the text, and *bound methods*, which couple together a function and the object on which that method will be invoked. A bound method value is already associated with its first argument, the instance on which it was invoked, which will be named `self` when the method is called.

We can see the difference in the interactive interpreter by calling `type` on the returned values of dot expressions. As an attribute of a class, a method is just a function, but as an attribute of an instance, it is a bound method:

```
>>> type(Account.deposit)
<class 'function'>
>>> type(spock_account.deposit)
<class 'method'>
```

These two results differ only in the fact that the first is a standard two-argument function with parameters `self` and `amount`. The second is a one-argument method, where the name `self` will be bound to the object named `spock_account` automatically when the method is called, while the parameter `amount` will be bound to the argument passed to the method. Both of these values, whether function values or bound method values, are associated with the same `deposit` function body.

We can call `deposit` in two ways: as a function and as a bound method. In the former case, we must supply an argument for the `self` parameter explicitly. In the latter case, the `self` parameter is bound automatically.

```
>>> Account.deposit(spock_account, 1001)  # The deposit function takes 2 arguments
1011
>>> spock_account.deposit(1000)           # The deposit method takes 1 argument
2011
```

The function `getattr` behaves exactly like dot notation: if its first argument is an object but the name is a method defined in the class, then `getattr` returns a bound method value. On the other hand, if the first argument is a class, then `getattr` returns the attribute value directly, which is a plain function.

Naming Conventions. Class names are conventionally written using the CapWords convention (also called CamelCase because the capital letters in the middle of a name look like humps). Method names follow the standard convention of naming functions using lowercased words separated by underscores.

In some cases, there are instance variables and methods that are related to the maintenance and consistency of an object that we don't want users of the object to see or use. They are not part of the abstraction defined by a class, but instead part of the implementation. Python's convention dictates that if an attribute name starts with an underscore, it should only be accessed within methods of the class itself, rather than by users of the class.

2.5.4 Class Attributes

Some attribute values are shared across all objects of a given class. Such attributes are associated with the class itself, rather than any individual instance of the class. For instance, let us say that a bank pays interest on the balance of accounts at a fixed interest rate. That interest rate may change, but it is a single value shared across all accounts.

Class attributes are created by assignment statements in the suite of a `class` statement, outside of any method definition. In the broader developer community, class attributes may also be called class variables or static variables. The following class statement creates a class attribute for `Account` with the name `interest`.

```
>>> class Account:
        interest = 0.02            # A class attribute
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        # Additional methods would be defined here
```

This attribute can still be accessed from any instance of the class.

```
>>> spock_account = Account('Spock')
>>> kirk_account = Account('Kirk')
>>> spock_account.interest
0.02
>>> kirk_account.interest
0.02
```

However, a single assignment statement to a class attribute changes the value of the attribute for all instances of the class.

```
>>> Account.interest = 0.04
>>> spock_account.interest
0.04
>>> kirk_account.interest
0.04
```

Attribute names. We have introduced enough complexity into our object system that we have to specify how names are resolved to particular attributes. After all, we could easily have a class attribute and an instance attribute with the same name.

As we have seen, a dot expression consists of an expression, a dot, and a name:

```
<expression> . <name>
```

To evaluate a dot expression:

1. Evaluate the `<expression>` to the left of the dot, which yields the *object* of the dot expression.
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If `<name>` does not appear among instance attributes, then `<name>` is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

In this evaluation procedure, instance attributes are found before class attributes, just as local names have priority over global in an environment. Methods defined within the class are combined with the object of the dot expression to form a bound method during the fourth step of this evaluation procedure. The procedure for looking up a name in a class has additional nuances that will arise shortly, once we introduce class inheritance.

Attribute assignment. All assignment statements that contain a dot expression on their left-hand side affect attributes for the object of that dot expression. If the object is an instance, then assignment sets an instance attribute. If the object is a class, then assignment sets a class attribute. As a consequence of this rule, assignment to an attribute of an object cannot affect the attributes of its class. The examples below illustrate this distinction.

If we assign to the named attribute `interest` of an account instance, we create a new instance attribute that has the same name as the existing class attribute.

```
>>> kirk_account.interest = 0.08
```

and that attribute value will be returned from a dot expression.

```
>>> kirk_account.interest
0.08
```

However, the class attribute `interest` still retains its original value, which is returned for all other accounts.

```
>>> spock_account.interest
0.04
```

Changes to the class attribute `interest` will affect `spock_account`, but the instance attribute for `kirk_account` will be unaffected.

```
>>> Account.interest = 0.05  # changing the class attribute
>>> spock_account.interest     # changes instances without like-named instance attributes
0.05
>>> kirk_account.interest     # but the existing instance attribute is unaffected
0.08
```

2.5.5 Inheritance

When working in the object-oriented programming paradigm, we often find that different types are related. In particular, we find that similar classes differ in their amount of specialization. Two classes may have similar attributes, but one represents a special case of the other.

For example, we may want to implement a checking account, which is different from a standard account. A checking account charges an extra $1 for each withdrawal and has a lower interest rate. Here, we demonstrate the desired behavior.

```
>>> ch = CheckingAccount('Spock')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # withdrawals decrease balance by an extra charge
14
```

A `CheckingAccount` is a specialization of an `Account`. In OOP terminology, the generic account will serve as the base class of `CheckingAccount`, while `CheckingAccount` will be a subclass of `Account`. (The terms

parent class and *superclass* are also used for the base class, while *child class* is also used for the subclass.)

A subclass *inherits* the attributes of its base class, but may *override* certain attributes, including certain methods. With inheritance, we only specify what is different between the subclass and the base class. Anything that we leave unspecified in the subclass is automatically assumed to behave just as it would for the base class.

Inheritance also has a role in our object metaphor, in addition to being a useful organizational feature. Inheritance is meant to represent *is-a* relationships between classes, which contrast with *has-a* relationships. A checking account *is-a* specific type of account, so having a `CheckingAccount` inherit from `Account` is an appropriate use of inheritance. On the other hand, a bank *has-a* list of bank accounts that it manages, so neither should inherit from the other. Instead, a list of account objects would be naturally expressed as an instance attribute of a bank object.

2.5.6 Using Inheritance

First, we give a full implementation of the `Account` class, which includes docstrings for the class and its methods.

```
>>> class Account:
        """A bank account that has a non-negative balance."""
        interest = 0.02
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            """Increase the account balance by amount and return the new balance."""
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            """Decrease the account balance by amount and return the new balance."""
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

A full implementation of `CheckingAccount` appears below. We specify inheritance by placing an expression that evaluates to the base class in parentheses after the class name.

```
>>> class CheckingAccount(Account):
        """A bank account that charges for withdrawals."""
        withdraw_charge = 1
        interest = 0.01
        def withdraw(self, amount):
            return Account.withdraw(self, amount + self.withdraw_charge)
```

Here, we introduce a class attribute `withdraw_charge` that is specific to the `CheckingAccount` class. We assign a lower value to the `interest` attribute. We also define a new `withdraw` method to override the behavior defined in the `Account` class. With no further statements in the class suite, all other behavior is inherited from the base class `Account`.

```
>>> checking = CheckingAccount('Sam')
>>> checking.deposit(10)
10
>>> checking.withdraw(5)
4
>>> checking.interest
0.01
```

The expression `checking.deposit` evaluates to a bound method for making deposits, which was defined in the `Account` class. When Python resolves a name in a dot expression that is not an attribute of the instance, it looks up the name in the class. In fact, the act of "looking up" a name in a class tries to find that name in every base class in the inheritance chain for the original object's class. We can define this procedure recursively. To look up a name in a class.

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

In the case of `deposit`, Python would have looked for the name first on the instance, and then in the `CheckingAccount` class. Finally, it would look in the `Account` class, where `deposit` is defined. According to our evaluation rule for dot expressions, since `deposit` is a function looked up in the class for the `checking` instance, the dot expression evaluates to a bound method value. That method is invoked with the argument 10, which calls the deposit method with `self` bound to the `checking` object and `amount` bound to 10.

The class of an object stays constant throughout. Even though the `deposit` method was found in the `Account` class, `deposit` is called with `self` bound to an instance of `CheckingAccount`, not of `Account`.

Calling ancestors. Attributes that have been overridden are still accessible via class objects. For instance, we implemented the `withdraw` method of `CheckingAccount` by calling the `withdraw` method of `Account` with an argument that included the `withdraw_charge`.

Notice that we called `self.withdraw_charge` rather than the equivalent `CheckingAccount.withdraw_charge`. The benefit of the former over the latter is that a class that inherits from `CheckingAccount` might override the withdrawal charge. If that is the case, we would like our implementation of `withdraw` to find that new value instead of the old one.

Interfaces. It is extremely common in object-oriented programs that different types of objects will share the same attribute names. An *object interface* is a collection of attributes and conditions on those attributes. For example, all accounts must have `deposit` and `withdraw` methods that take numerical arguments, as well as a `balance` attribute. The classes `Account` and `CheckingAccount` both implement this interface. Inheritance specifically promotes name sharing in this way. In some programming languages such as Java, interface implementations must be explicitly declared. In others such as Python, Ruby, and Go, any object with the appropriate names implements an interface.

The parts of your program that use objects (rather than implementing them) are most robust to future changes if they do not make assumptions about object types, but instead only about their attribute names. That is, they use the object abstraction, rather than assuming anything about its implementation.

For example, let us say that we run a lottery, and we wish to deposit $5 into each of a list of accounts. The following implementation does not assume anything about the types of those accounts, and therefore works equally well with any type of object that has a `deposit` method:

```
>>> def deposit_all(winners, amount=5):
        for account in winners:
            account.deposit(amount)
```

The function `deposit_all` above assumes only that each `account` satisfies the account object abstraction, and so it will work with any other account classes that also implement this interface. Assuming a particular class of account would violate the abstraction barrier of the account object abstraction. For example, the following implementation will not necessarily work with new kinds of accounts:

```
>>> def deposit_all(winners, amount=5):
        for account in winners:
            Account.deposit(account, amount)
```

We will address this topic in more detail later in the chapter.

2.5.7 Multiple Inheritance

Python supports the concept of a subclass inheriting attributes from multiple base classes, a language feature called *multiple inheritance*.

Suppose that we have a `SavingsAccount` that inherits from `Account`, but charges customers a small fee every time they make a deposit.

```
>>> class SavingsAccount(Account):
        deposit_charge = 2
        def deposit(self, amount):
            return Account.deposit(self, amount - self.deposit_charge)
```

Then, a clever executive conceives of an `AsSeenOnTVAccount` account with the best features of both `CheckingAccount` and `SavingsAccount`: withdrawal fees, deposit fees, and a low interest rate. It's both a checking and a savings account in one! "If we build it," the executive reasons, "someone will sign up and pay all those fees. We'll even give them a dollar."

```
>>> class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
        def __init__(self, account_holder):
            self.holder = account_holder
            self.balance = 1           # A free dollar!
```

In fact, this implementation is complete. Both withdrawal and deposits will generate fees, using the function definitions in `CheckingAccount` and `SavingsAccount` respectively.

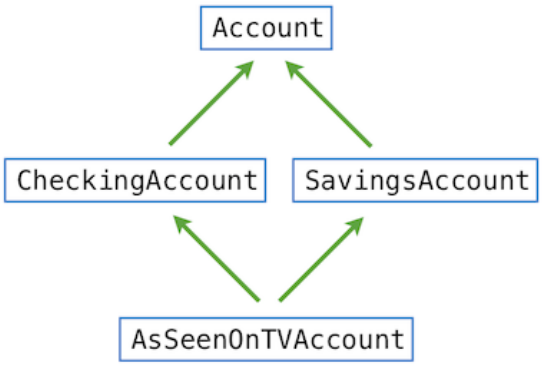
```
>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)           # $2 fee from SavingsAccount.deposit
19
>>> such_a_deal.withdraw(5)           # $1 fee from CheckingAccount.withdraw
13
```

Non-ambiguous references are resolved correctly as expected:

```
>>> such_a_deal.deposit_charge
2
>>> such_a_deal.withdraw_charge
1
```

But what about when the reference is ambiguous, such as the reference to the `withdraw` method that is defined in both `Account` and `CheckingAccount`? The figure below depicts an *inheritance graph* for the

`AsSeenOnTVAccount` class. Each arrow points from a subclass to a base class.



For a simple "diamond" shape like this, Python resolves names from left to right, then upwards. In this example, Python checks for an attribute name in the following classes, in order, until an attribute with that name is found:

```
AsSeenOnTVAccount, CheckingAccount, SavingsAccount, Account, object
```

There is no correct solution to the inheritance ordering problem, as there are cases in which we might prefer to give precedence to certain inherited classes over others. However, any programming language that supports multiple inheritance must select some ordering in a consistent way, so that users of the language can predict the behavior of their programs.

Further reading. Python resolves this name using a recursive algorithm called the C3 Method Resolution Ordering. The method resolution order of any class can be queried using the `mro` method on all classes.

```
>>> [c.__name__ for c in AsSeenOnTVAccount.mro()]
['AsSeenOnTVAccount', 'CheckingAccount', 'SavingsAccount', 'Account', 'object']
```

The precise algorithm for finding method resolution orderings is not a topic for this text, but is described by Python's primary author with a reference to the original paper.

2.5.8 The Role of Objects

The Python object system is designed to make data abstraction and message passing both convenient and flexible. The specialized syntax of classes, methods, inheritance, and dot expressions all enable us to formalize the object metaphor in our programs, which improves our ability to organize large programs.

In particular, we would like our object system to promote a *separation of concerns* among the different aspects of the program. Each object in a program encapsulates and manages some part of the program's state, and each class statement defines the functions that implement some part of the program's overall logic. Abstraction barriers enforce the boundaries between different aspects of a large program.

Object-oriented programming is particularly well-suited to programs that model systems that have separate but interacting parts. For instance, different users interact in a social network, different characters interact in a game, and different shapes interact in a physical simulation. When representing such systems, the objects in a program often map naturally onto objects in the system being modeled, and classes represent their types and relationships.

On the other hand, classes may not provide the best mechanism for implementing certain abstractions. Functional abstractions provide a more natural metaphor for representing relationships between inputs and outputs. One should not feel compelled to fit every bit of logic in a program within a class, especially when defining independent functions for manipulating data is more natural. Functions can also enforce a separation of concerns.

Multi-paradigm languages such as Python allow programmers to match organizational paradigms to appropriate problems. Learning to identify when to introduce a new class, as opposed to a new function, in order to simplify or modularize a program, is an important design skill in software engineering that deserves careful attention.

Continue: 2.6 Implementing Classes and Objects