Chapter

1

Basic Concepts

In traditional programming, we start with a problem to solve. We figure out how to break the problem into smaller parts, then each part into smaller parts still. At each stage we think about how to *do* things. To do something means first doing one thing, then another, then yet another. So, we divide and conquer with an emphasis on *doing*.

In the object-oriented approach, we again start with a problem to solve. We then try to figure out what objects the system has, what their responsibility is, and how they interact. We still divide and conquer, but the emphasis is now on objects and their interactions.

Objects

What is an object? In the real world, we can think of objects as *things*: an apple, a car, a person, a house. In the world of programming, we have objects that model real world objects, and we also have objects that exist to make our life easier, such as an input field, a text string, a collection, a number, a file, a window, or a process. The main qualifications for an object are that it can be named and that it is distinguishable from other types of objects. Let's look at the example of a stock that is traded on a Stock Exchange.

What are the properties of an object? An object usually has some data, and it usually has some behavior — it can do things. Our stock object has some data: it holds the name of the stock, its symbol, the current trading price, the number of shares traded in the most recent trade, and the total number of shares traded today. It also has behavior: it can tell you the most recent trading price and it can accumulate the total of shares traded.

A Stock Exchange application will have many, many stock objects, one for each symbol traded on the exchange. It will interact with the stock objects by sending them *messages*. For example, if it wants to know the current price of a stock, let's say the XYX stock, it would send the `price` message to the XYZ stock object. In response, the stock object looks in its data and returns the current price.

If the Stock Exchange gets information across the wire about a trade involving XYZ stock, it sends the message `traded: aCount price: aPrice` to the XYZ stock object (we'll talk a lot more about messages in Chapter 2, Messages). When the stock object receives the message, it will update its data — in this case it will

update the current price, the number of shares most recently traded, and the total traded for the day. *An object consists of its data plus the messages it understands.*

Our stock object consists of the data: stock name, symbol, current price, most recent trade volume, and trade volume for the day, and the messages: `stockName`, `symbol`, `price`, `lastTraded`, `totalTraded`, and `traded:price:` (plus a few more). The data is stored in *instance variables*. Each stock object has its own copies of the instance variables, so it can store different values for price, symbol, etc.

Encapsulation

When you send the message `traded: aNumberOfShares price: aPrice` to a particular stock object, it updates its instance variables *price*, *lastTraded*, and *totalTraded*. There is *no* other way to change these variables[1]. No other object can see or modify our XYZ stock object's data except by sending messages. Some messages will cause the stock object to update its data, and some messages will cause it to return the values. But no other object can access the data directly. The data is *encapsulated* in the object Figure 1-1 shows a representation of this encapsulation.

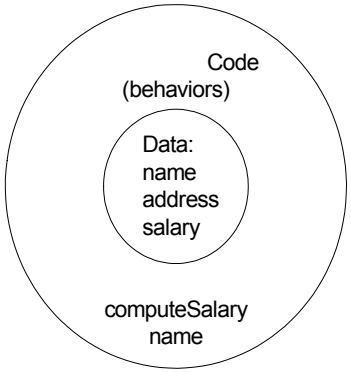

Figure 1-1.
Encapsulation.

The fact that the data is encapsulated means that we can change the way that it is stored. As long as we retain the public interface that we have defined — the messages to which the object responds, we can do what we like internally. One day we might decide to store the price in dollars and cents, the next day in cents, the next day in eighths of a dollar, and so on. As long as we also modify the way that `price` manipulates the data before returning it, we have maintained the public interface despite changing the object internals.

Classes

The Class as code respository

Suppose our program makes use of two *OrderedCollection*s — collections that store items in the order they were added. One keeps track of phone messages that we have received but not dealt with, and the other keeps track of things we must do — action items. Among others, we will need messages to add items to our collections, and to get the first item from a collection.

[1] Actually there is, but it involves meta-programming.

We don't want to write `add:` and `first` twice, once for each object. Instead, we need a mechanism to write the code once so that it can be used by both OrderedCollections. This is where the concept of a *class* comes in. A class is basically a blueprint or a template for what the object looks like: what variables it has and what messages it understands. We define an OrderedCollection class and that's where we write the code for `add:` and `first`. So you can consider the OrderedCollection class to be a repository for the code that is executed when you send these messages. Write the code once, and all OrderedCollections (in our case, the two examples above), can execute the code.

The code is stored in a *method*. When an object receives a message, it executes a *method* with the same name. So, we write methods and the methods are stored with the class. Figure 1-2 shows an example of an Employee class, showing how the class acts as a template and as a code repository.

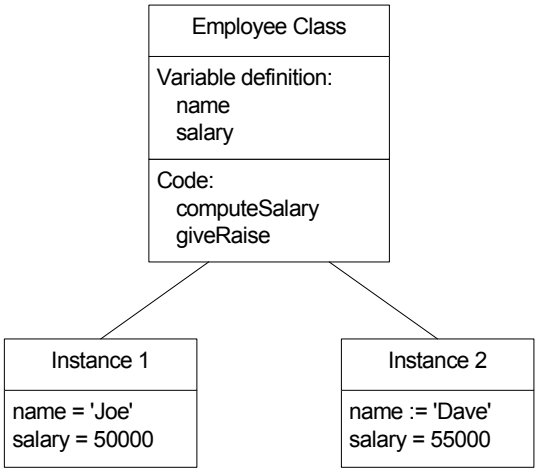

Figure 1-2.
The class as code repository.

The Class as factory

The next question is how our OrderedCollections are created. They don't appear by magic after all. The answer is that we ask the OrderedCollection *class* to create our two *instances* of an OrderedCollection. The following code shows two instance of OrderedCollection being created and assigned to variables (`:=` is the assignment operator).

```
phoneMessages := OrderedCollection new.
actionItems := OrderedCollection new.
```

So, besides being a repository for the code, the class is also a factory. Just as a car factory produces cars, an OrderedCollection factory (class) produces OrderedCollections. The factory contains blueprints for creating objects, and a template for what the object looks like — for the data it contains.

Now let's step back a moment and look at the code above which creates the instances of OrderedCollection. Notice that we send the `new` message to OrderedCollection. Remember that we get objects to do things by sending them messages. It looks remarkably like the OrderedCollection class is also an object, and in fact, this is the case. Not only are our individual instances of an OrderedCollection objects, but so is the factory that creates them. We give the name *class* to the factory object, and the name *instance* to each object that the factory creates. So in our example we have an OrderedCollection class which creates two instances of an OrderedCollection.

Because the class contains a template for individual instances of OrderedCollection, each OrderedCollection has its own copies of the instance variables firstIndex and lastIndex. And because the class is the repository for the code, each instance of OrderedCollection uses the code from the class.

The Class as abstractor

A class abstracts out the common behavior and common data of the objects that it creates. It provides a place where the common behavior and common instance variables can be defined. The instance variables are simply slots; no data is contained in them until instances of the class are created. *A class is a factory with blueprints for the instances it creates. It is also a code repository.*

Inheritance

Now let's look at another type of collection — a *SortedCollection*. Our action item list is better represented by a SortedCollection because we'd prefer to have all the high priority items appear before the low priority ones.

The big difference between a SortedCollection and an OrderedCollection is that in the former the items are sorted based on rules that you can specify. However, a lot of behavior is similar and therefore a lot of the code should be identical. It would be a shame to have to duplicate all the OrderedCollection code for a SortedCollection. Not only would it be a lot of work, but it would be a maintenance nightmare to also update the SortedCollection code if you make changes to OrderedCollection.

We would like an instance of SortedCollection to use the code that is already written for OrderedCollection, so that if the code changes, SortedCollections also get the changes. We would like to share the code where it makes sense to share, such as finding the first or last item in the collection. We'd like to have different code where the behavior is different, such as when adding an item to the collection. We would like a SortedCollection to *inherit* the behavior of an OrderedCollection where they are similar.

Fortunately we can do this by setting up an inheritance relationship between the classes. In the Object Oriented world, we can say that one class is a *subclass* of another class. So in our example, SortedCollection is a subclass of OrderedCollection. This allows SortedCollection to inherit all the code and instance variables of OrderedCollection. For example, if you want to loop through (iterate over) all the items in a SortedCollection, you send it the `do:` message, which is defined in OrderedCollection. SortedCollection inherits the code and its instance does exactly what an instance of OrderedCollection would do.

If you didn't write any specific code for a SortedCollection, it would inherit everything that is written for OrderedCollection. In fact, if we don't change some behavior, there's no point in having a new class at all. Fortunately, SortedCollections do have some different behavior. There are two types of different behavior. First, some messages need to do different things. For example, sending the `add:` message should add an object to the end of an OrderedCollection, but a SortedCollection should add the object in a position based on the sorting rule for the collection. If we do:

```
orderedCollection := OrderedCollection new.
orderedCollection add: 'xyz'.
orderedCollection add: 'def'.
orderedCollection add: 'abc'.
```

and inspect the collection, the strings will be ordered 'xyz', 'def', 'abc', in the order we added them. On the other hand, if we do:

```
sortedCollection := SortedCollection new.
sortedCollection add: 'xyz'.
sortedCollection add: 'def'.
sortedCollection add: 'abc'.
```

and inspect the collection, the strings are ordered 'abc', 'def', 'xyz', in the correct sort sequence for strings. So, in a SortedCollection we don't want to inherit our superclass's code for `add:`. Instead, we write our own `add:` method and override the one defined in OrderedCollection.

The second way we want different behavior is to add behavior — to do something that our superclass doesn't do. For example, we need to be able to specify the sorting algorithm that should be used by an instance of SortedCollection. We add behavior very easily, simply by writing a new method for SortedCollection. In the example of a sort algorithm, we write the `sortBlock:` method which stores the new algorithm for future additions and also resorts the collection according to the new algorithm. Figure 1-3 shows an example of inherited, overridden, and added methods.

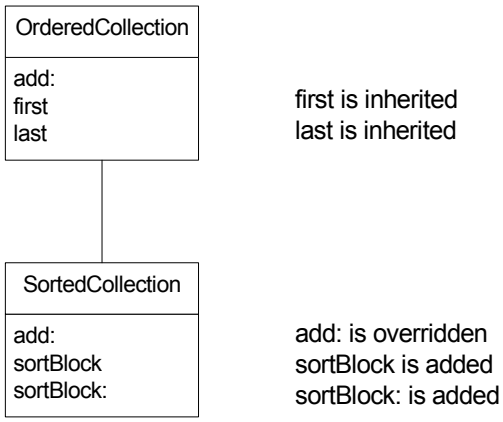

Figure 1-3.
Inheritance.

Polymorphism

Remember the `add:` message that is different for OrderedCollection and SortedCollection? There are other types of collection such as *Set*, *Bag*, and *LinkedList*, each of which implements its own version of `add:`.

What this means is that you can have a collection and not care much what type of collection it is; you simply send it the `add:` message and it will add an object to itself in the appropriate way. Another example might be a window that displays graphical objects. Rather than have the window know how to display a circle and a square, the window would simply send a message to the graphical object (for example, `graphicalObject displayYourselfOn: self`). The graphicalObject might be a square or circle, but the window doesn't care. It simply sends the same message regardless, and relies on the graphicalObject knowing how to display itself. In a procedural language you might write something like:

```
if (graphicalObject isSquare)
    displaySquare (graphicalObject)
else if (graphicalObject isCircle)
    displayCircle (graphicalObject)
```

Using polymorphism, we might simply write:

```
graphicalObject displayYourselfOn: self.
```

To use another example, we might have an Office object which sees Person objects coming in to work. One approach would be for the Office object to ask the Person object what kind of person it is. If it is a Programmer object, the Office object would tell it to start programming (`programmer startProgramming`). If it is a Receptionist object, the Office object would tell it to answer the phones (`receptionist answerPhones`). If it is a Manager object, the Office object would tell it to shuffle papers (`manager shufflePapers`).

This approach has the disadvantage of having a lot of code whose sole purpose is to check what type of Person object just walked in, then having to modify this code when a new type of person is added to the program. The better approach is for the Office object to have no interest in the type of person that walked in, and to simply tell the person to get to work (`person getToWork`). If a Programmer object receives this message, it will start programming; If a Receptionist object gets the message, it will start answering phones; if a Manager object gets the message, it will start shuffling papers. Now when you add a new type of person object, you just have to make sure that it responds to the `getToWork` message and does the appropriate thing. We have put the responsibility where it belongs. Figure 1-4 shows an example of polymorphism.

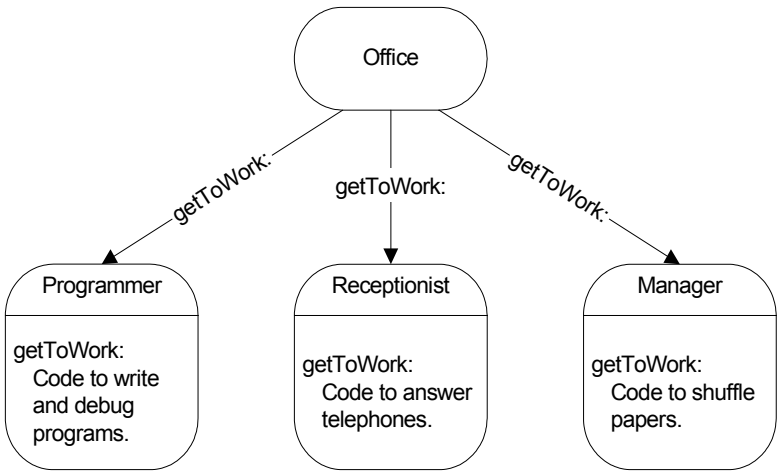

Figure 1-4.
Polymorphism.

This ability to have the same message understood differently by many objects means that we can get out of *decision making* mode and into *commanding* mode. This is a fundamental difference between procedural thinking and object-oriented thinking. Interpreting the the same message differently is called *polymorphism*. It works because of the difference between a *message* and a *method*. When a message is sent to an object, the object looks up the message name (*selector* in Smalltalk terms) in the list of messages it responds to. Associated with the message selector is a method — some lines of code that will be executed. So, the same message selector will be associated with different code for different classes.

To me, the defining feature of object-oriented programming is the ability to simply tell an object to do something, rather than getting information then doing different things based on the information. Polymorphism is a key part of this ability. We'll go into some explicit techniques in Chapter 28, Eliminating Procedural Code.

Abstract Superclasses

Now let's extend the idea of inheritance. OrderedCollection is a subclass of *SequenceableCollection*. Other subclasses of SequenceableCollection include *ArrayedCollection* and *LinkedList*. OrderedCollections, Arrays, and LinkedLists have some common behavior which has been coded in the class SequenceableCollection. However, you can't actually create an instance of SequenceableCollection (well you can, but you'll get errors if you try to do anything with it).

The class SequenceableCollection exists as a place to put the code that is common to the classes just mentioned. The common behavior has been abstracted out and placed in an abstract superclass, one that should not be instantiated — i.e., should not create instances of itself. For example, the methods `copyFrom:to:` and `occurrencesOf:` are both written in SequenceableCollection and inherited by its subclasses.

So, an abstract superclass is a class that has no instances of itself, but which exists as a respository for common code. The abstract superclass SequenceableCollection itself has an abstract superclass, *Collection*. Collection is also the superclass for *Set* and *Bag*, collections that don't have the concept of a sequence. Collection provides behavior that is common to all collections, such as `isEmpty`, `collect:`, `do:`, and `includes:`. (Some of the subclasses override these methods to provide a more appropriate implementation for themselves. However, many of the subclasses inherit the behavior directly.) Figure 1-5 shows a small part of the Collection hierarchy.

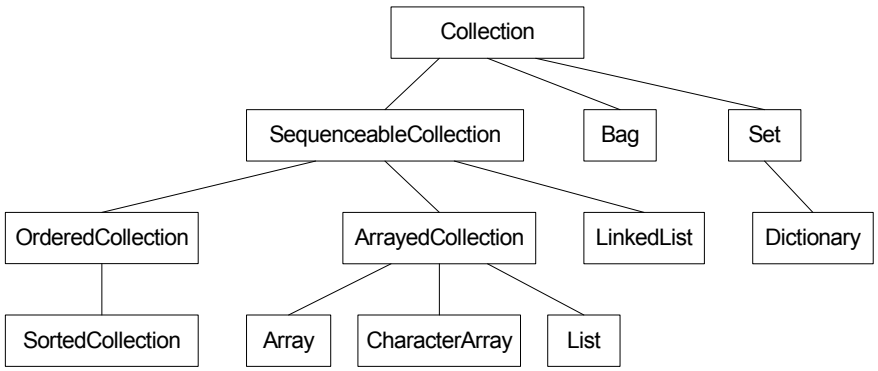

Figure 1-5.
The Collection hierarchy.

Summary

- Objects encapsulate data and behavior (code).
- Classes are factories that have blueprints for creating instances. They are repositories for the code that is executed by their instances.
- Classes are arranged in an inheritance hierarchy that allows objects to inherit behavior (and code) from other classes up their hierarchy chain.
- Work gets done by objects sending messages to other objects, telling them to do something or to return something.
- Many different objects understand the same message but do different things when they receive the message. That is, they execute different methods (polymorphism).