19

Object-Oriented Programming

BECAUSE OBJECT-ORIENTED PROGRAMMING (OR OOP) is so popular, and because many of the widely used OOP languages—such as C++, C#, Java, and Objective-C—are based on the C language, a brief introduction to this topic is presented here. The chapter starts with an overview about the concepts of OOP, and then shows you a simple program in three of the four aforementioned OOP languages (I picked the three that contain the word "C"!). The idea here is not to teach you how to program in these languages or even to describe their main features so much as it is to give you a quick taste.

What Is an Object Anyway?

An object is a *thing*. Think about object-oriented programming as a thing and something you want to do to that thing. This is in contrast to a programming language such as C, more formally known as a procedural programming language. In C, you typically think about what you want to do first (and maybe write some functions to accomplish those tasks), and then you worry about the objects—almost the opposite from object orientation.

As an example from everyday life, assume you own a car. That car is obviously an object—one that you own. You don't have just any car; you have a particular car that was manufactured from the factory, perhaps in Detroit, perhaps in Japan, or perhaps someplace else. Your car has a vehicle identification number (VIN) that uniquely identifies your car.

In object-oriented parlance, *your* car is an *instance* of a car. And continuing with the terminology, *car* is the name of the *class* from which this instance was created. So, each time a new car gets manufactured, a new instance from the class of cars gets created. Each instance of the car is also referred to as an *object*.

Now, your car might be silver, it might have a black interior, it might be a convertible or hardtop, and so on. In addition, there are certain things, or actions, you do with your car. For example, you drive your car, you fill it up with gas, you (hopefully) wash your car, you take it in for service, and so on. This is depicted in Table 19.1.

Table 19.1 **Actions on Objects**

Object	What You Do with It
Your car	Drive it
	Fill it with gas
	Wash it
	Service it

The actions listed in Table 19.1 can be done with your car, and they can also be done with other cars. For example, your sister can drive her car, wash it, fill it up with gas, and so on.

Instances and Methods

A unique occurrence of a class is an instance. The actions that you perform are called *methods*. In some cases, a method can be applied to an instance of the class or to the class itself. For example, washing your car applies to an instance (in fact, all of the methods listed in Table 19.1 are considered instance methods). Finding out how many different types of cars a manufacturer makes applies to the class, so it is a class method.

In C++, you invoke a method on an instance using the following notation:

```
Instance.method ();
```

A C# method is invoked with the same notation as follows:

```
Instance.method ();
```

An Objective-C message call follows this format:

```
[Instance  method]
```

Go back to the previous list and write a message expression in this new syntax. Assume that `yourCar` is an object from the `Car` class. Table 19.2 shows what message expressions might look like in the three OOP languages.

Table 19.2 **Message Expressions in OOP Languages**

C++	C#	Objective-C	Action
`yourCar.drive()`	`yourCar.drive()`	`[yourCar drive]`	Drive your car
`yourCar.getGas()`	`yourCar.getGas()`	`[yourCar getGas]`	Put gas in your car
`yourCar.wash()`	`yourCar.wash()`	`[yourCar wash]`	Wash your car
`yourCar.service()`	`yourCar.service()`	`[yourCar service]`	Service your car

And if your sister has a car, called `suesCar`, for example, then she can invoke the same methods on her car, as follows:

`suesCar.drive()`	`suesCar.drive()`	`[suesCar drive]`

This is one of the key concepts behind object-oriented programming (that is, applying the same methods to different objects).

Another key concept, known as polymorphism, allows you to send the same message to instances from different classes. For example, if you have a `Boat` class, and an instance from that class called `myBoat`, then polymorphism allows you to write the following message expressions in C++:

```
myBoat.service()
myBoat.wash()
```

The key here is that you can write a method for the `Boat` class that knows about servicing a boat, that can be (and probably is) completely different from the method in the `Car` class that knows how to service a car. This is the key to polymorphism.

The important distinction for you to understand about OOP languages versus C, is that in the former case you are working with objects, such as cars and boats. In the latter, you are typically working with functions (or procedures). In a so-called procedural language like C, you might write a function called `service` and then inside that function write separate code to handle servicing different vehicles, such as cars, boats, or bicycles. If you ever want to add a new type of vehicle, you have to modify all functions that deal with different vehicle types. In the case of an OOP language, you just define a new class for that vehicle and add new methods to that class. You don't have to worry about the other vehicle classes; they are independent of your class, so you don't have to modify their code (to which you might not even have access).

The classes you work with in your OOP programs will probably not be cars or boats. More likely, they'll be objects such as windows, rectangles, clipboards, and so on. The messages you'll send (in a language like C#) will look like this:

```
myWindow.erase()                    Erase the window
myRect.getArea()                    Calculate the area of the rectangle
userText.spellCheck()               Spell check some text
deskCalculator.setAccumulator(0.0)  Clear the accumulator
favoritePlaylist.showSongs()        Show songs in favorite playlist
```

Writing a C Program to Work with Fractions

Suppose you need to write a program to work with fractions. Perhaps you need to deal with adding, subtracting, multiplying them, and so on. You could define a structure to hold a fraction, and then develop a set of functions to manipulate them.

The basic setup for a fraction using C might look like Program 19.1. Program 19.1 sets the numerator and denominator and then displays the value of the fraction.

Program 19.1 **Working with Fractions in C**

```
// Simple program to work with fractions
#include <stdio.h>

typedef struct {
    int numerator;
    int denominator;
} Fraction;

int main (void)
{
     Fraction myFract;

     myFract.numerator = 1;
     myFract.denominator = 3;

     printf ("The fraction is %i/%i\n", myFract.numerator, myFract.denominator);

     return 0;
}
```

Program 19.1 **Output**

```
The fraction is 1/3
```

The next three sections illustrate how you might work with fractions in Objective-C, C++, and C#, respectively. The discussion about OOP that follows the presentation of Program 19.2 applies to OOP in general, so you should read these sections in order.

do just that. The first message statement sends the `setNumerator:` message to `myFract`. The argument that is supplied is the value `1`. Control is then sent to the `setNumerator:` method you defined for your `Fraction` class. The Objective-C runtime system knows that it is the method from this class to use because it knows that `myFract` is an object from the `Fraction` class.

Inside the `setNumerator:` method, the single program line in that method takes the value passed in as the argument and stores it in the instance variable `numerator`. So, you have effectively set the numerator of `myFract` to `1`.

The message that invokes the `setDenominator:` method on `myFract` follows next, and works in a similar way.

With the fraction being set, Program 19.2 then calls the two getter methods `numerator` and `denominator` to retrieve the values of the corresponding instance variables from `myFract`. The results are then passed to `printf` to be displayed.

The program next invokes the `print` method. This method displays the value of the fraction that is the receiver of the message. Even though you saw in the program how the numerator and denominator could be retrieved using the getter methods, a separate `print` method was also added to the definition of the `Fraction` class for illustrative purposes.

The last message in the program

```
[myFract free];
```

frees the memory that was used by your `Fraction` object.

Defining a C++ Class to Work with Fractions

Program 19.3 shows how a program to implement a `Fraction` class might be written using the C++ language. C++ has become an extremely popular programming language for software development. It was invented by Bjarne Stroustroup at Bell Laboratories, and was the first object-oriented programming language based on C—at least to my knowledge!

Program 19.3 **Working with Fractions in C++**

```
#include <iostream>

class Fraction
{
 private:
    int numerator;
    int denominator;

 public:
    void setNumerator (int num);
    void setDenominator (int denom);
    int  Numerator (void);
```

Program 19.3 **Continued**

```
    int  Denominator (void);
    void print (Fraction f);
};

void Fraction::setNumerator (int num)
{
    numerator = num;
}

void Fraction::setDenominator (int denom)
{
    denominator = denom;
}

int Fraction::Numerator (void)
{
    return numerator;
}

int Fraction::Denominator (void)
{
    return denominator;
}

void Fraction::print (void)
{
    std::cout << "The value of the fraction is " << numerator << '/'
              << denominator << '\n';
}

int main (void)
{
    Fraction  myFract;

    myFract.setNumerator (1);
    myFract.setDenominator (3);

    myFract.print ();

    return 0;
}
```

Program 19.3 **Output**

```
The value of the fraction is 1/3
```

The C++ members (instance variables) `numerator` and `denominator` are labeled `private` to enforce data encapsulation; that is, to prevent them from being directly accessed from outside the class.

The `setNumerator` method is declared as follows:

```
void Fraction::setNumerator (int num)
```

The method is preceded by the notation `Fraction::` to identify that it belongs to the `Fraction` class.

A new instance of a `Fraction` is created like a normal variable in C, as in the following declaration in `main`:

```
Fraction  myFract;
```

The numerator and denominator of the fraction are then set to 1 and 3, respectively, with the following method calls:

```
myFract.setNumerator (1);
myFract.setDenominator (3);
```

The value of the fraction is then displayed using the fraction's `print` method.

Probably the oddest-appearing statement from Program 19.3 occurs inside the `print` method as follows:

```
std::cout << "The value of the fraction is " << numerator << '/'
          << denominator << '\n';
```

`cout` is the name of the standard output stream, analogous to `stdout` in C. The `<<` is known as the *stream insertion operator*, and it provides an easy way to get output. You might recall that `<<` is also C's left shift operator. This is one significant aspect of C++: a feature known as *operator overloading* that allows you to define operators that are associated with a class. Here, the left shift operator is overloaded so that when it is used in this context (that is, with a stream as its left operand), it invokes a method to write a formatted value to an output stream, instead of trying to actually perform a left shift operation.

As another example of overloading, you might want to override the addition operator `+` so that if you try to add two fractions together, as in

```
myFract + myFract2
```

an appropriate method from your `Fraction` class is invoked to handle the addition.

Each expression that follows the `<<` is evaluated and written to the standard output stream. In this case, first the string `"The value of the fraction is"` gets written, followed by the fraction's numerator, followed by a `/`, the fraction's denominator, and then a newline character.

The C++ language is rich with features. Consult Appendix E, "Resources," for recommendations on a good tutorial.

Note that in the previous C++ example, the getter methods `Numerator ()` and `Denominator ()` were defined in the `Fraction` class but were not used.