Stephen G. Kochan

Updated for **Xcode 5** and **iOS 7**

Programming in Objective-C

Sixth Edition

Programming in Objective-C

Sixth Edition

```
{
   // To add two fractions:
   // a/b + c/d = ((a*d) + (b*c)) / (b * d)

   numerator = numerator * f.denominator + denominator * f.numerator;
   denominator = denominator * f.denominator;
}

-(void) reduce
{
   int  u = numerator;
   int  v = denominator;
   int  temp;

   while (v != 0) {
      temp = u % v;
      u = v;
      v = temp;
   }

   numerator /= u;
   denominator /= u;
}

@end
```

Program 7.4 **Test File:** `main.m`

```
#import "Fraction.h"

int main (int argc, char * argv[])
{
   @autoreleasepool {
      Fraction *aFraction = [[Fraction alloc] init];
      Fraction *bFraction = [[Fraction alloc] init];

      [aFraction setTo: 1 over: 4];   // set 1st fraction to 1/4
      [bFraction setTo: 1 over: 2];   // set 2nd fraction to 1/2

      [aFraction print];
      NSLog (@"+");
      [bFraction print];
      NSLog (@"=");

      [aFraction add: bFraction];
```

```
      // reduce the result of the addition and print the result

      [aFraction reduce];
      [aFraction print];
   }

   return 0;
}
```

Program 7.4 **Output**

```
1/4
+
1/2
=
3/4
```

That's better!

The `self` Keyword

In Program 7.4, we decided to reduce the fraction outside of the `add:` method. We could have done it inside `add:` as well; the decision was completely arbitrary. However, how would we go about identifying the fraction to be reduced? What fraction do we want to reduce anyway? We want to reduce the same fraction that we sent the `add:` message to.

We know how to identify instance variables inside a method directly by name, but we don't know how to directly identify the receiver of the message. Luckily, there is a way to do that.

You can use the keyword `self` to refer to the object that is the receiver of the current message. If inside your `add:` method you wrote

```
[self reduce];
```

the `reduce` method would be applied to the `Fraction` object that was the receiver of the `add:` message, which is what you want. You will see throughout this book how useful the `self` keyword can be, and it's used all the time in iOS programming. For now, you use it in your `add:` method. Here's what the modified method looks like:

```
- (void) add: (Fraction *) f
{
   // To add two fractions:
   // a/b + c/d = ((a*d) + (b*c)) / (b * d)

   numerator = numerator * f.denominator + denominator * f.numerator;
```

```
   denominator = denominator * f.denominator;

   [self reduce];
}
```

After the addition is performed, the fraction is reduced. The `reduce` message gets sent to the receiver of the `add:` message. So, if your test program contains this line of code

```
[aFraction add: bFraction];
```

then `self` refers to `aFraction` when the `add:` method executes, and so that is the fraction that will be reduced.

Allocating and Returning Objects from Methods

We noted that the `add:` method changes the value of the object that is receiving the message. Let's create a new version of `add:` that instead makes a new fraction to store the result of the addition. In this case, we need to return the new `Fraction` to the message sender. Here is the definition for the new `add:` method:

```
-(Fraction *) add: (Fraction *) f
{
   // To add two fractions:
   // a/b + c/d = ((a*d) + (b*c)) / (b * d)

   // result will store the result of the addition
   Fraction   *result = [[Fraction alloc] init];

   result.numerator = numerator * f.denominator +
                           denominator * f.numerator;
   result.denominator = denominator * f.denominator;

   [result reduce];

   return result;
}
```

The first line of your method definition is this:

```
-(Fraction *) add: (Fraction *) f
```

It says that your `add:` method will return a `Fraction` object and that it will take one as its argument as well. The argument will be added to the receiver of the message, which is also a `Fraction`. Note that you need to change your interface section to reflect the fact that the `add:` method now returns a `Fraction` object.

The method allocates and initializes a new `Fraction` object called `result` to store the result of the addition.

The method performs the addition as before, assigning the resulting numerator and denominator to your newly allocated `Fraction` object `result`. After reducing the result, you return its value to the sender of the message with the `return` statement. Note that this time we don't want to reduce the receiver, because we're not changing it. Instead we want to reduce `result`, which is why the message is sent to that object this time around.

Program 7.5 tests your new `add:` method.

Program 7.5 **Test File:** `main.m`

```
#import "Fraction.h"

int main (int argc, char * argv[])
{
   @autoreleasepool {
      Fraction *aFraction = [[Fraction alloc] init];
      Fraction *bFraction = [[Fraction alloc] init];

      Fraction *resultFraction;

      [aFraction setTo: 1 over: 4];   // set 1st fraction to 1/4
      [bFraction setTo: 1 over: 2];   // set 2nd fraction to 1/2

      [aFraction print];
      NSLog (@"+");
      [bFraction print];
      NSLog (@"=");

      resultFraction = [aFraction add: bFraction];
      [resultFraction print];
   }

   return 0;
}
```

Program 7.5 **Output**

```
1/4
+
1/2
=
3/4
3/4
```

Some explanation is in order here. First, you define two `Fraction`s (`aFraction` and `bFraction`) and set their values to `1/4` and `1/2`, respectively. You also define a `Fraction` called `resultFraction`. This variable stores the result of your addition operation that follows.

The following line of code sends the `add:` message to `aFraction`, passing along the `Fraction` `bFraction` as its argument:

```
resultFraction = [aFraction add: bFraction];
```

Inside the method, a new `Fraction` object is allocated and the resulting addition is performed. The result that is stored in the `Fraction` object `result` is then returned by the method, where it is then stored in the variable `resultFraction`.

You may have noticed that we never allocated (or initialized) a `Fraction` object inside main for `resultFraction`; that's because the `add:` method allocated the object for us and then returned the reference to that object. That reference was then stored in `resultFraction`. So `resultFraction` ends up storing the reference to the `Fraction` object that we allocated in the `add:` method. This paragraph is important! It's worth reviewing until you fully understand it.

Extending Class Definitions and the Interface File

You might not need to work with fractions, but these examples have shown how you can continually refine and extend a class by adding new methods. You could hand your `Fraction.h` interface file to someone else working with fractions, and it would be sufficient for that person to be able to write programs to deal with fractions. If that person needed to add a new method, he could do so either directly, by extending the class definition, or indirectly, by defining his own subclass and adding his own new methods. You learn how to do that in the next chapter.

Exercises

1. Add the following methods to the `Fraction` class to round out the arithmetic operations on fractions. Reduce the result within the method in each case:

   ```
   // Subtract argument from receiver
   -(Fraction *) subtract: (Fraction *) f;
   // Multiply receiver by argument
   -(Fraction *) multiply: (Fraction *) f;
   // Divide receiver by argument
   -(Fraction *) divide: (Fraction *) f;
   ```

2. Modify the `print` method from your `Fraction` class so that it takes an additional `BOOL` argument that indicates whether the fraction should be reduced for display. If it is to be reduced (that is if the argument is `YES`), be sure not to make any permanent changes to the fraction itself.

3. Will your `Fraction` class work with negative fractions? For example, can you add `-1/4` and `-1/2` and get the correct result? When you think you have the answer, write a test program to try it.

4. Modify the `Fraction`'s print method to display fractions greater than `1` as mixed numbers. For example, the fraction `5/3` should be displayed as `1 2/3`.

5. Remove the `@synthesize` directive from Program 7.2 and modify the program to handle the new names given to the instance variables by the compiler.

6. Exercise 6 in Chapter 4, "Data Types and Expressions," defined a new class called `Complex` for working with complex imaginary numbers. Add a new method called `add:` that can be used to add two complex numbers. To add two complex numbers, you simply add the real parts and the imaginary parts, as shown here:

   ```
   (5.3 + 7i) + (2.7 + 4i) = 8 + 11i
   ```

   Have the `add:` method store and return the result as a new `Complex` number, based on the following method declaration:

   ```
   -(Complex *) add: (Complex *) complexNum;
   ```

7. Given the `Complex` class developed in exercise 6 of Chapter 4 and the extension made in exercise 6 of this chapter, create separate `Complex.h` and `Complex.m` interface and implementation files. Create a separate test program file to test everything.