Extracted from:

Beginning Mac Programming

Develop with Objective-C and Cocoa

The Pragmatic Programmers

Beginning Mac Programming

Develop with Objective-C and Cocoa

Tim Isted

Edited by Colleen Toporek

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-51-4

ISBN-13: 978-1-934356-51-7

Printed on acid-free paper.

P1.0 printing, March 2010

Version: 2010-5-16

6.3 Class Methods

You have just seen how to refactor the code for circumference generation into a separate method. It seems a little strange, however, to have that method as part of the NotifyingClass object. Mathematical calculations don't seem to have much to do with notifying the user.

Furthermore, this circumference calculation is the kind of thing we might want to reuse in the future. We might have another part of our application (or even a different application altogether) that needs to calculate a circumference given a radius, and it would be strange to have to link to or generate a NotifyingClass object just to perform this calculation.

The alternative is to factor the circumferenceFromRadius: method into a separate utility class, and we'll do that now. We can also avoid having to get hold of an instance of that new class by writing our code into what is known as a *class method*.

Rather than calling the method on an *instance* of the object like this:

```
    [someInstanceOfNotifyingClass circumferenceFromRadius:5.0];
e.g.,
    [self circumferenceFromRadius:5.0];
```

we can just call the method on the *name* of the class itself, like this:

```
    [ClassName circumferenceFromRadius:5.0];
e.g.,
    [MathUtilities circumferenceFromRadius:5.0];
```

If we write a class called MathUtilities, we don't need to create an instance of that class to use its class methods.

Writing a New Class

Let's try this out now by creating a new class in the current project. Right-click (or ^-click) the Classes group in the Xcode project browser for TextApp, choose Add > New File..., and pick *Objective-C class*. Name this new class "MathUtilities," and tell Xcode to generate the necessary .h file for you.

Once the files are created, we can add a new method signature into the interface for the MathUtilities class. Back in Section 4.1, *Defining a New Method*, on page 58, we saw a selection of method signatures from the NSObject interface. Some of these had a + sign at the front, and some had a - sign. It is this + or - that specifies whether a method is a class

method or an instance method. So far we've been working with methods with a - at the front, like this:

```
- (float)generateValue
{
    «generation code»
}
```

which are methods that can be called on an *instance* of the class. To write our math utility method, however, we're going to need to use the + *class* method specifier.

We're now working with two separate classes, each with an interface and an implementation file. Take care to make sure that you put the right code in the right file! Change the interface for the MathUtilities class (MathUtilities.h) by adding the following:

```
@interface MathUtilities : NSObject {

}

+ (float)circumferenceFromRadius:(float)radius;

@end
```

Next, write the method implementation (MathUtilities.m) like this:

```
@implementation MathUtilities

+ (float)circumferenceFromRadius:(float)radius
{
    float circumference = 2 * pi * radius;
    return circumference;
}

@end
```

Finally, we need to change our NotifyingClass code to call this new class method. We should probably remove the old circumferenceFromRadius: code from this class to avoid any confusion, so first remove the method signature in the interface file (NotifyingClass.h) so it looks like this:

```
@interface NotifyingClass : NSObject {
    IBOutlet NSTextView *textView;
    IBOutlet NSTextField *textField;
}

- (IBAction)displaySomeText:(id)sender;
- (float)generateValue;

@end
```

To call our new circumferenceFromRadius: class method, we need to change the call inside the NotifyingClass's generateValue method from [self circumferenceFromRadius:radius] to [MathUtilities circumferenceFromRadius: radius].

It should now be pretty clear why it's so important to follow the naming convention of capitalized class names and noncapitalized variable names. It's possible, for example, to realize instantly that [MathUtilities circumferenceFromRadius:radius] is a call to a class method because of the capitalization of MathUtilities.

Change your implementation for NotifyingClass (NotifyingClass.m) so that it looks something like this:

```
@implementation NotifyingClass

- (IBAction)displaySomeText:(id)sender
{
    float circumference = [self generateValue];

    [textView insertText:[NSString
        stringWithFormat:@"The circumference is: %f\n", circumference]];
}

- (float)generateValue
{
    float radius = [textField floatValue];
    float circumference = [MathUtilities circumferenceFromRadius:radius];
    return circumference;
}

@end
```

With these changes made, let's try to build the project and run the application to make sure everything still works. Sadly, you'll be greeted by an error in Xcode, as shown in Figure 6.4, on the following page stating "error: 'MathUtilities' undeclared"—this is a slightly strange error, but it indicates that Xcode has no idea what a MathUtilities object is within this NotifyingClass file.

To solve this problem, we need to tell Xcode what the *interface* to a MathUtilities object looks like. How do we do this? Well, we need to tell it to look inside the MathUtilities.h *interface file*. Back near the beginning of the book, you might remember that you saw a statement looking like this:

```
#import <Cocoa/Cocoa.h>
```

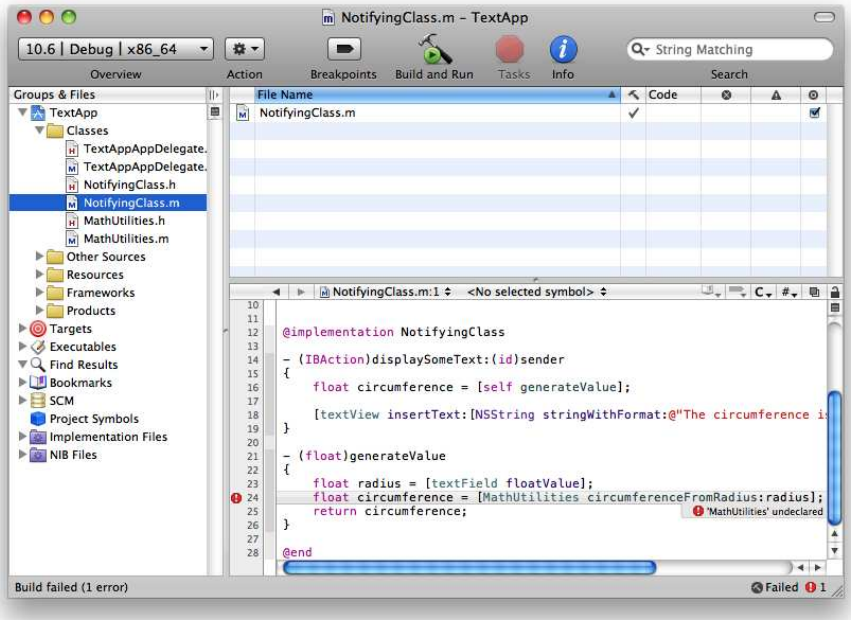

Figure 6.4: The error in Xcode about our use of the MathUtilities class

This appeared in the main.m file for the project. Subsequently, you might have noticed **#import** statements like this one at the top of each file we've worked with. The NotifyingClass.m file, for example, includes the statement #import "NotifyingClass.h" at the top to tell the compiler to include the interface description for the NotifyingClass class.

So, to tell the compiler about the MathUtilities class, we just need to add in an **#import** statement for the MathUtilities.h interface file like this:

```
#import "NotifyingClass.h"
#import "MathUtilities.h"

@implementation NotifyingClass
«implementation continues»
```

Now, when you build the project, the error disappears, and everything behaves as expected.

If we wanted, we could reuse the MathUtilities class in any future project just by including its interface and implementation files and using the relevant **#import** statement.

Class Method Limitations

Class methods are great when you have useful utility code, but since they aren't attached to any particular instance of a class, they obviously have no access to any of the instance variables on a class. If we'd defined a class method for NotifyingClass, for example, that method wouldn't have been able to access the textView or textField outlets, since those outlets have to be set for each particular instance. If you tried to access them, Xcode would complain and refuse to compile your code.

We'll see a number of examples of class methods in later chapters of this book when we use several Apple-provided utility methods for classes in the Cocoa Framework.

6.4 Passing Values by Reference

You might remember from Section 5.4, *Using Memory Addresses for Access*, on page 95 that we mentioned it was possible to allow methods to access variables that aren't currently "in scope" by letting them know the *address* of the variable.

One of the main uses of this is to allow you to return more than one value when a method finishes. Our current generateValue method just returns the calculated circumference. It might be nice to be able to pass back the value that was used to generate the circumference in the first place, but a **return** statement only can be used to return a single value or object.

The solution in this case is to declare a variable in our displaySomeText: method that will eventually hold the radius supplied by the user. We'll pass the address of this variable when we call the generateValue method so that the generateValue method can change the value of the variable held at that address.

Let's start by changing the method signature for generateValue. It needs to accept the address of a *scalar* variable (i.e., a pointer) as its only argument. Since we're dealing with a **float** variable, that's the type of pointer we need to use.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Beginning Mac Programming's Home Page
http://pragprog.com/titles/tibmac
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/tibmac.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)