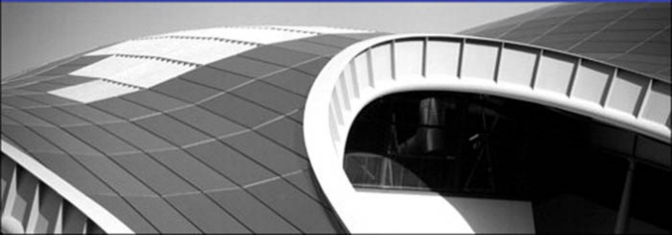
Kevin Hoffman

Windows® Phone 7 for iPhone® Developers

Developer's Library



Windows® Phone 7 for iPhone® Developers

Kevin Hoffman

Addison-Wesley

800 East 96th Street, Indianapolis, Indiana 46240 USA

Windows® Phone 7 for iPhone® Developers

Copyright © 2012 by Pearson Education, Inc.

ISBN-13: 978-0-672-33434-4
ISBN-10: 0-672-33434-8

Library of Congress cataloging-in-publication data is on file.

Printed in the United States of America

First Printing: August 2011

Trademarks

Warning and Disclaimer

Bulk Sales

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editors
Jovana San Nicolas-Shirley and Elaine Wiley

Copy Editor
Barbara Hacha

Indexer
Lisa Stumpf

Proofreader
Seth Kerney

Technical Editors
Bruce Johnson
Nate Dudek

Publishing Coordinator
Cindy Teeters

Cover Designer
Gary Adair

Senior Compositor
Gloria Schurick

❖

I want to dedicate this book to the women in my life:

Angelica, Isabella, and Jerrah.

Behind every good man is an even better woman, and behind every good author is a woman with the patience of a saint and a perpetually running coffeemaker.

❖

Table of Contents

About the Author

Kevin Hoffman (Windsor, CT) is an enterprise programmer who has extensive experience with both Windows Phone 7/Windows Mobile and Apple's iPhone platforms. Currently chief systems architect for Oakleaf Waste Management, he specializes in mobile and cloud development. He writes *The .NET Addict's Blog*, served as editor-in-chief of *iPhone Developer's Journal*, presented twice at Apple's World Wide Developer's Conference, and has authored and co-authored several books, including *WPF Control Development Unleashed: Building Advanced User Experiences* and *ASP.NET 4 Unleashed*.

Acknowledgments

Thanks also go to the staff at Pearson, in particular to Neil Rowe, who has impeccable taste in beer and has somehow managed to put up with me for years.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:	feedback@samspublishing.com
Mail:	Neil Rowe Executive Editor Sams Publishing 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Object-Oriented Programming

Certainly not every good program is object-oriented, and not every object-oriented program is good.

Bjarne Stroustrup

This chapter will cover some of the core concepts of object-oriented programming (OOP) and how they apply to both iPhone and Windows Phone 7 programming. Regardless of the type of application you plan to write, or the platform on which you plan to write it, you will need to utilize and understand some basic object-oriented principles and concepts.

Most developers are already familiar with concepts such as inheritance, contract-based (interface) programming, members, methods, encapsulation, and more. However, those concepts each have different implementations and even different terminology on different platforms. This chapter will help clear things up and show you how the core tenets of OOP are implemented in iOS and WP7.

Why OOP?

If you're reading this book, you probably do not need to be convinced that object-oriented programming is a good thing. However, as the chapter quote from Bjarne Stroustrup so eloquently puts it—not all OOP is a good thing. Like any tool, if used inappropriately, it can make a royal mess of any well-intentioned project.

We create classes as a way to group logically related data and behavior that's all geared toward performing a certain task. When these classes are written properly, they exemplify the benefits of OOP: increased reuse and the creation of testable, reliable, predictable applications that are easy to build, easy to maintain, and easy to change.

On mobile devices, we have classes that encapsulate information and behaviors for all kinds of things from reading GPS coordinates, to displaying text, to accepting input. Without a carefully architected suite of classes that abstract all the capabilities of a device, we would never be able to rapidly and reliably produce software for that device.

The same is true for the applications we build and the code we write. Without our own ability to create reusable classes, even the relatively small amount of code we write for mobile applications would be impossible to maintain and deliver applications on time and on budget.

Building a Class

Throughout the rest of this chapter we're going to be building a class that has progressively more functionality. The goal is to walk you through the process of designing and building a full-featured class for both the iPhone and Windows Phone 7 so that you can compare and contrast and map your existing iPhone skills (if any) to how classes are created in C#.

To demonstrate class creation, we need some behavior and data that we want to model. Throughout the rest of the chapter we'll be working on a class that might show up in a game. Because we are good little programmers that have been following the advice of those who know better, this class will model just behavior, logic, and data encapsulation and will not have anything to do with UI. In short, this class is a pure model and not a view.

The class we're going to build as a starting point is a class that models the logic, data encapsulation, and behavior of an object that can participate in combat in a hypothetical mobile game. We'll call this class a *Combatant*.

To start with, we'll create an empty class. In Listings 3.1 and 3.2, respectively, you can see the code for the Objective-C header (.h) and implementation (.m) files. Listing 3.3 shows this same (utterly empty, and completely useless up to this point) class in C#.

Listing 3.1 **Combatant.h**

```
@interface Combatant : NSObject {

}

@end
```

In the preceding code, you can see the stock contents of an Objective-C header file as they would look immediately after creating an empty class using an Xcode template. There really isn't much to see here. The important thing to keep in mind when learning C# is that C# classes do not separate the code into header and implementation files.

Listing 3.2 **Combatant.m**

```
#import "Combatant.h"

@implementation Combatant

@end
```

The preceding code is the implementation file for an Objective-C class. This is where all the actual implementation code goes, whereas the header file is used to allow other code that references this class to know how the class behaves and what data it exposes.

Listing 3.3 **Combatant.cs**

```
using System;

namespace Chapter3
{
    public class Combatant
    {
    }
}
```

Finally, Listing 3.3 shows the same empty class implemented in C#. If you created your own empty class by adding one to a WP7 project, you probably noticed a whole bunch of extra `using` statements. For clarity, I removed those from Listing 3.3. There's not much to see here, and we haven't gotten to any of the fun stuff. The purpose of this section was to help you get your head around the difference between how classes are stored on disk in iOS and C#, and we will progressively go through more OOP comparisons throughout this chapter.

Encapsulating Data

One of the most important things that any class can do is to encapsulate data. This is one of the main reasons for the original use of object-oriented programming.

Data encapsulation involves a few key concepts that each programming language implements differently:

- Store member variables
- Provide wrappers around accessing those variables
- Add scope and security (for example, read-only) to member variables

Objective-C in its earlier days had somewhat limited support for what most modern programming languages call properties, but now has full and robust capabilities for data encapsulation that rival those of C#.

The best way to see data encapsulation in action is to look at some code with member variables in it. In these next few code listings, we're going to add member variables to the `Combatant` class that will model some of the properties of a combatant that we know the game engine might need, such as hit points, armor class, damage class, and a few other details.

Listing 3.4 shows how we might declare properties in the header file of an iOS application.

Listing 3.4 **Combatant.h with Member Variables**

```
@interface Combatant : NSObject {
}

@property(nonatomic, assign) int maxHitPoints;
@property(nonatomic, assign) int currentHitPoints;
@property(nonatomic, assign) int armorClass;
@property(nonatomic, assign) int damageClass;
@property(nonatomic, retain) NSString *combatantName;
```

This should be fairly familiar to most iOS developers. We have a couple of int-based properties to store values such as hit points and armor class, and there is a string property for storing the combatant name. Using the `@property` syntax, we can specify that the autogenerated accessor for the `combatantName` property will automatically retain the string. In C#, we don't need to worry about retaining strings as a precaution against unintended disposal like we do in Objective-C. Listing 3.5 shows the implementation that automatically synthesizes the getter and setter accessors for the properties declared in Listing 3.4.

Listing 3.5 **Combatant.m with Member Variables**

```
#import "Combatant.h"

@implementation Combatant
    @synthesize maxHitPoints, currentHitPoints, armorClass, damageClass,
combatantName;
@end
```

In the implementation (.m) file, I'm using the `@synthesize` keyword to instruct Xcode that it should autogenerate accessors for those properties. How Xcode does so is governed by the information in the header file in the `@property` declaration. If I wanted to, I could manually override this autogeneration process and supply my own accessor for specific properties, or even replace a get or a set accessor.

Listing 3.6 shows the C# version of the `Combatant` class, including the automatic implementation of the property get and set accessors.

Listing 3.6 **Combatant.cs with Member Variables**

```
using System;
using System.Net;
using System.Windows;

namespace Chapter3
{
    public class Combatant
    {
        public int MaxHitpoints { get; set; }
        public int CurrentHitPoints { get; set; }
        public int ArmorClass { get; set; }
        public int DamageClass { get; set; }
        public string Name { get; set; }|
        public Point Location { get; set; }

        public int HitPointPercent
        {
            get
            {
                double pct =
                 (double)CurrentHitPoints / (double)MaxHitpoints;
                return (int)Math.Floor(pct * 100);
            }
        }
    }
}
```

Listing 3.6 shows the basic `Combatant` class with several public properties that use a shortcut syntax available in C#. This shortcut syntax allows the developer to leave the get and set implementations blank. When these accessors are left blank, the compiler will automatically generate a private member variable to back the public property and do all the required plumbing on behalf of the developer. This type of "automatic property" is also available in iOS.

The `public` keyword in front of each of the property names indicates that the property is visible and accessible from any class in any assembly. If the keyword were changed to `internal`, the properties would be available only to other classes within that assembly. Finally, the `private` keyword indicates that only that class has access to those members. You will learn about the `protected` keyword later in the chapter when we get into inheritance and object hierarchies.

The last property, `HitPointPercent`, shows an example of how you can create a read-only property that computes its value dynamically based on other properties. This shows another example of data encapsulation in that it allows the developer to hide the complexity of a calculation behind a simple property. In this example it's a simple percentage calculation, but you can imagine how this kind of technique can come in handy when modeling complex business objects with very detailed rules and logic. Also note that we have to typecast each of the integer values in the division calculation to a floating point value; otherwise, the `/` operator would assume integer division and return 0 instead of a fractional value.

Adding Behavior

Now that you've got a class that encapsulates data, you want to add behavior to it. Behavior in classes is added in the form of methods, which are functions that execute within the scope of an instance of a class.

Methods in Objective-C typically have their signature defined in the header (.h) file and the implementation defined in the implementation (.m) file. In C#, the method is defined directly on the class and there is no header used for exposing the method signature.

Some of the behavior that we want to add to our `Combatant` class might include attacking another combatant and moving. Listings 3.7 through 3.9 illustrate how we go about adding methods to the class to give our class some behavior. A good general rule is to think of members (or properties) as nouns on a model, whereas behaviors (or methods) should be considered verbs on a model.

Listing 3.7 **Combatant.h with Behavior**

```
@interface Combatant : NSObject {
}

- (void)attack:(Combatant *)target;

@property(nonatomic, assign) int maxHitPoints;
@property(nonatomic, assign) int currentHitPoints;
@property(nonatomic, assign) int armorClass;
@property(nonatomic, assign) int damageClass;
@property(nonatomic, retain) NSString *combatantName;
```

Listing 3.7 shows the header for the `Combatant` class, including the property declarations as well as a method signature for the `Attack` method.

Listing 3.8 **Combatant.m with Behavior**

```
#import "Combatant.h"

@implementation Combatant
    @synthesize maxHitPoints, currentHitPoints, armorClass, damageClass,
combatantName;

- (void)attack:(Combatant *)target
{
    // obviously this should be more complicated...
    target.currentHitPoints -= rand() % 20;
}
@end
```

Listing 3.8 shows the Objective-C implementation of the `Attack` method. This method operates on a supplied instance of another combatant to allow it to do damage to the other target. It's important to keep in mind here that in Listing 3.8's Objective-C code and in Listing 3.9's C# code, both classes are using encapsulated accessors to manipulate the other objects; they are not modifying internal variables directly.

Listing 3.9 **Combatant.cs with Behavior**

```
using System;
using System.Net;
using System.Windows;
using System.Diagnostics;

namespace Chapter3
{
    public class Combatant
    {
        public int MaxHitpoints { get; set; }
        public int CurrentHitPoints { get; internal set; }
        public int ArmorClass { get; set; }
        public int DamageClass { get; set; }
        public string Name { get; set; }
        public Point Location { get; private set; }

        public int HitPointPercent
        {
            get
            {
                double pct =
                 (double)CurrentHitPoints / (double)MaxHitpoints;
                return (int)Math.Floor(pct * 100);
            }
        }

        public void MoveTo(Point newLocation)
        {
            this.Location = newLocation;
            Debug.WriteLine("Combatant {0} just moved to ({1},{2})",
                this.Name,
                this.Location.X,
                this.Location.Y);
        }

        public void Attack(Combatant target)
        {
            Random r = new Random();
            // obviously oversimplified algorithm...
            int damage =
             (this.DamageClass - target.ArmorClass) * r.Next(20);
            target.CurrentHitPoints -= damage;
        }
    }
}
```

In Listing 3.9 there are a couple of minor changes to the access modifiers for some of the class members, and we have the implementation of the `Attack` and `MoveTo` methods. What you might notice is that the `CurrentHitPoints` property now has an access modifier of `internal` for the set accessor. This means that only classes within the same Assembly as `Combatant` are able to modify that property. This allows your "game engine" to freely tweak combatant health but does not allow code outside the core engine to modify that data directly. This forces all changes to hit points to go through only authorized routes.

Additionally, the `Location` property now has a private access modifier. This means that only the `Combatant` class itself can modify its own location. This forces changes to the `Combatant`'s location to go through the `MoveTo` method, which is the only acceptable means for moving a `Combatant`.

The reason I mention these here is because C# has much finer-grained control over access modifiers for methods and members than Objective-C, allowing you to place much more firm control over which code can and cannot affect certain pieces of data. Although this might seem like an unimportant detail, it becomes incredibly important when you are writing code that other developers will consume. An entire type of "accidental side effect" bugs can be eliminated by preventing unwanted changes to your member variables and properties.

Inheritance

In this section of the chapter we're going to take a look at how we can use inheritance to create specialized derivatives of the original `Combatant` class. For example, we might want to create a particular type of combatant that cannot move, such as an automated turret or a stationary cannon. Another kind we might want to create might be combatants that get an extra attack because they are so quick. Finally, we might want to create something completely unusual, such as a drunken attacker who never hits for more than one point of damage at a time.

Figure 3.1 shows the class hierarchy diagram that we want to create from the existing `Combatant` class. As you can see, we want to create three new classes:

- `ReallyDangerousCombatant`
- `StationaryCombatant`
- `DrunkenCombatant`

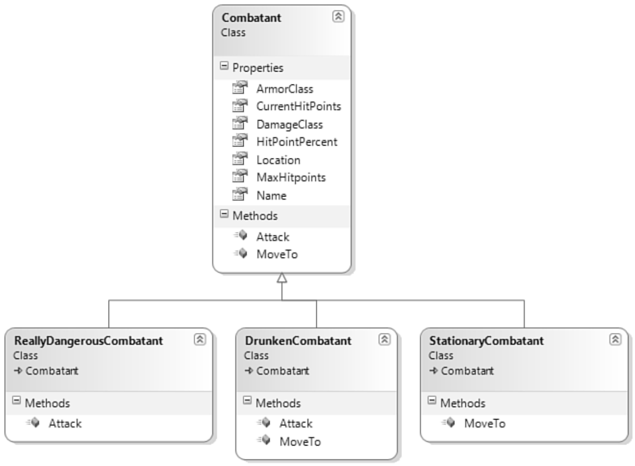


Figure 3.1 Class inheritance hierarchy diagram.

Building these classes in either C# or Objective-C is pretty straightforward. C# gives us a bit more fine-grained control over what the inheriting types can do and see, so we'll make more use of the access modifiers than we do in Objective-C.

In Listings 3.10 and 3.11, I show a sample derivative class called `ReallyDangerousCombatant`, written in Objective-C, that does double damage. This is inheritance in its most basic form—creating a child class that provides behavior that supersedes that of the parent.

Listing 3.10 **ReallyDangerousCombatant.h**

```
#import "Combatant.h"

@interface ReallyDangerousCombatant : Combatant {
}
@end
```

And the implementation of the "really dangerous" combatant class:

Listing 3.11 **ReallyDangerousCombatant.m**

```
#import "ReallyDangerousCombatant.h"

@implementation ReallyDangerousCombatant

- (void)attack:(Combatant *)target
{
    [super attack:target];
    target.currentHitPoints -= 12;
}
```

In Listing 3.11, you can see that the really dangerous combatant first asks its parent class (indicated by the `super` keyword) to attack the target. Then it does its own damage to the target. This really dangerous implementation will always do 12 more damage than a regular combatant would do because of the inheritance hierarchy.

In the interest of saving space and spending more time focusing on the C# implementations of these classes, I won't include the listings for DrunkenCombatant.h, DrunkenCombatant.m, StationaryCombatant.h, and StationaryCombatant.m. The following three listings show the C# implementations for the new derived classes.

Listing 3.12 **ReallyDangerousCombatant.cs**

```
namespace Chapter3
{
    public class ReallyDangerousCombatant : Combatant
    {
        public override void Attack(Combatant target)
        {
            base.Attack(target);

            // attack again for good measure!
            base.Attack(target);
        }
    }
}
```

Listing 3.13 shows how we can use inheritance and child classes to make a combatant that is so drunk it can't possibly win a fight and has an incredibly hard time moving where the game tells it to move:

Listing 3.13 **DrunkenCombatant.cs**

```
using System;
using System.Windows;

namespace Chapter3
{
    public class DrunkenCombatant : Combatant
    {
        public override void Attack(Combatant target)
        {
            target.CurrentHitPoints -= 1; // never do any real damage
        }

        public override void MoveTo(Point newLocation)
        {
            Random r = new Random();
            Point realLocation =
              new Point(r.NextDouble() * 30, r.NextDouble() * 30);

            this.Location = realLocation;
        }
    }
}
```

And now let's take a look at using inheritance to create a combatant that refuses to move at all (such as a stationary turret gun):

Listing 3.14 **StationaryCombatant.cs**

```
using System.Windows;

namespace Chapter3
{
    public class StationaryCombatant : Combatant
    {
        public override void MoveTo(Point newLocation)
        {
            // do nothing
        }

    }
}
```

No matter how many times the game engine might ask this combatant to move, it will do nothing in response.

Inheritance Versus Switching on Data

This is an argument that continues today, no matter your choice of platform or language, so long as it supports OOP. Take the example of a `StationaryCombatant`. What we've done is build a child class such that any time it is asked to move, it simply refuses. Another alternative to this might be to create a Boolean property called `IsStationary`. Then the base class can check the status of the `IsStationary` property in its `Move` method. This prevents the need for creating an entire subclass for the purpose of stationary objects.

This might seem like a simpler solution at first. But this is the top of a slippery slope. Fairly quickly, your simple base class becomes little more than a garbage truck filled with properties and data—a massive storage bin that holds information that might be used only by 1% of all instances of that object. This is just the beginning of the troubles.

Now your simple `Move` method has become convoluted and filled with enormous if statements. In many cases, the logic can become nested and nearly impossible to read. When someone goes to make a change to your `Move` method, it can potentially break functionality for specialized instances of your class (such as for `Combatants` where `IsStationary` is true). Several design patterns are violated by these giant if statements, but in the interest of keeping things simple in this chapter, I won't go into their names or definitions here.

To sum up: If you can solve your specialization issues with inheritance and interfaces (discussed in the next section), that is often a much cleaner, more maintainable and reliable solution than filling a single "bloat master" class with inordinate amounts of properties and logic.

Programming with Contracts

Contracts are different from class implementations. A contract merely defines the requirements of a particular class; it does not actually control the class implementation. To continue the combatant analogy: A `Combatant` base class defines behavior that all child classes can inherit. A `Combatant` contract defines the behavior and data that must be implemented by any class wanting to call itself a combatant.

Let's walk through an example while assuming we're still building a game engine. If we go with a straight inheritance hierarchy, we might be limiting the types of interactions we can model. Assuming single inheritance (which is the case for both Objective-C and C#), anything that can do damage to a player (via the `Attack` method) must inherit from `Combatant`. This presents us with a problem: What about complex inanimate objects with a unique inheritance hierarchy but that cannot inherit from `Combatant`?

Let's say the player is walking through an alley and is struck by a car. Vehicles in this game might require their own inheritance hierarchy, probably starting with a base class such as `Vehicle` or `MovingObject`. Given that we don't have the capability to do multiple inheritance, how do we allow noncombatant objects to do damage to players without messing up the `Combatant` object hierarchy? The answer is contracts.

Contracts are called *protocols* in Objective-C and *interfaces* in C#, but they serve identical purposes. Contracts define a minimum set of required properties or methods that must be implemented by a particular class. They do not enforce any restrictions on inheritance hierarchies. It is critically important here to remember that two classes, each with entirely different inheritance hierarchies, can implement the same interface.

So let's take a look at the `Combatant` class. The `Attack` method does two things (which might give us a clue that we can start refactoring there): It figures out how much damage to do to an opponent, and then it asks the other combatant to take that damage. If we take out the function of taking the actual damage and make that a requirement on an interface called `ITakesDamage`, we start getting some real flexibility in our game engine. This interface has a requirement that anything implementing that interface must implement a method called `TakeDamage`.

Listing 3.15 shows the `ITakesDamage` interface in C#, and Listing 3.16 shows the new `Combatant` class, refactored to separate out the concern of doing damage to be something that satisfies the interface requirement.

Listing 3.15 **ITakesDamage.cs**

```
namespace Chapter3
{
    public interface ITakesDamage
    {
        void TakeDamage(int hitPoints);
    }
}
```

Listing 3.16 shows the refactored `Combatant` class to implement the interface.

Listing 3.16 **Combatant.cs (Refactored to Implement IDoesDamage)**

```
using System;
using System.Net;
using System.Windows;
using System.Diagnostics;

namespace Chapter3
{
    public class Combatant : ITakesDamage
    {
        public int MaxHitpoints { get; set; }
        public int CurrentHitPoints { get; private set; }
        public int ArmorClass { get; set; }
        public int DamageClass { get; set; }
        public string Name { get; set; }
        public Point Location { get; protected set; }

        public Combatant()
        {
            this.CurrentHitPoints = this.MaxHitpoints;
        }

        public int HitPointPercent
        {
            get
            {
                double pct =
                  (double)CurrentHitPoints / (double)MaxHitpoints;
                return (int)Math.Floor(pct * 100);
            }
        }

        public virtual void MoveTo(Point newLocation)
        {
            this.Location = newLocation;
            Debug.WriteLine("Combatant {0} just moved to ({1},{2})",
                this.Name,
                this.Location.X,
                this.Location.Y);
        }

        public virtual void Attack(Combatant target)
        {
            Random r = new Random();
            int damage =
              (this.DamageClass - target.ArmorClass) * r.Next(20);
            target.TakeDamage(damage);
        }


        public void TakeDamage(int hitPoints)
        {
            this.CurrentHitPoints -= hitPoints;
        }
    }
}
```

The new `Attack` method on the `Combatant` class now determines the amount of damage to be done and then calls the `TakeDamage` method to affect the target. Now that the `Combatant` class isn't the only thing in the game engine that can be damaged (anything that implements `ITakesDamage` can now be harmed), we can create classes like the `PincushionTarget` (shown in Listing 3.17), which can be harmed by players but is not a combatant.

Listing 3.17 **PincushionTarget.cs**

```
public class PincushionTarget : ITakesDamage
{
    void TakeDamage(int hitPoints)
    {
       // take points out of pincushion target
    }
}
```

For reference, Listing 3.18 shows what the protocol definition might look like in Objective-C. Objective-C does not use the uppercase "I" prefix naming convention but rather uses the word "Protocol" as a postfix. To achieve a similar goal in Objective-C, we would create a protocol called `TakesDamageProtocol` like the one shown in Listing 3.19. I show you this because protocols are used extensively throughout iOS and in UIKit, so recognizing how those patterns translate into C# patterns can be very useful.

Listing 3.18 **TakesDamageProtocol.h**

```
@protocol TakesDamageProtocol
- (void)takeDamage:(int)hitPoints;
@end
```

Namespaces Versus Naming Conventions

As you have been going through the samples in this chapter, you might have noticed that the C# classes always exist within a namespace. Namespaces in C# are designed specifically to avoid naming collisions as well as to aid in organizing hierarchies of logically connected classes, enums, and structs.

In Objective-C and throughout all of Apple's libraries, either for the iPhone or for traditional Mac development, you will see that there are no namespaces. The decision was made for Objective-C to not support the concept of namespaces (which are available in many OOP languages, including C# and C++). Instead, Apple has opted for a standard by which classes belonging to a particular family, purpose, product, or company all begin with a common two-letter prefix in all capital letters.

For example, a combatant class written by someone named Kevin Hoffman might actually be called `KHCombatant` rather than simply `Combatant`. Further, that same class written by a company called Exclaim Computing might be written as `ECCombatant` or `XCCombatant`.

Naming collisions within iOS applications are rare because you will encounter these collisions only if your application is making use of a library, framework, or class that contains classes named identically to yours. The rare chances of naming collisions in this situation are usually eliminated with the use of the two-letter prefix naming convention.

Extending Other People's Classes

The last topic I want to cover in this chapter is the capability to extend classes written by other developers or companies without actually having the source code to those classes. Keep in mind that in both iOS and C#, extensions to third-party classes can access only class members and methods to which your code would normally have access. In other words, you cannot use class extensions to circumvent encapsulation methods surrounding private or protected members.

C# gives us a facility called *static extensions*, which is roughly analogous to the concept of categories in the Objective-C world. When a static extension to a specific class is in scope of the current code block, that code block can invoke methods on the extension class as if those methods actually belonged to the original class.

Let's assume that we didn't write the `Combatant` class and that it's sealed and we cannot inherit from it. We want to be able to add a method to the `Combatant` class that makes the object move in a square, as if it was square dancing. Perhaps this effect is a custom spell that can be cast on our combatants that gives them an irresistible urge to get their hoedown on.

We can accomplish this custom extension to the `Combatant` class using the code in Listing 3.19.

Listing 3.19 **CombatantSquareDancingExtension.cs**

```
using System;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

namespace Chapter3
{
    public static class CombatantSquareDancingExtension
    {
        public static void SquareDance(this Combatant dancer)
        {
            Point origin = dancer.Location;
            Point step1 =
              new Point(origin.X, origin.Y - 1); // move forward
            Point step2 =
              new Point(step1.X - 1, step1.Y); // move left
            Point step3 =
              new Point(step2.X, step2.Y + 1); // move back
            Point step4 =
              new Point(step3.X + 1, step3.Y); // move right
        }
    }
}
```

Although there are no hard and fast rules about naming conventions used for building extension classes, I like to follow a simple naming scheme: *[OriginalClass][ExtensionDescription]Extension*. The presence of the suffix Extension immediately tells me and any other developer looking at this class that it is not a complete class on its own; rather, it provides additional functionality to some other class. The class being extended is the first section of the name of the extension class. So in our example, a class that extends the `Combatant` class by providing square dancing capabilities would be called `CombatantSquareDancingExtension`.

If the code in Listing 3.19 is in scope, it should be perfectly legal (and possibly amusing) to invoke the `SquareDance` method of any object that is (directly or through inheritance) of type `Combatant`.

Summary

In this chapter, you were given a brief overview of some of the core concepts of object-oriented programming (OOP) and how they apply to the tasks of building iOS and Windows Phone 7 applications using Objective-C and C#, respectively.

Those of you who have been programming for a long time might have found some of these concepts remedial, but the key here was to introduce you to the syntax required for day-to-day programming in an object-oriented programming language such as C#. As you progress through the rest of the book, we will be using things like encapsulation, inheritance, properties, methods, interfaces, and extensions extensively, and if you don't have a firm grasp on those concepts now, the rest of the book will be very difficult to read.

If you have read through this chapter and feel as though you now have a decent understanding of how to write OOP code in C#, you are well prepared to dive deeper into the world of building Windows Phone 7 applications and continue with the rest of the book.