Building XNA 2.0 Games: A Practical Guide for Independent Game Development

ISBN-13 (pbk): 978-1-4302-0979-9

ISBN-13 (electronic): 978-1-4302-0980-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Fabio Claudio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Kevin Goff, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositors: Susan Glinert and Octal Publishing, Inc.
Proofreader: Nancy Sixsmith
Indexer: Carol Burbo
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com

CHAPTER 1

A .NET Snapshot
Coding 101

Prior to writing a game, or any application for that matter, it is extremely important to know how to program! This chapter provides a brief overview of some core programming concepts as they pertain to .NET and the C# language. If you are not familiar with C#, .NET, or object-oriented design, we suggest that you first spend some time reading and exploring other books dedicated to those subjects. If you have some experience developing on the .NET platform, you may wish to skip this chapter entirely. You won't miss much if you just want to develop a sweet game!

The .NET Platform

All of .NET (pronounced "dot net") is called a *platform*, because it is much more than some code, a software development kit (SDK), or a set of languages. The platform consists of a set of goals developed by Microsoft to tackle cross-platform development and create a way to enable rapid application development. In marketing terms, .NET makes developers' jobs easier by letting them focus on implementing functionality rather than developing the core mechanics of an application.

The goal of .NET is to provide a large umbrella for which *managed* languages can be written, compiled, and run with greater ease than ever before. One of the major strengths of the .NET platform is that it is inherently cross-platform.

The platform encompasses a wide range of three-letter acronyms (TLAs), a few of which you should know and understand while programming.

When you or some other developer writes an application or library in a .NET language, it is compiled into an *assembly*. This assembly, no matter what type, can be used by other .NET assemblies. This allows developers to easily reference and use code written in multiple managed languages, such as C#, Visual Basic .NET (VB .NET), or Managed C++. This is due to the fact that the code written in these languages compiles down into the *Common Intermediate Language* (CIL), a low-level language that resembles assembly. CIL is not an assembly language, however; it represents the code itself, rather than CPU-specific instructions. The fact that CIL represents the actual code, instead of optimized and cryptic assembly code, allows it to be decompiled

easily into a high-level language. The CIL is a very important middle step in the platform because it unifies all the languages under the umbrella, providing interoperability, so that the multiple pieces of software can communicate.

How is the CIL used and why is it so inherently cross-platform? Because the CIL provides the middle ground between a high-level language and machine code, which is platform-specific, the .NET platform needs to some layer that can interpret the code and run it. Assemblies written for the platform run under the *Common Language Runtime* (CLR), which compiles and uses CIL code *just-in-time* (JIT) for execution, as illustrated in Figure 1-1. One of the strengths of executing code in this way is that it makes for incredibly easy debugging. It allows a developer to stop execution at any time and run code line by line.

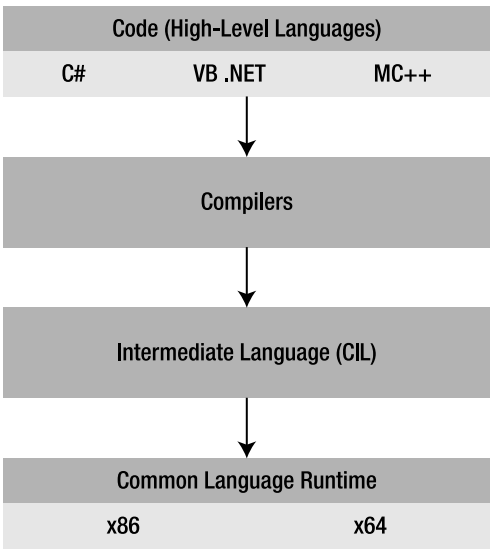

Figure 1-1. *The process of producing a .NET assembly from source code*

What about the languages, then? You now know that languages fit under some umbrella called the CIL and that the intermediate language can be run on a special runtime, but how does this all play out? It turns out that the glue that holds the languages together is yet another TLA. The *Common Type System* (CTS) provides a base layer of types and functionality that is global to all .NET languages. Figure 1-2 provides a high-level view of how the type system and languages are laid out.

The CTS is provided by another assembly, `mscorlib.dll`, which can be referenced in any .NET project. Using Lutz Roeder's .NET Reflector (which can be downloaded from `http://www.aisto.com/roeder/dotnet/`), it is possible to look at what `mscorlib.dll` actually contains. If you do look at it, you will notice all the common types for each language, such as Boolean, Int32, and Byte. Figure 1-3 shows an example of the Boolean type within the CTS library.

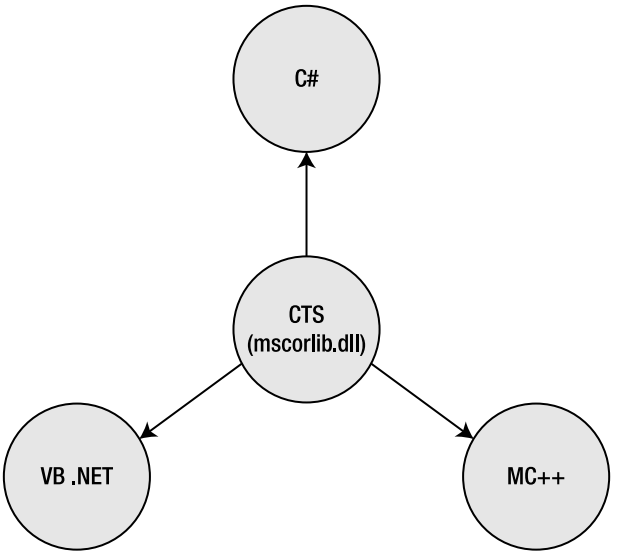

Figure 1-2. *How the CTS relates to your source code*

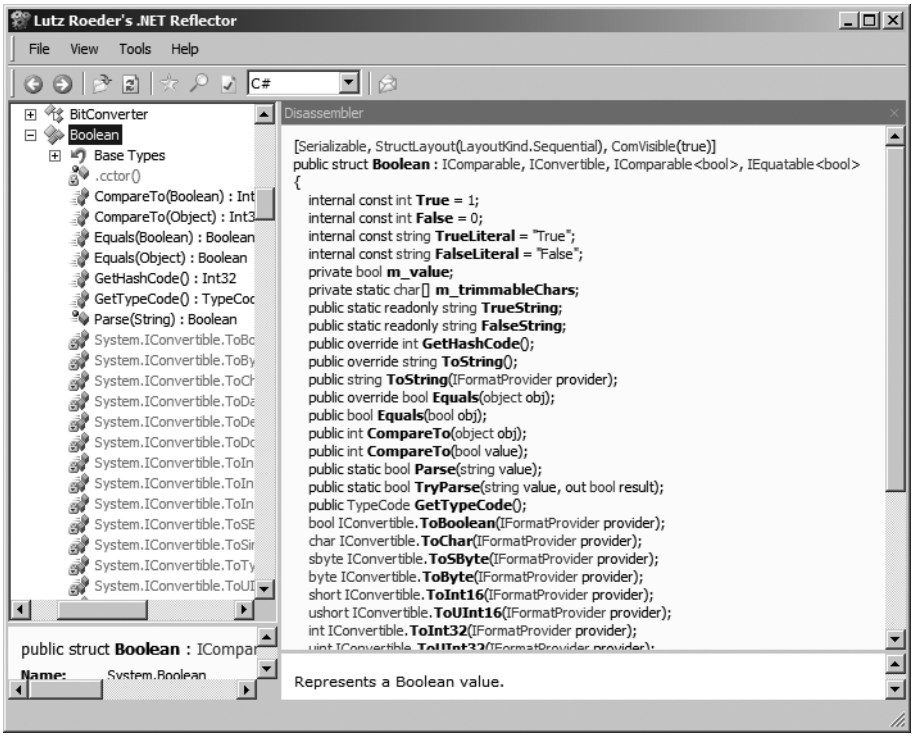

Figure 1-3. *A quick snapshot of mscorlib.dll in .NET Reflector*

Now that you understand what the `mscorlib.dll` library provides, you know one part of what is called the .NET Framework. In general, a framework is a set of libraries composed of types, methods, algorithms, and resources that developers can use to create applications. Inverting the diagram shown in Figure 1-1, you can see that assemblies reference and use each other to actually create a program. These libraries and all this technology are useless without some knowledge of how to use them. This is where object-oriented programming and design come in and guide us to the greener side of application development.

Variables

As developers, we use *variables*, *fields*, *members*, or whatever else you decide to call them to hold stuff for us. When we wish to count from one to ten, or know when a user has clicked something, we use variables to hold the data. Each variable has what we call a *type*, which determines exactly what the variable can hold. For example, a variable of type `int`, or an integer, can hold whole numbers. A variable of type `double` or `float` can hold decimal values. C# is very specific about how we declare and use these variables. For instance, we add two numbers in a certain way:

```
int myValue = 4;
int myValue2 = 3;
int myResult = myValue + myValue2;
```

Notice how we always declare a variable by putting the type first and the name second. It is important to note that the name of a variable can never start with anything but a letter. Thus, the following are illegal declarations:

```
int 3myValue;
int #myValue;
```

After the first character, you can use numbers and underscores. The capitalization does not matter and is done in a certain way for readability. The general convention is to start each word with a capital letter. Here are some valid declarations:

```
int my_VALUE;
int m_value;
int myvalue;
int MyValue;
```

After declaring a variable, we can assign it a value by using the equal sign. The variable we are setting is always to the left of the equal sign, and the value we are generating is to the right. So, in the first example in this section, we are setting the variable `myResult` equal to the sum of `myValue` and `myValue2`.

We can separate the declaration of a variable from when we set it. These types of variables are known as *value types* due to how they are stored in memory. Basically, there are two places a variable can be stored: the stack and the managed heap. Value types, for the most part, are stored on the stack because it is quick and dirty. Bigger types, known as *object* or *reference types*, are stored on the heap and require the use of the `new` operator, as you will see in examples later in this chapter. Table 1-1 shows a short list of common types and their uses.

Table 1-1. *Some Common .NET Types*

Type	Example	Use
`bool`	`bool myBoolean = true;`	True or false; represents a bit (0 or 1)
`byte`	`byte myByte = 3;`	Eight bits in length;. whole number between 0 and 255
`short`	`short myShort = 3;`	Small integers (–32,768 to 32,767); 16 bits in length
`int`	`int myValue = 3;`	Whole numbers; 32 bits in length
`double`	`double myDbl = 3.0;`	Precise real numbers
`float`	`float myFloat = 3.0f;`	Real numbers
`char`	`char myCharacter = '3';`	Single ASCII characters
`string`	`string myString = "333";`	Many characters

What if we want to convert from one variable type to another? A problem exists with going from types like an `int` to a `byte`. Clearly, all the data inside an `int` cannot fit inside a `byte`. Similarly, a `string` cannot just fit inside a `char`, because a `string` is many characters put together. Fortunately, we have type casting to help us fit in as much as possible. To type cast, we put the type we want to cast to in parentheses in front of the variable we wish to cast, as in the following example:

```
int myInteger = 254;
byte myByte = 1;
myByte = myInteger; // This is invalid!
myByte = (byte)myInteger; // This will work!
```

Be warned that when you move from a more precise type like `int` or `double` to a less precise type like `byte` or `float`, you can lose some of your data. Furthermore, when doing mathematical operations, it is possible to overflow or underflow a variable. Let's rework the previous example to demonstrate how this works:

```
int myInteger = 254;
byte myByte = 10;
myByte += (byte)myInteger;
```

In this case, the `byte`, `myByte`, will actually roll past 255 and be set back to zero because 254 + 10 is more than the total amount (255) a `byte` can hold. Similarly, a `char` can hold only one character from a `string`.

You may also have noticed a new way to do addition. It is possible to combine math operations and set operations into a single operator. The previous example uses the += operator because we want to add `myInteger` onto what `myByte` already is.

Playing around with these variables can be interesting, but in order to have some real fun, you need to be able to create and use objects.

Object-Oriented Programming

For now, we are concerned only with C# 2.0, which is available in Visual Studio 2005 and later. This is due to the fact that the XNA Framework does not support C# 3.0 or the .NET 3.5 Framework natively, especially on the Xbox 360, where a custom version of the Compact Framework is used. C# (pronounced "cee sharp") is known as an object-oriented programming (OOP) language because it relies on the ability to format code within sections called *objects*.

You can think of objects as anything you can perform an action on or anything that has an attribute associated with it. Relating to the real world, we can consider physical items as objects. Consider the idea of representing a box as an object. The core idea behind OOP is the notion of relationships. In the case of a cardboard box, we can say that a cardboard box *is a* box. The *is a* relationship tells us that something can be classified as something more generic. This relationship is called *inheritance* and is essential for OOP languages. When one object inherits another, it takes on some of its properties and methods as its own. Here is how our box object looks in C# code:

```
class Box
{
    /// <summary>
    /// Describes the height of the box.
    /// </summary>
    public int Height;

    /// <summary>
    /// Describes the width of the box.
    /// </summary>
    public int Width;

    /// <summary>
    /// Describes the length of the box.
    /// </summary>
    public int Length;
}
class CardboardBox : Box
{
    /// <summary>
    /// Describes the thickness of the cardboard.
    /// </summary>
    public int Thickness;
}
```

This code also contains a second essential part of OOP: the *has a* relationship. In the case of a box, we can say that `Box` *has a* `Height`, `Width`, and `Length`. In the case of a `CardboardBox`, we can say it *has a* `Thickness`, `Height`, `Width`, and `Length`. The *has a* relationship can give us a lot of information about an object or allow us to perform an action on an object.

Boxes are boring unless they have something inside them! Let's say that we ordered something from our favorite online store and it just arrived. How would we open it in code? We can do this by giving the `Box` object a *method*, which is a block of code that can be called from inside or outside the object. Defining a method is incredibly simple. We name it and then define what it does.

Before you read the next block of code, we should cover something that is important to understand from here on out. We can say that any object can be considered as a *type*. A type describes the name of the object, as well as what it contains, what it is, and what it can do. In our example, we say that the `Box` is a class type and has a `Width`, `Height`, and `Length`. When we declare a method, we need to give it what is referred to as a *return type*. When the method is called by code somewhere, it should do some work and then return some value. In the following example, we use a special type called `void`, which describes nothing; that is, the method does not need to return a value.

```
class Box
{
    // ...

    /// <summary>
    /// Opens the box.
    /// </summary>
    public void Open()
    {
    }
}
```

You may have noticed the use of the `public` keyword in the code examples. Another big idea in OOP is the notion of *scope*. In essence, scope defines who can do what from where. In the previous code examples, we have made everything `public` so that code outside the `Box` and `CardboardBox` classes can use the defined items. The following are a few other scopes:

- *Public*: Anyone can call the method or use the member.
- *Private*: Only the class itself can see, call, or use the item.
- *Protected*: The class itself as well as child classes (`CardboardBox` is a child class) can see, call, or use the item.
- *Internal*: Similar to public, but only code within the assembly can see, call, or use the item.

These scopes are very useful when writing code. The following is an example where the `Box` class uses *properties* instead of public fields to hold and maintain data. Properties are a quick way of writing access methods for a private field and can contain any standard code.

```
class Box
{
    // ...
    private int height;
```

```
    /// <summary>
    /// Gets or sets the height of the box.
    /// </summary>
    public int Height
    {
        get { return height; }
        set { height = value; }
    }
}
```

This block of code shows the `Height` property. The `Width` and `Length` properties are written in a similar manner.

Our `Box` class is awesome in that it has the ability to be opened, but it doesn't do anything. Let's add a new property to the class that lets us know whether the box has been opened. We then will change this property in the `Open` method, essentially opening the box!

```
class Box
{
    // ...
    /// <summary>
    /// Describes whether or not the box has been opened.
    /// </summary>
    private bool isOpened = false;

    /// <summary>
    /// Opens the box.
    /// </summary>
    public void Open()
    {
        if (IsOpened)
            return;

        IsOpened = true;
    }

    /// <summary>
    /// Gets whether or not the box has been opened.
    /// </summary>
    public bool IsOpened
    {
        get { return isOpened; }
        private set { isOpened = value; }
    }
}
```

Here, you see a few other OOP concepts. The first is that we can set the scope of both the getter and setter individually. *Getters* and *setters* are used, unsurprisingly, to get and set fields. They come in handy by letting us control how data is accessed. In the preceding example, we can check the `IsOpened` value from outside the class, but if we want to set it, it must be done from within the class.

We check the value of the `IsOpened` property in the `Open()` method, to see if the box has already been opened before trying to open it. The `if` statement uses what is known as *Boolean logic* to decide what to do.

Controlling Flow with Boolean Logic (If Statements)

A Boolean value can be either true or false, on or off. Thus, if the `IsOpened` Boolean is true, the method returns. If the `IsOpened` Boolean is set to false, it opens the box by setting the `IsOpened` Boolean to true. This means that we are able to open the box a maximum of one time.

We can combine several Boolean statements to create larger and more complex statements. This is done with Boolean operators. The two main operators are And, which requires both statements to be true, and Or, which requires at least one of the statements to be true. When considering two statements, A and B, Table 1-2 describes how you can combine them.

Table 1-2. *How Boolean Logic Works*

A	B	A && B (And)	A || B (Or)
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Suppose that we allow a `Box` to take on a new attribute describing whether or not it has a top. Clearly, we cannot open a box that does not have a top, so a check to see if the `Box` is opened *or* has no top is necessary when trying to open it. The following code is based on a `Box` class with a new Boolean property named `HasTop`.

```
class Box
{
    // ...
    public void Open()
    {
        if (IsOpened || !HasTop)
            return;

        IsOpened = true;
    }
}
```

Notice that in this case, we want to check the opposite of what the `HasTop` property provides. We do this using the negation operator (`!`), which makes a true value false and a false value true. We read the `if` statement here as "if the box is opened or does not have a top, we cannot open it."

What if we also want to remove the top when a `Box` is opened? We would need to handle the case where the box isn't opened and does have a top. We can add on to the `if` statement using the keyword `else`, which allows us to extend the statement with alternative checks. Each `if` and `else if` statement is checked in order until an option evaluates to true or no options are valid.

```
class Box
{
    // ...
    public void Open()
    {
        // If it is opened or does not have a top, we can't do anything
        if (IsOpened || !HasTop)
            return;
        // Otherwise, if it is not opened and has a top, "remove" the top.
        else if (!IsOpened && HasTop)
            HasTop = false;

        IsOpened = true;
    }
}
```

Using the Box Object

Now that we have our awesome `Box` and `CardboardBox` objects, we should actually use them. Open a new instance of your favorite C# integrated development environment (IDE) and create a new console project. In Visual Studio 2005, this is done by choosing New Project on the startup screen or by selecting File ➤ New ➤ Project. Figure 1-4 shows the Visual Studio New Project dialog box with the Console Application template selected.

When the project is created, you should see a window to the right titled Solution Explorer. This window shows all the files included in your project and solution. A *solution* is simply a super project that can contain several projects. A *project* is a container for code and is compiled into a single executable or library. If you do not see this window, select View ➤ Solution Explorer.

Open `Program.cs` by double-clicking the file in Solution Explorer. This is the file that will actually run your application. Currently, it contains a single method called `Main` in a class named `Program`.

Now we need to add two files for our objects to the project. This can be done a few ways:

- By right-clicking the project and selecting Add ➤ New Item
- By clicking the Add New Item icon in the toolbar
- By selecting Project ➤ Add New Item

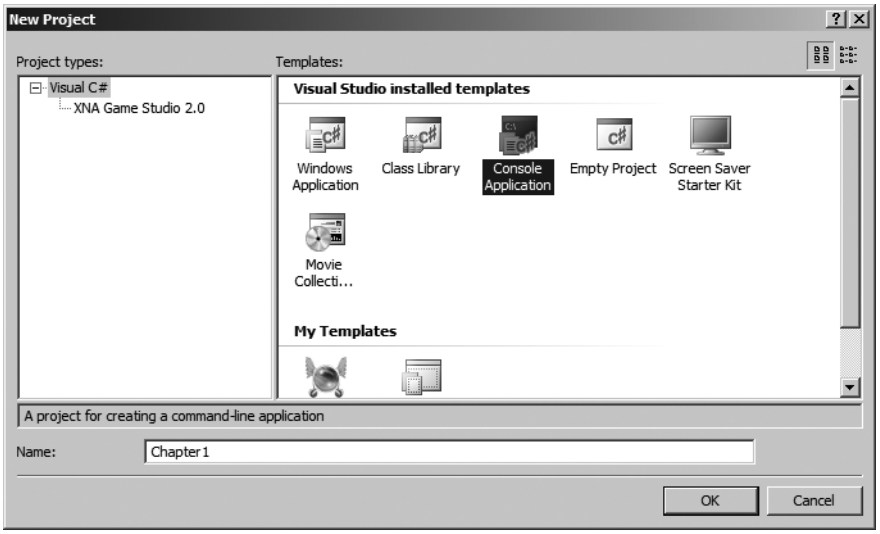

Figure 1-4. *Creating a console application*

You should see a dialog box similar to the one shown in Figure 1-5. Select Class, enter a name (we used `Box.cs`), and select Add. Repeat this process for the `CardboardBox` object.

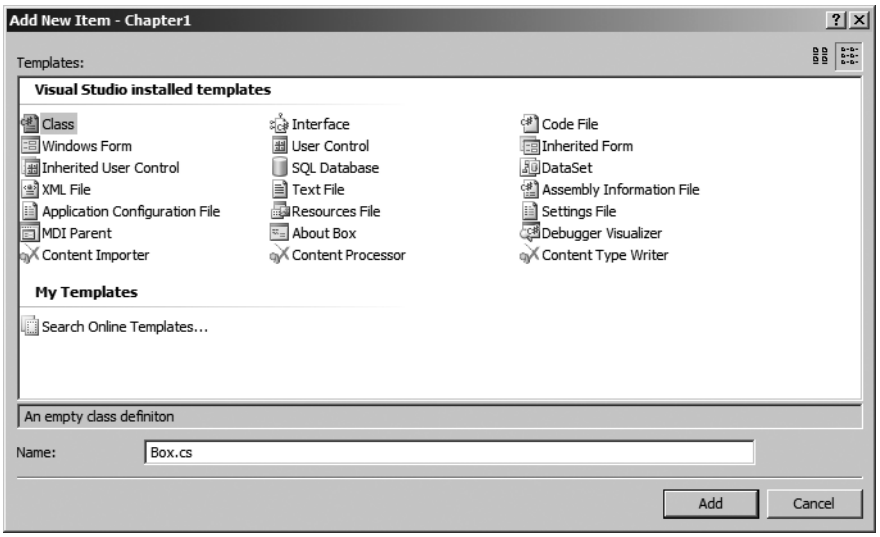

Figure 1-5. *Adding a class file*

Fill in the class files using the code provided previously in this chapter. If you run into trouble, remember that the full source code for all the examples in this book is available from the Source Code/Download section of the Apress web site (`http://www.apress.com`).

Now that we have two files with two objects, we need to write some code to actually do something. Open `Program.cs` again and in the `Main` method, add the following code:

```
class Program
{
    static void Main(string[] args)
    {
        Box box = new Box();
        box.Width = 10;
        box.Height = 10;
        box.Length = 10;

        CardboardBox cbBox = new CardboardBox();
        cbBox.Width = 10;
        cbBox.Height = 10;
        cbBox.Length = 10;
        cbBox.Thickness = 2;

        Console.WriteLine("Box Volume: {0}", box.Width * box.Height * box.Length);
        Console.WriteLine("Cardboard Box Volume: {0}",
            cbBox.Width * cbBox.Height * cbBox.Length - cbBox.Thickness * 6);
    }
}
```

Notice the use of a new keyword, `new`, which creates an instance of each object for us. We can use the object only after we have instantiated it. After creating each object, we set the `Width`, `Height`, and `Length` properties. Because a `Box` is not a `CardboardBox`, we cannot set the `Thickness` property (it doesn't exist). However, because a `CardboardBox` is a `Box`, we can set the `Width`, `Height`, and `Length` properties.

At the end of method, you see `Console.WriteLine(...);`, which is a call to a static method inside an object. A *static* method or property does not require the object to be instantiated in order to use it. Consider it to be an item that is unique and consistent across all instances of the object. In this case, we are using a method to write text to a console window, as shown in Figure 1-6.

After you are finished writing code, select Debug ➤ Start Without Debugging from the menu bar, or press Ctrl+F5.

Debugging

When you start an application without debugging, you are telling Visual Studio to run the application and ignore errors as much as possible. If an error occurs, it will not jump into the code and help you figure out what is going on. It will also ignore any breakpoints in the code.

A *breakpoint* is as simple as it sounds: a point in the code where Visual Studio will stop execution and jump into the code, allowing you to manually step through it line by line. Test this by inserting a breakpoint on the line where the first `Box` is created. You can do this by clicking in the gray area to the left of the text editor; a red circle should appear. Now simply run the program with debugging by selecting Debug ➤ Start Debugging or pressing F5.

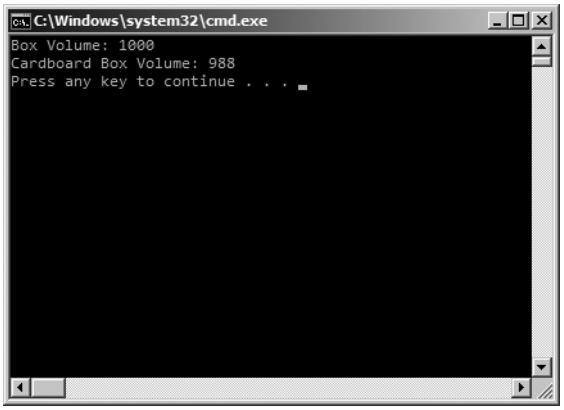

Figure 1-6. *Running our program*

When the application is run, it will immediately jump back into the code and highlight the current line it is on. You are now debugging your code! Move your mouse over various code elements, and you will see a box pop up, telling you more about each item. If you hover over `box` in `Box box = new Box();`, you will see that the pop-up reads `null`. This tells you that the `box` item has not been created yet.

In the Debug menu, look up the shortcut key for Step Over and select it. You will see the code start to execute line by line. As you continue to press the Step Over key, you can hover your mouse over objects and properties and see how they change.

Controlling Flow with Arrays and Looping

Now suppose that we have a lot of boxes to open. If you wanted to buy many copies of this book, each one coming in its own box, you would need a way to open them all easily. For this, we use an array.

An *array*, in its most basic form, is simply a collection of items maintained in one location. We can loop through the array, which allows us to visit each item in order. This can be done via the standard `for` loop:

```
Box[] boxes = new Box[10];
for (int i = 0; i < 10; i++)
    boxes[i] = new Box();

Box[] boxes = new Box[10];
for (int i = 9; i >= 0; i--)
    boxes[i] = new Box();
```

This code creates an array of ten boxes, loops through each item in the array, and instantiates it.

Each spot in the array is said to be at a certain *index*, or position in that collection. In C#, arrays are *zero-based*, meaning that the first element is always index 0 and the last element has an index of *length* – 1.

The `for` loop can be dissected into three distinct elements:

- *Initialization*: Sets up the counter field.
- *Continuing condition*: Provides a Boolean statement that decides whether or not the loop can continue.
- *Incrementing statement*: Moves the counter field toward a value that makes the continuing condition evaluate to false.

Loops are useful anytime many instances of the same object need to be stored. These objects are stored in arrays, lists, or collections.

There are a couple of ways to declare and use arrays. The preceding code blocks show one way. There are also classes that can help maintain collections. Two notable classes are `Queue` and `Stack`, located in the `System.Collections.Generic` namespace. A *queue* is known as a first-in, first-out (FIFO) structure, because the first item to be put in the queue will be the first item taken out. This can be likened to how a line at a coffee shop works: the first customer in line is the first customer served. A *stack* is the opposite of a queue, in that it is a first-in, last-out (FILO). The stack structure can be likened to how a stack of lunch trays works in a cafeteria, where the first tray put on the stack is the last tray to be picked up, and the last tray to be put on the stack is the first to be used.

Instead of the `for` loop, we could also use a `while` or a `do-while` loop. This approach involves setting up a counter to help us index through the array. For an array of ten items, the indices of the items will range from 0 to 9, or one less than the total amount. The difference between a `while` and a `do-while` loop is when the check to exit the loop happens, as demonstrated in the following code:

```
int index = 0;
do{
     boxes[index].Open(); index++;
}while(index < boxes.Length);

int index = 0;
while(index < boxes.Length){
     boxes[index].Open(); index++;
}
```

But `while` loops are used much less often than standard `for` loops. One feature of the C# language that makes the handling of objects in a collection easier is called the `foreach` structure. This looping method allows us to pick out certain types within a collection and use them without needing to worry about counters. Here is a simple example:

```
foreach(Box box in boxes)
     box.Open();
```

It really is that simple! However, there are some caveats to using the `foreach` loop in place of a normal `for` loop. The biggest problem is that you can't modify or remove objects from the collection. In order to do this, you should use a `for` loop as in the prior examples, but loop through the array backward. Having said this, the `foreach` is great for updating objects or drawing them, because it gives us quick and easy access to the objects we need.

There is one major flaw with how we have been creating our arrays so far: they aren't easy to modify. There is no simple way to add new elements or remove old ones. The next section describes how to use generics to create modifiable lists of a certain type.

Using Generics and Events

Generics, or *template classes*, are a way of defining a type such as a class based on another type. This relationship is known as the *of a* relationship, because you can say that you are declaring a variable as a "`List` of `Box` objects," where `List` is the generic class and `Box` is the class you are using.

Let's define our own little box collection using generics. First we need to create a class to handle items in a collection. Start up a new console application in Visual Studio and add a class called `ListBase`. Now we need to add the necessary namespaces and declare the class type. We use the letter `T` to signify the generic type, but you can use any other letter. In fact, it is possible to declare multiple generic types for a class.

```
using System;
using System.Collections.Generic;

namespace Generics
{
    class ListBase<T> : List<T>
    {
    }
}
```

Note that we do not do much with the generic parameter; we simply pass it on up to the base class. Now we need to create some functionality in this class.

One problem with the built-in generic collection types (`List`, `Dictionary`, `Queue`, `Stack`, and so on) is that they do not have events for when an item is added or removed. An *event* is a way of notifying code outside the class that something has happened. For example, in a Windows application, the window fires a `MouseDown` event whenever the user clicks the mouse on the window. This allows us to know about and handle events as they happen. To create an event, we use what is known as a *delegate*, which is nothing more than a function turned into a type. Here is the delegate we will be using (defined in `System.dll`):

```
public delegate void EventHandler(object sender, EventArgs e);
```

You should notice right away that much of this looks like a normal function. It has scope (`public`), a return type (`void`), a name (`EventHandler`), and two parameters (`sender`, `e`). The main difference is the addition of an extra keyword: `delegate`. This tells the compiler that this function is to be used as a type.

Let's use this delegate in our `ListBase` class to declare some events. We do this using another new keyword: `event`.

```
class ListBase<T> : List<T>
{
    public event EventHandler ItemAdded;
    public event EventHandler ItemRemoved;
```

```
    public ListBase()
    {
        ItemAdded += OnItemAdded;
        ItemRemoved += OnItemRemoved;
    }

    ~ListBase()
    {
        ItemAdded -= OnItemAdded;
        ItemRemoved -= OnItemRemoved;
    }

    protected virtual void OnItemAdded(object sender, EventArgs e)
    {
    }

    protected virtual void OnItemRemoved(object sender, EventArgs e)
    {
    }
}
```

A lot of things are going on here. First, we have declared our events with the `new` keyword. Next, in our constructor, we add our methods (event listeners) to the events using the `+=` operator. After this, we declare the deconstructor for the class, which is called when the memory is being freed. It is good practice to always remove listeners from events when you are finished with them to make sure there are no lingering references. Last, but certainly not least, we declare the bodies of our event listeners. Do you notice anything familiar? The parameters and return type match that of the delegate definition. Also note that we have set the scope to `protected` and that it is possible to change the name of the parameters.

Now that we have events, we need to use them. Despite the `ListBox` class having many more methods for adding and removing items, we are going to rewrite only two of them. You can rewrite functionality provided by a base class by using the `new` operator.

```
public new void Add(T item)
{
    base.Add(item);

    if (ItemAdded != null)
        ItemAdded.Invoke(item, EventArgs.Empty);
}

public new bool Remove(T item)
{
    bool returnValue = base.Remove(item);
```

```
    if (ItemRemoved != null)
        ItemRemoved.Invoke(item, EventArgs.Empty);

    return returnValue;
}
```

Here, we have a method, `Add`, that uses the generic type as a parameter. Notice how it is used as a type (like `int`, `bool`, and so on). This is the meat and bones of using generics. After calling the base class's methods, we invoke the event and pass in the item as the first parameter and empty arguments as the second.

Now let's see how to use this class! In `Program.cs`, we will add some code to declare and use a list:

```
static void Main(string[] args)
{
    ListBase<int> myIntegers = new ListBase<int>();
    myIntegers.ItemAdded += OnItemAdded;

    myIntegers.Add(3);
}

static void OnItemAdded(object sender, EventArgs args)
{
    Console.WriteLine("Item added: {0}", sender);
}
```

We declare a variable of a generic type in much the same way as we declare any other variable, except for the addition of the extra type inside the carets. After coding this, the output should be pretty clear. It should say `Item added: 3`.

As an exercise, we suggest playing around with events and the `ListBase` class to add functionality for the other `Add` and `Remove` methods. You can test these by adding event handlers and trying to add and remove items in different ways.

Conclusion

This chapter was intended to give you a little taste of what .NET and C# can do. In the coming chapters, we will be applying the ideas we have discussed here to game development. As you progress through this book, the core concepts you have read about here will become increasingly important. Thankfully, we will not need to introduce many more core concepts and can get right down and dirty with creating a game with the XNA Framework. Now let's make some games!