Table of Contents

[ILDASM and ILASM - Part 2 8](#_Toc380498228)

[Strong naming an assembly - Part 3 10](#_Toc380498229)

[What is GAC. How and when to install an assembly into GAC - Part 4 11](#_Toc380498230)

[How .NET finds the assemblies during program execution - Part 5 13](#_Toc380498231)

[What is DLL HELL in .NET 14](#_Toc380498232)

[Lesson 4: Control Statements - Loops 16](#_Toc380498233)

[**The *while* Loop** 16](#_Toc380498234)

[**The *do* Loop** 17](#_Toc380498235)

[**The *for* Loop** 19](#_Toc380498236)

[**The *foreach* Loop** 20](#_Toc380498237)

[**Summary** 21](#_Toc380498238)

[Lesson 7: Introduction to Classes 22](#_Toc380498239)

[Lesson 8: Class Inheritance 25](#_Toc380498240)

[Lesson 9: Polymorphism 28](#_Toc380498241)

[**Summary** 30](#_Toc380498242)

[Lesson 10: Properties 31](#_Toc380498243)

[**Overview of Properties** 31](#_Toc380498244)

[**Traditional Encapsulation Without Properties** 32](#_Toc380498245)

[**Encapsulating Type State with Properties** 33](#_Toc380498246)

[Creating Read-Only Properties 34](#_Toc380498247)

[Creating a Write-Only Property 36](#_Toc380498248)

[Creating Auto-Implemented Properties 37](#_Toc380498249)

[**Summary** 38](#_Toc380498250)

[Lesson 11: Indexers 38](#_Toc380498251)

[**Summary** 43](#_Toc380498252)

[Lesson 12: Structs 43](#_Toc380498253)

[**What is a *struct*?** 43](#_Toc380498254)

[**Creating a Custom *struct* Type** 44](#_Toc380498255)

[**Using a *struct*** 45](#_Toc380498256)

[**Overloading *struct* Constructors** 46](#_Toc380498257)

[**Adding a Method to a *struct*** 48](#_Toc380498258)

[**Calling a *struct* Method** 49](#_Toc380498259)

[**Summary** 51](#_Toc380498260)

[Lesson 13: Interfaces 51](#_Toc380498261)

[Lesson 14: Introduction to Delegates and Events 54](#_Toc380498262)

[**Delegates** 54](#_Toc380498263)

[**Events** 58](#_Toc380498264)

[**Summary** 60](#_Toc380498265)

[Lesson 15: Introduction to Exception Handling 61](#_Toc380498266)

[Syntax 61](#_Toc380498267)

[Exception Classes in C# 62](#_Toc380498268)

[Handling Exceptions 63](#_Toc380498269)

[Lesson 16: Using Attributes 66](#_Toc380498270)

[**Why Attributes?** 66](#_Toc380498271)

[**Attribute Basics** 67](#_Toc380498272)

[**Attribute Parameters** 69](#_Toc380498273)

[**Attribute Targets** 70](#_Toc380498274)

[**Summary** 71](#_Toc380498275)

[Lesson 17: Enums 71](#_Toc380498276)

[**Enums Defined** 71](#_Toc380498277)

[**Creating an Enum** 72](#_Toc380498278)

[**Using Enums** 73](#_Toc380498279)

[**Enum tricks** 74](#_Toc380498280)

[**Summary** 78](#_Toc380498281)

[Lesson 18: Overloading Operators 78](#_Toc380498282)

[**About Operator Overloading** 78](#_Toc380498283)

[**When Not to Use Operator Overloading** 79](#_Toc380498284)

[**Implementing an Overloaded Operator** 80](#_Toc380498285)

[**Operator Rules** 82](#_Toc380498286)

[**Summary** 82](#_Toc380498287)

[Lesson 19: Encapsulation 83](#_Toc380498288)

[**What is Encapsulation and How Does It Benefit Me?** 84](#_Toc380498289)

[**Type Member Access Modifiers** 84](#_Toc380498290)

[**Opening Type Members to *public* Access** 85](#_Toc380498291)

[**Hiding Type Members with *private* Access** 86](#_Toc380498292)

[**Access for Derived Types with the *protected* Access Modifier** 86](#_Toc380498293)

[**A Quick Word on *internal* and *protected internal* Access Modifiers** 89](#_Toc380498294)

[**Access Modifiers for Types** 89](#_Toc380498295)

[**Summary** 90](#_Toc380498296)

[Lesson 20: Introduction to Generic Collections 91](#_Toc380498297)

[**What Can Generics Do For Me?** 92](#_Toc380498298)

[**Creating Generic *List<T>* Collections** 92](#_Toc380498299)

[**Working with *Dictionary<TKey, TValue>* Collections** 93](#_Toc380498300)

[**Summary** 95](#_Toc380498301)

[Lesson 21: Anonymous Methods 96](#_Toc380498302)

[**How Do Anonymous Methods Benefit Me?** 97](#_Toc380498303)

[**Implementing an Anonymous Method** 97](#_Toc380498304)

[**Using Delegate Parameters with Anonymous Methods** 98](#_Toc380498305)

[**Summary** 99](#_Toc380498306)

[Lesson 22: Topics on C# Type 99](#_Toc380498307)

[**Why Type Safety?** 100](#_Toc380498308)

[**Using the Cast Operator for Conversions** 100](#_Toc380498309)

[**Understanding Reference Types** 101](#_Toc380498310)

[**Understanding Value Types** 102](#_Toc380498311)

[**Reference Type and Value Type Differences** 104](#_Toc380498312)

[**Summary** 104](#_Toc380498313)

[Lesson 23: Working with Nullable Types 104](#_Toc380498314)

[**Understanding the Problem with Value Types and Null Values** 104](#_Toc380498315)

[**Declaring Nullable Types** 105](#_Toc380498316)

[**Working with Nullable Types** 105](#_Toc380498317)

[**Summary** 106](#_Toc380498318)

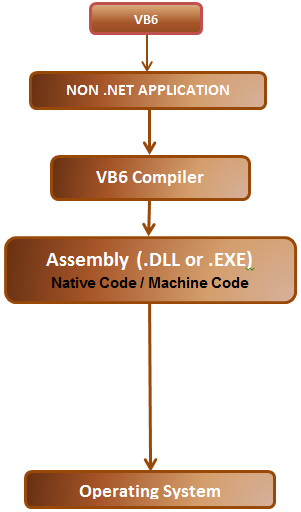
[Dictionary Objects: 106](#_Toc380498319)

[ContainsKey 107](#_Toc380498320)

[KeyValuePair 108](#_Toc380498321)

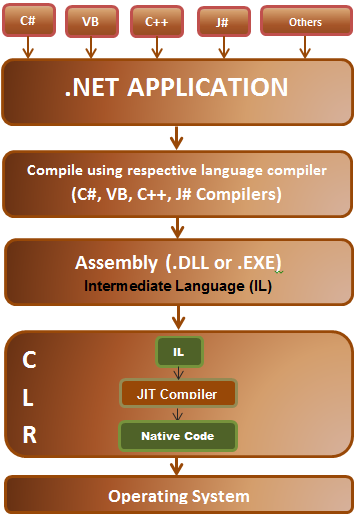
[Keys 110](#_Toc380498322)

.**NET Program Execution - Part 1**

Let us first understand how VB6 or C++ programs (Non Dotnet applications) used to execute.  
We know that computers only understand machine level code. Machine level code is also called as native or binary code. So, when we execute a VB6 or C++ program, the respective language compiler, compiles the respective language source code into native code, which can then be understood by the underlying operating system and hardware. This process is depicted in the image below.  
  
  
  


Native code is specific (native) to the operating system on which it is generated. If you take this compiled native code and try to run on another operating system it will fail. So the problem with this style of program execution is that, it is not portable from one platform to another platform.

Let us now understand, how a .Net program executes. Using dotnet we can create different types of applications. A few of the common types of .NET applications include Web, Windows, Console and Mobile Applications. Irrespective of the type of the application, when you execute any .NET application the following things happens:-  
  
  
1. The .NET application gets compiled into Intermediate language (IL). IL is also referred as Common Intermediate language (CIL) and Microsoft Intermediate language (MSIL). Both .NET and non .NET applications generate an assembly. Assemblies have an extension of .DLL or .EXE. For example if you compile a windows or Console application, you get a .EXE, where as when we compile a web or Class library project we get a .DLL. The difference between a .NET and NON .NET assembly is that, DOTNET Assembly is in intermediate language format where as NON DOTNET assembly is in native code format.  
  
  
2. NON DOTNET applications can run directly on top of the operating system, where as DOTNET applications run on top of a virtual environment called as Common Language Runtime (CLR). CLR contains a component called Just In-Time Compiler (JIT), which will convert the Intermediate language into native code which the underlying operating system can understand.  
   
So, in .NET the application execution consists of 2 steps  
1. Language compiler, compiles the Source Code into Intermediate Language (IL)  
2. JIT compiler in CLR converts, the IL into native code which can then be run on the underlying operating system.  
  
  
This process is shown in the image below.



Since, a .NET assembly is in intermediate Language format and not native code; .NET assemblies are portable to any platform, as long as the target platform has the Common Language Runtime (CLR). The target platform's CLR converts the intermediate Language into native code that the underlying operating system can understand. Intermediate Language is also called as managed code. This is because CLR manages the code that runs inside it. For example, in a VB6 program, the developer is responsible for de-allocating the memory consumed by an object. If a programmer forgets to de-allocate memory, we may run into hard to detect out of memory exceptions. On the other hand a .NET programmer need not worry about de-allocating the memory consumed by an object. Automatic memory management, also known as garbage collection is provided by CLR. Apart, from garbage collection, there are several other benefits provided by the CLR, which we will discuss in a later session. Since, CLR is managing and executing the Intermediate Language, it (IL) is also called as managed code.  
  
  
.NET supports different programming languages like C#, VB, J#, and C++. C#, VB, and J# can only generate managed code (IL), whereas C++ can generate both managed code (IL) and un-managed code (Native code).  
  
  
The native code is not stored permanently anywhere, after we close the program the native code is thrown away. When we execute the program again, the native code gets generated again.  
  
  
.NET program is similar to java program execution. In java we have byte codes and JVM (Java Virtual Machine), where as in .NET we Intermediate Language and CLR (Common Language Runtime)

# ILDASM and ILASM - Part 2

From Part 1 of DotNet basics videos, we understood that, compiling any .NET application would produce an assembly. Assemblies have an extension of .DLL or .EXE. For example if you compile a windows or Console application, you get a .EXE, whereas when we compile a web or Class library project we get a .DLL.   
  
The entire source code of a project is compiled into Intermediate Language and packaged into the assembly. A .NET assembly consists of **Manifest** and **Intermediate language**. Manifest contains metadata about the assembly like the name, version, culture and strong name information. Metadata also contains information about the referenced assemblies. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.

**Some information in the assembly manifest can be modified using attributes. For example to modify the version number follow these steps**  
**1.** Expand the properties folder in solution explorer. Every project in .NET has a properties folder.  
**2.** Open AssemblyInfo.cs file that is present under properties folder.  
**3.** In this file, you should see AssemblyVersion attribute, which is defaulted to 1.0.0.0. Change this to 2.0.0.0 and rebuild the solution.  
**4.** Now open the assembly using ILDASM.exe  
  
  
**To peek inside an assembly with ILDASM follow these steps.**  
**1.** Navigate to Visual Studio Command Prompt (Start -> All Programs -> Microsoft Visual Studio 2010 -> Visual Studio Tools -> Right Click on Visual Studio Command Prompt 2012 and select "Run as Administrator")  
**2.** Once you have the "Visual Studio Command Prompt 2012" open, type in the following command and press enter  
**Ildasm.exe C:\YourDirectoryPath\YourAssembly.exe**  
  
  
This command should open the assembly and you will find the manifest and the types (classes, structs etc..) in the assembly. At the bottom you can see the version of the assembly.  
  
  
**If you want to save the Intermediate Language to a text file.**  
**1.** Select File Menu from the ILDASM tool.  
**2.** Select Dump and you will see "Dump Options Window"  
**3.** Click OK on "Dump Options Window"  
**4.** Now enter the file name of your choice. For this example let's enter sample and save it to the C: drive.  
**5.** Now navigate to C: drive in windows explorer and you should see Sample.il  
**6.** Open Sample.il with notepad and you should see assembly metadata and IL(Intermediate Language).  
  
  
**If you want to rebuild an Assembly from the Sample.il file we can use a tool ILASM.exe**  
**1.** Type the following command in "Visual Studio Command Prompt" and press enter  
**ILASM.exe C:\Sample.il**  
**2.** Now navigate to C: drive in windows explorer and you should see Sample.exe  
  
  
We use ILDASM (Intermediate Language Disassembler) to peek at the assembly manifest and IL. You can also use this tool to export manifest and IL to a text file.  
  
  
We use ILASM.exe (Intermediate Language Assembler) to reconstruct an assembly from a text file that contains manifest and IL

# Strong naming an assembly - Part 3

**Strong naming an assembly or Signing an assembly with strong name.**  
  
  
**In .NET assemblies can be broadly classified into 2 types**  
**1.** Weak Named Assemblies  
**2.** Strong Named Assemblies  
  
  
**An assembly name consists of 4 Parts**  
**1.** Simple textual name.  
**2.** Version number.  
**3.** Culture information (If provided, otherwise the assembly is language neutral)  
**4.** Public key token

We use AssemblyVersion attribute to specify the Assembly version. The default is 1.0.0.0. The version number of an assembly consists of the following four parts:  
**1.** Major Version  
**2.** Minor Version  
**3.** Build Number  
**4.** Revision Number  
  
  
You can specify all the values or you can default the Revision and Build Numbers by using the '\*' as shown below:  
[assembly: AssemblyVersion("2.1.\*")]

AssemblyCulture attribute is used for specifying the culture. By default an assembly is language neutral, as the AssemblyCulture attribute contains empty string. If you specify any string other than an empty string for the culture parameter, the assembly becomes a satellite assembly. In fact, compilers use this attribute to distinguish between main assembly (language neutral) and a satellite assembly. We will talk about satellite assemblies in a later session.  
  
  
We use AssemblyKeyFile attribute to sign the assembly with a strong name. To the constructor of AssemblyKeyFile attribute, we need to pass the path of the key file, that contains the private and public key. To generate the key file

**1. Open Visual Studio Command Prompt**  
**2. Type the command and press enter: sn.exe -k c:\KeyFile.snk**  
  
  
The key file with name KeyFile.snk should be generated in the C: drive. In SN.exe, SN stands for Strong Name. Key files have the extension of .snk  
  
  
Finally, In AssemblyInfo.cs file of the project, specify AssemblyKeyFile attribute as shown below and build the project. This process will strongly name an assembly.  
[assembly: AssemblyKeyFile("KeyFile.snk")]  
  
  
**A strongly named assembly should have all of the following**  
**1.** The textual assembly name.  
**2.** The assembly Version number.  
**3.** The assembly should have been signed with private/public key pair.  
  
  
If the assembly is not signed with private/public key pair, the assembly is weak named and not guaranteed to be unique, and may cause DLL hell. Strong named assemblies are guaranteed to be unique and solve DLL hell. You cannot install an assembly into GAC unless, the assembly is strongly named.  
  
  
**In the upcoming video sessions we will discuss**  
**1.** What is GAC. How and when to install an assembly into GAC?  
**2.** What is DLL HELL?  
**3.** How is DLL HELL solved with .NET

# What is GAC. How and when to install an assembly into GAC - Part 4

**GAC** stands for **Global Assembly Cache** and contains strong named assemblies. Assemblies in the GAC can be shared by all applications running on that machine, without having to copy the assembly locally. It is recommended to install an assembly into GAC, only when required and shared by applications, otherwise they should be kept private. You shouldn't add an assembly into the GAC, if you wish to deploy your application to another machine using XCopy deployment. This is because in XCopy deployment, we only copy the application files to the target machine and not the GAC contents. XCopy deployment is simply copying files from one location to another.

With the introduction of .NET 4.0, we have 2 GAC's. One for DotNet 2.0 to 3.5 assemblies and the other for .NET 4.0 assemblies. The following are the paths for the 2 GAC's  
**1.** C:\Windows\Assembly - For .NET 2.0 - 3.5 assemblies  
**2.** C:\WINDOWS\Microsoft.NET\assembly - For .NET 4.0 assemblies

To install an assembly into the GAC, the assembly must be strongly named, otherwise you get an error stating - Failure adding assembly to the cache: Attempt to install an assembly without a strong name. There are 2 ways to install an assembly into GAC.  
**1.** Simply Drag and Drop  
**2.** Use GacUtil.exe (GAC Utility Tool)  
  
  
To install an assembly using gacutil, use the following command. This command installs SampleAssembly.dll into the GAC. If you have build this project using .NET framwork 4.0 then look in C:\WINDOWS\Microsoft.NET\assembly, else look in C:\Windows\Assembly.  
Gacutil -i C:\SampleProject\SampleAssembly.dll  
  
  
**Note:** If you are using Visual Studio 2010, then by default the target framework for any new project is .NET 4.0. If you want to change the target framework, right click the project and select properties. In the properties window, you can change the target framework version.  
  
  
To uninstall an assembly from the GAC, using GAC utility, use the following command.   
Gacutil -u MyClassLibrary  
  
  
If there are multiple versions of MyClassLibrary assembly, in the GAC, then all these versions will be removed by the above command. If you want to remove only one of the assemblies then specify the full name as shown below.  
gacutil -u ClassLibrary,Version=1.0.0.0,PublicKeyToken=eeaabf36d7783129  
  
  
**Note:** Please make sure there are no spaces between Comma(,) and the words "Version" and PublicKeyToken, otherwise you get an error stating Unknown option: Version=1.0.0.0. Also, don't specify the assembly extension (.dll or .exe) when uninstalling, otherwise the assembly will not be uninstalled. You will just get a message stating Number of assemblies uninstalled = 0

# How .NET finds the assemblies during program execution - Part 5

**1.** .NET figures out what version is needed : Usually the information about the dependant assemblies is present in the application's assembly manifest. CLR checks the application configuration file, publisher policy file(if exists), and machine config file for information that overrides the version information stored in the calling assembly's manifest.   
**2.** .NET searches GAC (Global Assembly Cache) : .NET searches GAC only if the assembly is strongly named.  
**3.** If the assembly is not found in the GAC, and if there is a .config file, then .NET searches the location in the cofiguration file, else .NET searches directory containing the executable (.EXE)  
**4.** If the assembly is not found, the application terminates with error.  
  
  
**Note: Version checking is not done for Weakly Named Assemblies (Assemblies without a strong name)**

# What is DLL HELL in .NET

Let us try and understand DLL HELL problem with an example. Please refer to the image below.

|  |
| --- |
| http://1.bp.blogspot.com/-GdfFIGaLmQs/TeU4FcNPW6I/AAAAAAAAAGk/lPEo1eZNHm0/s1600/DllHell.png |

**1.** I have 2 applications, A1 and A2 installed on my computer.   
  
**2.** Both of these applications use shared assembly shared.dll  
  
**3.** Now, I have a latest version of Application - A2 available on the internet.  
  
**4.** I download the latest version of A2 and install it on my machine.  
  
**5.** This new installation has over written Shared.dll, which is also used by Application - A1.  
  
**6.** Application - A2 works fine, but A1 fails to work, because the newly installed Shared.dll is not backward compatible.  
  
So, DLL HELL is a problem where one application will install a new version of the shared component that is not backward compatible with the version already on the machine, causing all the other existing applications that rely on the shared component to break. With .NET versioning we do not have DLL HELL problem anymore.  
  
[How is the DLL HELL problem solved in .NET, is another very important .net interview question. Click here to read.](http://venkataspinterview.blogspot.com/2011/06/how-is-dll-hell-problem-solved-in-net.html)

In short, the dll hell problem is solved in .NET by signing the shared assemblies with strong name. [Please follow this article, to understand the process of strong naming an assembly.](http://venkataspinterview.blogspot.com/2011/06/what-is-process-for-strong-naming.html)  
  
In dot net all the shared assemblies are usually in the GAC. GAC stands for Global Assembly Cache. The path for GAC is **C:\[OperatingSystemDirectory]\assembly**. For example on my computer the path is **C:\WINDOWS\assembly**. The image below shows the shared assemblies in the GAC.

|  |
| --- |
| http://3.bp.blogspot.com/-3h9wj4-LqZU/TeaQG75xBjI/AAAAAAAAAGo/j6cSV-ZZHsA/s1600/GAC.png |

Only strong named assemblies can be copied into GAC. Strong named assemblies in .NET has 4 pieces in its name as listed below.  
**1.** Simple Textual Name  
**2.** Version Number  
**3.** Culture  
**4.** Public Key Token  
  
  
All these four pieces put together, is called as the fully qualified name of the assembly. In the GAC image above **Accessibility assembly** has a **version of 2.0.0.0**.  
  
  
**Now consider the example below:**  
**1**. I have 2 applications, **Application - A1** and **Application - A2** which relies on the shared assembly **Accessibility.dll (Version 2.0.0.0)** as shown in the image below.

|  |
| --- |
| http://4.bp.blogspot.com/-Wy2MvPARtTA/TeaROJ-kFYI/AAAAAAAAAGs/kQODaRaugfQ/s1600/Dll+Hell+Solved.png |

**2.** Now, I have a latest version of **Application - A2** available on the internet.  
**3.** I download the latest version of **A2** and install it on my machine.   
**4.** This new installation copies a newer version of **Accessibility.dll** into the GAC with version **3.0.0.0.**  
**5.** So, in the GAC we now have **2 versions of Accessibility.dll.**   
**6.** **Application - A1** continues to use **Accessibility.dll (version 2.0.0.0)** and **Application - A2 uses Accessibility.dll (version 3.0.0.0)**   
**7.** So, now the assemblies are able to reside side by side in the GAC. For this reason dot net assemblies are also said to be supporting **side by side execution.**

# Lesson 4: Control Statements - Loops

In the last lesson, you learned how to create a simple loop by using the *goto* statement. I advised you that this is not the best way to perform loops in C#. The information in this lesson will teach you the proper way to execute iterative logic with the various C# looping statements. Its goal is to meet the following objectives:

* Learn the *while* loop.
* Learn the *do* loop.
* Learn the *for* loop.
* Learn the *foreach* loop.
* Complete your knowledge of the *break* statement.
* Teach you how to use the *continue* statement.

**The *while* Loop**

A *while* loop will check a condition and then continues to execute a block of code as long as the condition evaluates to a boolean value of *true*. Its syntax is as follows: *while (<boolean expression>) { <statements> }.* The statements can be any valid C# statements. The boolean expression is evaluated before any code in the following block has executed. When the boolean expression evaluates to *true*, the statements will execute. Once the statements have executed, control returns to the beginning of the *while* loop to check the boolean expression again.

When the boolean expression evaluates to *false*, the *while* loop statements are skipped and execution begins after the closing brace of that block of code. Before entering the loop, ensure that variables evaluated in the loop condition are set to an initial state. During execution, make sure you update variables associated with the boolean expression so that the loop will end when you want it to. Listing 4-1 shows how to implement a *while* loop.

**Listing 4-1. The While Loop: WhileLoop.cs**

using System;  
  
class WhileLoop  
{  
public static void Main()  
{  
int myInt = 0;  
  
while (myInt < 10)  
{  
Console.Write("{0} ", myInt);  
myInt++;  
}  
Console.WriteLine();  
}  
}

Listing 4-1 shows a simple *while* loop. It begins with the keyword *while*, followed by a boolean expression. All control statements use boolean expressions as their condition for entering/continuing the loop. This means that the expression must evaluate to either a *true* or *false* value. In this case we are checking the *myInt* variable to see if it is less than (<) 10. Since *myInt* was initialized to 0, the boolean expression will return *true* the first time it is evaluated. When the boolean expression evaluates to *true*, the block immediately following the boolean expression will be executed.

Within the *while* block we print the number and a space to the console. Then we increment (++) *myInt* to the next integer. Once the statements in the *while* block have executed, the boolean expression is evaluated again. This sequence will continue until the boolean expression evaluates to *false*. Once the boolean expression is evaluated as *false*, program control will jump to the first statement following the *while* block. In this case, we will write the numbers 0 through 9 to the console, exit the *while* block, and print a new line to the console.

**The *do* Loop**

A *do* loop is similar to the *while* loop, except that it checks its condition at the end of the loop. This means that the *do* loop is guaranteed to execute at least one time. On the other hand, a *while* loop evaluates its boolean expression at the beginning and there is generally no guarantee that the statements inside the loop will be executed, unless you program the code to explicitly do so. One reason you may want to use a *do* loop instead of a *while* loop is to present a message or menu such as the one in Listing 4-2 and then retrieve input from a user.

**Listing 4-2. The Do Loop: DoLoop.cs**

using System;  
  
class DoLoop  
{  
public static void Main()  
{  
string myChoice;  
  
do  
{  
// Print A Menu  
Console.WriteLine("My Address Book\n");  
  
Console.WriteLine("A - Add New Address");  
Console.WriteLine("D - Delete Address");  
Console.WriteLine("M - Modify Address");  
Console.WriteLine("V - View Addresses");  
Console.WriteLine("Q - Quit\n");  
  
Console.WriteLine("Choice (A,D,M,V,or Q): ");  
  
// Retrieve the user's choice  
myChoice = Console.ReadLine();  
  
// Make a decision based on the user's choice  
switch(myChoice)  
{  
case "A":  
case "a":  
Console.WriteLine("You wish to add an address.");  
break;  
case "D":  
case "d":  
Console.WriteLine("You wish to delete an address.");  
break;  
case "M":  
case "m":  
Console.WriteLine("You wish to modify an address.");  
break;  
case "V":  
case "v":  
Console.WriteLine("You wish to view the address list.");  
break;  
case "Q":  
case "q":  
Console.WriteLine("Bye.");  
break;  
default:  
Console.WriteLine("{0} is not a valid choice", myChoice);  
break;  
}  
  
// Pause to allow the user to see the results  
Console.Write("press Enter key to continue...");  
Console.ReadLine();  
Console.WriteLine();  
} while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to quit  
}  
}

Listing 4-2 shows a *do* loop in action. The syntax of the *do* loop is *do { <statements> } while (<boolean expression>);*. The statements can be any valid C# programming statements you like. The boolean expression is the same as all others we've encountered so far. It returns either *true* or *false*.

In the *Main* method, we declare the variable *myChoice* of type *string*. Then we print a series of statements to the console. This is a menu of choices for the user. We must get input from the user, which is in the form of a *Console.ReadLine* method which returns the user's value into the *myChoice* variable. We must take the user's input and process it. A very efficient way to do this is with a *switch* statement. Notice that we've placed matching upper and lower case letters together to obtain the same functionality. This is the only legal way to have automatic fall through between cases. If you were to place any statements between two cases, you would not be able to fall through. Another point is that we used the *default:* case, which is a very good habit for the reasons stated in [Lesson 3: Control Statements - Selection](http://csharp-station.com/Tutorial/CSharp/Lesson03).

**The *for* Loop**

A *for* loop works like a *while* loop, except that the syntax of the *for* loop includes initialization and condition modification. *for* loops are appropriate when you know exactly how many times you want to perform the statements within the loop. The contents within the *for* loop parentheses hold three sections separated by semicolons *(<initializer list>; <boolean expression>; <iterator list>) { <statements> }*.

The initializer list is a comma separated list of expressions. These expressions are evaluated only once during the lifetime of the *for* loop. This is a one-time operation, before loop execution. This section is commonly used to initialize an integer to be used as a counter.

Once the initializer list has been evaluated, the *for* loop gives control to its second section, the boolean expression. There is only one boolean expression, but it can be as complicated as you like as long as the result evaluates to *true* or *false*. The boolean expression is commonly used to verify the status of a counter variable.

When the boolean expression evaluates to *true*, the statements within the curly braces of the *for* loop are executed. After executing *for* loop statements, control moves to the top of loop and executes the iterator list, which is normally used to increment or decrement a counter. The iterator list can contain a comma separated list of statements, but is generally only one statement. Listing 4-3 shows how to implement a *for* loop. The purpose of the program is to print only odd numbers less than 10.

**Listing 4-3. The For Loop: ForLoop.cs**

using System;  
  
class ForLoop  
{  
public static void Main()  
{  
for (int i=0; i < 20; i++)  
{  
if (i == 10)  
break;  
  
if (i % 2 == 0)  
continue;  
  
Console.Write("{0} ", i);  
}  
Console.WriteLine();  
}  
}

Normally, *for* loop statements execute from the opening curly brace to the closing curly brace without interruption. However, in Listing 4-3, we've made a couple exceptions. There are a couple *if* statements disrupting the flow of control within the *for* block.

The first *if* statement checks to see if *i* is equal to 10. Now you see another use of the *break* statement. Its behavior is similar to the selection statements, as discussed in [Lesson 3: Control Statements - Selection](http://csharp-station.com/Tutorial/CSharp/Lesson03). It simply breaks out of the loop at that point and transfers control to the first statement following the end of the *for* block.

The second *if* statement uses the remainder operator to see if *i* is a multiple of 2. This will evaluate to *true* when *i* is divided by 2 with a remainder equal to zero, (0). When *true*, the *continue* statement is executed, causing control to skip over the remaining statements in the loop and transfer back to the iterator list. By arranging the statements within a block properly, you can conditionally execute them based upon whatever condition you need.

When program control reaches either a *continue* statement or end of block, it transfers to the third section within the *for* loop parentheses, the iterator list. This is a comma separated list of actions that are executed after the statements in the *for* block have been executed. Listing 4-3 is a typical action, incrementing the counter. Once this is complete, control transfers to the boolean expression for evaluation.

Similar to the *while* loop, a *for* loop will continue as long as the boolean expression is *true*. When the boolean expression becomes *false*, control is transferred to the first statement following the *for* block.

For this tutorial, I chose to implement *break* and *continue* statements in Listing 4-3 only. However, they may be used in any of the loop statements.

**The *foreach* Loop**

A *foreach* loop is used to iterate through the items in a list. It operates on arrays or collections such as ArrayList, which can be found in the System.Collections namespace. The syntax of a foreach loop is *foreach (<type> <iteration variable> in <list>) { <statements> }*. The type is the type of item contained in the list. For example, if the type of the list was *int[]* then the type would be *int*.

The iteration variable is an identifier that you choose, which could be anything but should be meaningful. For example, if the list contained an array of people's ages, then a meaningful name for item name would be age.

The *in* keyword is required.

As mentioned earlier, the list could be either an array or a collection. You learned about arrays in [Lesson 02: Operators, Types, and Variables](http://csharp-station.com/Tutorial/CSharp/Lesson02). You can also iterate over C# generic collections also, described in [Lesson 20: Introduction to Generic Collections](http://csharp-station.com/Tutorial/CSharp/Lesson20).

While iterating through the items of a list with a *foreach* loop, the list is read-only. This means that you can't modify the iteration variable within a *foreach* loop. There is a subtlety here; Later, you'll learn how to create custom types, called class and struct, that can contain multiple fields. You can change the fields of the class or struct, but not the iteration variable for the class or struct itself in a *foreach* loop.

On each iteration through a *foreach* loop the list is queried for a new value. As long as the list can return a value, this value will be put into the read-only iteration variable, causing the statements in the *foreach* block to be executed. When the collection has been fully traversed, control will transfer to the first executable statement following the end of the *foreach* block. Listing 4-4 demonstrates how to use a *foreach* loop.

**Listing 4-4. The ForEach Loop: ForEachLoop.cs**

using System;  
  
class ForEachLoop  
{  
public static void Main()  
{  
string[] names = {"Cheryl", "Joe", "Matt", "Robert"};  
  
foreach (string person in names)  
{  
Console.WriteLine("{0} ", person);  
}  
}  
}

In Listing 4-4, the first thing we've done inside the *Main* method is declare and initialize the *names* array with 4 *strings*. This is the list used in the *foreach* loop.

In the *foreach* loop, we've used a *string* variable, *person*, as the item name, to hold each element of the *names* array. As long as there are names in the array that have not been returned, the *Console.WriteLine* method will print each value of the *person* variable to the screen.

**Summary**

Loops allow you to execute a block of statements repeatedly. C# offers several statements to construct loops with, including the *while*, *do*, *for*, and *foreach* loops. *while* loops execute a block of statements as long as an expression is *true*, *do* loops execute a block of statements at least once and then keep going as long as a condition is *true*, *for* loops execute a block of statements a specified amount of times, and *foreach* loops execute a block of statements for each item in a collection. Normally a block of statements will execute from beginning to end. However, the normal flow of a loop can be changed with the *break* and *continue* statements.

So far, the only method you've seen in this tutorial is the Main method, which is the entry point of a C# application. However, you are probably wanting to write larger programs to test your new knowledge. This requires breaking up the code into methods to keep it organized and logical. For this, I invite you to return for [Lesson 5: Introduction to Methods](http://csharp-station.com/Tutorial/CSharp/Lesson05), where you can learn new techniques of organizing your code.

# Lesson 7: Introduction to Classes

This lesson introduces you to C# Classes. Our objectives are as follows:

* Implement Constructors.
* Know the difference between instance and *static* members.
* Understand Destructors.
* Familiarization with Class Members.

Since the beginning of this tutorial, you have been using classes. By now, you should have a sense of what a *class* is for and how to specify one. This lesson will build upon what you already know and introduce the various *class* members.

Classes are declared by using the keyword *class* followed by the *class* name and a set of *class* members surrounded by curly braces. Every *class* has a constructor, which is called automatically any time an instance of a *class* is created. The purpose of constructors is to initialize *class* members when an instance of the *class* is created. Constructors do not have return values and always have the same name as the *class*. Listing 7-1 is an example of a *class*.

**Listing 7-1. Example C# Classes: Classes.cs**

// Namespace Declaration

using System;  
  
// helper class  
class OutputClass   
{  
string myString;  
  
// Constructor  
public OutputClass(string inputString)   
{  
myString = inputString;  
}  
  
// Instance Method  
publicvoid printString()   
{  
Console.WriteLine("{0}", myString);  
}  
  
// Destructor  
~OutputClass()   
{  
// Some resource cleanup routines  
}  
}  
  
// Program start class  
class ExampleClass   
{  
// Main begins program execution.  
publicstatic void Main()   
{  
// Instance of OutputClass  
OutputClass outCl = new OutputClass("This is printed by the output class.");  
  
// Call Output class' method  
outCl.printString();   
}  
}

Listing 7-1 shows two classes. The top *class*, *OutputClass*, has a constructor, instance method, and a destructor. It also had a field named *myString*. Notice how the *OutputClass* constructor is used to initialize data members of the *class*. In this case, the *OutputClass* constructor accepts a *string* argument, *inputString*. This *string* is copied to the *class* field *myString*.

Constructors are not mandatory, as indicated by the implementation of *ExampleClass*. In this case, a default constructor is provided. A default constructor is simply a constructor with no arguments. However, a constructor with no arguments is not always useful. To make default constructors more useful, you can implement them with initializers. Here is an example:

public OutputClass() : this("Default Constructor String") { }

Imagine this constructor was included in *class* *OutputClass* from Listing 7-1. This default constructor is followed by an initializer. The colon, ":", marks the beginning of the initializer, followed by the *this* keyword. The *this* keyword refers to this particular object. It effectively makes a call to the constructor of the same object it is defined in. After the *this* keyword is a parameter list with a *string*. The action taken by the initializer above is to invoke the *OutputClass* constructor that takes a *string* type as an argument. The initializer helps you to ensure your *class* fields are initialized when a *class* is instantiated.

The example above illustrates how a class can have multiple constructors. The specific constructor called depends on the number of parameters and the type of each parameter.

In C#, there are two types of *class* members, instance and *static*. Instance *class* members belong to a specific occurrence of a *class*. Every time you declare an object of a certain *class*, you create a new instance of that *class*. The *ExampleClass* *Main()* method creates an instance of the *OutputClass* named *outCl*. You can create multiple instances of *OutputClass* with different names. Each of these instances are separate and stand alone. For example, if you create two *OutputClass* instances as follows:

OutputClass oc1 = new OutputClass("OutputClass1");  
OutputClass oc2 = new OutputClass("OutputClass2");

You create two separate instances of *OutputClass* with separate *myString* fields and separate *printString()* methods. On the other hand, if a *class* member is *static*, you can access it simply by using the syntax *<classname>.<static class member>*. The instance names are *oc1* and *oc2*.

Suppose *OutputClass* had the following *static* method:

public static void staticPrinter()   
{  
Console.WriteLine("There is only one of me.");  
}

Then you could call that function from *Main()* like this:

OutputClass.staticPrinter();

You must call *static* class members through their *class* name and not their instance name. This means that you don't need to instantiate a class to use its *static* members. There is only ever one copy of a *static* class member. A good use of *static* members is when there is a function to be performed and no intermediate state is required, such as math calculations. Matter of fact, the .NET Frameworks Base Class Library includes a *Mathclass* that makes extensive use of *static* members.

Another type of constructor is the *static* constructor. Use *static* constructor to initialize *static* fields in a *class*. You declare a *static* constructor by using the keyword *static* just in front of the constructor name. A *static* constructor is called before an instance of a *class* is created, before a *static* member is called, and before the *static* constructor of a derived class (covered in a later chapter). They are called only once.

*OutputClass* also has a destructor. Destructors look just like constructors, except they have a tilde, "~", in front of them. They don't take any parameters and do not return a value. Destructors are places where you could put code to release any resources your class was holding during its lifetime. They are normally called when the C# garbage collector decides to clean your object from memory.

**Note:** You've probably noticed the use of the *public* modifier (an access modifier), meaning that a class member can be accessed from other classes. When used on a class, it means that the class can be accessed by DLLs outside of the Assembly (which is commonly a \*.exe or \*.dll file). [Lesson 19: Encapsulation](http://csharp-station.com/Tutorial/CSharp/Lesson19) discusses access modifiers in more depth.

So far, the only class members you've seen are Fields, Methods, Constructors, and Destructors. Here is a complete list of the types of members you can have in your classes:

* Constructors
* Destructors
* Fields
* Methods
* Properties
* Indexers
* Delegates
* Events
* Nested Classes

Those items not covered in this lesson will be covered in later lessons.

In summary, you can declare instance and *static* constructors. You know how to initialize class fields. When there is no need to instantiate an object, you can create *static* class members. You can also declare destructors for cleaning up resources.

# Lesson 8: Class Inheritance

This lesson teaches about C# Inheritance. Our objectives are as follows:

* Implement Base Classes.
* Implement Derived Classes.
* Initialize Base Classes from Derived Classes.
* Learn How to Call Base Class Members.
* Learn How to Hide Base Class Members.

Inheritance is one of the primary concepts of object-oriented programming. It allows you to reuse existing code. Through effective employment of reuse, you can save time in your programming.

**Listing 8-1. Inheritance: BaseClass.cs**

using System;  
  
public class ParentClass  
{  
public ParentClass()  
{  
Console.WriteLine("Parent Constructor.");  
}  
  
public void print()  
{  
Console.WriteLine("I'm a Parent Class.");  
}  
}  
  
public class ChildClass : ParentClass  
{  
public ChildClass()  
{  
Console.WriteLine("Child Constructor.");  
}  
  
public static void Main()  
{  
ChildClass child = new ChildClass();  
  
child.print();  
}  
}

Output:

Parent Constructor.

Child Constructor.

I'm a Parent Class.

Listing 8-1 shows two classes. The top class is named *ParentClass* and the main class is called *ChildClass*. What we want to do is create a child class, using existing code from *ParentClass*.

First we must declare our intention to use *ParentClass* as the base class of *ChildClass*. This is accomplished through the *ChildClass* declaration public class *ChildClass : ParentClass*. The base class is specified by adding a colon, ":", after the derived class identifier and then specifying the base class name.

**Note:** C# supports single class inheritance only. Therefore, you can specify only one base class to inherit from. However, it does allow multiple *interface* inheritance, a subject covered in a later lesson.

*ChildClass* has exactly the same capabilities as *ParentClass*. Because of this, you can also say *ChildClass* "is" a *ParentClass*. This is shown in the *Main()* method of *ChildClass* when the *print()* method is called. *ChildClass* does not have its own *print()* method, so it uses the *ParentClass* *print()* method. You can see the results in the 3rd line of output.

Base classes are automatically instantiated before derived classes. Notice the output from Listing 8-1. The *ParentClass* constructor executed before the *ChildClass* constructor.

**Listing 8-2. Derived Class Communicating with Base Class: BaseTalk.cs**

using System;

public class Parent

{

string parentString;

public Parent()

{

Console.WriteLine("Parent Constructor.");

}

public Parent(string myString)

{

parentString = myString;

Console.WriteLine(parentString);

}

public void print()

{

Console.WriteLine("I'm a Parent Class.");

}

}

public class Child : Parent

{

public Child() : base("From Derived")

{

Console.WriteLine("Child Constructor.");

}

public new void print()

{

base.print();

Console.WriteLine("I'm a Child Class.");

}

public static void Main()

{

Child child = new Child();

child.print();

((Parent)child).print();

}

}

Output:

From Derived

Child Constructor.

I'm a Parent Class.

I'm a Child Class.

I'm a Parent Class.

Derived classes can communicate with base classes during instantiation. Listing 8-2 shows how this is done at the child constructor declaration. The colon, ":", and keyword *base* call the base class constructor with the matching parameter list. If the code had not appended *base("From Derived")* to the *Derived* constructor, the code would have automatically called *Parent()*. The first line of output shows the base class constructor being called with the *string* "From Derived".

Sometimes you may want to create your own implementation of a method that exists in a base class. The *Child* class does this by declaring its own *print()* method. The *Child* *print()* method hides the *Parent* *print()* method. The effect is the *Parent* *print()* method will not be called, unless we do something special to make sure it is called.

Inside the *Child* *print()* method, we explicitly call the *Parentprint()* method. This is done by prefixing the method name with "*base."*. Using the *base* keyword, you can access any of a base class *public* or *protected* class members. The output from the *Child* *print()* method is on output lines 3 and 4.

Another way to access base class members is through an explicit cast. This is done in the last statement of the *Child* class *Main()* method. Remember that a derived class is a specialization of its base class. This fact allows us to perform a cast on the derived class, making it an instance of its base class. The last line of output from Listing 8-2 shows the *Parent* *print()* method was indeed executed.

Notice the *new* modifier on the *Child* class *print()* method. This enables this method to hide the *Parent* class *print()* method and explicitly states your intention that you don't want polymorphism to occur. Without the *new* modifier, the compiler will produce a warning to draw your attention to this. See the next lesson for a detailed discussion of polymorphism.

In summary, you know how to create a derived/base class relationship. You can control instantiation of your base class and call its methods either implicitly or explicitly. You also understand that a derived class is a specialization of its base class.

# Lesson 9: Polymorphism

This lesson teaches about Polymorphism in C#. Our objectives are as follows:

* Learn What Polymorphism Is.
* Implement a Virtual Method.
* Override a Virtual Method.
* Use Polymorphism in a Program.

Another primary concept of object-oriented programming is Polymorphism. It allows you to invoke derived class methods through a base class reference during run-time. This is handy when you need to assign a group of objects to an array and then invoke each of their methods. They won't necessarily have to be the same object type. However, if they're related by inheritance, you can add them to the array as the inherited type. Then if they all share the same method name, that method of each object can be invoked. This lesson will show you how to accomplish this.

**Listing 9-1. A Base Class With a Virtual Method: DrawingObject.cs**

using System;  
  
public class DrawingObject  
{  
public virtual void Draw()  
{  
Console.WriteLine("I'm just a generic drawing object.");  
}  
}

Listing 9-1 shows the *DrawingObject* class. This will be the base class for other objects to inherit from. It has a single method named *Draw()*. The *Draw()* method has a *virtual* modifier. The *virtual* modifier indicates to derived classes that they can override this method. The *Draw()* method of the *DrawingObject* class performs a single action of printing the statement, "I'm just a generic drawing object.", to the console.

**Listing 9-2. Derived Classes With Override Methods: Line.cs, Circle.cs, and Square.cs**

using System;  
  
public class Line : DrawingObject  
{  
public override void Draw()  
{  
Console.WriteLine("I'm a Line.");  
}  
}  
  
public class Circle : DrawingObject  
{  
public override void Draw()  
{  
Console.WriteLine("I'm a Circle.");  
}  
}  
  
public class Square : DrawingObject  
{  
public override void Draw()  
{  
Console.WriteLine("I'm a Square.");  
}  
}

Listing 9-2 shows three classes. These classes inherit the *DrawingObject* class. Each class has a *Draw()* method and each *Draw()* method has an *override* modifier. The *override* modifier allows a method to *override* the *virtual* method of its base class at run-time. The *override* will happen only if the class is referenced through a base class reference. Overriding methods must have the same signature, name and parameters, as the *virtual* base class method it is overriding.

**Listing 9-3. Program Implementing Polymorphism: DrawDemo.cs**

using System;  
  
public class DrawDemo  
{  
public static int Main( )  
{  
DrawingObject[] dObj = new DrawingObject[4];  
  
dObj[0] = new Line();  
dObj[1] = new Circle();  
dObj[2] = new Square();  
dObj[3] = new DrawingObject();  
  
foreach (DrawingObject drawObj in dObj)  
{  
drawObj.Draw();  
}  
  
return 0;  
}  
}

Listing 9-3 shows a program that uses the classes defined in Listing 9-1 and Listing 9-2. This program implements polymorphism. In the *Main()* method of the *DrawDemo* class, there is an array being created. The type of object in this array is the *DrawingObject* class. The array is named *dObj* and is being initialized to hold four objects of type *DrawingObject*.

Next the *dObj* array is initialized. Because of their inheritance relationship with the *DrawingObject* class, the *Line*, *Circle*, and *Square* classes can be assigned to the *dObj* array. Without this capability, you would have to create an array for each type. Inheritance allows derived objects to act like their base class, which saves work.

After the array is initialized, there is a *foreach* loop that looks at each element of the array. Within the *foreach* loop the *Draw()* method is invoked on each element of the *dObj* array. Because of polymorphism, the run-time type of each object is invoked. The type of the reference object from the *dObj* array is a *DrawingObject*. However, that doesn't matter because the derived classes *override* the *virtual* *Draw()* method of the *DrawingObject* class. This makes the overriden *Draw()* methods of the derived classes execute when the *Draw()* method is called using the *DrawingObject* base class reference from the *dObj* array. Here's what the output looks like:

Output:

I'm a Line.

I'm a Circle.

I'm a Square.

I'm just a generic drawing object.

The *override* *Draw()* method of each derived class executes as shown in the *DrawDemo* program. The last line is from the *virtual* *Draw()* method of the *DrawingObject* class. This is because the actual run-time type of the fourth array element was a *DrawingObject* object.

The code in this lesson can be compiled with the following command line:

csc DrawDemo.cs DrawingObject.cs Circle.cs Line.cs Square.cs

It will create the file *DrawDemo.exe*, which defaulted to the name of the first file on the command line.

**Summary**

You should now have a basic understanding of polymorphism. You know how to define a *virtual* method. You can implement a derived class method that overrides a *virtual* method. This relationship between *virtual* methods and the derived class methods that *override* them enables polymorphism. This lesson showed how to use this relationship between classes to implement polymorphism in a program.

I invite you to return for [Lesson 10: Properties](http://csharp-station.com/Tutorial/CSharp/Lesson10).

# Lesson 10: Properties

This lesson teaches C# Properties. Our objectives are as follows:

* Understand What Properties Are For.
* Implement a Property.
* Create a Read-Only Property.
* Create a Write-Only Property.
* Create an auto-implemented property.

**Overview of Properties**

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties

this type of protection while also letting you access the property just like it was a field.

Another benefit of properties over fields is that you can change their internal implementation over time. With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you can change the implementation. For example, if a customer has an ID that is originally stored as an int, you might have a requirements change that made you perform a validation to ensure that calling code could never set the ID to a negative value. If it was a field, you would never be able to do this, but a property allows you to make such a change without breaking code. Now, lets see how to use properties.

**Traditional Encapsulation Without Properties**

Languages that don't have properties will use methods (functions or procedures) for encapsulation. The idea is to manage the values inside of the object, state, avoiding corruption and misuse by calling code. Listing 10-1 demonstrates how this traditional method works, encapsulating *Customer* information via accessor methods.

**Listing 10-1. An Example of Traditional Class Field Access**

using System;

public class Customer

{

private int m\_id = -1;

public int GetID()

{

return m\_id;

}

public void SetID(int id)

{

m\_id = id;

}

private string m\_name = string.Empty;

public string GetName()

{

return m\_name;

}

public void SetName(string name)

{

m\_name = name;

}

}

public class CustomerManagerWithAccessorMethods

{

public static void Main()

{

Customer cust = new Customer();

cust.SetID(1);

cust.SetName("Amelio Rosales");

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.GetID(),

cust.GetName());

Console.ReadKey();

}

}

Listing 10-1 shows the traditional method of accessing class fields. The *Customer* class has four methods, two for each private field that the class encapsulates: *m\_id* and *m\_name*. As you can see, *SetID* and *SetName* assign a new values and *GetID* and *GetName* return values.

Observe how *Main* calls the *SetXxx* methods, which sets *m\_id* to *1* and *m\_name* to "Amelio Rosales" in the *Customer* instance, *cust*. The call to *Console.WriteLine* demonstrates how to read *m\_id* and *m\_name* from *cust*, via *GetID* and *GetName* method calls, respectively.

This is such a common pattern, that C# has embraced it in the form of a language feature called properties, which you'll see in the next section.

**Encapsulating Type State with Properties**

The practice of accessing field data via methods was good because it supported the object-oriented concept of encapsulation. For example, if the type of *m\_id* or *m\_name* changed from an *int* type to *byte*, calling code would still work. Now the same thing can be accomplished in a much smoother fashion with properties, as shown in Listing 10-2.

**Listing 10-2. Accessing Class Fields With Properties**

using System;

public class Customer

{

private int m\_id = -1;

public int ID

{

get

{

return m\_id;

}

set

{

m\_id = value;

}

}

private string m\_name = string.Empty;

public string Name

{

get

{

return m\_name;

}

set

{

m\_name = value;

}

}

}

public class CustomerManagerWithProperties

{

public static void Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

Listing 10-2 shows how to create and use a property. The *Customer* class has the *ID* and *Name* property implementations. There are also private fields named *m\_id* and *m\_name;* which *ID* and *Name*, respectively, encapsulate. Each property has two accessors, *get* and *set*. The *get* accessor returns the value of a field. The *set* accessor sets the value of a field with the contents of *value*, which is the value being assigned by calling code. The *value* shown in the accessor is a C# reserved word.

When setting a property, just assign a value to the property as if it were a field. The *CustomerManagerWithProperties* class uses the *ID* and *Name* properties in the *Customer* class. The first line of *Main* instantiates a *Customer* object named *cust*. Next the value of the *m\_id* and *m\_name* fields of *cust* are set by using the *ID* and *Name* properties.

To read from a property, use the property as if it were a field. *Console.WriteLine* prints the value of the *m\_id* and *m\_name* fields of *cust*. It does this by calling the *ID* and *Name* properties of *cust*.

This was a read/write property, but you can also create read-only properties, which you'll learn about next.

## Creating Read-Only Properties

Properties can be made read-only. This is accomplished by having only a *get* accessor in the property implementation. Listing 10-3 demonstrates how you can create a read-only property.

**Listing 10-3. Read-Only Properties**

using System;

public class Customer

{

private int m\_id = -1;

private string m\_name = string.Empty;

public Customer(int id, string name)

{

m\_id = id;

m\_name = name;

}

public int ID

{

get

{

return m\_id;

}

}

public string Name

{

get

{

return m\_name;

}

}

}

public class ReadOnlyCustomerManager

{

public static void Main()

{

Customer cust = new Customer(1, "Amelio Rosales");

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

The *Customer* class in Listing 10-3 has two read-only properties, *ID* and *Name*. You can tell that each property is read-only because they only have *get* accessors. At some time, values for the *m\_id* and *m\_name* must be assigned, which is the role of the constructor in this example.

The *Main* method of the *ReadOnlyCustomerManager* class instantiates a new *Customer* object named *cust*. The instantiation of *cust* uses the constructor of *Customer* class, which takes *int* and *string* type parameters. In this case, the values are *1* and *"Amelio Rosales"*. This initializes the *m\_id* and *m\_name* fields of *cust*.

Since the *ID* and *Name* properties of the *Customer* class are read-only, there is no other way to set the value of the *m\_id* and *m\_name* fields. If you inserted *cust.ID = 7* into the listing, the program would not compile, because *ID* is read-only; the same goes for *Name*. When the *ID* and *Name* properties are used in *Console.WriteLine*, they work fine. This is because these are read operations which only invoke the *get* accessor of the *ID* and *Name* properties.

One question you might have now is "If a property can be read-only, can it also be write-only?" The answer is yes, and explained in the next section.

## Creating a Write-Only Property

You can assign values to, but not read from, a write-only property. A write-only property only has a *set* accessor. Listing 10-4 shows you how to create and use write-only properties.

**Listing 10-4. Write-Only Properties**

using System;

public class Customer

{

private int m\_id = -1;

public int ID

{

set

{

m\_id = value;

}

}

private string m\_name = string.Empty;

public string Name

{

set

{

m\_name = value;

}

}

public void DisplayCustomerData()

{

Console.WriteLine("ID: {0}, Name:

{1}", m\_id, m\_name);

}

}

public class WriteOnlyCustomerManager

{

public static void Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

cust.DisplayCustomerData();

Console.ReadKey();

}

}

This time, the *get* accessor is removed from the *ID* and *Name* properties of the *Customer* class, shown in Listing 10-1. The *set* accessors have been added, assigning *value* to the backing store fields, *m\_id* and *m\_name*.

The *Main* method of the *WriteOnlyCustomerManager* class instantiates the *Customer* class with a default constructor. Then it uses the *ID* and *Name* properties of *cust* to set the *m\_id* and *m\_name* fields of *cust* to *1* and *"Amelio Rosales"*, respectively. This invokes the *set* accessor of *ID* and *Name* properties from the *cust* instance.

When you have a lot of properties in a class or struct, there can also be a lot of code associated with those properties. In the next section, you'll see how to write properties with less code.

## Creating Auto-Implemented Properties

The patterns you see here, where a property encapsulates a property with *get* and *set* accessors, without any other logic is common. It is more code than we should have to write for such a common scenario. That's why C# 3.0 introduced a new syntax for a property, called an *auto-implemented property*, which allows you to create properties without *get* and *set* accessor implementations. Listing 10-5 shows how to add auto-implemented properties to a class.

**Listing 10-5. Auto-Implemented Properties**

using System;

public class Customer

{

public int ID { get; set; }

public string Name { get; set; }

}

public class AutoImplementedCustomerManager

{

static void Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

Notice how the *get* and *set* accessors in Listing 10-5 do not have implementations. In an auto-implemented property, the C# compiler creates the backing store field behind the scenes, giving the same logic that exists with traditional properties, but saving you from having to use all of the syntax of the traditional property. As you can see in the *Main* method, the usage of an auto-implemented property is exactly the same as traditional properties, which you learned about in previous sections.

**Summary**

You now know what properties are for and how they're used. Traditional techniques of encapsulation have relied on separate methods. Properties allow you to access objects state with field-like syntax. Properties can be made read-only or write-only. You also learned how to write properties with less code by using auto-implemented properties.

I invite you to return for [Lesson 11: Indexers](http://csharp-station.com/Tutorial/CSharp/Lesson11).

# Lesson 11: Indexers

This lesson teaches C# Indexers. Our objectives are as follows:

* Understand What Indexers Are For.
* Implement an Indexer.
* Overload Indexers.
* Understand How to Implement Multi-Parameter Indexers.

Indexers allow your class to be used just like an array. On the inside of a class, you manage a collection of values any way you want. These objects could be a finite set of class members, another array, or some complex data structure. Regardless of the internal implementation of the class, its data can be obtained consistently through the use of indexers. Here's an example.

**Listing 11-1. An Example of An Indexer: IntIndexer.cs**

using System;  
  
///<summary>  
/// A simple indexer example.  
///</summary>  
class IntIndexer  
{  
privatestring[] myData;  
  
public IntIndexer(int size)  
{  
myData = new string[size];  
  
for (int i=0; i < size; i++)  
{  
myData[i] = "empty";  
}  
}  
  
public string this[int pos]  
{  
get  
{  
return myData[pos];  
}  
set  
{  
myData[pos] = value;  
}  
}  
  
static void Main(string[] args)  
{  
int size = 10;  
  
IntIndexer myInd = new IntIndexer(size);  
  
myInd[9] = "Some Value";  
myInd[3] = "Another Value";  
myInd[5] = "Any Value";  
  
Console.WriteLine("\nIndexer Output\n");  
  
for (int i=0; i < size; i++)  
{  
Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);  
}  
}  
}

Listing 11-1 shows how to implement an Indexer. The *IntIndexer* class has a *string* array named *myData*. This is a private array that external users can't see. This array is initialized in the constructor, which accepts an *int* *size* parameter, instantiates the *myData* array, and then fills each element with the word "empty".

The next class member is the Indexer, which is identified by the *this* keyword and square brackets, *this[int pos]*. It accepts a single position parameter, *pos*. As you may have already guessed, the implementation of an Indexer is the same as a Property. It has *get* and *set* accessors that are used exactly like those in a Property. This indexer returns a *string*, as indicated by the *string* return value in the Indexer declaration.

The *Main()* method simply instantiates a new *IntIndexer* object, adds some values, and prints the results. Here's the output:

Indexer Output

myInd[0]: empty

myInd[1]: empty

myInd[2]: empty

myInd[3]: Another Value

myInd[4]: empty

myInd[5]: Any Value

myInd[6]: empty

myInd[7]: empty

myInd[8]: empty

myInd[9]: Some Value

Using an *integer* is a common means of accessing arrays in many languages, but the C# Indexer goes beyond this. Indexers can be declared with multiple parameters and each parameter may be a different type. Additional parameters are separated by commas, the same as a method parameter list. Valid parameter types for Indexers include *integers*, *enums*, and *strings*. Additionally, Indexers can be overloaded. In listing 11-2, we modify the previous program to accept overloaded Indexers that accept different types.

**Listing 11-2. Overloaded Indexers: OvrIndexer.cs**

using System;  
  
///<summary>  
/// Implements overloaded indexers.  
///</summary>  
class OvrIndexer  
{  
privatestring[] myData;  
privateint arrSize;  
  
public OvrIndexer(int size)  
{  
arrSize = size;  
myData = new string[size];  
  
for (int i=0; i < size; i++)  
{  
myData[i] = "empty";  
}  
}  
  
public string this[int pos]  
{  
get  
{  
return myData[pos];  
}  
set  
{  
myData[pos] = value;  
}  
}  
  
public string this[string data]  
{  
get  
{  
int count = 0;  
  
for (int i=0; i < arrSize; i++)  
{  
if (myData[i] == data)  
{  
count++;  
}  
}  
return count.ToString();  
}  
set  
{  
for (int i=0; i < arrSize; i++)  
{  
if (myData[i] == data)  
{  
myData[i] = value;  
}  
}  
}  
}  
  
static void Main(string[] args)  
{  
int size = 10;  
OvrIndexer myInd = new OvrIndexer(size);  
  
myInd[9] = "Some Value";  
myInd[3] = "Another Value";  
myInd[5] = "Any Value";  
  
myInd["empty"] = "no value";  
  
Console.WriteLine("\nIndexer Output\n");  
  
for (int i=0; i < size; i++)  
{  
Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);  
}  
  
Console.WriteLine("\nNumber of \"no value\" entries: {0}", myInd["no value"]);  
}  
}

Listing 11-2 shows how to overload Indexers. The first Indexer, with the *int* parameter, *pos*, is the same as in Listing 11-1, but there is a new Indexer that takes a *string* parameter. The *get* accessor of the new indexer returns a *string* representation of the number of items that match the parameter value, *data*. The *set* accessor changes each entry in the array that matches the *data* parameter to the value that is assigned to the Indexer.

The behavior of the overloaded Indexer that takes a *string* parameter is demonstrated in the *Main()* method of Listing 11-2. It invokes the *set* accessor, which assigns the value of "no value" to every member of the *myInd* class that has the value of "empty". It uses the following command: *myInd["empty"] = "no value";*. After each entry of the *myInd* class is printed, a final entry is printed to the console, indicating the number of entries with the "no value" string. This happens by invoking the *get* accessor with the following code: *myInd["no value"]*. Here's the output:

Indexer Output

myInd[0]: no value

myInd[1]: no value

myInd[2]: no value

myInd[3]: Another Value

myInd[4]: no value

myInd[5]: Any Value

myInd[6]: no value

myInd[7]: no value

myInd[8]: no value

myInd[9]: Some Value

Number of "no value" entries: 7

The reason both Indexers in Listing 11-2 can coexist in the same class is because they have different signatures. An Indexer signature is specified by the number and type of parameters in an Indexers parameter list. The class will be smart enough to figure out which Indexer to invoke, based on the number and type of arguments in the Indexer call. An indexer with multiple parameters would be implemented something like this:

public objectthis[int param1, ..., int paramN]  
{  
get  
{  
// process and return some class data  
}  
set  
{  
// process and assign some class data  
}  
}

**Summary**

You now know what Indexers are for and how they're used. You can create an Indexer to access class members similar to arrays. Overloaded and multi-parameter Indexers were also covered.

I invite you to return for [Lesson 12: Structs](http://csharp-station.com/Tutorial/CSharp/Lesson12).

# Lesson 12: Structs

This lesson will teach you about the C# *struct*. Our objectives are as follows:

* Understand the Purpose of *structs*.
* Implement a *struct*.
* Use a *struct*.

**What is a *struct*?**

A *struct* is a value type. To help understand the *struct*, it's helpful to make a comparison with *classes*, as described in [Lesson 7: Introduction to Classes](http://csharp-station.com/Tutorial/CSharp/Lesson07) and subsequent chapters. While a *struct* is a value type, a *class* is a reference type. Value types hold their value in memory where they are declared, but reference types hold a reference to an object in memory. If you copy a *struct*, C# creates a new copy of the object and assigns the copy of the object to a separate *struct* instance. However, if you copy a *class*, C# creates a new copy of the reference to the object and assigns the copy of the reference to the separate *class* instance. *Structs* can't have destructors, but *classes* can have destructors. Another difference between a *struct* and *class* is that a *struct* can't have implementation inheritance, but a *class* can, as described in [Lesson 8: Class Inheritance](http://csharp-station.com/Tutorial/CSharp/Lesson08). Although a *struct* can't have implementation inheritance, it can have *interface* inheritance, as described in [Lesson 13: Interfaces](http://csharp-station.com/Tutorial/CSharp/Lesson13), which is the next lesson following this one. [Lesson 22: Topics on C# Type](http://csharp-station.com/Tutorial/CSharp/Lesson22), digs deeper into the differences between value and reference types, providing code that demonstrates the concepts that are introduced here.

The .NET Framework includes many types that are *structs*, including many of the built-in types. For example, a *System.Int32* is a C# *int*, a *System.Single* is a C# *float*, and a *System.Bool* is a C# *bool*. The C# built-in types are aliases for .NET Framework types, giving you language-specific syntax. If you look at the documentation for any of these .NET Framework types, you'll see them declared as *struct* types. That means you'll need to recognize what a *struct* type is when you see it, which the next section helps with by showing you how to create your own custom *struct* type.

**Creating a Custom *struct* Type**

While the behavior of *class* and *struct* types are very different, their syntax is similar. You declare the type and its members with the primary visual difference being that a *struct* uses the keyword *struct* and a *class* uses the keyword *class*. The example in Listing 12-1 demonstrates how to define a custom *struct*. In this case, the *struct* is a *Rectangle* with *Width* and *Height* properties, similar to what you might use to represent a rectangular shape on a screen.

**Listing 12-1. Defining a *struct***

/// <summary>

/// Custom struct type, representing

a rectangular shape

/// </summary>

struct Rectangle

{

/// <summary>

/// Backing Store for Width

/// </summary>

private int m\_width;

/// <summary>

/// Width of rectangle

/// </summary>

public int Width

{

get

{

return m\_width;

}

set

{

m\_width = value;

}

}

/// <summary>

/// Backing store for Height

/// </summary>

private int m\_height;

/// <summary>

/// Height of rectangle

/// </summary>

public int Height

{

get

{

return m\_height;

}

set

{

m\_height = value;

}

}

}

As you can see, the *Rectangle* *struct* in Listing 12-1 looks very much like a class with a couple properties, except that it uses the keyword *struct*, instead of the keyword *class*, to declare that *Rectangle* is a *struct*.

**Using a *struct***

To use a *struct*, instantiate the *struct* and use it just like a *class*. Listing 12-2 shows how to instantiate the *Rectangle* *struct* and access its properties.

**Listing 12-2. Using a Struct**

using System;

/// <summary>

/// Example of declaring and using

a struct

/// </summary>

class StructExample

{

/// <summary>

/// Entry point: execution starts

here

/// </summary>

static void Main()

{

// instantiate a new Rectangle struct

// where Width is set to 1 and Height

is set to 3

Rectangle rect1 = new Rectangle();

rect1.Width = 1;

rect1.Height = 3;

// show the value of Width and Height

for rect1

Console.WriteLine("rect1: {0}:{1}", rect1.Width, rect1.Height);

Console.ReadKey();

}

}

The code in the *Main* method of Listing 12-2 instantiates a new *Rectanglestruct* and sets its *Height* and *Width* properties. The experience is similar to how a *class* can be used. Here's the output:

rect1: 1:3

An alternate way of instantiating a *struct* and setting its properties is with an object initializer, shown below:

// you can also use object

initialization syntax

Rectangle rect11 = new Rectangle

{

Width = 1,

Height = 3

};

Notice that the object initializer uses curly braces and sets properties via a comma-separated list of name/value pairs.

**Overloading *struct* Constructors**

The two previous examples of instantiating a *struct*, via constructor only and via object initializer, used the default (parameterless) constructor of the *struct*. The default constructor is implicitly defined by C# and you can't implement the default constructor yourself. The default constructor initializes all *struct* fields to default values. i.e. integrals are 0, floating points are 0.0, and booleans are false. If you need custom constructor overloads, you can add new constructors, as long as they have one or more parameters. Listing 12-3 shows a customization of the *Rectangle* *struct* from Listing 12-1 that includes a constructor overload.

**Listing 12-3: Overloading a *struct* Constructor**

/// <summary>

/// Custom struct type, representing

a rectangular shape

/// </summary>

struct Rectangle

{

/// <summary>

/// Backing Store for Width

/// </summary>

private int m\_width;

/// <summary>

/// Width of rectangle

/// </summary>

public int Width

{

get

{

return m\_width;

}

set

{

m\_width = value;

}

}

/// <summary>

/// Backing store for Height

/// </summary>

private int m\_height;

/// <summary>

/// Height of rectangle

/// </summary>

public int Height

{

get

{

return m\_height;

}

set

{

m\_height = value;

}

}

**/// <summary> /// Instantiate rectangle struct with**

**dimensions /// </summary>**

**/// <param name="width">Width**

**to make new rectangle</param>**

**/// <param name="height">Height to make new rectangle</param>**

**public Rectangle(int width,**

**int height) { m\_width = width; m\_height = height; }**

}

The highlighted portion of code in Listing 12-3 is a constructor overload. Constructors are named the same as their containing struct, which is *Rectangle* in this case. This *Rectangle* constructor overload has two parameters, which it assigns to backing stores that are encapsulated by properties for calling code. Listing 12-4 shows an example of how you would use the constructor overload from Listing 12-3 to instantiate a new *Rectangle*.

**Listing 12-4: Instantiating a *struct* Through a Constructor Overload**

using System;

/// <summary>

/// Example of declaring and using

a struct

/// </summary>

class StructExample

{

/// <summary>

/// Entry point: execution starts

here

/// </summary>

static void Main()

{

// instantiate a new Rectangle struct

// where Width is set to 5 and Height

is set to 7

Rectangle rect2 = new Rectangle(5, 7);

// show the value of Width and Height

for rect2

Console.WriteLine("rect2: {0}:{1}", rect2.Width, rect2.Height);

Console.ReadKey();

}

}

The code in the *Main* method of Listing 12-4 instantiates a *Rectanglestruct* and displays the values set via the constructor overload. When instantiating *rect2*, the code passes the values 5 and 7 as arguments. From the constructor in Listing 12-3, you can see that the *Width* of *rect2* will be set to 5 and the *Height* of *rect2* will be set to 7. Here's the output from Listing 12-4:

rect2: 5:7

**Adding a Method to a *struct***

All of the examples so far showed how you can add properties and constructors to a *struct*, but you can also add methods to a *struct*. Defining a method in a *struct* is the same as defining a method in a *class*. Listing 12-5 shows the *Rectangle* *struct* with a method named *Add*.

**Listing 12-5: Adding a Method to a *struct***

/// <summary>

/// Custom struct type, representing

a rectangular shape

/// </summary>

struct Rectangle

{

/// <summary>

/// Backing Store for Width

/// </summary>

private int m\_width;

/// <summary>

/// Width of rectangle

/// </summary>

public int Width

{

get

{

return m\_width;

}

set

{

m\_width = value;

}

}

/// <summary>

/// Backing store for Height

/// </summary>

private int m\_height;

/// <summary>

/// Height of rectangle

/// </summary>

public int Height

{

get

{

return m\_height;

}

set

{

m\_height = value;

}

}

/// <summary>

/// Instantiate rectangle struct

with dimensions

/// </summary>

/// <param name="width">Width

to make new rectangle</param>

/// <param name="height">Height

to make new rectangle</param>

public Rectangle(int width, int height)

{

m\_width = width;

m\_height = height;

}

**/// <summary>**

**/// Increase the size of this rectangle by the size of the specified rectangle**

**/// </summary>**

**/// <param name="rect">Rectangle that will be added to this rectangle</param>**

**/// <returns>New rectangle created by adding rect to this rectangle</returns>**

**public Rectangle Add(Rectangle rect)**

**{**

**// create instance of rectangle struct with default constructor**

**Rectangle newRect = new Rectangle();**

**// add matching axes and assign to new Rectangle struct**

**newRect.Width = Width + rect.Width; newRect.Height = Height + rect.Height;**

**// return new Rectangle struct**

**return newRect;**

**}**

}

The highlighted code in Listing 12-5 is a method named *Add*. It might or might not make sense to add two *Rectangle* structs together, but the example demonstrates how to define a method in a *struct*. In this case, the *Add* method will increase the *Height* and *Width* of the current *Rectangle* instance by adding the *Height* and *Width* in the *rect* parameter. The result of the method is a new *Rectangle* with the added properties.

**Calling a *struct* Method**

You can call the *Add* method, from Listing 12-5, through an instance of a *Rectangle* *struct*. Listing 12-6 shows how to instantiate two *Rectanglestructs*, call the *Add* method and assign the result of the *Add* method call to another *Rectangle* struct.

**Listing 12-6: Calling a *struct* Method**

using System;

/// <summary>

/// Example of declaring and using

a struct

/// </summary>

class StructExample

{

/// <summary>

/// Entry point: execution starts

here

/// </summary>

static void Main()

{

// instantiate a new Rectangle struct

// where Width is set to 1 and Height is set to 3

Rectangle rect1 = new Rectangle();

rect1.Width = 1;

rect1.Height = 3;

// show the value of Width and Height for rect1

Console.WriteLine("rect1: {0}:{1}", rect1.Width, rect1.Height);

// instantiate a new Rectangle struct

// where Width is set to 5 and Height is set to 7

Rectangle rect2 = new Rectangle(5, 7);

// show the value of Width and Height for rect2

Console.WriteLine("rect2: {0}:{1}", rect2.Width, rect2.Height);

**// invoke the Add method on the rect1 Rectangle struct instance,**

**// passing the rect2 Rectangle struct instance as an argument**

**// and assigning the new copy of the value returned by the**

**// Add method to the rect3 Rectangle struct.**

**Rectangle rect3 = rect1.Add(rect2);**

// show the value of Width and Height for rect3

Console.WriteLine("rect3: {0}:{1}", rect3.Width, rect3.Height);

Console.ReadKey();

}

}

In the *Main* method of Listing 12-6, the code instantiates *rect1* and *rect2*, which are both *Rectangle* *structs*, assigning values to their *Height* and *Width* properties. The *struct* instantiation examples should be familiar by now because they are the same as earlier examples. What's useful about Listing 12-6 is the highlighted code, which shows how to invoke the *Add* method of the *Rectangle* *struct*. The code invokes the *Add* method of the *rect1* instance and passes *rect2* as the *Rectangle* *struct* to be added to *rect1*. The *Add* method in Listing 12-5 shows what happens when this code executes. In Listing 12-6, the return value of the *Add* method is assigned to *rect3*, which is a larger *Rectangle* with each of its sides equal to the sum of the individual sides of *rect1* and *rect2*. Here's the output:

rect1: 1:3

rect2: 5:7

rect3: 6:10

**Summary**

This lesson described what a *struct* was and identified a few differences between *class* and *struct* types. You learned how to create a *struct*. You can instantiate a *struct* either via a default constructor or a custom constructor overload that you write. You also saw how to implement properties and methods in structs.

I invite you to return for [Lesson 13: Interfaces](http://csharp-station.com/Tutorial/CSharp/Lesson13).

# Lesson 13: Interfaces

This lesson teaches C# *Interfaces*. Our objectives are as follows:

* Understand the Purpose of *Interfaces.*
* Define an *Interface.*
* Use an *Interface.*
* Implement Interface *Inheritance*.

An *interface* looks like a class, but has no implementation. The only thing it contains are declarations of *events*, *indexers*, *methods* and/or *properties*. The reason *interfaces* only provide declarations is because they are inherited by *classes* and *structs*, which must provide an implementation for each interface member declared.

So, what are *interfaces* good for if they don't implement functionality? They're great for putting together plug-n-play like architectures where components can be interchanged at will. Since all interchangeable components implement the same *interface*, they can be used without any extra programming. The *interface* forces each component to expose specific public members that will be used in a certain way.

Because *interfaces* must be implemented by derived *classes* and *structs*, they define a contract. For instance, if class *foo* implements the *IDisposable* interface, it is making a statement that it guarantees it has the *Dispose()* method, which is the only member of the *IDisposable* interface. Any code that wishes to use class *foo* may check to see if class *foo* implements *IDisposable*. When the answer is *true*, then the code knows that it can call *foo.Dispose()*. Listing 13-1 shows how to define an interface:

##### Listing 13-1. Defining an Interface: MyInterface.cs

interface IMyInterface  
{  
void MethodToImplement();  
}

Listing 13-1 defines an *interface* named *IMyInterface*. A common naming convention is to prefix all *interface* names with a capital "I". This *interface* has a single method named *MethodToImplement()*. This could have been any type of method declaration with different parameters and return types. I just chose to declare this method with no parameters and a *void* return type to make the example easy. Notice that this method does not have an implementation (instructions between curly braces - *{}*), but instead ends with a semi-colon, "*;*". This is because the *interface* only specifies the signature of methods that an inheriting *class* or *struct* must implement. Listing 13-2 shows how this *interface* could be used.

##### Listing 13-2. Using an Interface: InterfaceImplementer.cs

class InterfaceImplementer : IMyInterface  
{  
staticvoid Main()  
{  
InterfaceImplementer iImp = new InterfaceImplementer();  
iImp.MethodToImplement();  
}  
  
public void MethodToImplement()  
{  
Console.WriteLine("MethodToImplement() called.");  
}  
}

The *InterfaceImplementer* *class* in Listing 13.2 implements the *IMyInterfaceinterface*. Indicating that a *class* inherits an *interface* is the same as inheriting a *class*. In this case, the following syntax is used:

class InterfaceImplementer : IMyInterface

Now that this *class* inherits the *IMyInterface* interface, it must implement its members. It does this by implementing the *MethodToImplement()* method. Notice that this method implementation has the exact same signature, parameters and method name, as defined in the *IMyInterface* interface. Any difference between the method signature in the interface and the method signature in the implementing *class* or *struct* will cause a compiler error. Additionally, a *class* or *struct* that inherits an interface must include all interface members; You will receive a compiler error if you don't implement all interface members.

Interfaces may also inherit other interfaces. Listing 13-3 shows how inherited interfaces are implemented.

##### Listing 13-3. Interface Inheritance: InterfaceInheritance.cs

using System;  
  
interface IParentInterface  
{  
void ParentInterfaceMethod();  
}  
  
interface IMyInterface : IParentInterface  
{  
void MethodToImplement();  
}  
  
class InterfaceImplementer : IMyInterface  
{  
staticvoid Main()  
{  
InterfaceImplementer iImp = new InterfaceImplementer();  
iImp.MethodToImplement();  
iImp.ParentInterfaceMethod();  
}  
  
public void MethodToImplement()  
{  
Console.WriteLine("MethodToImplement() called.");  
}  
  
publicvoid ParentInterfaceMethod()  
{  
Console.WriteLine("ParentInterfaceMethod() called.");  
}  
}

The code in listing 13.3 contains two *interfaces*: *IMyInterface* and the *interface* it inherits, *IParentInterface*. When one *interface* inherits another, any implementing *class* or *struct* must implement every *interface* member in the entire inheritance chain. Since the *InterfaceImplementerclass* in Listing 13-3 inherits from *IMyInterface*, it also inherits *IParentInterface*. Therefore, the *InterfaceImplementer* class must implement the *MethodToImplement()* method specified in the *IMyInterface* interface and the *ParentInterfaceMethod()* method specified in the *IParentInterface* interface.

#### Summary

You now understand what *interfaces* are. You can implement an *interface* and use it in a class. *Interfaces* may also be inherited by other *interface*. Any *class* or *struct* that inherits an *interface* must also implement all members in the entire *interface* inheritance chain.

I invite you to return for [Lesson 14: Introduction to Delegates and Events](http://csharp-station.com/Tutorial/CSharp/Lesson14).

# Lesson 14: Introduction to Delegates and Events

This lesson introduces *delegates* and *events*. Our objectives are as follows:

* Understand What a *Delegate* Is
* Understand What an *Event* Is
* Implement *Delegates*
* Fire *Events*

**Delegates**

During previous lessons, you learned how to implement reference types using language constructs such as classes and interfaces. These reference types allowed you to create instances of objects and use them in special ways to accomplish your software development goals. Classes allow you to create objects that contained members with attributes or behavior. Interfaces allow you to declare a set of attributes and behavior that all objects implementing them would publicly expose. Today, I'm going to introduce a new reference type called a *delegate*.

A *delegate* is a C# language element that allows you to reference a method. If you were a C or C++ programmer, this would sound familiar because a *delegate* is basically a function pointer. However, developers who have used other languages are probably wondering, "Why do I need a reference to a method?". The answer boils down to giving you maximum flexibility to implement any functionality you want at runtime.

Think about how you use methods right now. You write an algorithm that does its thing by manipulating the values of variables and calling methods directly by name. What if you wanted an algorithm that was very flexible, reusable, and allowed you to implement different functionality as the need arises? Furthermore, let's say that this was an algorithm that supported some type of data structure that you wanted to have sorted, but you also want to enable this data structure to hold different types. If you don't know what the types are, how could you decide an appropriate comparison routine? Perhaps you could implement an *if/then/else* or *switch* statement to handle well-known types, but this would still be limiting and require overhead to determine the type. Another alternative would be for all the types to implement an interface that declared a common method your algorithm would call, which is actually a nice solution. However, since this lesson is about *delegates*, we'll apply a *delegate* solution, which is quite elegant.

You could solve this problem by passing a *delegate* to your algorithm and letting the contained method, which the *delegate* refers to, perform the comparison operation. Such an operation is performed in Listing 14-1.

**Listing 14-1. Declaring and Implementing a Delegate: SimpleDelegate.cs**

using System;  
  
// this is the delegate declaration  
public delegate int Comparer(object obj1, object obj2);  
  
public class Name  
{  
publicstring FirstName = null;  
publicstring LastName = null;  
  
public Name(string first, string last)  
{  
FirstName = first;  
LastName = last;  
}  
  
// this is the delegate method handler  
public static int CompareFirstNames(object name1, object name2)  
{  
string n1 = ((Name)name1).FirstName;  
string n2 = ((Name)name2).FirstName;  
  
if (String.Compare(n1, n2) > 0)  
{  
return 1;  
}  
else if (String.Compare(n1, n2) < 0)  
{  
return -1;  
}  
else  
{  
return 0;  
}  
}  
  
public override string ToString()  
{  
return FirstName + " " + LastName;  
}  
}  
  
class SimpleDelegate  
{  
Name[] names = new Name[5];  
  
public SimpleDelegate()  
{  
names[0] = new Name("Joe", "Mayo");  
names[1] = new Name("John", "Hancock");  
names[2] = new Name("Jane", "Doe");  
names[3] = new Name("John", "Doe");  
names[4] = new Name("Jack", "Smith");  
}  
  
static void Main(string[] args)  
{  
SimpleDelegate sd = new SimpleDelegate();  
  
// this is the delegate instantiation  
Comparer cmp = new Comparer(Name.CompareFirstNames);  
  
Console.WriteLine("\nBefore Sort: \n");  
  
sd.PrintNames();  
  
// observe the delegate argument  
sd.Sort(cmp);  
  
Console.WriteLine("\nAfter Sort: \n");  
  
sd.PrintNames();  
}  
  
// observe the delegate parameter  
public void Sort(Comparer compare)  
{  
object temp;  
  
for (int i=0; i < names.Length; i++)  
{  
for (int j=i; j < names.Length; j++)  
{  
// using delegate "compare" just like  
// a normal method  
if ( compare(names[i], names[j]) > 0 )  
{  
temp = names[i];  
names[i] = names[j];  
names[j] = (Name)temp;  
}  
}  
}  
}  
  
public void PrintNames()  
{  
Console.WriteLine("Names: \n");  
  
foreach (Name name in names)  
{  
Console.WriteLine(name.ToString());  
}  
}  
}

The first thing the program in Listing 14-1 does is declare a *delegate*. *Delegate* declarations look somewhat like methods, except they have the *delegate* modifier, are terminated with a semi-colon (*;*), and have no implementation. Below, is the *delegate* declaration from Listing 14-1.

public delegate int Comparer(object obj1, object obj2);

This *delegate* declaration defines the signature of a delegate handler method that this *delegate* can refer to. The delegate handler method, for the *Comparerdelegate*, can have any name, but must have a first parameter of type *object*, a second parameter of type *object*, and return an *int* type. The following method from Listing 14-1 shows a delegate handler method that conforms to the signature of the *Comparer* *delegate*.

public static int CompareFirstNames(object name1, object name2)  
{  
...  
}

**Note:** The *CompareFirstNames* method calls *String.Compare* to compare the *FirstName* properties of the two *Name* instances. The *String* class has many convenience methods, such as *Compare*, for working with strings. Please don't allow the implementation of this method to interfere with learning how delegates work. What you should concentrate on is that *CompareFirstNames* is a handler method that a delegate can refer to, regardless of the code inside of that method.

To use a *delegate*, you must create an instance of it. The instance is created, similar to a class instance, with a single parameter identifying the appropriate delegate handler method, as shown below.

Comparer cmp = new Comparer(Name.CompareFirstNames);

The *delegate*, *cmp*, is then used as a parameter to the *Sort()* method, which uses it just like a normal method. Observe the way the *delegate* is passed to the *Sort()* method as a parameter in the code below.

sd.Sort(cmp);

Using this technique, any delegate handler method may be passed to the *Sort()* method at run-time. i.e. You could define a method handler named *CompareLastNames()*, instantiate a new *Comparer delegate* instance with it, and pass the new *delegate* to the *Sort()* method.

**Events**

Traditional Console applications operate by waiting for a user to press a key or type a command and press the *Enter* key. Then they perform some pre-defined operation and either quit or return to the original prompt that they started from. This works, but is inflexible in that everything is hard-wired and follows a rigid path of execution. In stark contrast, modern GUI programs operate on an event-based model. That is, some event in the system occurs and interested modules are notified so they can react appropriately. With Windows Forms, there is not a polling mechanism taking up resources and you don't have to code a loop that sits waiting for input. It is all built into the system with events.

A C# *event* is a class member that is activated whenever the event it was designed for occurs. I like to use the term "fires" when the *event* is activated. Anyone interested in the *event* can register and be notified as soon as the *event* fires. At the time an *event* fires, registered methods will be invoked.

*Events* and *delegates* work hand-in-hand to provide a program's functionality. It starts with a class that declares an *event*. Any class, including the same class that the *event* is declared in, may register one of its methods for the *event*. This occurs through a *delegate*, which specifies the signature of the method that is registered for the *event*. The *delegate* may be one of the pre-defined .NET *delegates* or one you declare yourself. Whichever is appropriate, you assign the *delegate* to the *event*, which effectively registers the method that will be called when the *event* fires. Listing 14-2 shows a couple different ways to implement *events*.

**Listing 14-2. Declaring and Implementing Events: Eventdemo.cs**

using System;  
using System.Drawing;  
using System.Windows.Forms;  
  
// custom delegate  
public delegate void Startdelegate();  
  
class Eventdemo : Form  
{  
// custom event  
publicevent Startdelegate StartEvent;  
  
public Eventdemo()  
{  
Button clickMe = new Button();  
  
clickMe.Parent = this;  
clickMe.Text = "Click Me";  
clickMe.Location = new Point(  
(ClientSize.Width - clickMe.Width) /2,  
(ClientSize.Height - clickMe.Height)/2);  
  
// an EventHandler delegate is assigned  
// to the button's Click event  
clickMe.Click += new EventHandler(OnClickMeClicked);  
  
// our custom "Startdelegate" delegate is assigned  
// to our custom "StartEvent" event.  
StartEvent += new Startdelegate(OnStartEvent);  
  
// fire our custom event  
StartEvent();  
}  
  
// this method is called when the "clickMe" button is pressed  
publicvoid OnClickMeClicked(object sender, EventArgs ea)  
{  
MessageBox.Show("You Clicked My Button!");  
}  
  
// this method is called when the "StartEvent" Event is fired  
publicvoid OnStartEvent()  
{  
MessageBox.Show("I Just Started!");  
}  
  
static void Main(string[] args)  
{  
Application.Run(new Eventdemo());  
}  
}

**Note:** If you're using Visual Studio or another IDE, remember to add references to System.Drawing.dll and System.Windows.Forms.dll before compiling Listing 14.2 or just add the code to a Windows Forms project. Teaching the operation of Visual Studio or other IDE's is out-of-scope for this tutorial.

You may have noticed that Listing 14-2 is a Windows Forms program. Although I haven't covered Windows Forms in this tutorial, you should know enough about C# programming in general that you won't be lost. To help out, I'll give a brief explanation of some of the parts that you may not be familiar with.

The *Eventdemo* class inherits *Form*, which essentially makes it a Windows Form. This automatically gives you all the functionality of a Windows Form, including Title Bar, Minimize/Maximize/Close buttons, System Menu, and Borders. A lot of power, that inheritance thing, eh?

The way a Windows Form's application is started is by calling the *Run()* method of the *static* *Application* object with a reference to the *form* object as its parameter. This starts up all the underlying Windows plumbing, displays the GUI, and ensures that *events* are fired as appropriate.

Let's look at the custom *event* first. Below is the *event* declaration, which is a member of the *Eventdemo* class. It is declared with the *event* keyword, a *delegate* type, and an *event* name.

public event Startdelegate StartEvent;

Anyone interested in an *event* can register by hooking up a *delegate* for that *event*. On the next line, we have a *delegate* of type *Startdelegate*, which the *event* was declared to accept, hooked up to the *StartEventevent*. The *+=* syntax registers a *delegate* with an *event*. To unregister with an *event*, use the *-=* with the same syntax.

StartEvent += new Startdelegate(OnStartEvent);

Firing an *event* looks just like a method call, as shown below:

StartEvent();

This was how to implement *events* from scratch, declaring the *event* and *delegate* yourself. However, much of the *event* programming you'll do will be with pre-defined *events* and *delegates*. This leads us to the other *event* code you see in Listing 14-2, where we hook up an *EventHandlerdelegate* to a *Button* *Click* event.

clickMe.Click += new EventHandler(OnClickMeClicked);

The *Click* event already belongs to the *Button* class and all we have to do is reference it when registering a *delegate*. Similarly, the *EventHandlerdelegate* already exists in the *System* namespace of the .NET Frameworks Class Library. All you really need to do is define your callback method (delegate handler method) that is invoked when someone presses the *clickMe* button. The *OnClickMeClicked()* method, shown below, conforms to the signature of the *EventHandler* *delegate*, which you can look up in the .NET Framework Class Library reference.

public void OnClickMeClicked(object sender, EventArgs ea)  
{  
MessageBox.Show("You Clicked My Button!");  
}

Any time the *clickMe* button is pressed with a mouse, it will fire the *Clickevent*, which will invoke the *OnClickMeClicked()* method. The *Button* class takes care of firing the *Click* *event* and there's nothing more you have to do. Because it is so easy to use pre-defined *events* and *delegates*, it would be a good idea to check if some exist already that will do what you need, before creating your own.

**Summary**

This completes this lesson, which was an introduction to *delegates* and *events*. You learned how to declare and implement *delegates*, which provide dynamic run-time method invocation services. You also know how to declare *events* and use them in a couple different scenarios. One way is to declare your own *event*, *delegate*, and callback method from scratch. Another way is to use pre-existing *events* and *delegates* and only implement the callback method, which will save you time and make coding easier.

I invite you to return for [Lesson 15: Introduction to Exception Handling](http://csharp-station.com/Tutorial/CSharp/Lesson15).

# Lesson 15: Introduction to Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally** and **throw**.

* **try**: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
* **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
* **finally**: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
* **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

## Syntax

Assuming a block will raise and exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

try

{

// statements causing exception

}

catch( ExceptionName e1 )

{

// error handling code

}

catch( ExceptionName e2 )

{

// error handling code

}

catch( ExceptionName eN )

{

// error handling code

}

finally

{

// statements to be executed

}

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

## Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class:

|  |  |
| --- | --- |
| **Exception Class** | **Description** |
| System.IO.IOException | Handles I/O errors. |
| System.IndexOutOfRangeException | Handles errors generated when a method refers to an array index out of range. |
| System.ArrayTypeMismatchException | Handles errors generated when type is mismatched with the array type. |
| System.NullReferenceException | Handles errors generated from deferencing a null object. |
| System.DivideByZeroException | Handles errors generated from dividing a dividend with zero. |
| System.InvalidCastException | Handles errors generated during typecasting. |
| System.OutOfMemoryException | Handles errors generated from insufficient free memory. |
| System.StackOverflowException | Handles errors generated from stack overflow. |

## Handling Exceptions

C# provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch** and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs:

Create your own Console Application and add the following Code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace ExceptionHandling

{

class Program

{

int result;

Program()

{

result = 0;

}

public void division(int num1, int num2)

{

try

{

result = num1 / num2;

}

catch (DivideByZeroException e)

{

Console.WriteLine("Exception caught: {0}",e);

}

finally

{

Console.WriteLine("Result: {0}", result);

}

}

static void Main(string[] args)

{

int num1, num2;

Program obj = new Program();

Console.WriteLine("Enter num1");

num1 = Convert.ToInt16(Console.ReadLine());

Console.WriteLine("Enter num2");

num2 = Convert.ToInt16(Console.ReadLine());

obj.division(num1, num2);

}

}

}

**User-Defined Exceptions:**

You can create your own exceptions. User-defined exceptions must inherit from the Exception base class. We will create another separate class that inherits from the Exception base class. Create a new Console Application and name it UserDefinedExceptions. After the project is created, click the Add New Item button in the toolbar and browse for the Class template. Name the file NegativeNumberException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System;    namespace UserDefinedExceptions  {  class NegativeNumberException : Exception  {  public NegativeNumberException()  : base("The operation will result to a negative number.")  {  }    public NegativeNumberException(string message)  : base(message)  {  }    public NegativeNumberException(string message, Exception inner)  : base(message, inner)  {  }  }  } |

Example 1 - NegativeNumberException Class

You can see in line 5 that our custom class inherits from the Exception class. As a convention, user-defined exception class names must be appended with the word "Exception" and it should define 3 constructors. The first one is the parameterless constructor, the second constructor accepts a string argument which defines the error message, and the third constructor accepts the error message, and the inner exception the caused the current exception.

Let's try to use our very own exception class. In the Program.cs file, use the following codes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | using System;    namespace UserDefinedExceptions  {  class Program  {  public static void Main()  {  int firstNumber, secondNumber, difference;    Console.Write("Enter the first number: ");  firstNumber = Int32.Parse(Console.ReadLine());    Console.Write("Enter the second number: ");  secondNumber = Int32.Parse(Console.ReadLine());    difference = firstNumber - secondNumber;    try  {  if (difference < 0)  {  throw new NegativeNumberException();  }  }  catch (NegativeNumberException error)  {  Console.WriteLine(error.Message);  }  }  }  } |

Example 2 - Using NegativeNumberException

Enter the first number: 10Enter the second number: 11

The operation will result to a negative number.

Since yielding a negative number as a result will not issue any kind of exception, we need to manually throw it ourselves . We ask two values from the user (line 11-15). We then calculated the difference of the two (line 17). Inside the try block, we tested if the result of the operation is a negative number (line 21). If it is, then we throw a new instance of the NegativeNumberException class (line 23). This will then be catched by the catch block and show the error message (line 24-26).

# Lesson 16: Using Attributes

This lesson explains how to use C# attributes. Our objectives are as follows:

* Understand what attributes are and why they're used
* Apply various attributes with multiple or no parameters
* Use assembly, type member, and type level attributes

**Why Attributes?**

Attributes are elements that allow you to add declarative information to your programs. This declarative information is used for various purposes during runtime and can be used at design time by application development tools. For example, there are attributes such as DllImportAttribute that allow a program to communicate with the Win32 libraries. Another attribute, ObsoleteAttribute, causes a compile-time warning to appear, letting the developer know that a method should no longer be used. When building Windows forms applications, there are several attributes that allow visual components to be drag-n-dropped onto a visual form builder and have their information appear in the properties grid. Attributes are also used extensively in securing .NET assemblies, forcing calling code to be evaluated against pre-defined security constraints. These are just a few descriptions of how attributes are used in C# programs.

The reason attributes are necessary is because many of the services they provide would be very difficult to accomplish with normal code. You see, attributes add what is called metadata to your programs. When your C# program is compiled, it creates a file called an assembly, which is normally an executable or DLL library. Assemblies are self-describing because they have metadata written to them when they are compiled. Via a process known as reflection, a program's attributes can be retrieved from its assembly metadata. Attributes are classes that can be written in C# and used to decorate your code with declarative information. This is a very powerful concept because it means that you can extend your language by creating customized declarative syntax with attributes.

This tutorial will show how to use pre-existing attributes in C# programs. Understanding the concepts and how to use a few attributes, will help in finding the multitude of other pre-existing attributes in the .NET class libraries and use them also.

**Attribute Basics**

Attributes are generally applied physically in front of type and type member declarations. They're declared with square brackets, "[" and "]", surrounding the attribute such as the following *ObsoleteAttribute* attribute:

[ObsoleteAttribute]

The "Attribute" part of the attribute name is optional. So the following is equivalent to the attribute above:

[Obsolete]

You'll notice that the attribute is declared with only the name of the attribute, surrounded by square brackets. Many attributes have parameter lists, that allow inclusion of additional information that customizes a program even further. Listing 16.1 shows various ways of how to use the *ObsoleteAttribute* attribute.

**Listing 16-1. How to Use Attributes: BasicAttributeDemo.cs**

using System;  
  
class BasicAttributeDemo  
{  
[Obsolete]  
publicvoid MyFirstdeprecatedMethod()  
{  
Console.WriteLine("Called MyFirstdeprecatedMethod().");  
}  
  
[ObsoleteAttribute]  
publicvoid MySecondDeprecatedMethod()  
{  
Console.WriteLine("Called MySecondDeprecatedMethod().");  
}  
  
[Obsolete("You shouldn't use this method anymore.")]  
publicvoid MyThirdDeprecatedMethod()  
{  
Console.WriteLine("Called MyThirdDeprecatedMethod().");  
}  
  
// make the program thread safe for COM  
[STAThread]  
staticvoid Main(string[] args)  
{  
BasicAttributeDemo attrDemo = new BasicAttributeDemo();  
  
attrDemo.MyFirstdeprecatedMethod();  
attrDemo.MySecondDeprecatedMethod();  
attrDemo.MyThirdDeprecatedMethod();  
}  
}

Examining the code in listing 16-1 reveals that the *ObsoleteAttribute* attribute was used a few different ways. The first usage appeared on the *MyFirstdeprecatedMethod()* method and the second usage appeared in the *MySecondDeprecatedMethod()* method as follows:

[Obsolete]  
publicvoid MyFirstdeprecatedMethod()  
...  
[ObsoleteAttribute]  
publicvoid MySecondDeprecatedMethod()  
...

The only difference between the two attributes is that *MySecondDeprecatedMethod()* method contains the "Attribute" in the attribute declaration. The results of both attributes are exactly the same. Attributes may also have parameters, as shown in the following declaration:

[Obsolete("You shouldn't use this method anymore.")]  
publicvoid MyThirdDeprecatedMethod()  
...

This adds customized behavior to the *ObsoleteAttribute* attribute which produces different results from the other *ObsoleteAttribute* attribute declarations. The results of all three *ObsoleteAttribute* attributes are shown below. These are the warnings that are emitted by the C# compiler when the program is compiled:

>csc BasicAttributeDemo.cs

Microsoft (R) Visual C# .NET Compiler version 7.10.2292.4

for Microsoft (R) .NET Framework version 1.1.4322

Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

BasicAttributeDemo.cs(29,3): warning CS0612:

'BasicAttributeDemo.MyFirstdeprecatedMethod()' is obsolete

BasicAttributeDemo.cs(30,3): warning CS0612:

'BasicAttributeDemo.MySecondDeprecatedMethod()' is obsolete

BasicAttributeDemo.cs(31,3): warning CS0618:

'BasicAttributeDemo.MyThirdDeprecatedMethod()' is obsolete: 'You

shouldn't use this method anymore.'

As you can see, the *ObsoleteAttribute* attribute caused the *MyThirdDeprecatedMethod()* method to emit the message that was a parameter to the *ObsoleteAttribute* attribute of that method in the code. The other attributes simply emitted standard warnings.

Listing 16-1 also contains another attribute you're likely to see, the *STAThreadAttribute* attribute. You'll often see this attribute applied to the *Main()* method, indicating that this C# program should communicate with unmanaged COM code using the Single Threading Apartment . It is generally safe to use this attribute all the time because you never know when a 3rd party library you're using is going to be communicating with COM. The following excerpt shows how to use the *STAThreadAttribute* attribute:

[STAThread]  
staticvoid Main(string[] args)  
...

**Attribute Parameters**

Attributes often have parameters that enable customization. There are two types of parameters that can be used on attributes, positional and named. Positional parameters are used when the attribute creator wishes the parameters to be required. However, this is not a hard and fast rule because the *ObsoleteAttribute* attribute has a second positional parameter named error of type *bool* that we can omit as demonstrated in Listing 16-1. That attribute could have been written with the second positional parameter to force a compiler error instead of just a warning as follows:

[Obsolete("You shouldn't use this method anymore.", true)]  
publicvoid MyThirdDeprecatedMethod()  
...

The difference between positional parameters and named parameters are that named parameters are specified with the name of the parameter and are always optional. The *DllImportAttribute* attribute is one you are likely to see that has both positional and named parameters (Listing 16-2).

**Listing 16-2. Using Positional and Named Attribute Parameters: AttributeParamsDemo.cs**

using System;  
using System.Runtime.InteropServices;  
  
class AttributeParamsDemo  
{  
[DllImport("User32.dll", EntryPoint="MessageBox")]  
staticextern int MessageDialog(int hWnd, string msg, string caption, int msgType);  
  
[STAThread]  
staticvoid Main(string[] args)  
{  
MessageDialog(0, "MessageDialog Called!", "DllImport Demo", 0);  
}  
}

The *DllImportAttribute* attribute in Listing 16-2 has one positional parameter, *"User32.dll"*, and one named parameter, *EntryPoint="MessageBox"*. Positional parameters are always specified before any named parameters. When there are named parameters, they may appear in any order. This is because they are marked with the parameter name like in the *DllImportAttribute* attribute, *EntryPoint="MessageBox"*. Since the purpose of this lesson is to explain how to use attributes in general, I won't go into the details of the *DllImportAttribute* attribute, which has extra parameters that require knowledge of Win32 and other details that don't pertain to this lesson. Many other attributes can be used with both positional and named parameters.

**Attribute Targets**

The attributes shown so far have been applied to methods, but there are many other C# language elements that you can use attributes with. table 16-1 outlines the C# language elements that attributes may be applied to. They are formally called attribute "targets".

|  |  |
| --- | --- |
| **Attribute Target** | **Can be Applied To** |
| all | everything |
| assembly | entire assembly |
| class | classes |
| constructor | constructors |
| delegate | delegates |
| enum | enums |
| event | events |
| field | fields |
| interface | interfaces |
| method | methods |
| module | modules (compiled code that can be part of an assembly) |
| parameter | parameters |
| property | properties |
| returnvalue | return values |
| struct | structures |

Whenever there is ambiguity in how an attribute is applied, you can add a target specification to ensure the right language element is decorated properly. An attribute that helps ensure assemblies adhere to the Common Language Specification (CLS) is the *CLSCompliantAttribute* attribute. The CLS is the set of standards that enable different .NET languages to communicate. Attribute targets are specified by prefixing the attribute name with the target and separating it with a colon (*:*). Listing 16-3 shows how to use the *CLSCompliantAttribute* attribute and apply it to the entire assembly.

**Listing 16-3. Using Positional and Named Attribute Parameters: AttributeTargetdemo.cs**

using System;  
  
[assembly:CLSCompliant(true)]  
  
public class AttributeTargetdemo  
{  
publicvoid NonClsCompliantMethod(uint nclsParam)  
{  
Console.WriteLine("Called NonClsCompliantMethod().");  
}  
  
[STAThread]  
staticvoid Main(string[] args)  
{  
uint myUint = 0;  
  
AttributeTargetdemo tgtdemo = new AttributeTargetdemo();  
  
tgtdemo.NonClsCompliantMethod(myUint);  
}  
}

The code in Listing 16-3 will generate a compiler warning because of the *uint* type parameter declared on the *NonClsCompliantMethod()* method. If you change the *CLSCompliantAttribute* attribute to false or change the type of the *NonClsCompliantMethod()* method to a CLS compliant type, such as *int*, the program will compile without warnings.

The point about Listing 16-3 is that the *CLSCompliantAttribute* attribute is decorated with an attribute target of *"assembly"*. This causes all members of this assembly to be evaluated according to the *CLSCompliantAttribute* attribute setting. To limit the scope of the *CLSCompliantAttribute*, apply it to either the *AttributeTargetdemo* class or *NonClsCompliantMethod()* method directly.

**Summary**

Attributes are C# language elements that decorate program elements with additional metadata that describes the program. This metadata is then evaluated at different places, such as runtime or design time for various purposes. The examples in this lesson showed how the *ObsoleteAttribute* attribute could be used to generate compile time warnings for deprecated code. Through applying the *DllImportAttribute* attribute, you could see how to apply both positional and named parameters to an attribute. Attributes may also be used to decorate various different types of program elements with a target descriptor. The example applied the *CLSCompliantAttribute* attribute to an entire assembly. However, it could have also been applied to different program elements with applicable target descriptors to limit its scope.

I invite you to return for [Lesson 17: Enums](http://csharp-station.com/Tutorial/CSharp/Lesson17).

# Lesson 17: Enums

This lesson explains how to use C# enums. Our objectives are as follows:

* Understand what an enum is
* Be able to create new enum types
* Learn how to use enums
* Gain familiarity with System.Enum type methods

**Enums Defined**

Enums are strongly typed constants. They are essentially unique types that allow you to assign symbolic names to integral values. In the C# tradition, they are strongly typed, meaning that an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members are the same. Along the same lines, integral types and enums are not implicitly interchangable. All assignments between different enum types and integral types require an explicit cast.

Enums lend themselves to more maintainable code because they are symbolic, allowing you to work with integral values, but using a meaningful name to do so. For example, what type of code would you rather work with - a set of values named North, South, East, and West or the set of integers 0, 1, 2, and 3 that mapped to the same values, respectively? Enums make working with strongly typed constants via symbolic names easy.

Enums are value types, which means they contain their own value, can't inherit or be inherited from, and assignment copies the value of one enum to another. You will see in this lesson and elsewhere that enums are used and referred to with both lower case, *enum*, and upper case, *Enum*. The relationship between the two is that the C# type, *enum*, inherits the Base Class Library (BCL) type, *Enum*. Use the C# type, *enum*, to define new enums and use the BCL type, *Enum*, to implement static enum methods.

**Creating an Enum**

The .NET Framework Class Library contains many enums and examples of how they are used. For example, every time you put an icon on a *MessageBox*, you use the *MessageBoxIcon* enum. For a list of available enums in the .NET Framework Class Library, look at the documentation for the *Enum* class and click on the Derived Classes link.

Whenever there are situations where you are using a set of related numbers in a program, consider replacing those numbers with enums. It will make a program more readable and type safe. Listing 17-1 contains an *enum* definition and code that uses that enum in a *switch* statement. Instead of using the numbers 0, 1, and 2 in the *switch* statement, the code is more meaningful through the use of the *Volume* enum.

**Listing 17-1. Creating and Using an Enum: EnumSwitch.cs**

using System;  
  
// declares the enum  
public enum Volume  
{  
Low,  
Medium,  
High  
}  
  
// demonstrates how to use the enum  
  
class EnumSwitch  
{  
static void Main()  
{  
// create and initialize   
// instance of enum type  
Volume myVolume = Volume.Medium;  
  
// make decision based  
// on enum value  
switch (myVolume)  
{  
case Volume.Low:  
Console.WriteLine("The volume has been turned Down.");  
break;  
case Volume.Medium:  
Console.WriteLine("The volume is in the middle.");  
break;  
case Volume.High:  
Console.WriteLine("The volume has been turned up.");  
break;  
}  
Console.ReadLine();  
}  
}

Listing 17-1 contains a definition for an enum. Notice that it is declared with the *enum* keyword, has a type identifier (*Volume*), and contains a comma separated list of values enclosed within curly braces.

This enum is of type *Volume* and we use it to declare the *myVolume* variable in the *Main* method. Since an enum is a value type, we can assign a value (*Volume.Medium*) to it directly, similar to the simple types such as *int* or *double*. Once the *myVolume* variable is declared and initialized, it is used in the *switch* statement.Each of the *case* statements represent a unique member of the *Volume* enum.

Any time a member of the *Volume* enum is used, it is fully qualified with the "*Volume*" identifier to guarantee type safety. For example, if there were a *Meat* enum in scope, then *Meat.Medium* would definitely have different semantics than *Volume.Medium*. With both enums in scope, it would be ambiguous to just use the *Medium* identifier without type qualification. Using the type identifier ensures such mistakes are not made.

**Using Enums**

An enum is typically specified as shown in Listing 17-1, but may be customized by changing its base type and member values. By default, the underlying type of an enum is *int*. This default may be changed by specifying a specific base when declaring the enum. You would specify a different base if the enum was used extensively and there was an opportunity for space savings by selecting a smaller type. Another reason may be if you wanted the underlying type of the enum to correspond to another type in your program and you wanted to explicitly cast between the two without loss of precision. Valid base types include *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long*, and *ulong*.

Another modification you can make to an enum is to set the value of any enum member. By default, the first member of an enum takes the value of zero. If this value doesn't make sense for your enum, you can change it to one or some other number. Additionally, you can change any of the members of an enum to any value that is valid for its base type. Unassigned enum members have a value that is one more than their predecessor. Listing 17-2 shows how to modify the base type and member values of an enum.

**Listing 17-2. Setting the Enum Base and Initializing Members: EnumBaseAndMembers.cs**

using System;  
  
// declares the enum  
public enum Volume : byte  
{  
Low = 1,  
Medium,  
High  
}  
  
class EnumBaseAndMembers  
{  
staticvoid Main()  
{  
// create and initialize   
// instance of enum type  
Volume myVolume = Volume.Low;  
  
// make decision based  
// on enum value  
switch (myVolume)  
{  
case Volume.Low:  
Console.WriteLine("The volume has been turned Down.");  
break;  
case Volume.Medium:  
Console.WriteLine("The volume is in the middle.");  
break;  
case Volume.High:  
Console.WriteLine("The volume has been turned up.");  
break;  
}  
Console.ReadLine();  
}  
}

The *Volume* enum in Listing 17-2 shows how to modify the base type and members of an enum.Its base type is changed to *byte* with the *: <type>* syntax following the enum identifier, *Volume*.This ensures that the *Volume* enum may only have members with values that are valid for type *byte*.

The first member of the *Volume* enum, *Low*, has its value changed to 1. The same syntax, *<member> = <value>*, may be applied to any member of the enum. You are restricted from creating forward references, circular references, and duplicate references in enum members.

The default values of the *Volume* enum are *Low*=0, *Medium*=1, and *High*=2 because the first member of an enum defaults to 0 and the following members default to one more than their predecessor. However, the *Volume* enum in Listing 17-2 has its *Low* member set to 1, which means that *Medium*=2 and *High*=3.

**Enum tricks**

Enum types implicitly inherit the System.Enum type in the Base Class Library (BCL). This also means that you can use the members of System.Enum to operate on enum types. This section does just that, showing some useful tips and tricks to use with enums in your programs.

A common requirement with enums is to convert between the enum and a variable of its base type. For example, if you are getting input in the form of an *int* from a user or a file stream, then you can cast it to an enum and use it in a meaningful way in your program. You can also get a complete list of enum member names or enum values, which is useful if you have logic that needs to iterate through every enum member. Listing 17-3 shows how to perform conversions between enums and their base types and how to use some of the System.Enum type members.

**Listing 17-3. Enum Conversions and using the System.Enum Type: Enumtricks.cs**

using System;  
  
// declares the enum  
public enum Volume : byte  
{  
Low = 1,  
Medium,  
High  
}  
  
// shows different ways  
// to work with enums  
class Enumtricks  
{  
staticvoid Main(string[] args)  
{  
// instantiate type  
Enumtricks enumtricks = new Enumtricks();  
  
// demonstrates explicit cast  
// of int to Volume  
enumtricks.GetEnumFromUser();  
  
// iterate through Volume enum by name  
enumtricks.ListEnumMembersByName();  
  
// iterate through Volume enum by value  
enumtricks.ListEnumMembersByValue();  
  
Console.ReadLine();  
}  
  
// demonstrates explicit cast  
// of int to Volume  
publicvoid GetEnumFromUser()  
{  
Console.WriteLine("\n----------------");  
Console.WriteLine("Volume Settings:");  
Console.WriteLine("----------------\n");  
  
Console.Write(@"  
1 - Low  
2 - Medium  
3 - High  
  
Please select one (1, 2, or 3): ");  
  
// get value user provided  
string volString = Console.ReadLine();  
int volInt = Int32.Parse(volString);  
  
// perform explicit cast from  
// int to Volume enum type  
Volume myVolume = (Volume)volInt;  
  
Console.WriteLine();  
  
// make decision based  
// on enum value  
switch (myVolume)  
{  
case Volume.Low:  
Console.WriteLine("The volume has been turned Down.");  
break;  
case Volume.Medium:  
Console.WriteLine("The volume is in the middle.");  
break;  
case Volume.High:  
Console.WriteLine("The volume has been turned up.");  
break;  
}  
  
Console.WriteLine();  
}  
  
// iterate through Volume enum by name  
publicvoid ListEnumMembersByName()  
{  
Console.WriteLine("\n---------------------------- ");  
Console.WriteLine("Volume Enum Members by Name:");  
Console.WriteLine("----------------------------\n");  
  
// get a list of member names from Volume enum,  
// figure out the numeric value, and display  
foreach (string volume in Enum.GetNames(typeof(Volume)))  
{  
Console.WriteLine("Volume Member: {0}\n Value: {1}",   
volume, (byte)Enum.Parse(typeof(Volume), volume));  
}  
}  
  
// iterate through Volume enum by value  
publicvoid ListEnumMembersByValue()  
{  
Console.WriteLine("\n----------------------------- ");  
Console.WriteLine("Volume Enum Members by Value:");  
Console.WriteLine("-----------------------------\n");  
  
// get all values (numeric values) from the Volume  
// enum type, figure out member name, and display  
foreach (byte val in Enum.GetValues(typeof(Volume)))  
{  
Console.WriteLine("Volume Value: {0}\n Member: {1}",   
val, Enum.GetName(typeof(Volume), val));  
}  
}  
}

The code in Listing 17-3 includes three method calls to *GetEnumFromUser*, *ListEnumMembersByName*, and *ListEnumMembersByValue*. Each of these methods demonstrate a different aspect of using System.Enum to work with enums.

The *GetEnumFromUser* method shows how to obtain *int* input and translate it to an appropriate enum type. Converting an *int* to an enum makes the code more readable and type safe. The following is an excerpt from Listing 17-3 that shows the pertinent part of the code that performs the conversion:

// get value user provided  
string volString = Console.ReadLine();  
int volInt = Int32.Parse(volString);  
  
// perform explicit cast from  
// int to Volume enum type  
Volume myVolume = (Volume)volInt;

After the program displays a menu, it prompts the user for a selection in the form of a number (1, 2, or 3). When the user makes a selection and presses the Enter key, the code reads the value with *Console.ReadLine*, which returns the value as a *string* type. Since you can only cast an *int* to a *Volume* enum type, the user's input must be converted from a *string* to an *int* with the *Int32.Parse* method. Converting the *in*t to a *Volume* enum type is simply a matter of applying a cast operation during assignment.

To get all the members of an enum at the same time, you can use the *GetNames* method of the *System.Enum* type, which returns a *string* array of the names of all an enum's members. An excerpt from the *ListEnumMembersByName* method in Listing 17.3 that shows this appears below:

// get a list of member names from Volume enum,  
// figure out the numeric value, and display  
foreach (string volume in Enum.GetNames(typeof(Volume)))  
{  
Console.WriteLine("Volume Member: {0}\n Value: {1}",   
volume, (byte)Enum.Parse(typeof(Volume), volume));  
}

Because *GetNames* returns an array of *strings*, it is easy to use in a loop statement such as *foreach*. Something you may be curious about in the code above is the second parameter to the *WriteLine* method's format string. Given the enum type and a string representation of the member name, you can use the *Enum.Parse* method to get the underlying value of that member. Because the *Volume* enum's base type is byte, the return value from *Enum.Parse* must be cast to a *byte* before assignment, forcing the numeric representation of the enum value to appear. If we would have omitted the *byte* cast, the output would be the Volume enum member, which would then be converted to a string representation of the member name, which is not what the code intended to show.

Instead of getting names of all the members of an enum, you may have a reason to get all the values of the enum at one time. The code below, from the *ListEnumMembersByValue* method in Listing 17.3, shows how to accomplish this:

// get all values (numeric values) from the Volume  
// enum type, figure out member name, and display  
foreach (byte val in Enum.GetValues(typeof(Volume)))  
{  
Console.WriteLine("Volume Value: {0}\n Member: {1}",   
val, Enum.GetName(typeof(Volume), val));  
}

Given the type of the enum, the *GetValues* method of System.Enum will return an array of the given enum's base type, which in this case is *byte*. While iterating through this list, each member is printed to the console showing its value and name. The name is obtained by using the *GetName* method of System.Enum, which accepts an enum type and value for which to get the corresponding name of.

**Summary**

Enums are lists of strongly typed constants with members that are symbolic names, corresponding to an underlying integral type. Enum base types can be changed and member values can be specified. The System.Enum .NET Framework Class Library type is the base class of enum types and contains methods that allow you to work with enums in different ways, such as working with a list of names or values, converting from value to name, and converting from name to value. For more information on the System.Enum type, see the .NET Framework SDK documentation.

I invite you to return for [Lesson 18: Overloading Operators](http://csharp-station.com/Tutorial/CSharp/Lesson18).

# Lesson 18: Overloading Operators

This lesson shows you how to overload C# operators. Our objectives are as follows:

* Understand what operator overloading is
* Determine when it is appropriate to overload an operator
* Learn how to overload an operator
* Familiarize yourself with rules for operator overloading

**About Operator Overloading**

In [Lesson 2](http://csharp-station.com/Tutorial/CSharp/Lesson02), you learned what operators were available in C#, which included *+* (plus), *-* (minus), *^* (exclusive or), and others. Operators are defined for the built-in types, but that's not all. You can add operators to your own types, allowing them to be used much like the operators with the built-in C# types.

To understand the need for operator overloading, imagine that you need to perform matrix math operations in your program. You could instantiate a couple 2-dimensional arrays and do what you need. However, add the requirement for the matrix behavior to be reusable. Because you need to do the same thing in other programs and want to take advantage of the fact that you have already written the code, you will want to create a new type.

So, you create a Matrix type, which could be a class or a struct. Now consider how this Matrix type would be used. You would want to initialize two or more Matrix instances with data and then do a mathematical operation with them, such as add or get a dot product. To accomplish the mathematical operation, you could implement an *Add()*, *DotProduct()*, and other methods to get the job done. Using the classes would look something like this:

Matrix result = mat1.Add(mat2); // instance

or

Matrix result = Matrix.Add(mat1, mat2); // static

or even worse

Matrix result = mat1.DotProduct(mat2).DotProduct(mat3); // and so on...

The problem with using methods like this is that it is cumbersome, verbose, and unnatural for the problem you are trying to solve. It would be much easier to have a *+* operator for the add operation and a *\** operator for the dot product operation. The following shows how the syntax appears using operators:

Matrix result = mat1 + mat2;

or

Matrix result = mat1 \* mat2;

or even better

Matrix result = mat1 \* mat2 \* mat3 \* mat4;

This is much more elegant and easier to work with. For a single operation, one could argue that the amount of work to implement one syntax over the other is not that great. However, when chaining multiple mathematical operations, the syntax is much simpler. Additionally, if the primary users of your type are mathematicians and scientists, operators are more intuitive and natural.

**When Not to Use Operator Overloading**

A lot of the discussion, so far, has emphasized the need to write code and implement types in the simplest and most natural way possible. A very important concept to remember is that although operators are simple, they are not always natural. In the example above it made sense to use operators with the *Matrix* type. This is similar to the reason why operators make sense with the built-in types such as *int* and *float*. However, it is easy to abuse operators and create convoluted implementations that are hard for anyone, including the original author, to understand.

For an example of a bad implementation, consider a *Car* class that needs an implementation allowing you to park the car in a garage. It would be a mistake to think that the following implementation was smart:

Car mySedan = new Car();

Garage parkingGarage = new Garage();

mySedan = mySedan + parkingGarage; // park car in the garage

This is bad code. If you ever have the temptation to do something like this - don't. No one will truly understand what it means and they will not think it is clever. Furthermore, it hurts the maintainability of the application because it is so hard to understand what the code does. Although the comment is there, it doesn't help much and if it wasn't there, it would be even more difficult to grasp the concept of adding a *Car* and a *Garage*.

The idea is this: Use operators where they lend understanding and simplicity to a type. Otherwise, do not use them.

**Implementing an Overloaded Operator**

The syntax required to implement an overloaded operator is much the same as a static method with a couple exceptions. You must use the *operator* keyword and specify the operator symbol being overloaded. Here's a skeleton example of how the dot product operator could be implemented:

public static Matrix operator \*(Matrix mat1, Matrix mat2)

{

// dot product implementation

}

Notice that the method is static. Use the keyword *operator* after specifying the return type, *Matrix* in this case. Following the *operator* keyword, the actual operator symbol is specified and then there is a set of parameters to be operated on. See Listing 18-1 for a full example of how to implement and use an overloaded operator.

**Listing 18-1. Implementing an Overloaded Operator: Matrix.cs**

using System;

class Matrix

{

public const int DimSize = 3;

private double[,] m\_matrix = new double[DimSize, DimSize];

// allow callers to initialize

public double this[int x, int y]

{

get { return m\_matrix[x, y]; }

set { m\_matrix[x, y] = value; }

}

// let user add matrices

public static Matrix operator +(Matrix mat1, Matrix mat2)

{

Matrix newMatrix = new Matrix();

for (int x=0; x < DimSize; x++)

for (int y=0; y < DimSize; y++)

newMatrix[x, y] = mat1[x, y] + mat2[x, y];

return newMatrix;

}

}

class MatrixTest

{

// used in the InitMatrix method.

public static Random m\_rand = new Random();

// test Matrix

static void Main()

{

Matrix mat1 = new Matrix();

Matrix mat2 = new Matrix();

// init matrices with random values

InitMatrix(mat1);

InitMatrix(mat2);

// print out matrices

Console.WriteLine("Matrix 1: ");

PrintMatrix(mat1);

Console.WriteLine("Matrix 2: ");

PrintMatrix(mat2);

// perform operation and print out

results

Matrix mat3 = mat1 + mat2;

Console.WriteLine();

Console.WriteLine("Matrix 1 + Matrix

2 = ");

PrintMatrix(mat3);

Console.ReadLine();

}

// initialize matrix with random values

public static void InitMatrix(Matrix mat)

{

for (int x=0; x < Matrix.DimSize; x++)

for (int y=0; y < Matrix.DimSize; y++)

mat[x, y] = m\_rand.NextDouble();

}

// print matrix to console

public static void PrintMatrix(Matrix mat)

{

Console.WriteLine();

for (int x=0; x < Matrix.DimSize; x++)

{

Console.Write("[ ");

for (int y=0; y < Matrix.DimSize; y++)

{

// format the output

Console.Write("{0,8:#.000000}", mat[x, y]);

if ((y+1 % 2) < 3)

Console.Write(", ");

}

Console.WriteLine(" ]");

}

Console.WriteLine();

}

}

Similar to the skeleton example of the dot product operator, the Matrix class in Listing 18-1 contains an operator overload for the + operator. For your convenience, I've extracted the pertinent overload implementation in the code below:

// let user add matrices

public static Matrix operator +(Matrix mat1, Matrix mat2)

{

Matrix newMatrix = new Matrix();

for (int x=0; x < DimSize; x++)

for (int y=0; y < DimSize; y++)

newMatrix[x, y] = mat1[x, y] + mat2[x, y];

return newMatrix;

}

The operator is *static*, which is the only way it can and should be declared because an operator belongs to the type and not a particular instance. There are just a few rules you have to follow when implementing operator overloads. What designates this as an operator is the use of the keyword *operator*, followed by the *+* symbol. The parameter types are both of the enclosing type, *Matrix*. The implementation of the operator overload creates a new instance of the return type and performs a matrix add.

**Operator Rules**

C# enforces certain rules when you overload operators. One rule is that you must implement the operator overload in the type that will use it. This is sensible because it makes the type self-contained.

Another rule is that you must implement matching operators. For example, if you overload ==, you must also implement !=. The same goes for <= and >=.

When you implement an operator, its compound operator works also. For example, since the *+* operator for the *Matrix* type was implemented, you can also use the *+=* operator on *Matrix* types.

**Summary**

Operator overloading allows you to implement types that behave like the built-in types when using operators. Be sure to use operators in a way that is natural and understandable for the type. Syntax for implementing operators is much like a static method, but includes the *operator* keyword and the operator symbol in place of an identifier. Additionally, there are rules, such as maintaining symmetry,for using operators, which encourage construction of robust types.

I invite you to return for [Lesson 19: Encapsulation](http://csharp-station.com/Tutorial/CSharp/Lesson19).

# Lesson 19: Encapsulation

Earlier in this tutorial, you learned about two of the important principles of object-oriented programming, [Inheritance](http://csharp-station.com/Tutorial/CSharp/Lesson08) and [Polymorphism](http://csharp-station.com/Tutorial/CSharp/Lesson09). Now that you've seen much of the syntax of C#, I'll show you how C# supports the another of the object-oriented principles - Encapsulation. This lesson will discuss Encapsulation with the following objectives:

* Understand the object-oriented principle of Encapsulation.
* Learn the available modifiers for type members.
* Protect object state through properties.
* Control access to methods.
* Learn how to modify types for assembly encapsulation

**What is Encapsulation and How Does It Benefit Me?**

In object-oriented programming, you create objects that have state and behavior. An object's state is the data or information it contains. For example, if you have a *BankAccount* object, its state could be *Amount* and *CustomerName*. Behavior in an object is often represented by methods. For example, the BankAccount object's behavior could be *Credit*, *Debit*, and *GetAmount*. This sounds like a nice definition of an object, and it is, but you must also consider how this object will be used.

When designing an object, you must think about how others could use it. In a best-case scenario any program using the object would be well designed and the code would never change. However, the reality is that programs do change often and in a team environment many people touch the same code at one time or another. Therefore, it is beneficial to consider what could go wrong as well as the pristine image of how the object \*should\* be used.

In the case of the *BankAccount* object, examine the situation where code outside of your object could access a *decimal* *Amount* field or a *string* *CustomerName* field. At the point of time that the code is written, everything would work well. However, later in the development cycle, you realize that the *BankAccount* object should keep track of an *intCustomerID* rather than *string CustomerName* because you don't want to duplicate relationships between information (or some other valid reason to alter the definition of internal state). Such changes cause a rippling effect in your code because it was built to use the *BankAccount* class, as originally designed (with *CustomerName* being a string), and you must now change code that accesses that state throughout your entire application.

The object-oriented principle of Encapsulation helps avoid such problems, allowing you to hide internal state and abstract access to it though type members such as methods, properties, and indexers. Encapsulation helps you reduce coupling between objects and increases the maintainability of your code.

**Type Member Access Modifiers**

An access modifier allows you to specify the visibility of code outside a type or assembly. Access modifiers can be applied to either types or type members. A later section on Type Access Modifiers discusses modifiers that can be applied to types. This section discusses those modifiers that apply to type members and how they affect visibility.

Generally, you should hide the internal state of your object from direct access from outside code. Then implement other members, such as methods and properties, that wrap that state. This allows the internal implementation of the state to change at will, while the members wrapping the state can still return a representation of the state that doesn't change. This means that outside code will access your object via members that wrap state and be guaranteed that the type of information extracted is consistent. Additionally, because external code doesn't have access to the internal state of your object, they can't alter that state in an inconsistent manner that could break the way your object works.

The first step in encapsulating object state is to determine what type of access that outside code should have to the members of your type. This is performed with access modifiers. The type of access granted varies from no external access at all to full public access and a few variations in between the extremes. Table 19-1 lists all of the type member access modifiers and explains their meaning.

**Table 19-1. Type member access modifiers control what code has access to a specified type member.**

|  |  |
| --- | --- |
| **Access Modifier** | **Description (who can access)** |
| private | Only members within the same type. (default for type members) |
| protected | Only derived types or members of the same type. |
| internal | Only code within the same assembly. Can also be code external to object as long as it is in the same assembly. (default for types) |
| protected internal | Either code from derived type or code in the same assembly. Combination of protected OR internal. |
| public | Any code. No inheritance, external type, or external assembly restrictions. |

As you've learned from previous lessons of the C# Tutorial, types contain several types of members, including constructors, properties, indexers, methods, and others. Rather than show you an exhaustive list of all of the permutations of access modifiers you can use with these members, I'll take a more practical approach and describe a sub-set of access modifiers used on properties and methods.

**Opening Type Members to *public* Access**

You've seen the *public* access modifier used in earlier parts of the C# Tutorial. Any time the *public* access modifier is used on a type member, calling code will be able to access the type member. If you make your type member public, you are giving everyone permission to use it. Listing 19-1 shows an example of using the *public* access modifier on a method.

**Listing 19-1. Declaring a Method with a public Access Modifier: BankAccountPublic.cs**

using System;  
  
class BankAccountPublic  
{  
publicdecimal GetAmount()  
{  
return1000.00m;  
}  
}

The *GetAmount()* method in Listing 19-1 is public meaning that it can be called by code that is external to this class. Now, you can write the following code, elsewhere in your program, to use this method:

BankAccountPublic bankAcctPub = newBankAccountPublic();  
  
// call a public method  
decimal amount = bankAcctPub.GetAmount();

All you need to do, as shown above, is create an instance of the class that contains the method and then call the method through that instance. Because it is *public*, you won't have a problem. Remember that the default access for a type member is *private*, which we'll talk about next. This means that if you forget the *public* modifier, and didn't use any modifier at all, you would receive a compiler error.

**Hiding Type Members with *private* Access**

A *private* type member is one that can only be accessed by members within the same type. For example, if the *BankAccount* class has a private member, only other members of the *BankAccount* class can access or call that member.

Although the default access for type members is *private*, I prefer to be explicit about my intentions when declaring type members and include the access modifier, rather than rely on defaults. I think it makes the code easier to read and makes it clear to other developers what my true intention is. Listing 19-2 shows how to use the *private* access modifier and offers an example of why you would want to use it.

**Listing 19-2. Declaring a private Field: BankAccountPrivate.cs**

using System;  
  
class BankAccountPrivate  
{  
privatestring m\_name;  
  
publicstring CustomerName  
{  
get { return m\_name; }  
set { m\_name = value; }  
}  
}

It's common to encapsulate the state of your type with properties. In fact, I always wrap my type state in a property. In Listing 19-2, you can see how the name of the customer is held in the *m\_name* field, but it is wrapped (encapsulated) with the *CustomerName* property. Because *m\_name* is declared as *private*, code outside the *BankAccountPrivate* class can't access it directly. They must use the *public* *CustomerName* property instead.

Now you can change the implementation of *m\_name* in any way you want. For example, what if you wanted it to be an ID of type *int* and the *CustomerName* property would do a search to find the name or what if you wanted to have first and last name values that the *CustomerName* property could concatenate. There are all kinds of things happening to your code in maintenance that will causes implementation to change. The point is that *private* members allow the implementation to change without constraining the implementation or causing rippling effects throughout your code base that would have occurred if that external code had access to the members of your type.

The *private* and *public* access modifiers are at the two extremes of access, either denying all external access or allowing all external access, respectively. The other access modifiers are like different shades of gray between these two extremes, including the *protected* modifier, discussed next.

**Access for Derived Types with the *protected* Access Modifier**

In some ways, the *protected* access modifier acts like both the *private* and *public* access modifiers. Like *private*, it only allows access to members within the same type, except that it acts like *public* only to derived types. Said another way, *protected* type members can only be accessed by either members within the same type or members of derived types.

Returning to the *BankAccount* example, what if you needed to call code to close an account? Furthermore, what if there were different types of accounts? Each of these different account types would have their own logic for closing, but the basic process would be the same for all account types. If this sounds to you like the description of Polymorphism, you would be on the right track. Back in [Lesson 9](http://csharp-station.com/Tutorial/CSharp/Lesson09), we discussed polymorphism and how it allows us to treat different classes the same way. You may want to refer to [Lesson 9](http://csharp-station.com/Tutorial/CSharp/Lesson09) for a refresher before looking at the next example.

In the case of closing an account, there are several things that need to be done like calculating interest that is due, applying penalties for early withdrawal, and doing the work to remove the account from the database. Individually, you don't want any code to call methods of the *BankAccount* class unless all of the methods are called and each method is called in the right order. For example, what if some code called the method to delete the account from the database and didn't calculate interest or apply penalties? Someone would lose money. Also, if the calling code were to delete the account first then the other methods would run into errors because the account information isn't available. Therefore, you need to control this situation and Listing 19-3 shows how you can do it.

**Listing 19-3. Declaring protected Methods: BankAccountProtected.cs**

using System;  
  
class BankAccountProtected  
{  
publicvoid CloseAccount()  
{  
ApplyPenalties();  
CalculateFinalInterest();  
DeleteAccountFromDB();  
}  
  
protectedvirtualvoid ApplyPenalties()  
{  
// deduct from account  
}  
  
protectedvirtualvoid CalculateFinalInterest()  
{  
// add to account  
}  
  
protectedvirtualvoid DeleteAccountFromDB()  
{  
// send notification to data entry personnel  
}  
}

The most important parts of Listing 19-3 are that the *CloseAccount* method is public and the other methods are protected. Any calling code can instantiate *BankAccountProtected*, but it can only call the *CloseAccount* method. This gives you protection from someone invoking the behavior of your object in inappropriate ways. Your business logic is sound.

At the end of this section, you'll see an example of how to call the code in Listing 19-3. For now, it is essential that you see how the other pieces fit together first.

If you only wanted the *BankAccountProtected* class to operate on its own members, you could have made the *protected* methods *private* instead. However, this code supports a framework where you can have different account types such as Savings, Checking, and more. You will be able to add new account types in the future because the *BankAccountProtected* class is designed to support them with *protected* virtual methods. Listings 19-4 and 19-5 show you the *SavingsAccount* and *CheckingAccount* classes that derive from the *BankAccountProtected* class.

**Listing 19-4. Derived SavingsAccount Class Using protected Members of its Base Class: SavingsAccount.cs**

using System;  
  
class SavingsAccount : BankAccountProtected  
{  
protectedoverridevoid ApplyPenalties()  
{  
Console.WriteLine("Savings Account Applying Penalties");  
}  
  
protectedoverridevoid CalculateFinalInterest()  
{  
Console.WriteLine("Savings Account Calculating Final Interest");  
}  
  
protectedoverridevoid DeleteAccountFromDB()  
{  
base.DeleteAccountFromDB();  
Console.WriteLine("Savings Account Deleting Account from DB");  
}  
}

Notice how *SavingsAccount* derives from *BankAccountProtected*. *SavingsAccount* can access any of the *protected* members of the *BankAccountProtected* class which is its base class. It demonstrates this fact via the call to *base.DeleteAccountFromDB* in it's *DeleteAccountFromDB* method. If the inheritance part of Listing 19-4 is a little confusing, you can visit [Lesson 8: Class Inheritance](http://csharp-station.com/Tutorial/CSharp/Lesson08) for a refresher and better understanding. Each method of *SavingsAccount* has the protected access modifier also, which simply means that classes derived from *SavingsAccount* can access those *SavingsAccount* members with the *protected* access modifier. The same situation exists with the *CheckingAccount* class, shown in Listing 19-5.

**Listing 19-5. Derived CheckingAccount Class Using protected Members of its Base Class: CheckingAccount.cs**

using System;  
  
class CheckingAccount : BankAccountProtected  
{  
protectedoverridevoid ApplyPenalties()  
{  
Console.WriteLine("Checking Account Applying Penalties");  
}  
  
protectedoverridevoid CalculateFinalInterest()  
{  
Console.WriteLine("Checking Account Calculating Final Interest");  
}  
  
protectedoverridevoid DeleteAccountFromDB()  
{  
base.DeleteAccountFromDB();  
Console.WriteLine("Checking Account Deleting Account from DB");  
}  
}

The *CheckingAccount* class in Listing 19-5 is implemented similar to *SavingsAccount* from Listing 19-4. If you were writing this, the difference would be that the methods of each class would have unique implementations. For example, the business rules associated with the final interest calculation would differ, depending on whether the account type was checking or savings.

Notice the call to the base class method in the *DeleteAccountFromDB* method in *CheckingAccount*. Just like *SavingsAccount*, *CheckingAccount* has access to *BankAccountProtected*'s protected method because it is a derived class. This is a common pattern in polymorphism because derived classes often have a responsibility to call virtual base class methods to ensure critical functionality has the opportunity to execute. You would consult the method documentation to see if this was necessary. Without a *protected* access modifier, your only option would have been to make the base class method *public*; which, as explained earlier, is dangerous.

To use the code from Listings 19-3, 19-4, and 19-5, you can implement the following code:

BankAccountProtected[] bankAccts = newBankAccountProtected[2];  
bankAccts[0] = newSavingsAccount();  
bankAccts[1] = newCheckingAccount();  
  
foreach (BankAccountProtected acct in bankAccts)  
{  
// call public method, which invokes protected virtual methods  
acct.CloseAccount();  
}

Since both *SavingsAccount* and *CheckingAccount* derive from *BankAccountProtected*, you can assign them to the *bankAccts* array. They both override the *protectedvirtual* methods of *BankAccountProtected*, so it is the *SavingsAccount* and *CheckingAccount* methods that are called when *CloseAccount* in *BankAccountProtected* executes. Remember that the only reason the methods of *SavingsAccount* and *CheckingAccount* can call their *virtual* base class methods, as in the case of *DeleteAccountFromDB*, is because the *virtual* base class methods are marked with the *protected* access modifier.

**A Quick Word on *internal* and *protected internal* Access Modifiers**

In practice, most of the code you write will involve the *public*, *private*, and *protected* access modifiers. However, there are two more access modifiers that you can use in more sophisticated scenarios: *internal* and *protected internal*.

You would use *internal* whenever you created a separate class library and you don't want any code outside of the library to access the code with *internal* access. The *protected internal* is a combination of the two access modifiers it is named after, which means either *protected* **or** *internal*.

**Access Modifiers for Types**

So far, the discussion of access modifiers has only applied to the members of types. However, the rules are different for the types themselves. When talking about types, I'm referring to all of the C# types, including classes, structs, interfaces, delegates, and enums. Nested types, such as a class defined within the scope of a class, are considered type members and fall under the same access rules as other type members.

Types can have only two access modifiers: *public* or *internal*. The default, if you don't specify the access modifier, is *internal*. Looking at all of the classes used in this lesson, you can see that they are *internal* because they don't have an access modifier. You can explicitly specify *internal* like this:

internal class InternalInterestCalculator  
{  
// members go here  
}

Perhaps the *InternalInterestCalculator*, shown above, has special business rules that you don't want other code to use. Now, it is in a class library of its own and can only be accessed by other code inside of that same class library (DLL).

**Note:** To be more specific, *internal* means that only code in the same assembly can access code marked as *internal*. However, discussing the definition of an assembly is outside the scope of this lesson, so I am simplifying the terminology.

If you declared a class inside of a class library that you wanted other code to use, you would give it a *public* access modifier. The following code shows an example of applying the *public* access modifier to a type:

public class BankAccountExternal  
{  
// members go here  
}

Clearly, a bank account is something you would want to access from outside of a class library. Therefore, it only makes sense to give it a *public* access modifier as shown in the *BankAccountExternal* class above.

**Tip:** A common gottcha in Visual Studio occurs when you create a new class in a class library. The default template doesn't include an access modifier. Then, when you try to write code that uses the new class in your program (which references the class library), you get a compiler error saying that the class doesn't exist. Well, you know it exists because you just wrote it and are looking at the page. If you've already seen the clue I've given you so far, you'll key on the fact that the default template left out the access modifier on the type. This makes the class default to *internal*, which can't be seen outside of the assembly. The fix is to give the class a *public* modifier, like the *BankAccountExternal* class above.

**Summary**

Encapsulation is an object-oriented principle of hiding the internal state and behavior of an object, making your code more maintainable. In C#, you can manage encapsulation with access modifiers. For example, the *public* access modifier allows access to any code but the *private* access modifier restricts access to only members of a type. Other access modifiers restrict access in the range somewhere between *public* and *private*. While you can use any of the access modifiers on type members, the only two access modifiers you can use on types are the *public* and *internal*.

I invite you to return for [Lesson 20: Introduction to Generic Collections](http://csharp-station.com/Tutorial/CSharp/Lesson20).

# Lesson 20: Introduction to Generic Collections

All the way back in Lesson 02, you learned about arrays and how they allow you to add and retrieve a collection of objects. Arrays are good for many tasks, but C# v2.0 introduced a new feature called generics. Among many benefits, one huge benefit is that generics allow us to create collections that allow us to do more than allowed by an array. This lesson will introduce you to generic collections and how they can be used. Here are the objectives for this lesson:

* Understand how generic collections can benefit you
* Learn how to create and use a generic List
* Write code that implements a generic Dictionary

**What Can Generics Do For Me?**

Throughout this tutorial, you've learned about types, whether built-in (*int*, *float*, *char*) or custom (*Shape*, *Customer*, *Account*). In .NET v1.0 there were collections, such as the *ArrayList* for working with groups of objects. An *ArrayList* is much like an array, except it could automatically grow and offered many convenience methods that arrays don't have. The problem with *ArrayList* and all the other .NET v1.0 collections is that they operate on type *object*. Since all objects derive from the *object* type, you can assign anything to an *ArrayList*. The problem with this is that you incur performance overhead converting value type objects to and from the *object* type and a single *ArrayList* could accidentally hold different types, which would cause hard to find errors at runtime because you wrote code to work with one type. Generic collections fix these problems.

A generic collection is strongly typed (type safe), meaning that you can only put one type of object into it. This eliminates type mismatches at runtime. Another benefit of type safety is that performance is better with value type objects because they don't incur overhead of being converted to and from type *object*. With generic collections, you have the best of all worlds because they are strongly typed, like arrays, and you have the additional functionality, like *ArrayList* and other non-generic collections, without the problems.

The next section will show you how to use a generic *List* collection.

**Creating Generic *List<T>* Collections**

The pattern for using a generic *List* collection is similar to arrays. You declare the *List*, populate its members, then access the members. Here's a code example of how to use a *List*:

List<int> myInts = new List<int>();

myInts.Add(1);

myInts.Add(2);

myInts.Add(3);

for (int i = 0; i < myInts.Count; i++)

{

Console.WriteLine("MyInts: {0}", myInts[i]);

}

The first thing you should notice is the generic collection *List<int>*, which is referred to as List of int. If you looked in the documentation for this class, you would find that it is defined as *List<T>*, where *T* could be any type. For example, if you wanted the list to work on *string* or *Customer* objects, you could define them as *List<string>* or *List<Customer>* and they would hold only *string* or *Customer* objects. In the example above, *myInts* holds only type *int*.

Using the *Add* method, you can add as many *int* objects to the collection as you want. This is different from arrays, which have a fixed size. The *List<T>* class has many more methods you can use, such as *Contains*, *Remove*, and more.

There are two parts of the *for* loop that you need to know about. First, the condition uses the *Count* property of *myInts*. This is another difference between collections and arrays in that an array uses a *Length* property for the same thing. Next, the way to read from a specific position in the *List<T>* collection, *myInts[i]*, is the exact same syntax you use with arrays.

The next time you start to use a single-dimension array, consider using a *List<T>* instead. That said, be sure to let your solution fit the problem and use the best tool for the job. i.e. it's common to work with *byte[]* in many places in the .NET Framework.

**Working with *Dictionary<TKey, TValue>* Collections**

Another very useful generic collection is the *Dictionary*, which works with key/value pairs. There is a non-generic collection, called a *Hashtable* that does the same thing, except that it operates on type *object*. However, as explained earlier in this lesson, you want to avoid the non-generic collections and use thier generic counterparts instead. The scenario I'll use for this example is that you have a list of *Customers* that you need to work with. It would be natural to keep track of these *Customers* via their *CustomerID*. The *Dictionary* example will work with instances of the following *Customer* class:

public class Customer

{

public Customer(int id, string name)

{

ID = id;

Name = name;

}

private int m\_id;

public int ID

{

get { return m\_id; }

set { m\_id = value; }

}

private string m\_name;

public string Name

{

get { return m\_name; }

set { m\_name = value; }

}

}

The *Customer* class above has a constructor to make it easier to initialize. It also exposes its state via public properties. It isn't very sophisticated at this point, but that's okay because its only purpose is to help you learn how to use a *Dictionary* collection. The following example populates a *Dictionary* collection with *Customer* objects and then shows you how to extract entries from the *Dictionary*:

Dictionary<int, Customer> customers = new Dictionary<int, Customer>();

Customer cust1 = new Customer(1, "Cust 1");

Customer cust2 = new Customer(2, "Cust 2");

Customer cust3 = new Customer(3, "Cust 3");

customers.Add(cust1.ID, cust1);

customers.Add(cust2.ID, cust2);

customers.Add(cust3.ID, cust3);

foreach (KeyValuePair<int, Customer> custKeyVal in customers)

{

Console.WriteLine(

"Customer ID: {0}, Name: {1}",

custKeyVal.Key,

custKeyVal.Value.Name);

}

The *customers* variable is declared as a *Dictionary<int, Customer>*. Considering that the formal declaration of *Dictionary* is *Dictionary<TKey, TValue>*, the meaning of *customers* is that it is a *Dictionary* where the key is type *int* and the value is type *Customer*. Therefore, any time you add an entry to the *Dictionary*, you must provide the key because it is also the key that you will use to extract a specified *Customer* from the *Dictionary*.

I created three *Customer* objects, giving each an *ID* and a *Name*. I'll use the *ID* as the key and the entire *Customer* object as the value. You can see this in the calls to *Add*, where *custX.ID* is added as the key (first parameter) and the *custX* instance is added as the value (second parameter).

Extracting information from a *Dictionary* is a little bit different. Iterating through the *customers* *Dictionary* with a *foreach* loop, the type returned is *KeyValuePair<TKey, TValue>*, where *TKey* is type *int* and *TValue* is type *Customer* because those are the types that the *customersDictionary* is defined with.

Since *custKeyVal* is type *KeyValuePair<int, Customer>* it has *Key* and *Value* properties for you to read from. In our example, *custKeyVal.Key* will hold the *ID* for the *Customer* instance and *custKeyVal.Value* will hold the whole *Customer* instance. The parameters in the *Console.WriteLine* statement demonstrate this by printing out the *ID*, obtained through the *Key* property, and the *Name*, obtained through the *Name* property of the *Customer* instance that is returned by the *Value* property.

The *Dictionary* type is handy for those situations where you need to keep track of objects via some unique identifier. For your convenience, here's Listing 20-1, shows how both the *List* and *Dictionary* collections work.

**Listing 20-1. Introduction to Using Generic Collections with an Example of the List<T> and Dictionary<TKey, TValue> Generic Collections**

using System;

using System.Collections.Generic;

public class Customer

{

public Customer(int id, string name)

{

ID = id;

Name = name;

}

private int m\_id;

public int ID

{

get { return m\_id; }

set { m\_id = value; }

}

private string m\_name;

public string Name

{

get { return m\_name; }

set { m\_name = value; }

}

}

class Program

{

static void Main(string[] args)

{

List<int> myInts = new List<int>();

myInts.Add(1);

myInts.Add(2);

myInts.Add(3);

for (int i = 0; i < myInts.Count; i++)

{

Console.WriteLine("MyInts: {0}", myInts[i]);

}

Dictionary<int, Customer> customers = new Dictionary<int, Customer>();

Customer cust1 = new Customer(1, "Cust 1");

Customer cust2 = new Customer(2, "Cust 2");

Customer cust3 = new Customer(3, "Cust 3");

customers.Add(cust1.ID, cust1);

customers.Add(cust2.ID, cust2);

customers.Add(cust3.ID, cust3);

foreach (KeyValuePair<int, Customer> custKeyVal in customers)

{

Console.WriteLine(

"Customer ID: {0}, Name: {1}",

custKeyVal.Key,

custKeyVal.Value.Name);

}

Console.ReadKey();

}

}

Whenever coding with the generic collections, add a *using System.Collections.Generic* declaration to your file, just as in Listing 20-1.

**Summary**

Generic collections give you the best of all worlds with the strong typing of arrays and flexibility of non-generic collections. There are many more generic collections to choose from also, such as *Stack*, *Queue*, and *SortedDictionary*. Look in the *System.Collections.Generic* namespace for other generic collections.

I invite you to return for [Lesson 21: Anonymous Methods](http://csharp-station.com/Tutorial/CSharp/Lesson21).

# Lesson 21: Anonymous Methods

In [Lesson 14: Introduction to Delegates](http://csharp-station.com/Tutorial/CSharp/Lesson14), you learned about delegates and how they enable you to connect handlers to events. For C# v2.0, there is a new language feature, called anonymous methods, that are similar to delegates, but require less code. While you learn about anonymous methods, we'll cover the following objectives:

* Understand the benefits of anonymous methods
* Learn how to implement an anonymous method
* Implement anonymous methods that use delegate parameters

**How Do Anonymous Methods Benefit Me?**

An anonymous method is a method without a name - which is why it is called anonymous. You don't declare anonymous methods like regular methods. Instead they get hooked up directly to events. You'll see a code example shortly.

To see the benefit of anonymous methods, you need to look at how they improve your development experience over using delegates. Think about all of the moving pieces there are with using delegates: you declare the delegate, write a method with a signature defined by the delegate interface, declare the event based on that delegate, and then write code to hook the handler method up to the delegate. With all this work to do, no wonder programmers, who are new to C# delegates, have to do a double-take to understand how they work.

Because you can hook an anonymous method up to an event directly, a couple of the steps of working with delegates can be removed. The next section shows you how this works.

**Implementing an Anonymous Method**

An anonymous method uses the keyword, delegate, instead of a method name. This is followed by the body of the method. Typical usage of an anonymous method is to assign it to an event. Listing 21-1 shows how this works.

**Listing 21-1. Implementing an Anonymous Method**

using System.Windows.Forms;

public partial class Form1 : Form

{

public Form1()

{

Button btnHello = new Button();

btnHello.Text = "Hello";

btnHello.Click +=

delegate

{

MessageBox.Show("Hello");

};

Controls.Add(btnHello);

}

}

The code in Listing 21-1 is a Windows Forms application. It instantiates a *Button* control and sets its *Text* to "Hello". Notice the combine, *+=*, syntax being used to hook up the anonymous method. You can tell that it is an anonymous method because it uses the *delegate* keyword, followed by the method body in curly braces.

Essentially, you have defined a method inside of a method, but the body of the anonymous method doesn't execute with the rest of the code. Because you hook it up to the event, the anonymous method doesn't execute until the *Click* event is raised. When you run the program and click the Hello button, you'll see a message box that say's "Hello" - courtesy of the anonymous method.

Using *Controls.Add*, adds the new button control to the window. Otherwise the window wouldn't know anything about the *Button* and you wouldn't see the button when the program runs.

**Using Delegate Parameters with Anonymous Methods**

Many event handlers need to use the parameters of the delegate they are based on. The previous example didn't use those parameters, so it was more convenient to not declare them, which C# allows. Listing 21-2 shows you how to use parameters if you need to.

**Listing 21-2. Using Parameters with Anonymous Methods**

using System;

using System.Windows.Forms;

public partial class Form1 : Form

{

public Form1()

{

Button btnHello = new Button();

btnHello.Text = "Hello";

btnHello.Click +=

delegate

{

MessageBox.Show("Hello");

};

**Button btnGoodBye = new Button();**

**btnGoodBye.Text = "Goodbye"**;

**btnGoodBye.Left = btnHello.Width + 5;**

**btnGoodBye.Click +=**

**delegate**(**object sender, EventArgs e)**

**{**

**string message = (sender as Button).Text;**

**MessageBox.Show(message);**

**};**

**Controls.Add(btnHello);**

**Controls.Add(btnGoodBye);**

}

}

The bold parts of Listing 21-2 show another *Button* control added to the code from Listing 21-1. Besides changing the text, *btnGoodBye* is moved to the right of *btnHello* by setting it's *Left* property to 5 pixels beyond the right edge of *btnHello*. If we didn't do this, *btnGoodBye* would cover *btnHello* because both of their *Top* and *Left* properties would default to 0.

Beyond implementation details, the real code for you to pay attention to is the implementation of the anonymous method. Notice that the *delegate* keyword now has a parameter list. this parameter list must match the delegate type of the event that the anonymous method is being hooked up to. The delegate type of the *Click* event is *EventHandler*, which has the following signature:

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

Notice the *EventHandler* parameters. Now, here's how the *Button* control's *Click* event is defined:

public event EventHandler Click;

Notice that the delegate type of the *Click* event is *EventHandler*. This is why the anonymous method, assigned to *btnGoodBye.Click* in Listing 21-2, must have the same parameters as the *EventHandler* delegate.

**Summary**

Anonymous methods are a simplified way for you to assign handlers to events. They take less effort than delegates and are closer to the event they are associated with. You have the choice of either declaring the anonymous method with no parameters or you can declare the parameters if you need them.

I invite you to return for [Lesson 22: Topics on C# Type](http://csharp-station.com/Tutorial/CSharp/Lesson22).

# Lesson 22: Topics on C# Type

Throughout this tutorial, you've seen many different types, including those that are part of C# and custom designed types. If you've taken the samples and worked on them yourself, extending and writing your own programs, you are likely to have experienced errors associated with type. For example, you can't assign a *double* to an *int* without using a cast operator to perform the conversion. Another feature of C# concerns the semantic differences between reference and value types. Such problems should make you wonder why this is so and that's what this lesson is for. Here are the objectives for this lesson:

* Understand the need for type safety
* See how to convert one type to another
* Learn about reference types
* Learn about value types
* Comprehend the semantic differences between reference and value types

**Why Type Safety?**

In untyped languages, such as scripting languages, you can assign one variable to another and the compiler/interpreter will use an intelligent algorithm to figure out how the assignment should be done. If the assignment is between two variables of the same type, all is good. However, if the assignment is between different types, you could have serious problems.

For example, if you assigned an *int* value to a *float* variable it would convert okay because the fractional part of the new *float* would just be zero. However, if you went the other way and assigned a *float* value to an *int* variable, that would most likely be a problem. You would lose all of the precision of the original *float* value. Consider the damage that could be caused if the *float* value represented a chemical ingredient, an engineering measurement, or a financial value. Finding such an error would be difficult and particularly expensive, especially if the error didn't show up until your application was in production (already being used by customers).

**Using the Cast Operator for Conversions**

In [Lesson 02: Operators, Types, and Variables](http://csharp-station.com/Tutorial/CSharp/Lesson02), you learned about C# types and operators. It explained the size and precision of the various types and there is a list of available operators. The cast operator, *(x)*, is listed first as a primary operator in Table 2-4. When you must convert a type that doesn't fit, it must be done via what is called an explicit conversion, which uses the cast operator. Listing 22-1 has an example of an implicit conversion, which doesn't require the cast operator, and an explicit conversion.

**Listing 22-1. Cast Operators**

using System;

class Program

{

static void Main()

{

float lengthFloat = 7.35f;

// lose precision - explicit conversion

int lengthInt = (int)lengthFloat;

// no problem - implicit conversion

double lengthDouble = lengthInt;

Console.WriteLine("lengthInt = " + lengthInt);

Console.WriteLine("lengthDouble = " + lengthDouble);

Console.ReadKey();

}

}

Here's the output:

lengthInt = 7

lengthDouble = 7

Since a *float*, *lengthFloat*, has a fractional part but an *int*, *lengthInt*, doesn't; the types aren't compatible. Because of type safety, C# won't allow you to assign *lengthFloat* directly to *lengthInt*, which would be dangerous. For your protection, you must use a cast operator, *(int)*, to force the explicit conversion of *lengthFloat* to *lengthInt*. In the output, you can see that *lengthInt* is *7*, showing that it lost the fractional part of the *7.35f* value from *lengthFloat*.

The assignment from *lengthInt* to *lengthDouble* is safe because a double is 64-bit and an int is 32-bit, meaning that you won't lose information. Therefore, the conversion is implicit, meaning that you can perform the assignment without the cast operator.

**Understanding Reference Types**

Reference type variables are named appropriately (reference) because the variable holds a reference to an object. In C and C++, you have something similar that is called a pointer, which points to an object. While you can modify a pointer, you can't modify the value of a reference - it simply points at the object in memory.

An important fact you need to understand is that when you are assigning one reference type variable to another, only the reference is copied, not the object. The variable holds the reference and that is what is being copied. Listing 22-2 shows how this works.

**Listing 22-2. Reference Type Assignment**

using System;

class Employee

{

private string m\_name;

public string Name

{

get { return m\_name; }

set { m\_name = value; }

}

}

class Program

{

static void Main()

{

Employee joe = new Employee();

joe.Name = "Joe";

Employee bob = new Employee();

bob.Name = "Bob";

Console.WriteLine("Original Employee Values:");

Console.WriteLine("joe = " + joe.Name);

Console.WriteLine("bob = " + bob.Name);

// assign joe reference to bob variable

bob = joe;

Console.WriteLine();

Console.WriteLine("Values After Reference Assignment:");

Console.WriteLine("joe = " + joe.Name);

Console.WriteLine("bob = " + bob.Name);

joe.Name = "Bobbi Jo";

Console.WriteLine();

Console.WriteLine("Values After Changing One Instance:");

Console.WriteLine("joe = " + joe.Name);

Console.WriteLine("bob = " + bob.Name);

Console.ReadKey();

}

}

Here's the output:

Original Employee Values:

joe = Joe

bob = Bob

Values After Reference Assignment:

joe = Joe

bob = Joe

Values After Changing One Instance:

joe = Bobbi Jo

bob = Bobbi Jo

In Listing 22-2, I created two *Employee* instances, *joe* and *bob*. You can see in the output that the *Name* properties of both *Employee* instances each show their assigned values from when the objects were first created. After assigning *joe* to *bob*, the value of the *Name* properties of both instances are the same. This is what you might expect to see.

What might surprise you is the values that occur after assigning a value to the *Employee* instance variable named *joe*. If you look at the code closely, you'll notice that it doesn't change *bob* - only *Joe*. However, the results from the output show that the *Name* property in *bob* is the same as the *Name* property in *joe*. This demonstrates that after assigning *joe* to *bob*, both variables held references to the *joe* object. Only the reference was copied - not the object. This is why you see the results of printing *Name* in both *joe* and *bob* are the same because the change was on the object that they both refer to.

The following types are reference types:

* arrays
* class'
* delegates
* interfaces

**Note:** The primitive type, *string*, is also a reference type.

**Understanding Value Types**

Value type variables, as their name (value) suggests, hold the object value. A value type variable holds its own copy of an object and when you perform assignment from one value type variable to another, both the left-hand-side and right-hand-side of the assignment hold two separate copies of that value. Listing 22-3 shows how value type assignment works.

**Listing 22-3. Value Type Assignment**

using System;

struct Height

{

private int m\_inches;

public int Inches

{

get { return m\_inches; }

set { m\_inches = value; }

}

}

class Program

{

static void Main()

{

Height joe = new Height();

joe.Inches = 71;

Height bob = new Height();

bob.Inches = 59;

Console.WriteLine("Original Height Values:");

Console.WriteLine("joe = " + joe.Inches);

Console.WriteLine("bob = " + bob.Inches);

// assign joe value to bob variable

bob = joe;

Console.WriteLine();

Console.WriteLine("Values After Value Assignment:");

Console.WriteLine("joe = " + joe.Inches);

Console.WriteLine("bob = " + bob.Inches);

joe.Inches = 65;

Console.WriteLine();

Console.WriteLine("Values After Changing One Instance:");

Console.WriteLine("joe = " + joe.Inches);

Console.WriteLine("bob = " + bob.Inches);

Console.ReadKey();

}

}

Here's the output:

Original Height Values:

joe = 71

bob = 59

Values After Value Assignment:

joe = 71

bob = 71

Values After Changing One Instance:

joe = 65

bob = 71

In Listing 22-3, you can see that the *Inches* property of *bob* and *joe* are initially set to different values. After assigning *joe* to *bob*, a value copy occurs, where both of the variables have the same value, but are two separate copies. To demonstrate value assignment results, notice what happens after setting *joe* to *65*; The output shows that *bob* did not change, which demonstrates that value types hold distinct copies of their objects.

The following types are value types:

* enum
* struct

**Note:** All of the primitive types (int, char, double, etc.), except for string, are value types.

**Reference Type and Value Type Differences**

From the previous paragraphs, you might already see that there is a difference reference type and value type assignment. Reference types copy a reference to an object and value types copy the object. If you don't know this, then the effects can be surprising in your code when performing tasks such as making assignments and passing arguments to methods.

**Summary**

This lesson provided a few tips on working with types in C#. You should now have a better understanding of type safety and how it can help you avoid problems. This lesson showed you how to use a cast operator to perform conversions and explained the difference between explicit and implicit conversions. You also know that the type system is divided between reference types and value types. To demonstrate the differences between reference types and value types, this lesson provided examples that showed how both reference types and value types behave during assignment.

I invite you to return for [Lesson 23: Working with Nullable Types](http://csharp-station.com/Tutorial/CSharp/Lesson23).

# Lesson 23: Working with Nullable Types

Working with value types and data can sometimes be challenging because a value type doesn't normally hold a null value. This lesson shows you how to overcome this limitation with C# nullable types. Here's what you'll learn.

* Understand the problem that nullable types solve
* See how to declare a nullable type
* Learn how to use nullable types

**Understanding the Problem with Value Types and Null Values**

As explained in [Lesson 12: Structs](http://csharp-station.com/Tutorial/CSharp/Lesson12), the default value of a struct (value type) is some form of 0. This is another difference between reference types and value types, in addition to what was described in [Lesson 22: Topics on C# Type](http://csharp-station.com/Tutorial/CSharp/Lesson22). The default value of a reference type is *null*. If you're just writing C# code and managing your own data source, such as a file that holds data for your application, the default values for structs works fine.

In reality, most applications work with databases, which have their own type systems. The implications of working with database type systems is that you don't have a one-to-one mapping between C# and database types. One glaring difference is that database types can be set to *null*. A database has no knowledge of reference and value types, which are C# language (.NET Platform) concepts. This means that C# value type equivalents in the database, such as *int*, *decimal*, and *DateTime*, can be set to *null*.

Since a type in the database can be *null*, but your C# value type can't be *null*, you have to find some way to provide a translation in your C# code to account for *null* values. Effectively, the scheme you use will often be inconsistent from one program to another; something you often don't have a choice about. For example, what if you wanted to handle a *null* *DateTime* from SQL Server as the minimum *DateTime* value in C#. After that project, your next task would be to read data from a legacy Foxpro database, whose minimum *DateTime* value is different from SQL Server. Because of this lack of constency and potential confusion, C# 2.0 added nullable types, which are more elegant and natural for working with *null* data.

**Declaring Nullable Types**

To declare a value type as nullable, append a question mark, *?*, to the type name. Here's how to declare a *DateTime* variable as a nullable type:

DateTime? startDate;

A *DateTime* can't normally hold a *null* value, but the declaration above enables *startDate* to hold *null*, as well as any legal *DateTime* value. The proper terminology is to refer to the type of *startDate* as a nullable *DateTime*.

You can assign a normal value to *startDate* like this:

startDate = DateTime.Now;

or you can assign *null*, like this:

startDate = null;

Here's another example that declares and initializes a nullable *int*:

int? unitsInStock = 5;

The *unitsInStock* in the example above can be assigned a value of *null* also.

**Working with Nullable Types**

When you have nullable types, you'll want to check them to see if they're *null*. Here's an example that shows how you can check for a *null* value:

bool isNull = startDate == null;

Console.WriteLine("isNull: " + isNull);

The example above shows that you only need to use the equals operator to check for *null*. You could also make the equality check as part of an *if* statement, like this:

int availableUnits;

if (unitsInStock == null)

{

availableUnits = 0;

}

else

{

availableUnits = (int)unitsInStock;

}

**Note:** Notice the cast operator in the else clause above. An explicit conversion is required when assigning from nullable to non-nullable types.

That's several lines of code for something that appears to be such a common operation. Fortunately, there's a better way to perform the same task, using the coalesce operator, *??*, shown below:

int availableUnits = unitsInStock ?? 0;

The coalesce operator works like this: if the first value (left hand side) is *null*, then C# evaluates the second expression (right hand side).

**Summary**

This lesson explained how nullable types can be useful in your C# applications - especially when working with values from a database. You learned how to declare a nullable type and how to assign values, both *null* and non-*null* to nullable types. Another skill you learned was how to use nullable types by checking to see if their values are *null*. As you saw, the coalesce operator can be useful to help work with nullable type variables when you need to assign a valid value to a non-nullable type.

# Dictionary Objects:

The Dictionary provides fast lookups (with keys) to get values. With it, we use keys and values of any type, including int and string. Dictionary uses a special syntax form.

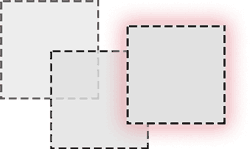
Types:

1. Dictionary(string, int)
2. Dictionary(int, string)

The Dictionary is composed of separate keys and values.

Example:

## ContainsKey



Next, you can check to see if a given string is present in a Dictionary with string keys.

It returns true if the key was found.

Tip:There is a more efficient method called TryGetValue. You should definitely use it when possible.

And:As its name implies, it tests for the key and then returns the value if it finds the key.

[TryGetValue](http://www.dotnetperls.com/trygetvalue)

**Program that uses ContainsKey: C#**

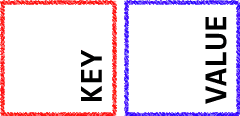
|  |
| --- |
| using System;  using System.Collections.Generic;  class Program  {  static void Main()  {  Dictionary<string, int> dictionary = new Dictionary<string, int>();  dictionary.Add("apple", 1);  dictionary.Add("windows", 5);  // See whether Dictionary contains this string.  if (dictionary.ContainsKey("apple"))  {  int value = dictionary["apple"];  Console.WriteLine(value);  }  // See whether Dictionary contains this string.  if (!dictionary.ContainsKey("acorn"))  {  Console.WriteLine(false);  }  }  } |

Output

1

False

## KeyValuePair



When Dictionary, or any object that implements IDictionary, is used in a foreach-loop, it returns an enumeration. In the case of Dictionary, this enumeration is in the form of KeyValuePair values.

[KeyValuePair](http://www.dotnetperls.com/keyvaluepair)

[Foreach](http://www.dotnetperls.com/foreach)

**Program that uses foreach on Dictionary: C#**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Example Dictionary again

Dictionary<string, int> d = new Dictionary<string, int>()

{

{"cat", 2},

{"dog", 1},

{"llama", 0},

{"iguana", -1}

};

// Loop over pairs with foreach

foreach (KeyValuePair<string, int> pair in d)

{

Console.WriteLine("{0}, {1}",

pair.Key,

pair.Value);

}

// Use var keyword to enumerate dictionary

foreach (var pair in d)

{

Console.WriteLine("{0}, {1}",

pair.Key,

pair.Value);

}

}

}

**Output**

cat, 2

dog, 1

llama, 0

iguana, -1

cat, 2

dog, 1

llama, 0

iguana, -1

The code creates a Dictionary with string keys and int values. The Dictionary stores animal counts. The program has a ShowDictionaryPair method. This method shows the foreach-loop and the KeyValuePair declaration.

**Tip:**In the foreach-loop, each KeyValuePair has two members, a string Key and an int Value.

Var

The final loop in the code shows how to make the syntax for looping simpler by using the var keyword. This is not desirable in some projects. But for KeyValuePair, it reduces source code size. It makes code easier to read.

[Var](http://www.dotnetperls.com/var)

[Var Dictionary](http://www.dotnetperls.com/var-dictionary)

### Keys

Here we use the Keys property. We then look through each key and lookup the values. This method is slower but has the same results. Using the Keys collection and putting it in an array or List is sometimes effective.

Tip:The Keys property returns a collection of type KeyCollection, not an actual List. We can convert it into a List.

**Program that gets Keys: C#**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

Dictionary<string, int> d = new Dictionary<string, int>()

{

{"cat", 2},

{"dog", 1},

{"llama", 0},

{"iguana", -1}

};

// Store keys in a List

List<string> list = new List<string>(d.Keys);

// Loop through list

foreach (string k in list)

{

Console.WriteLine("{0}, {1}",

k,

d[k]);

}

}

}

**Output**

cat, 2

dog, 1

llama, 0

iguana, -1