

# **C Sharp Programming**

# C Sharp Programming

[http://en.wikibooks.org/wiki/C\\_Sharp\\_Programming](http://en.wikibooks.org/wiki/C_Sharp_Programming)

This Book Is Generated By [WikiType](#)

using

[RenderX DiType](#), XML to PDF XSL-FO Formatter

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "[GNU Free Documentation License](#)".

## Table of Contents

1. C Sharp Programming.....	12
Introduction.....	12
Language Basics.....	13
Classes.....	13
The .NET Framework.....	14
Advanced Object-Orientation Concepts.....	14
Keywords.....	15
External links.....	16
2. Intro.....	17
Introduction.....	17
3. Basics.....	18
Basics.....	18
4. Structure.....	19
Structure.....	19
5. The .NET Framework.....	20
The .NET Framework.....	20
Console Programming.....	21
Console Programming.....	21
Output.....	21
Input.....	22
Error.....	22
Command line arguments.....	23
Windows Forms.....	25
6. Advanced.....	26
Advanced.....	26
7. Index.....	27
8. Foreword.....	28

## C Sharp Programming

---

Introduction.....	28
Standard.....	29
History.....	29
9. Introduction.....	30
10. Naming.....	33
Reasoning.....	33
Conventions.....	33
Namespace.....	33
Assemblies.....	34
Classes and Structures.....	34
Exception Classes.....	34
Interfaces.....	34
Functions.....	34
Properties and Public Member Variables.....	34
Parameters and Procedure-level Variables.....	35
Class-level Private and Protected Variables.....	35
Controls on Forms.....	35
Constants.....	35
Example.....	35
11. Syntax.....	37
Statements.....	37
Statement blocks.....	38
Comments.....	39
Case sensitivity.....	40
12. Variables.....	41
Fields, Local Variables, and Parameters.....	41
Fields.....	42
Local variables.....	42
Parameter.....	42

- Types.....43
  - Integral types.....43
  - Custom types.....47
  - Conversion.....48
  - Scope and extent.....48
- 13. Operators.....49
  - Arithmetic.....49
  - Logical.....50
  - Bitwise shifting.....52
  - Relational.....52
  - Assignment.....53
  - Short-hand Assignment.....54
  - Type information.....55
  - Pointer manipulation.....55
  - Overflow exception control.....56
  - Others.....56
- 14. Data structures.....57
  - Enumerations.....57
  - Structs.....58
  - Arrays.....60
- 15. Control.....62
  - Conditional statements.....62
    - The if statement.....62
    - The switch statement.....63
  - Iteration statements.....65
    - The do...while loop.....65
    - The for loop.....66
    - The foreach loop.....67
    - The while loop.....67

## C Sharp Programming

---

Jump statements.....	68
16. Exceptions.....	69
References.....	71
17. Namespaces.....	72
Nested namespaces.....	73
18. Classes.....	75
Methods.....	76
Constructors.....	76
Finalizers.....	78
Properties.....	79
Indexers.....	80
Events.....	80
Operator.....	81
Structures.....	81
Static classes.....	82
19. Objects.....	83
Introduction.....	83
Reference and Value Types.....	84
System.Object.....	85
Object basics.....	85
Constructors.....	85
Destructors.....	88
Abstract Class.....	89
Sub-heading.....	89
20. Encapsulation.....	90
Protection Levels.....	91
Private.....	91
Protected.....	91
Public.....	91

Internal.....	91
21. NET Framework overview.....	92
Introduction.....	92
Background.....	93
22. Inheritance.....	94
Inheritance.....	94
Subtyping Inheritance.....	94
Inheritance keywords.....	96
23. Interfaces.....	97
Additional Details.....	99
24. Delegates and Events.....	100
Introduction.....	100
Delegates.....	101
Events.....	103
25. Abstract classes.....	105
26. Partial classes.....	108
Partial Classes.....	108
27. Collections.....	110
Lists.....	110
LinkedLists.....	110
Queues.....	110
Stacks.....	111
Dictionaries.....	111
28. Generics.....	112
Generic Interfaces.....	112
Generic Classes.....	112
Generic lists.....	115
Generic linked lists.....	117
Generic queues.....	117

## C Sharp Programming

---

Generic stacks.....	117
Generic dictionaries.....	117
Generic Methods.....	117
Generic Delegates.....	117
Generic Events.....	117
29. Object Lifetime.....	118
Introduction.....	118
Garbage Collector.....	118
Managed Resources.....	119
Unmanaged Resources.....	119
Applications.....	120
Resource Acquisition Is Initialisation.....	122
30. Design Patterns.....	123
Table Of Contents (TOC).....	123
Factory Pattern.....	123
Singleton.....	126
31. abstract.....	128
32. as.....	129
See also.....	129
33. base.....	130
34. bool.....	132
35. break.....	133
36. byte.....	134
37. case.....	135
38. catch.....	136
39. char.....	137
40. class.....	138
41. const.....	139
42. continue.....	140



43. decimal.....	141
44. default.....	142
45. delegate.....	143
46. do.....	144
47. double.....	145
48. else.....	146
49. enum.....	148
50. event.....	149
51. explicit.....	150
General.....	150
Keyword.....	151
52. extern.....	152
53. false.....	154
54. finally.....	155
55. fixed.....	156
56. float.....	157
57. for.....	158
58. foreach.....	159
59. goto.....	160
60. if.....	161
61. implicit.....	163
General.....	163
Keyword.....	163
62. in.....	165
63. int.....	166
64. interface.....	167
65. internal.....	168
66. is.....	169
67. long.....	170

## C Sharp Programming

---

68. namespace.....	171
69. new.....	172
70. null.....	173
71. object.....	174
72. out.....	175
73. override.....	176
74. params.....	177
75. private.....	180
76. protected.....	181
77. public.....	182
78. readonly.....	183
79. ref.....	184
80. return.....	186
81. sbyte.....	187
82. sealed.....	188
83. short.....	189
84. sizeof.....	190
85. stackalloc.....	191
86. static.....	192
87. string.....	193
88. struct.....	194
89. switch.....	195
90. this.....	196
91. throw.....	197
92. true.....	198
93. try.....	199
94. typeof.....	200
95. uint.....	202
96. ulong.....	203

97. unchecked.....	204
98. unsafe.....	205
99. ushort.....	206
100. using.....	207
The directive.....	207
The statement.....	208
101. virtual.....	209
102. void.....	210
103. volatile.....	211
104. while.....	212
105. alias.....	213
106. get.....	214
107. partial.....	215
108. set.....	216
109. value.....	217
110. yield.....	218
GNU Free Documentation License.....	219

# C Sharp Programming

| [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C #



C sharp musical note

**C#** (pronounced "See Sharp") is a multi-purpose computer [programming language](#) suitable for all development needs. This WikiBook introduces C# language fundamentals and covers a variety of the base class libraries (BCL) provided by the Microsoft .NET Framework.

## Introduction

*Main introduction: [C Sharp Programming/Foreword](#)*

Although C# is derived from the [C programming language](#), it has features such as *garbage collection* that allow beginners to become proficient in C# more quickly than in C or C++. Similar to [Java](#), it is [object-oriented](#), comes with an extensive *class library*, and supports exception handling, multiple types of [polymorphism](#), and separation of interfaces from implementations. Those features, combined with its powerful development tools, multi-platform support, and *generics*, make C# a good choice for many types of software development projects: [rapid application development](#) projects, projects implemented by individuals or large or small teams, Internet applications, and projects with strict reliability requirements. Testing frameworks such as [NUnit](#) make C# amenable to [test-driven development](#) and thus a good language for use with [Extreme Programming](#) (XP). Its [strong typing](#) helps to prevent many programming errors that are common in weakly typed languages.

### Foreword

A description of the C# language and introduction to this Wikibook.

### **Getting started with C#**

A simple C# program and where to get tools to compile it.

## Language Basics

### **Naming conventions**

Quickly describes the generally accepted naming conventions for C#.

### **Basic syntax**

Describes the basics in how the applications you write will be interpreted.

### **Variables**

The entities used to store data of various shapes.

### **Operators**

Summarizes the operators, such as the '+' in addition, available in C#.

### **Data structures**

Enumerations, structs, and more.

### **Control statements**

Loops, conditions, and more. How the program flow is controlled.

### **Exceptions**

Responding to errors that can occur.

## Classes

### **Namespaces**

Giving your code its own space to live in.

### **Classes**

The blueprints of objects that describes how they should work.

### **Objects**

Cornerstones of any object-oriented programming language, objects are the tools you use to perform work.

### **Encapsulation and accessor levels**

Explains protection of object states by *encapsulation*.

## The .NET Framework

### **.NET Framework Overview**

An overview of the .NET class library used in C#.

### **Console Programming**

Input and Output using the console.

### **Windows Forms**

GUI Programming with Windows Forms.

## Advanced Object-Orientation Concepts

### **Inheritance**

Re-using existing code to improve or specialise the functionality of an object.

### **Interfaces**

Define a template, in which to base sub-classes from.

### **Delegates and Events**

Be informed about when an event happens and choose what method to call when it happens with delegates.

### **Abstract classes**

Build partially implemented classes.

### **Partial classes**

Split a class over several files to allow multiple users to develop, but also to stop code generators interfering with source code.

### **Collections**

Effectively manage (add, remove, find, iterate, etc.) large sets of data.

### **Generics**

Allow commonly used collections and classes to appear to have specialisation for your custom class.

### **Object Lifetime**

Learn about the lifetime of objects, where they are allocated and learn about garbage collection.

### **Design Patterns**

Learn commonly used design methodologies to simplify and/or improve your development framework.

## Keywords

### External links

- [Learning Visual C# in 5 minutes](#) A simple tutorial that teaches you a few basics.
- [A C# Tutorial](#) starts from basics and gives source code.
- [An Introduction to Mono Development](#) by Andrew Troelsen
- [Sharp Develop IDE](#) : A free IDE for C#, VB.NET and Boo projects on Microsoft's .NET platform.
- [Microsoft Visual C# Express Edition](#) : A free development environment created by Microsoft for writing C# Applications.
- [Mono Project](#) : A C# Development Environment for Linux, Windows, and other platforms.
- [Mono IDE](#) : An GNOME based IDE for Mono on Linux platforms.
- [C# Online.NET](#) - free, wiki-based C# and .NET encyclopedia and forums
- [C# Language Specification](#) download page at ECMA
- [C# Environment setup](#) Visual C# environment setup details from MSDN
- [C# FAQ](#) C# FAQ, Blogs and Forums.
- [Premium C# Tutorial](#) - A collection of complete programming tutorials
- [DotGNU Portable.NET](#) - A CLI/.NET built in accordance with the requirements of the GNU Project capable of running C# programs on many platforms and architectures.
- [.NET Book Zero](#) by [Charles Petzold](#) - free downloadable book on C# and .NET framework by one of the world's foremost authorities on Windows programming, Charles Petzold.



# Intro

## **C# Programming**

[Cover](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

C #



C sharp musical note

## Introduction

### **Foreword**

A description of the C# and introduction to this Wikibook.

### **Getting started with C#**

A simple C# program and where to get tools to compile it.

# Basics

## **C# Programming**

[Cover](#) | [Introduction](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

C#



C sharp musical note

## Basics

### **Basic syntax**

Describes the basics in how the applications you write will be interpreted.

### **Variables**

The entities used to store data of various shapes.

### **Operators**

Summarizes the operators, such as the '+' in addition, available in C#.

### **Data structures**

Enumerations, structs, and more.

### **Control statements**

Loops, conditions, and more. How the program flow is controlled.

# Structure

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C #



C sharp musical note

## Structure

### Namespaces

Giving your code its own space to live in.

### Classes

The blueprints of objects that describes how they should work.

### Objects

Cornerstones of any object-oriented programming language, objects are the tools you use to perform work.

### Encapsulation and accessor levels

Explains protection of object states by *encapsulation*.

# The .NET Framework

## **C# Programming**

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [Advanced Topics](#) | [Index](#)

C #



C sharp musical note

## The .NET Framework

### **.NET Framework Overview**

An overview of the .NET class library used in C#.

### **Console Programming**

Input and Output using the console.

### **Windows Forms**

GUI Programming with Windows Forms.

# Console Programming

## Console Programming

### Output

The example program below shows a couple of ways to output text:

```
using System;
public class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hello World!");           // relies on "using
System;"
        Console.Write("This is");
        Console.Write("... my first program!\n");
        System.Console.WriteLine("Goodbye World!"); // no "using" statement
required
    }
}
```

The above code displays the following text:

```
Hello World!
This is... my first program!
Goodbye World!
```

That text is output using the `System.Console` class. The `using` statement at the top allows the compiler to find the `Console` class without specifying the `System` namespace each time it is used.

## C Sharp Programming

---

The middle lines use the Write() method, which does not automatically create a new line. To specify a new line, we can use the sequence backslash-n ( \n ). If for whatever reason we wanted to really show the \n character instead, we add a second backslash ( \\n ). The backslash is known as the escape character in C# because it is not treated as a normal character, but allows us to encode certain special characters (like a new line character).

## Input

Input can be gathered in a similar method to outputting data using the Read() and ReadLine methods of that same System.Console class:

```
using System;
public class ExampleClass
{
    public static void Main()
    {
        Console.WriteLine("Greetings! What is your name?");
        Console.Write("My name is: ");
        string name = Console.ReadLine();
        Console.WriteLine("Nice to meet you, " + name);
        Console.Read();
    }
}
```

The above program requests the user's name and displays it back. The final Console.Read() waits for the user to enter a key before exiting the program.

## Error

The Error output is used to divert error specific messages to the console. To a novice user this may seem fairly pointless, as this achieves the same as **Output** (as above). If you decide to write an application that runs another application (for example a scheduler), you may wish to monitor the output of that program - more specifically, you may only wish to be notified only of the errors that occur. If you coded your program to write to the Console.Error stream whenever an error occurred, you can tell your scheduler program to monitor this stream, and feedback any information that is sent to it. Instead of the Console appearing with the Error messages, your program may wish to log these to a file.

You may wish to revisit this after studying Streams and after learning about the Process class.

## Command line arguments

Command line arguments are values that are passed to a console program before execution. For example, the Windows command prompt includes a `copy` command that takes two command line arguments. The first argument is the original file and the second is the location or name for the new copy. Custom console applications can have arguments as well.

```
using System;
public class ExampleClass
{
    public static void Main(string[] args)
    {
        Console.WriteLine("First Name: " + args[0]);
        Console.WriteLine("Last Name: " + args[1]);
        Console.Read();
    }
}
```

If the program above code is compiled to a program called *username.exe*, it can be executed from the command line using two arguments, e.g. "Bill" and "Gates":

```
C:\>username.exe Bill Gates
```

Notice how the `Main()` method above has a string array parameter. The program assumes that there will be two arguments. That assumption makes the program unsafe. If it is run without the expected number of command line arguments, it will crash when it attempts to access the missing argument. To make the program more robust, we make we can check to see if the user entered all the required arguments.

```
using System;
public class Test
{
    public static void Main(string[] args)
    {
        if(args.Length >= 1)
```

## C Sharp Programming

---

```
        Console.WriteLine(args[0]);
    if(args.Length >= 2)
        Console.WriteLine(args[1]);
    }
}
```

Try running the program with only entering your first name or no name at all. The `string.Length` property returns the total number of arguments. If no arguments are given, it will return zero.

You are also able to group a single argument together by using the `""` quote marks. This is particularly useful if you are expecting many parameters, but there is a requirement for including spaces (e.g. file locations, file names, full names etc)

```
using System;

class Test
{
    public static void Main(string[] args)
    {
        for(int index =0 ;index < args.Length; index++)
        {
            Console.WriteLine((index+1) + ": " + args[index]);
        }
    }
}
```

```
C:\> Test.exe Separate words "grouped together"
1: Separate
2: words
3: grouped together
```



# Windows Forms

The `System.Windows.Forms` namespace allows us to create Windows applications easily. The `Form` class is a particularly important part of that namespace because the form is the key graphical building block of Windows applications. It provides the visual frame that holds buttons, menus, icons, and title bars together. Integrated development environments (IDEs) like Visual C# and SharpDevelop can help create graphical applications, but it is important to know how to do so manually:

```
using System.Windows.Forms;
public class ExampleForm : Form    // inherits from System.Windows.Forms.Form
{
    public static void Main()
    {
        ExampleForm wikibooksForm = new ExampleForm();
        wikibooksForm.Text = "I Love Wikibooks"; // specify title of the form
        wikibooksForm.Width = 400;              // width of the window in pixels
    els
        wikibooksForm.Height = 300;             // height in pixels
        Application.Run(wikibooksForm);        // display the form
    }
}
```

The example above creates a simple Window with the text "I Love Wikibooks" in the title bar. Custom form classes like the example above inherit from the `System.Windows.Forms.Form` class. Setting any of the properties `Text`, `Width`, and `Height` is optional. Your program will compile and run successfully if you comment these lines out, but they allow us to add extra control to our form.

# Advanced

## **C# Programming**

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | | [Index](#)

C #



C sharp musical note

## Advanced

**Inheritance**

**Interfaces**

**Abstract Classes**

**Partial Classes**

**Generics**

**Object Lifetime**

# Index

## **C# Programming**

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) |

C #



C sharp musical note

# Foreword

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

**C#** (pronounced "See Sharp") is a multi-purpose computer [programming language](#) suitable for all development needs.

## Introduction

Although C# is derived from the [C programming language](#), it has features such as [garbage collection](#) that allow beginners to become proficient in C# more quickly than in C or C++. Similar to [Java](#), it is [object-oriented](#), comes with an extensive *class library*, and supports exception handling, multiple types of [polymorphism](#), and separation of interfaces from implementations. Those features, combined with its powerful development tools, multi-platform support, and *generics*, make C# a good choice for many types of software development projects: [rapid application development](#) projects, projects implemented by individuals or large or small teams, Internet applications, and projects with strict reliability requirements. Testing frameworks such as [NUnit](#) make C# amenable to [test-driven development](#) and thus a good language for use with [Extreme Programming \(XP\)](#). Its [strong typing](#) helps to prevent many programming errors that are common in weakly typed languages.

A large part of the power of C# (as with other .NET languages), comes with the common .NET Framework API, which provides a large set of classes, including ones for encryption, TCP/IP socket programming, and graphics. Developers can thus write part of an application in C# and another part in another .NET language (e.g. VB .NET), keeping the tools, library, and object-oriented development model while only having to learn the new language syntax.

Because of the similarities between C# and the C family of languages, as well as [Java](#), a developer with a background in object-oriented languages like C++ may find C# structure and syntax intuitive.

## Standard

Microsoft, with [Anders Hejlsberg](#) as Chief Engineer, created C# as part of their .NET initiative and subsequently opened its [specification](#) via the [ECMA](#). Thus, the language is open to implementation by other parties. Other implementations include [Mono](#) and [DotGNU](#).

C# and other .NET languages rely on an implementation of the [virtual machine](#) specified in the [Common Language Infrastructure](#), like Microsoft's *Common Language Runtime* (CLR). That virtual machine manages memory, handles object references, and performs Just-In-Time (JIT) compiling of [Common Intermediate Language](#) code. The virtual machine makes C# programs safer than those that must manage their own memory and is one of the reasons .NET language code is referred to as *managed code*. More like Java than C and C++, C# discourages explicit use of pointers, which could otherwise allow software bugs to corrupt system memory and force the operating system to halt the program forcibly with nondescript error messages.

## History

Microsoft's original plan was to create a rival to Java, named J++ but this was abandoned to create C#, codenamed "Cool".

Microsoft submitted C# to the ECMA standards group mid-2000.

C# 2.0 was released in late-2005 as part of Microsoft's development suite, Visual Studio 2005. The 2.0 version of C# includes such new features as generics, partial classes, and iterators.

[1][2]

# Introduction

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

To compile your first C# application, you will need a copy of a .NET Framework SDK installed on your PC.

There are two .NET frameworks available: Microsoft's and Mono's.

### Microsoft

For Windows, the .NET Framework SDK can be downloaded from Microsoft's [.NET Framework Developer Center](#). If the default Windows directory (the directory where Windows or WinNT is installed) is C:\WINDOWS, the .Net Framework SDK installation places the Visual C# .NET Compiler (csc) in the C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705 directory for version 1.0, the C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322 directory for version 1.1, **or** the C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727 directory for version 2.0.

### Mono

For Windows, Linux, or other Operating Systems, an installer can be downloaded from the [Mono website](#).

For Linux, a good compiler is csc which can be downloaded for free from [the DotGNU Portable.Net project](#) page. The compiled programs can then be run with ilrun.

If you are working on Windows it is a good idea to add the path to the folders that contain cs.exe or mcs.exe to the Path environment variable so that you do not need to type the full path each time you want to compile.

For writing C#.NET code, there are plenty of editors that are available. It's entirely possible to write C#.NET programs with a simple text editor, but it should be noted that this requires you

to compile the code yourself. Microsoft offers a wide range of code editing programs under the Visual Studio line that offer syntax highlighting as well as compiling and debugging capabilities. Currently C#.NET can be compiled in Visual Studio 2002 and 2003 (only supports the .NET Framework version 1.0 and 1.1) and Visual Studio 2005 (supports the .NET Framework 2.0 and earlier versions with some tweaking). Microsoft offers , four of which cost money. The Visual Studio C# Express Edition can be downloaded and used for free from [Microsoft's website](#).

The code below will demonstrate a C# program written in a simple text editor. Start by saving the following code to a text file called hello.cs:

```
using System;

namespace MyConsoleApplication
{
    class MyFirstClass
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello, ");
            Console.WriteLine("World!");

            Console.ReadLine();
        }
    }
}
```

To compile hello.cs, run the following from the command line:

- For standard Microsoft installations of .NET 2.0, run `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc.exe hello.cs`
- For Mono run `mcs hello.cs`.
- For users of csc, compile with `"csc -o <name>.exe <name>.cs"`.

Doing so will produce hello.exe. The following command will run hello.exe:

- On Windows, use `hello.exe`.
- On Linux, use `mono hello.exe` or `"ilrun <name>.exe"`.

Alternatively, in Visual C# express, you could just hit F5 or the green play button to run the code, even though that is for debugging.

Running hello.exe will produce the following output:

## C Sharp Programming

---

```
Hello,  
World!
```

The program will then wait for you to strike 'enter' before returning to the command prompt.

Note that the example above includes the System namespace via the using keyword. That inclusion allows direct references to any member of the System namespace without specifying its fully qualified name.

The first call to the WriteLine method of the Console class uses a fully qualified reference.

```
System.Console.WriteLine("Hello,");
```

The second call to that method shortens the reference to the Console class by taking advantage of the fact that the System namespace is included (with using System).

```
Console.WriteLine("World!");
```

C# is a fully object-oriented language. The following sections explain the syntax of the C# language as a beginner's course for programming in the language. Note that much of the power of the language comes from the classes provided with the .NET framework, which are not part of the C# language syntax *per se*.



# Naming

This section will define the naming conventions that are generally accepted by the C# development community. Some companies may define naming conventions that differ from this, but that is done on an individual basis and is generally discouraged. Some of the objects discussed in this section may be beyond the reader's knowledge at this point, but this section can be referred back to later.

## Reasoning

Much of the naming standards are derived from Microsoft's .NET Framework libraries. These standards have proven to make names readable and understandable "at a glance". By using the correct conventions when naming objects, you ensure that other C# programmers who read your code will easily understand what objects are without having to search your code for their definition.

## Conventions

### Namespace

Namespaces are named using Pascal Case with no underscores. This means the first letter of every word in the name is capitalized. For example: MyNewNamespace. Also, note that Pascal Case also denotes that acronyms of three or more letters should only have the first letter capitalized (MyXmlNamespace instead of MyXMLNamespace)

### Assemblies

If an assembly contains only one namespace, they should use the same name. Otherwise, Assemblies should follow the normal Pascal Case format.

### Classes and Structures

Pascal Case, no underscores or leading "C", "cls", or "I". Classes should not have the same name as the namespace in which they reside. Any acronyms of three or more letters should be pascal case, not all caps. Try to avoid abbreviations, and try to always use nouns.

### Exception Classes

Follow class naming conventions, but add Exception to the end of the name. In .Net 2.0, all classes should inherit from the *System.Exception* base class, and not inherit from the *System.ApplicationException*.

### Interfaces

Follow class naming conventions, but start the name with "I" and capitalize the letter following the "I". Example: IFoo The "I" prefix helps to differentiate between Interfaces and classes and also to avoid name collisions.

### Functions

Pascal Case, no underscores except in the event handlers. Try to avoid abbreviations. Many programmers have a nasty habit of overly abbreviating everything. This should be discouraged.

### Properties and Public Member Variables

Pascal Case, no underscores. Try to avoid abbreviations.

## Parameters and Procedure-level Variables

Camel Case. Try to avoid abbreviations. Camel Case is the same as Pascal case, but the first letter of the first word is lowercased.

## Class-level Private and Protected Variables

Camel Case with a leading underscore. Always indicate 'Protected' or 'Private' in the declaration. The leading underscore is the only controversial thing in this document. The leading character helps to prevent name collisions in constructors (a parameter and a private variable have the same name).

## Controls on Forms

Pascal Case with a prefix that identifies it as being part of the UI instead of a purely coded control (ex. a temporary variable). Many developers use "ui" as the prefix followed by a descriptive name such as "UserNameTextBox"

## Constants

Pascal Case. The use of SCREAMING\_CAPS is discouraged. This is a large change from earlier conventions. Most developers now realize that in using SCREAMING\_CAPS they betray more implementation than is necessary. A large portion of the .NET Framework Design Guidelines is dedicated to this discussion.

## Example

Here is an example of a class that uses all of these naming conventions combined.

```
using System;

namespace MyExampleNamespace
{
```

```
public class Customer : IDisposable
{
    private string _customerName;
    public string CustomerName
    {
        get
        {
            return _customerName;
        }
        set
        {
            _customerName = value;
            _lastUpdated = DateTime.Now;
        }
    }

    private DateTime _lastUpdated;

    public DateTime LastUpdated
    {
        get
        {
            return _lastUpdated;
        }
        private set
        {
            _lastUpdated = value;
        }
    }

    public void UpdateCustomer(string newName)
    {
        if( !newName.Equals(customerName))
        {
            CustomerName = newName;
        }
    }

    public void Dispose()
    {
        //Do nothing
    }
}
}
```

# Syntax

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

C# syntax looks quite similar to the syntax of Java because both inherit much of their syntax from C and C++. **The object-oriented** nature of C# requires the high-level structure of a C# program to be defined in terms of **classes**, whose detailed behaviors are defined by their *statements*.

## Statements

The basic unit of execution in a C# program is the *statement*. A statement can declare a variable, define an expression, perform a simple action by calling a method, **control the flow of execution** of other statements, create an object, or assign a value to a variable, property, or field. Statements are usually terminated by a semicolon.

Statements can be grouped into comma-separated statement lists or brace-enclosed statement blocks.

Examples:

```
int sampleVariable;           // declaring a variable
sampleVariable = 5;          // assigning a value
Method();                    // calling an instance method
SampleClass sampleObject = new SampleClass(); // creating a new instance of
an object
sampleObject.ObjectMethod(); // calling a member function of
an object
```

```
// executing a "for" loop with an embedded "if" statement
for(int i = 0; i < upperLimit; i++)
{
    if (SampleClass.SampleStaticMethodReturningBoolean(i))
    {
        sum += sampleObject.SampleMethodReturningInteger(i);
    }
}
```

## Statement blocks

A series of statements surrounded by curly braces form a *block* of code. Among other purposes, code blocks serve to limit the scope of variables defined within them. Code blocks can be nested and often appear as the bodies of methods.

```
private void MyMethod(int value)
{ // This block of code is the body of "MyMethod()"

    // The 'value' integer parameter is accessible to everything in the method

    int methodLevelVariable; // This variable is accessible to everything in
the method

    if (value == 2)
    {
        // methodLevelVariable is still accessible here

        int limitedVariable; // This variable is only accessible to code in the
if block

        DoSomeWork(limitedVariable);
    }

    // limitedVariable is no longer accessible here
} // Here ends the code block for the body of "MyMethod()".
```

# Comments

*Comments* allow inline documentation of source code. The C# compiler ignores comments. Three styles of comments are allowed in C#:

## Single-line comments

The `"/"` character sequence marks the following text as a single-line comment. Single-line comments, as one would expect, end at the first end-of-line following the `"/"` comment marker.

## Multiple-line comments

Comments can span multiple lines by using the multiple-line comment style. Such comments start with `"/**"` and end with `"*/"`. The text between those multi-line comment markers is the comment.

```
//This style of a comment is restricted to one line.  
/*  
    This is another style of a comment.  
    It allows multiple lines.  
*/
```

## XML Documentation-line comments

This comment is used to generate XML documentation. Each line of the comment begins with `"/"/"`.

```
/// <summary> documentation here </summary>
```

This is the most recommended type. Avoid using butterfly style comments. For example:

```
/**  
// Butterfly style documentation comments like this are not recommend-  
ed.  
/**
```

# Case sensitivity

C# is **case-sensitive**, including its variable and method names.

The variables `myInteger` and `MyInteger` below are distinct because C# is case-sensitive:

```
int myInteger = 3;  
int MyInteger = 5;
```

For example, C# defines a class `Console` to handle most operations with the console window. Writing the following code would result in a compiler error unless an object named `console` had been previously defined.

```
// Compiler error!  
console.WriteLine("Hello");
```

The following corrected code compiles as expected because it uses the correct case:

```
Console.WriteLine("Hello");
```



# Variables

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

**Variables** are used to store values. More technically, a variable **binds** an **object** (in the general sense of the term, i.e. a specific value) to an identifier (the variable's name) so that the object can be accessed later. Variables can, for example, store a value for later use:

```
string name = "Dr. Jones";  
Console.WriteLine("Good morning " + name);
```

In this example "name" is the identifier and "Dr. Jones" is the value that we bound to it. Also, each variable is declared with an explicit *type*. Only values whose types are compatible with the variable's declared type can be bound to (stored in) the variable. In the above example we stored "Dr. Jones" into a variable of the type string. This is a legal statement. However, if we had said `int name = "Dr. Jones"`, the compiler would have thrown an error telling us that you cannot implicitly convert between int and string. There are methods for doing this, but we will talk about them later.

## Fields, Local Variables, and Parameters

C# supports several program elements corresponding to the general programming concept of *variable*: *fields*, *parameters*, and *local variables*.

# Fields

**Fields**, sometimes called class-level variables, are variables associated with classes or structures. An *instance variable* is a field associated with an instance of the class or structure, while a *static variable*, declared with the **static** keyword, is a field associated with the type itself. Fields can also be associated with their class by making them *constants* (**const**), which requires a declaration assignment of a constant value and prevents subsequent changes to the field.

Each field has a visibility of *public*, *protected*, *internal*, *protected internal*, or *private* (from most visible to least visible).

# Local variables

Like fields, local variables can optionally be *constant* (**const**). Constant local variables are stored in the assembly data region, while non-constant local variables are stored (or referenced from) the stack. They thus have both a scope and an extent of the method or statement block that declares them.

# Parameter

**Parameters** are variables associated with a method.

An *in* parameter may either have its value passed in from the callee to the method's environment, so that changes to the parameter by the method do not affect the value of the callee's variable, or passed in by reference, so that changes to the variables will affect the value of the callee's variable. Value types (int, double, string) are passed in "by value" while reference types (objects) are passed in "by reference." Since this is the default for the C# compiler, it is not necessary to use .

An *out* parameter does not have its value copied, thus changes to the variable's value within the method's environment directly affect the value from the callee's environment. Such a variable is considered by the compiler to be *unbound* upon method entry, thus it is illegal to reference an *out* parameter before assigning it a value. It also **must** be assigned by the method in each valid (non-exceptional) code path through the method in order for the method to compile.

A *reference* parameter is similar to an *out* parameter, except that it is *bound* before the method call and it need not be assigned by the method.

A *params* parameter represents a variable number of parameters. If a method signature includes one, the *params* argument must be the last argument in the signature.

```
// Each pair of lines is what the definition of a method and a call of a
// method with each of the parameters types would look like.
// In param:
void MethodOne(int param1) //definition
MethodOne(variable);      //call

// Out param:
void MethodTwo(out string message) //definition
MethodTwo(out variable);          //call

// Reference param:
void MethodThree(ref int someFlag) //definition
MethodThree(ref theFlag)          //call

// Params
void MethodFour(params string[] names) //definition
MethodFour("Matthew", "Mark", "Luke", "John"); //call
```

## Types

Each **type** in C# is either a *value type* or a *reference type*. C# has several predefined ("built-in") types and allows for declaration of custom value types and reference types.

## Integral types

Because the type system in C# is unified with other languages that are CLI-compliant, each integral C# type is actually an alias for a corresponding type in the .NET framework. Although the names of the aliases vary between .NET languages, the underlying types in the .NET framework remain the same. Thus, objects created in assemblies written in other languages of the .NET Framework can be bound to C# variables of any type to which the value can be converted, per the conversion rules below. The following illustrates the cross-language compatibility of types by comparing C# code with the equivalent Visual Basic .NET code:

```
// C#
public void UsingCSharpTypeAlias()
{
```

## C Sharp Programming

---

```
int i = 42;
}
public void EquivalentCodeWithoutAlias()
{
    System.Int32 i = 42;
}
```

```
' Visual Basic .NET
Public Sub UsingVisualBasicTypeAlias()
    Dim i As Integer = 42
End Sub
Public Sub EquivalentCodeWithoutAlias()
    Dim i As System.Int32 = 42
End Sub
```

Using the language-specific type aliases is often considered more readable than using the fully-qualified .NET Framework type names.

The fact that each C# type corresponds to a type in the unified type system gives each *value type* a consistent size across platforms and compilers. That consistency is an important distinction from other languages such as C, where, e.g. a long is only guaranteed to be *at least as large as an int*, and is implemented with different sizes by different compilers. As *reference types*, variables of types derived from object (i.e. any class) are exempt from the consistent size requirement. That is, the size of *reference types* like System.IntPtr, as opposed to *value types* like System.Int, may vary by platform. Fortunately, there is rarely a need to know the actual size of a *reference type*.

There are two predefined *reference types*: object, an alias for the System.Object class, from which all other reference types derive; and string, an alias for the System.String class. C# likewise has several integral value types, each an alias to a corresponding value type in the System namespace of the .NET Framework. The predefined C# type aliases expose the methods of the underlying .NET Framework types. For example, since the .NET Framework's System.Int32 type implements a ToString() method to convert the value of an integer to its string representation, C#'s int type exposes that method:

```
int i = 97;
string s = i.ToString();
// The value of s is now the string "97".
```

Likewise, the `System.Int32` type implements the `Parse()` method, which can therefore be accessed via C#'s `int` type:

```
string s = "97";
int i = int.Parse(s);
// The value of i is now the integer 97.
```

The unified type system is enhanced by the ability to convert value types to reference types (*boxing*) and likewise to convert certain reference types to their corresponding value types (*unboxing*). This is also known as *casting*.

```
object boxedInteger = 97;
int unboxedInteger = (int)boxedInteger;
```

Boxing and casting are, however, not type-safe: the compiler won't generate an error if the programmer mixes up the types. In the following short example the mistake is quite obvious, but in complex programs it may be real hard to spot. Avoid boxing, if possible.

```
object getInteger = "97";
int anInteger = (int)getInteger; // no compile-time error, the program will
crash, however
```

The built-in C# type aliases and their equivalent .NET Framework types follow:

### Integers

<b>C# Alias</b>	<b>.NET Type</b>	<b>Size (bits)</b>	<b>Range</b>
sbyte	System.SByte	8	-128 to 127
byte	System.Byte	8	0 to 255
short	System.Int16	16	-32,768 to 32,767
ushort	System.UInt16	16	0 to 65,535
char	System.Char	16	A unicode character of code 0 to 65,535
int	System.Int32	32	-2,147,483,648 to 2,147,483,647
uint	System.UInt32	32	0 to 4,294,967,295
long	System.Int64	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	System.UInt64	64	0 to 18,446,744,073,709,551,615

### Floating-point

<b>C# Alias</b>	<b>.NET Type</b>	<b>Size (bits)</b>	<b>Precision</b>	<b>Range</b>
float	System.Single	32	7 digits	$1.5 \times 10^{-45}$ to $3.4 \times 10^{38}$
double	System.Double	64	15-16 digits	$5.0 \times 10^{-324}$ to $1.7 \times 10^{308}$
decimal	System.Decimal	128	28-29 decimal places	$1.0 \times 10^{-28}$ to $7.9 \times 10^{28}$

### Other predefined types

<b>C# Alias</b>	<b>.NET Type</b>	<b>Size (bits)</b>	<b>Range</b>
bool	System.Boolean	32	true or false, which aren't related to any integer in C#.
object	System.Object	32/64	Platform dependant (a pointer to an object).
string	System.String	16 * length	A unicode string with no special upper bound.

## Custom types

The predefined types can be aggregated and extended into custom types.

Custom *value types* are declared with the `struct` or `enum` keyword. Likewise, *custom reference types* are declared with the `class` keyword.

## Arrays

Although the number of dimensions is included in array declarations, the size of each dimension is not:

```
string[] s;
```

Assignments to an array variable (prior to the variable's usage), however, specify the size of each dimension:

```
s = new string[5];
```

As with other variable types, the declaration and the initialization can be combined:

```
string[] s = new string[5] ;
```

It is also important to note that like in Java, arrays are passed by reference, and not passed by value. For example, the following code snippet successfully swaps two elements in an integer array:

```
static void swap (int[] arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

# Conversion

Values of a given type may or may not be explicitly or implicitly convertible to other types depending on predefined conversion rules, inheritance structure, and explicit cast definitions.

## Predefined conversions

Many predefined value types have predefined conversions to other predefined value types. If the type conversion is guaranteed not to lose information, the conversion can be *implicit* (i.e. an explicit *cast* is not required).

## Inheritance polymorphism

A value can be implicitly converted to any class from which it inherits or interface that it implements. To convert a base class to a class that inherits from it, the conversion must be explicit in order for the conversion statement to compile. Similarly, to convert an interface instance to a class that implements it, the conversion must be explicit in order for the conversion statement to compile. In either case, the runtime environment throws a conversion exception if the value to convert is not an instance of the target type or any of its derived types.

# Scope and extent

The scope and extent of variables is based on their declaration. The scope of parameters and local variables corresponds to the declaring method or statement block, while the scope of fields is associated with the instance or class and is potentially further restricted by the field's access modifiers.

The extent of variables is determined by the runtime environment using implicit reference counting and a complex garbage collection algorithm.



# Operators

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

C# operators and their precedence closely resemble the operators in other languages of the C family.

Similar to C++, classes can *overload* most operators, defining or redefining the behavior of the operators in contexts where the first argument of that operator is an instance of that class, but doing so is often discouraged for clarity.

Following are the built-in behaviors of C# operators.

## Arithmetic

The following arithmetic operators operate on numeric operands (arguments a and b in the "sample usage" below).

Sample usage	Read	Explanation
$a + b$	<i>a plus b</i>	The binary operator $+$ returns the <b>sum</b> of its arguments.
$a - b$	<i>a minus b</i>	The binary operator $-$ returns the <b>difference</b> between its arguments.
$a * b$	<i>a times b</i>	The binary operator $*$ returns the <b>multiplicative product</b> of its arguments.
$a / b$	<i>a divided by b</i>	The binary operator $/$ returns the <b>quotient</b> of its arguments. If both of its operators are integers, it obtains that quotient using <i>integer division</i> (i.e. it drops any resulting remainder).
$a \% b$	<i>a mod b</i>	The binary operator $\%$ operates only on integer arguments. It returns the <b>remainder</b> of <i>integer division</i> of those arguments. (See <i>modular arithmetic</i> .)
$a++$	<i>a plus plus</i> or <i>Postincrement a</i>	The unary operator $++$ operates only on arguments that have an <i>l-value</i> . When placed <b>after</b> its argument, it increments that argument by 1 and returns the value of that argument before it was incremented.
$++a$	<i>plus plus a</i> or <i>Preincrement a</i>	The unary operator $++$ operates only on arguments that have an <i>l-value</i> . When placed <b>before</b> its argument, it increments that argument by 1 and returns the resulting value.
$a--$	<i>a minus minus</i> or <i>Postdecrement a</i>	The unary operator $--$ operates only on arguments that have an <i>l-value</i> . When placed <b>after</b> its argument, it decrements that argument by 1 and returns the value of that argument before it was decremented.
$--a$	<i>minus minus a</i> or <i>Predecrement a</i>	The unary operator $--$ operates only on arguments that have an <i>l-value</i> . When placed <b>before</b> its argument, it decrements that argument by 1 and returns the resulting value.

## Logical

The following logical operators operate on boolean or integral operands, as noted.

Sample usage	Read	Explanation
<code>a &amp; b</code>	<code>a bitwise and b</code>	The binary operator <code>&amp;</code> evaluates both of its operands and returns the <b>logical conjunction</b> ("AND") of their results. If the operands are integral, the logical conjunction is performed bitwise.
<code>a &amp;&amp; b</code>	<code>a and b</code>	The binary operator <code>&amp;&amp;</code> operates on boolean operands only. It evaluates its first operand. If the result is <i>false</i> , it returns <i>false</i> . Otherwise, it evaluates and returns the results of the second operand. Note that if evaluating the second operand would hypothetically have no side effects, the results are identical to the logical conjunction performed by the <code>&amp;</code> operator.
<code>a   b</code>	<code>a bitwise or b</code>	The binary operator <code> </code> evaluates both of its operands and returns the <b>logical disjunction</b> ("OR") of their results. If the operands are integral, the logical disjunction is performed bitwise.
<code>a    b</code>	<code>a or b</code>	The binary operator <code>  </code> operates on boolean operands only. It evaluates the first operand. If the result is <i>true</i> , it returns <i>true</i> . Otherwise, it evaluates and returns the results of the second operand. Note that if evaluating the second operand would hypothetically have no side effects, the results are identical to the logical disjunction performed by the <code> </code> operator.
<code>a ^ b</code>	<code>a x-or b</code>	The binary operator <code>^</code> returns the <b>exclusive or</b> ("XOR") of their results. If the operands are integral, the exclusive or is performed bitwise.
<code>!a</code>	<code>not a</code>	The unary operator <code>!</code> operates on a boolean operand only. It evaluates its operand and returns the <b>negation</b> ("NOT") of the result. That is, it returns <i>true</i> if <code>a</code> evaluates to <i>false</i> and it returns <i>false</i> if <code>a</code> evaluates to <i>true</i> .
<code>~a</code>	<code>bitwise not a</code>	The unary operator <code>~</code> operates on integral operands only. It evaluates its operand and returns the bitwise negation of the result. That is, <code>~a</code> returns a value where each bit is the negation of the corresponding bit in the result of evaluating <code>a</code> .

# Bitwise shifting

Sample usage	Read	Explanation
<code>a &lt;&lt; b</code>	<i>a left shift b</i>	The binary operator <code>&lt;&lt;</code> evaluates its operands and returns the resulting first argument left-shifted by the number of bits specified by the second argument. It discards high-order bits that shift beyond the size of its first argument and sets new low-order bits to zero.
<code>a &gt;&gt; b</code>	<i>a right shift b</i>	The binary operator <code>&gt;&gt;</code> evaluates its operands and returns the resulting first argument right-shifted by the number of bits specified by the second argument. It discards low-order bits that are shifted beyond the size of its first argument and sets new high-order bits to the sign bit of the first argument, or to zero if the first argument is unsigned.

# Relational

The binary relational operators `==`, `!=`, `<`, `>`, `<=`, and `>=` are used for relational operations and for type comparisons.

Sample usage	Read	Explanation
<code>a == b</code>	<i>a is equal to b</i>	For arguments of <i>value</i> type, the operator <code>==</code> returns <i>true</i> if its operands have the same value, false otherwise. For the <i>string</i> type, it returns <i>true</i> if the strings' character sequences match. For other <i>reference</i> types (types derived

		from System.Object), however, a == b returns <i>true</i> only if a and b reference the same object.
a != b	a is not equal to b	The operator != returns the logical negation of the operator ==. Thus, it returns <i>true</i> if a is not equal to b, and <i>false</i> if they are equal.
a < b	a is less than b	The operator < operates on integral types. It returns <i>true</i> if a is less than b, <i>false</i> otherwise.
a > b	a is greater than b	The operator > operates on integral types. It returns <i>true</i> if a is greater than b, <i>false</i> otherwise.
a <= b	a is less than or equal to b	The operator <= operates on integral types. It returns <i>true</i> if a is less than or equal to b, <i>false</i> otherwise.
a >= b	a is greater than or equal to b	The operator >= operates on integral types. It returns <i>true</i> if a is greater than or equal to b, <i>false</i> otherwise.

## Assignment

The assignment operators are binary. The most basic is the operator =. Not surprisingly, it assigns the value of its second argument to its first argument.

(More technically, the operator = requires for its first (*left*) argument an expression to which a value can be assigned (an *l-value*) and for its second (*right*) argument an expression which can be evaluated (an *r-value*). That requirement of an *assignable* expression to its left and a *bound* expression to its right is the origin of the terms *l-value* and *r-value*.)

The first argument of the assignment operator (=) is typically a variable. When that argument has a *value* type, the assignment operation changes the argument's underlying value. When the first argument is a *reference* type, the assignment operation changes the reference, so the first argument typically just refers to a different object but the object that it originally referenced does not change (except that it may no longer be referenced and may thus be a candidate for *garbage collection*).

Sample usage	Read	Explanation
<code>a = b</code>	<i>a equals (or set to) b</i>	The operator = evaluates its second argument and then assigns the results to (the <i>l-value</i> indicated by) its first argument.
<code>a = b = c</code>	<i>b set to c, and then a set to b</i>	Equivalent to <code>a = (b = c)</code> . When there are consecutive assignments, the right-most assignment is evaluated first, proceeding from right to left. In this example, both variables a and b have the value of c.

## Short-hand Assignment

The short-hand assignment operators shortens the common assignment operation of `a = a operator b` into `a operator= b`, resulting in less typing and neater syntax.

Sample usage	Read	Explanation
<code>a += b</code>	<i>a plus equals (or increment by) b</i>	Equivalent to <code>a = a + b</code> .
<code>a -= b</code>	<i>a minus equals (or decrement by) b</i>	Equivalent to <code>a = a - b</code> .
<code>a *= b</code>	<i>a multiply equals (or multiplied by) b</i>	Equivalent to <code>a = a * b</code> .
<code>a /= b</code>	<i>a divide equals (or divided by) b</i>	Equivalent to <code>a = a / b</code> .
<code>a %= b</code>	<i>a mod equals b</i>	Equivalent to <code>a = a % b</code> .
<code>a &amp;= b</code>	<i>a and equals b</i>	Equivalent to <code>a = a &amp; b</code> .
<code>a  = b</code>	<i>a or equals b</i>	Equivalent to <code>a = a   b</code> .
<code>a ^= b</code>	<i>a xor equals b</i>	Equivalent to <code>a = a ^ b</code> .
<code>a &lt;&lt;= b</code>	<i>a left-shift equals b</i>	Equivalent to <code>a = a &lt;&lt; b</code> .
<code>a &gt;&gt;= b</code>	<i>a right-shift equals b</i>	Equivalent to <code>a = a &gt;&gt; b</code> .

## Type information

Expression	Explanation
x is T	returns true if the variable x of base class type stores an object of derived class type T, or, if x is of type T. Else returns false.
x as T	returns (T)x ( <i>x cast to T</i> ) if the variable x of base class type stores an object of derived class type T, or, if x is of type T. Else returns null. Equivalent to x is T ? (T)x : null
sizeof(x)	returns the size of the value type x. Remarks: The sizeof operator can be applied only to value types, not reference types..
typeof(T)	returns a System.Type object describing the type. T must be the name of the type, and not a variable. Use the GetType method to retrieve run-time type information of variables.

## Pointer manipulation

*NOTE: Most C# developers agree that direct manipulation and use of pointers is not recommended in C#. The language has many built-in classes to allow you to do almost any operation you want. C# was built with memory-management in mind and the creation and use of pointers is greatly disruptive to this end. This speaks to the declaration of pointers and the use of pointer notation, not arrays. In fact, a program may only be compiled in "unsafe mode" if it uses pointers.*

Expression	Explanation
*a	<i>Indirection</i> operator. Allows access the object being pointed.
a->member	Similar to the '.' operator. Allows access to members of classes and structs being pointed.
a[]	Used to <i>index</i> a pointer.
&a	References the <i>address</i> of the pointer.
stackalloc	allocates memory on the stack.
fixed	Temporarily fixes a variable in order that its address may be found.

# Overflow exception control

<b>Expression</b>	<b>Explanation</b>
checked(a)	uses overflow checking on value a
unchecked(a)	avoids overflow checking on value a

# Others

<b>Expression</b>	<b>Explanation</b>
a.b	accesses member b of type or namespace a
a[b]	the value of index b in a
(a)b	casts the value b to type a
new a	creates an object of type a
a + b	if a and b are string types, concatenates a and b
a ? b : c	if a is true, returns the value of b, otherwise c
a ?? b	if a is null, returns b, otherwise returns a
@a	you can write a path without mentioning the special characters. (example: @"c:\\" instead of "c:\\")



# Data structures

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

There are various ways of grouping sets of data together in C#.

## Enumerations

An *enumeration* is a data type that *enumerates* a set of items by assigning to each of them an identifier (a name), while exposing an underlying base type for ordering the elements of the enumeration. The underlying type is int by default, but can be any one of the integral types except for char.

Enumerations are declared as follows:

```
enum Weekday { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
};
```

The elements in the above enumeration are then available as constants:

```
Weekday day = Weekday.Monday;
if (day == Weekday.Tuesday)
{
    Console.WriteLine("Time sure flies by when you program in C#!");
}
```

## C Sharp Programming

---

If no explicit values are assigned to the enumerated items as the example above, the first element has the value 0, and the successive values are assigned to each subsequent element. However, specific values from the underlying integral type can be assigned to any of the enumerated elements:

```
enum Age { Infant = 0, Teenager = 13, Adult = 18 };  
  
Age age = Age.Teenager;  
Console.WriteLine("You become a teenager at an age of {0}.", (int)age);
```

The underlying values of enumerated elements may go unused when the purpose of an enumeration is simply to group a set of items together, e.g., to represent a nation, state, or geographical territory in a more meaningful way than an integer could. Rather than define a group of logically related constants, it is often more readable to use an enumeration.

It may be desirable to create an enumeration with a base type other than int. To do so, specify any integral type besides char as with base class *extension* syntax after the name of the enumeration, as follows:

```
enum CardSuit : byte { Hearts, Diamonds, Spades, Clubs };
```

The enumeration type is also helpful if you need to output the value. By calling the `.ToString()` method on the enumeration, will output the enumerations name (e.g. `CardSuit.Hearts.ToString()` will output "Hearts").

## Structs

Structures (keyword `struct`) are light-weight objects. They are mostly used when only a data container is required for a collection of value type variables. *Structs* are similar to *classes* in that

they can have constructors, methods, and even implement interfaces, but there are important differences.

- *Structs* are value types while *classes* are reference types, which means they behave differently when passed into methods as parameters.
- *Structs* cannot support inheritance. While *structs* may appear to be limited with their use, they require less memory and can be less expensive if used in the proper way.
- *Structs* always have a default constructor, even if you don't want one. Classes allow you to hide the constructor away by using the "private" modifier, whereas structures *must* have one.

A struct can, for example, be declared like this:

```
struct Person
{
    public string name;
    public System.DateTime birthDate;
    public int heightInCm;
    public int weightInKg;
}
```

The Person *struct* can then be used like this:

```
Person dana = new Person();
dana.name = "Dana Developer";
dana.birthDate = new DateTime(1974, 7, 18);
dana.heightInCm = 178;
dana.weightInKg = 50;

if (dana.birthDate < DateTime.Now)
{
    Console.WriteLine("Thank goodness! Dana Developer isn't from the fu-
ture!");
}
```

It is also possible to provide *constructors* to structs to make it easier to initialize them:

```
using System;
struct Person
{
```

```
string name;
DateTime birthDate;
int heightInCm;
int weightInKg;

public Person(string name, DateTime birthDate, int heightInCm, int
weightInKg)
{
    this.name = name;
    this.birthDate = birthDate;
    this.heightInCm = heightInCm;
    this.weightInKg = weightInKg;
}

}

public class StructWikiBookSample
{
    public static void Main()
    {
        Person dana = new Person("Dana Developer", new DateTime(1974, 7, 18),
178, 50);
    }
}
```

*Structs* are really only used for performance reasons and/or if you intend to it by value. Structs work best when holding a total equal to or less than 16 bytes of data. If in doubt, use classes.

## Arrays

Arrays represent a set of items all belonging to the same type. The declaration itself may use a variable or a constant to define the length of the array. However, an array has a set length and it cannot be changed after declaration.

```
// an array whose length is defined with a constant
int[] integers = new int[20];

int length = 0;
System.Console.Write("How long should the array be? ");
System.Console.ReadLine(length);
// an array whose length is defined with a variable
```

```
// this array still can't change length after declaration  
double[] doubles = new double[length];
```

# Control

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

Conditional, iteration, jump, and exception handling statements control a program's flow of execution.

An iteration statement can create a loop using keywords such as `do`, `while`, `for`, `foreach`, and `in`.

A jump statement can be used to transfer program control using keywords such as `break`, `continue`, `return`, and `yield`.

An exception handling statement can be used to handle exceptions using keywords such as `throw`, `try-catch`, `try-finally`, and `try-catch-finally`.

## Conditional statements

A conditional statement decides whether to execute code based on conditions. The `if` statement and the `switch` statement are the two types of conditional statements in C#.

## The `if` statement

As with most of C#, the `if` statement has the same syntax as in C, C++, and Java. Thus, it is written in the following form:

```
if-statement ::= "if" "(" condition ")" if-body ["else" else-body]
```

*condition ::= boolean-expression*

*if-body ::= statement-or-statement-block*

*else-body ::= statement-or-statement-block*

The if statement evaluates its *condition* expression to determine whether to execute the *if-body*. Optionally, an else clause can immediately follow the *if body*, providing code to execute when the *condition* is *false*. Making the *else-body* another if statement creates the common *cascade* of if, else if, else if, else if, else if, else statements:

```
using System;

public class IfStatementSample
{
    public void IfMyNumberIs()
    {
        int myNumber = 5;
        if ( myNumber == 4 )
            Console.WriteLine("This will not be shown because myNumber is not
4.");
        else if( myNumber < 0 )
        {
            Console.WriteLine("This will not be shown because myNumber is not
negative.");
        }
        else if( myNumber % 2 == 0 )
            Console.WriteLine("This will not be shown because myNumber is not
even.");
        else
        {
            Console.WriteLine("myNumber does not match the coded conditions,
so this sentence will be shown!");
        }
    }
}
```

## The switch statement

The switch statement is similar to the statement from C, C++ and Java.

Unlike C, each case statement must finish with a jump statement (which can be break or goto or return). In other words, C# does not support "fall through" from one case statement to

## C Sharp Programming

---

the next (thereby eliminating a common source of unexpected behaviour in C programs). However "stacking" of cases is allowed, as in the example below. If goto is used, it may refer to a case label or the default case (e.g. goto case 0 or goto default).

The default label is optional. If no default case is defined, then the default behaviour is to do nothing.

A simple example:

```
switch (nCPU)
{
    case 0:
        Console.WriteLine("You don't have a CPU! :-");
        break;
    case 1:
        Console.WriteLine("Single processor computer");
        break;
    case 2:
        Console.WriteLine("Dual processor computer");
        break;
    // Stacked cases
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
        Console.WriteLine("A multi processor computer");
        break;
    default:
        Console.WriteLine("A seriously parallel computer");
        break;
}
```

A nice improvement over the C switch statement is that the switch variable can be a string. For example:

```
switch (aircraft_ident)
{
    case "C-FESO":
        Console.WriteLine("Rans S6S Coyote");
        break;
    case "C-GJIS":
        Console.WriteLine("Rans S12XL Airaile");
}
```



```
        break;
    default:
        Console.WriteLine("Unknown aircraft");
        break;
}
```

## Iteration statements

An iteration statement creates a *loop* of code to execute a variable number of times. The for loop, the do loop, the while loop, and the foreach loop are the iteration statements in C#.

### The do...while loop

The **do...while** loop likewise has the same syntax as in other languages derived from C. It is written in the following form:

```
do...while-loop ::= "do" body "while" "(" condition ")"
condition ::= boolean-expression
body ::= statement-or-statement-block
```

The do...while loop always runs its *body* once. After its first run, it evaluates its *condition* to determine whether to run its *body* again. If the *condition* is *true*, the *body* executes. If the *condition* evaluates to *true* again after the *body* has ran, the *body* executes again. When the *condition* evaluates to *false*, the do...while loop ends.

```
using System;

public class DoWhileLoopSample
{
    public void PrintValuesFromZeroToTen()
    {
        int number = 0;
        do
        {
            Console.WriteLine(number++.ToString());
        } while(number <= 10);
    }
}
```

```
}
```

The above code writes the integers from 0 to 10 to the console.

## The for loop

The **for** loop likewise has the same syntax as in other languages derived from C. It is written in the following form:

```
for-loop ::= "for" "(" initialization ";" condition ";" iteration ")" body  
initialization ::= variable-declaration | list-of-statements  
condition ::= boolean-expression  
iteration ::= list-of-statements  
body ::= statement-or-statement-block
```

The *initialization* variable declaration or statements are executed the first time through the for loop, typically to declare and initialize an index variable. The *condition* expression is evaluated before each pass through the *body* to determine whether to execute the body. It is often used to test an index variable against some limit. If the *condition* evaluates to *true*, the *body* is executed. The *iteration* statements are executed after each pass through the *body*, typically to increment or decrement an index variable.

```
public class ForLoopSample  
{  
    public void ForFirst100NaturalNumbers()  
    {  
        for(int i=0; i<100; i++)  
        {  
            System.Console.WriteLine(i.ToString());  
        }  
    }  
}
```

The above code writes the integers from 0 to 99 to the console.

## The foreach loop

The **foreach** statement is similar to the for statement in that both allow code to iterate over the items of collections, but the foreach statement lacks an iteration index, so it works even with collections that lack indices altogether. It is written in the following form:

*foreach-loop ::= "foreach" "(" variable-declaration "in" enumerable-expression ")" body*  
*body ::= statement-or-statement-block*

The *enumerable-expression* is an expression of a type that implements **IEnumerable**, so it can be an array or a *collection*. The *variable-declaration* declares a variable that will be set to the successive elements of the *enumerable-expression* for each pass through the *body*. The foreach loop exits when there are no more elements of the *enumerable-expression* to assign to the variable of the *variable-declaration*.

```
public class ForEachSample
{
    public void DoSomethingForEachItem()
    {
        string[] itemsToWrite = {"Alpha", "Bravo", "Charlie"};
        foreach (string item in itemsToWrite)
            System.Console.WriteLine(item);
    }
}
```

In the above code, the foreach statement iterates over the elements of the string array to write "Alpha", "Bravo", and "Charlie" to the console.

## The while loop

The **while** loop has the same syntax as in other languages derived from C. It is written in the following form:

*while-loop ::= "while" "(" condition ")" body*  
*condition ::= boolean-expression*  
*body ::= statement-or-statement-block*

## C Sharp Programming

---

The while loop evaluates its *condition* to determine whether to run its *body*. If the *condition* is *true*, the *body* executes. If the *condition* then evaluates to *true* again, the *body* executes again. When the *condition* evaluates to *false*, the while loop ends.

```
using System;

public class WhileLoopSample
{
    public void RunForAwhile()
    {
        TimeSpan durationToRun = new TimeSpan(0, 0, 30);
        DateTime start = DateTime.Now;
        while (DateTime.Now - start < durationToRun)
        {
            Console.WriteLine("not finished yet");
        }
        Console.WriteLine("finished");
    }
}
```

## Jump statements

A jump statement can be used to transfer program control using keywords such as `break`, `continue`, `return`, `yield`, and `throw`.

# Exceptions

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The exception handling system in the C# language allows the programmer to handle errors or anomalous situations in a structured manner that allows the programmer to separate the normal flow of the code from error-handling logic. An exception can represent a variety of abnormal conditions, including, for example, the use of a null object reference detected by the runtime system, or an invalid input string entered by a user and detected by application code. Code that detects an error condition is said to *throw* an exception and code that handles the error is said to *catch* the exception. An exception in C# is an object that encapsulates various information about the error that occurred, such as the stack trace at the point of the exception and a descriptive error message. All exception objects are instantiations of the `System.Exception` or a child class of it. There are many exception classes defined in the .NET Framework used for various purposes. Programmers may also define their own class inheriting from `System.Exception` or some other appropriate exception class from the .NET Framework.

Microsoft recommendations prior to version 2.0 recommended that a developer inherit from the `ApplicationException` exception class. After 2.0 was released, this recommendation was made obsolete and users should inherit from the `Exception` class<sup>[1]</sup>.

The following example demonstrates the basics of exception throwing and handling exceptions:

```
class ExceptionTest
{
    public static void Main(string[] args)
    {
        try
        {
            OrderPizza("pepperoni");
        }
    }
}
```

```
        OrderPizza("anchovies");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        Console.WriteLine("press enter to continue...");
        Console.ReadLine();
    }
}

private static void OrderPizza(string topping)
{
    if (topping != "pepperoni" && topping != "sausage")
    {
        throw new ArgumentException(
            String.Format("Unsupported pizza topping: {0}", topping));
    }
    Console.WriteLine("one {0} pizza ordered", topping);
}
}
```

When run, this example produces the following output:

```
one pepperoni pizza ordered
Unsupported pizza topping: anchovies
press enter to continue...
```

The Main() method begins by opening a **try** block. A **try** block is a block of code that may throw an exception that is to be caught and handled. Following the **try** block are one or more **catch** blocks. These blocks contain the exception handling logic. Each **catch** block contains an exception object declaration, similar to the way a method argument is declared, in this case, an **ApplicationException** named **e**. When an exception matching the type of the **catch** block is thrown, that exception object is passed in to the **catch** and available for it to use and even possibly re-throw. The **try** block calls the OrderPizza() method, which may throw an **ApplicationException**. The method checks the input string and, if it has an invalid value, an exception is thrown using the **throw** keyword. The **throw** is followed by the object reference representing the exception object to throw. In this case, the exception object is constructed on the spot. When the exception is thrown, control is transferred to the inner most **catch** block matching the exception type

thrown. In this case, it is one method in the call stack higher. Lastly, the Main() method contains a **finally** block after the **catch** block. The **finally** block is optional and contains code that is to be executed regardless of whether an exception is thrown in the associated **try** block. In this case, the **finally** just prompts the user to press enter, but normally it is used to release acquired resources or perform other cleanup activities.

## References

1. [↑ \[ApplicationException made obsolete\]](#)

# Namespaces

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

Namespaces are used to provide a "named space" in which your application resides. They're used especially to provide the C# compiler a context for all the named information in your program, such as variable names. Without namespaces, you wouldn't be able to make, e.g., a class named `Console`, as .NET already uses one in its `System` namespace. The purpose of namespaces is to solve this problem, and release thousands of names defined in the .NET Framework for your applications to use, along with making it so your application doesn't occupy names for other applications, if your application is intended to be used in conjunction with another. So namespaces exist to resolve ambiguities a compiler wouldn't otherwise be able to do.

Namespaces are easily defined in this way:

```
namespace MyApplication
{
    // The content to reside in the MyApplication namespace is placed here.
}
```

There is an entire hierarchy of namespaces provided to you by the .NET Framework, with the `System` namespace usually being by far the most commonly seen one. Data in a namespace is referred to by using the `.` operator, such as:

```
System.Console.WriteLine("Hello, world!");
```

This will call the `WriteLine` method that is a member of the `Console` class within the `System` namespace.



By using the using keyword, you explicitly tell the compiler that you'll be using a certain namespace in your program. Since the compiler would then know that, it no longer requires you to type the namespace name(s) for such declared namespaces, as you told it which namespaces it should look in if it couldn't find the data in your application.

So one can then type like this:

```
using System;

namespace MyApplication
{
    class MyClass
    {
        void ShowGreeting()
        {
            Console.WriteLine("Hello, world!"); // note how System is now not re-
required
        }
    }
}
```

Namespaces are global, so a namespace in one C# source file, and another with the same name in another source file, will cause the compiler to treat the different named information in these two source files as residing in the same namespace.

## Nested namespaces

Normally, your entire application resides under its own special namespace, often named after your application or project name. Sometimes, companies with an entire product series decide to use nested namespaces though, where the "root" namespace can share the name of the company, and the nested namespaces the respective project names. This can be especially convenient if you're a developer who has made a library with some usual functionality that can be shared across programs. If both the library and your program shared a parent namespace, that one would then not have to be explicitly declared with the using keyword, and still not have to be completely typed out. If your code was open for others to use, third party developers that may use your code would additionally then see that the same company had developed the library and the program. The developer of the library and program would finally also separate all the named information in their product source codes, for fewer headaches especially if common names are used.

## C Sharp Programming

---

To make your application reside in a nested namespace, you can show this in two ways. Either like this:

```
namespace CodeWorks
{
    namespace MyApplication
    {
        // Do stuff
    }
}
```

... or like this:

```
namespace CodeWorks.MyApplication
{
    // Do stuff
}
```

Both methods are accepted, and are identical in what they do.

# Classes

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

As in other object-oriented programming languages, the functionality of a C# program is implemented in one or more **classes**. The *methods* and *properties* of a class contain the code that defines how the class behaves.

C# classes support [information hiding](#) by [encapsulating](#) functionality in properties and methods and by enabling several types of [polymorphism](#), including *subtyping polymorphism* via [inheritance](#) and *parametric polymorphism* via [generics](#).

Several types of C# classes can be defined, including *instance* classes (*standard* classes that can be instantiated), *static* classes, and *structures*.

Classes are defined using the keyword "class" followed by an identifier to name the class. Instances of the class can then be created with the "new" keyword followed by the name of the class. The code below defines a class called Employee with properties Name and Age and with empty methods GetPayCheck() and Work(). It also defines a Sample class that instantiates and uses the Employee class:

```
public class Employee
{
    private string _name;
    private int _age;

    public string Name
    {
        set { _name = value; }
        get { return _name; }
    }
}
```

```
public int Age
{
    set { _age = value; }
    get { return _age; }
}

public void GetPayCheck()
{
}

public void Work()
{
}
}

public class Sample
{
    public static void Main()
    {
        Employee Marissa = new Employee();
        Marissa.Work();
        Marissa.GetPayCheck();
    }
}
```

## Methods

C# *methods* are class members containing code. They may have a return value and a list of **parameters**, as well as a *generic* type declaration. Like fields, methods can be *static* (associated with and accessed through the class) or *instance* (associated with and accessed through an object instance of the class).

## Constructors

A class's *constructors* control its initialization. A constructor's code executes to initialize an instance of the class when a program requests a new object of the class's type. Constructors often set properties of their classes, but they are not restricted to doing so.

Like other methods, a constructor can have *parameters*. To create an object using a constructor with parameters, the new command accepts parameters. The below code defines and then instantiates multiple objects of the Employee class, once using the constructor without parameters and once using the version with a parameter:

```
public class Employee
{
    public Employee()
    {
        System.Console.WriteLine("Constructed without parameters");
    }

    public Employee(string text)
    {
        System.Console.WriteLine(text);
    }
}

public class Sample
{
    public static void Main()
    {
        System.Console.WriteLine("Start");
        Employee Alfred = new Employee();
        Employee Billy = new Employee("Parameter for construction");
        System.Console.WriteLine("End");
    }
}
```

Output:

```
Start
Constructed without parameters
Parameter for construction
End
```

# Finalizers

The opposite of constructors, *finalizers* define the final behavior of an object and execute when the object is no longer in use. Although they are often used in C++ to free memory reserved by an object, they are less frequently used in C# due to the .NET Framework Garbage Collector. An object's finalizer, which takes no parameters, is called sometime after an object is no longer referenced, but the complexities of garbage collection make the specific timing of finalizers uncertain.

```
public class Employee
{
    public Employee(string text)
    {
        System.Console.WriteLine(text);
    }

    ~Employee()
    {
        System.Console.WriteLine("Finalized!");
    }

    public static void Main()
    {
        Employee Marissa = new Employee("Constructed!");
        Marissa = null;
    }
}
```

Output:

```
Constructed!
Finalized!
```

# Properties

C# **properties** are class members that expose functionality of methods using the syntax of *fields*. They simplify the syntax of calling traditional *get* and *set* methods (a.k.a. *accessor* methods). Like methods, they can be *static* or *instance*.

Properties are defined in the following way:

```
public class MyClass
{
    private int integerField = 3; // Sets integerField with a default value
of 3

    public int IntegerField
    {
        get {
            return integerField; // get returns the field you specify when
this property is assigned
        }
        set {
            integerField = value; // set assigns the value assigned to the
property of the field you specify
        }
    }
}
```

The C# keyword `value` contains the value assigned to the property. After a property is defined it can be used like a variable. If you were to write some additional code in the `get` and `set` portions of the property it would work like a method and allow you to manipulate the data before it is read or written to the variable.

```
using System;

public class MyProgram
{
    MyClass myClass = new MyClass;

    Console.WriteLine(myClass.IntegerField); // Writes 3 to the command line.

    myClass.IntegerField = 7; // Indirectly assigns 7 to the field myClass.in-
tegerField
}
```

```
}
```

Using properties in this way provides a clean, easy to use mechanism for protecting data.

## Indexers

C# **indexers** are class members that define the behavior of the *array access* operation (e.g. `list[0]` to access the first element of `list` even when `list` is not an array).

To create an indexer, use the **this** keyword as in the following example:

```
public string this[string key]
{
    get {return coll[key];}
    set {coll[key] = value;}
}
```

This code will create a string indexer that returns a string value. For example, if the class was `EmployeeCollection`, you could write code similar to the following:

```
EmployeeCollection e = new EmployeeCollection();
.
.
.
string s = e["Jones"];
e["Smith"] = "xxx";
```

## Events

C# **events** are class members that expose notifications to clients of the class.



# Operator

C# **operator** definitions are class members that define or redefine the behavior of basic C# operators (called implicitly or explicitly) on instances of the class.

# Structures

*Structures*, or *structs*, are defined with the `struct` keyword followed by an *identifier* to name the structure. They are similar to classes, but have subtle differences. *Structs* are used as lightweight versions of classes that can help reduce memory management efforts when working with small data structures. In most situations, however, using a standard *class* is a better choice.

The principal difference between *structs* and *classes* is that *instances* of *structs* are *values* whereas *instances* of *classes* are *references*. Thus when you pass a *struct* to a function by value you get a copy of the object so changes to it are not reflected in the original because there are now two distinct objects but if you pass an instance of a class by value then there is only one instance.

The Employee structure below declares a public and a private field. Access to the private field is granted through the public **property** "Name":

```
struct Employee
{
    private string name;
    public int age;

    public string Name
    {
        set { name = value; }
        get { return name; }
    }
}
```

# Static classes

**Static classes** are commonly used to implement a [Singleton Pattern](#). All of the methods, properties, and fields of a static class are also static (like the `WriteLine()` method of the `System.Console` class) and can thus be used without instantiating the static class:

```
public static class Writer
{
    public static void Write()
    {
        System.Console.WriteLine("Text");
    }
}

public class Sample
{
    public static void Main()
    {
        Writer.Write();
    }
}
```

# Objects

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

## Introduction

The .NET framework consists of several languages, all which follow the "object orientated programming" (OOP) approach to software development. This standard defines that all objects support

- Inheritance - the ability to inherit and extend existing functionality.
- Encapsulation - the ability to allow the user to only see specific parts, and to interact with it in specific ways.
- Polymorphism - the ability for an object to be assigned dynamically, but with some predicatability as to what can be done with the object.

Objects are synonymous with objects in the real world. Think of any object and think of how it looks and how it is measured and interacted with. When creating OOP languages, the reasoning was that if it mimics the thought process of humans, it would simplify the coding experience.

For example, let's consider a chair, and its dimensions, weight, colour and what it is made out of. In .NET, these values are called "Properties". These are values which define the object's state. Be careful, as there are two ways to expose values: Variables and Properties. The recommended approach is expose Properties and not variables.

So we have a real-world idea of the concept of an object. In terms of practicality for a computer to pass information about, passing around an object within a program would consume a lot of resources. Think of a car, how many properties that has - 100's, 1000's. A computer passing

this information about all the time will waste memory, processing time and therefore a bad idea to use. So objects come in 2 flavours:

- Reference types
- Value types

## Reference and Value Types

A reference type is like a pointer to the value. Think of it like a piece of paper with a street address on it, and the address leads to your house - your object with hundreds of properties. If you want to find it, go to where the address says! This is exactly what happens inside the computer. The reference is stored as a number, corresponding to somewhere in memory where the object exists. So instead of moving an object around - like building a replica house every time you want to look at it - you just look at the original.

A value type is the exact value itself. Values are great for storing small amounts of information: numbers, dates etc.

There are differences in the way they are processed, so we will leave that until a little later in the article.

As well as querying values, we need a way to interacting with the object so that some operation can be performed. Think of files - its all well and good knowing the length of the file, but how about *Read()* 'ing it? Therefore, we can use something called *methods* as a way of performing actions on an object.

An example would be a circle. The properties of a square are:

- Length
- Height

The "functions" (or *methods* in .NET) would be:

- Area ( = Length \* Width )
- Perimeter ( = 2 \* Length + 2 \* Width)

Methods vary from Properties because they require some transformation of data to achieve a result. Methods can either return a result (such as Area) or not. Like above with the chair, if you Sit() on the chair, there is no expected reaction, the chair just ... works!

## System.Object

To support the first rule of OOP - Inheritance, we define something that all objects will derive from - this is *System.Object*, known as Object object. This object defines some methods that all objects can use should they need too. These methods include:

- GetHashCode() - retrieve a number unique to that object.
- GetType() - retrieves information about the object like method names, the objects name etc.
- ToString() - convert the object to a textual representation - usually for outputting to the screen or file.

Since all objects derive from this class (whether you define it or not), any class will have these 3 methods ready to use. Since we always inherit from System.Object, or a class that itself inherits from System.Object, we therefore enhance and/or extend its functionality. Like in the real world that humans, cats, dogs, birds, fish are all an improved and specialised version of an "organism".

## Object basics

All objects by default are reference types. To support value types, objects must instead inherit from the *System.ValueType* abstract class, rather than System.Object.

## Constructors

When objects are created, they are initialized by the "constructor". The constructor sets up the object, ready for use. Because objects need to be created before being used, the constructor is created implicitly, unless it is defined differently by the developer. There are 3 types of constructor:

- Static Constructor
- Default constructor - takes no parameters.
- Overloaded constructor - takes parameters.

Overloaded constructors automatically remove the implicit default constructor, so a developer must explicitly define the default constructor if they want to use it.

### Static Constructor

A static constructor is first called when the runtime first accesses the class. Static variables are accessible at all times, so the runtime must initialize it on its first access. The example below, when stepping through in a debugger, will show that *static MyClass()* is only accessed when the *MyClass.Number* variable is accessed.

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

namespace StaticConstructors
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            int j = 0;
            Console.WriteLine("Static Number = " + MyClass.Number);
        }
    }

    class MyClass
    {
        private static int number;
        public static int Number { get { return number; } }
        static MyClass()
        {
            Random r = new Random();
            number = r.Next();
        }
    }
}
```

### Default Constructor

The default constructor takes no parameters and is implicitly defined if no other constructors exist. The code sample below show the before, and after result of creating a class.

```
//Created by the developer
```

```
class MyClass
{
}

//Created by the compiler
class MyClass : System.Object
{
    public MyClass() : base()
    {
    }
}
```

## Overloaded Constructors

To initialize objects in various forms, the constructors allow customization of the object by passing in parameters.

```
class MyClass
{
    private int number;
    public int Number { get { return number; } }

    public MyClass()
    {
        Random r = new Random();
        number = r.Next();
    }

    public MyClass(int seed)
    {
        Random r = new Random(seed);
        number = r.Next();
    }
}
```

## Calling other constructors

To minimise code, if another constructor implements the functionality better, you can instruct the constructor to call an overloaded (or default) constructor with specific parameters.

```
class MyClass
```

```
{
    private int number;
    public int Number { get { return number; } }

    public MyClass() :
        this ( DateTime.Now.Millisecond ) //Call the other constructor
        passing in a value.
    {
    }

    public MyClass(int seed)
    {
        Random r = new Random(seed);
        number = r.Next();
    }
}
```

Base classes constructors can also be called instead of constructors in the current instance

```
class MyException : Exception
{
    private int number;
    public int Number { get { return number; } }

    public MyException ( int errorNumber, string message, Exception innerException ) : base( message, innerException )
    {
        number = errorNumber;
    }
}
```

## Destructors

As well as being ""constructed"", objects can also perform cleanup when they are cleared up by the garbage collector. The garbage collector only runs when either directly invoked, or has reason to reclaim memory, therefore the destructor may not get the change to clean up resources for a long time. In this case, look into use of the Dispose() method, from the IDisposable interface.



Destructors are recognised via the use of the ~ symbol in front of a constructor with no access modifier e.g.

```
class MyException : Exception
{
    private int number;
    public int Number { get { return number; } }

    public MyException ( int errorNumber, string message, Exception innerException ) : base( message, innerException )
    {
        number = errorNumber;
    }

    ~MyException()
    {
    }
}
```

## Abstract Class

An abstract class is a class that was never intended to be instantiated directly, but only to serve as a base to other classes. An abstract class must/should contain at least one abstract method that child concrete classes are obliged to implement. A class should be made abstract if it makes no sense to create instances of that class. For example, an Employee can be an abstract class if there are concrete classes that represent all kinds of employees. It is never appropriate to instantiate it, but it does contain behavior (abstract functions and code) common to all employees.

## Sub-heading



*The C Sharp Programming/Objects module or this section of [C Sharp Programming](#) is a **stub**.*


*You can help Wikibooks by [expanding it](#).*

# Encapsulation

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



 C sharp musical note

Encapsulation is depriving of the user of a class information he does not need, and preventing him from manipulating objects in ways not intended by the designer.

A class element having *public protection level* is accessible to all code anywhere in the program. These methods and properties represent the operations allowed on the class to outside users.

Methods, data members (and other elements) with *private protection level* represent the internal state of the class (for variables), and operations which are not allowed to outside users.

For example:

```
public class Frog
{
    public void JumpLow() { Jump(1); }
    public void JumpHigh() { Jump(10); }

    private void Jump(int height) { _height += height; }

    private int _height = 0;
}
```

In this example, the public method the Frog class exposes are JumpLow and JumpHigh. Internally, they are implemented using the private Jump function that can jump to any height. This operation is not visible to an outside user, so he cannot make the frog jump 100 meters, only 10

or 1. The Jump private method is implemented by changing the value of a private data member `_height`, which is also not visible to an outside user. Some private data members are made visible by [C Sharp Programming/Properties](#).

## Protection Levels

### Private

Private members are only accessible within the class itself. A method in another class, even a class derived from the class with private members cannot access the members.

### Protected

Protected members can be accessed by the class itself and by any class derived from that class.

### Public

Public members can be accessed by any method in any class.

### Internal

Internal members are accessible only in the same assembly and invisible outside it.

# NET Framework overview

The .NET Framework is a common environment for building, deploying, and running Web Services, Web Applications, Windows Services and Windows Applications. The .NET Framework contains common class libraries - like ADO.NET, ASP.NET and Windows Forms - to provide advanced standard services that can be integrated into a variety of computer systems.

## Introduction

C# is a language in itself. It can perform mathematical and logical operation, variable assignment and other expected traits of a programming language. This in itself is not flexible enough for more complex applications. At some stage, the developer will want to interact with the host system whether it be reading files or downloading content from the internet.

The .NET framework is a toolset developed for the Windows platform to allow the developer to interact with the host system or any external entity whether it be another process, or another computer. The .NET platform is a Windows platform specific implementation. Other operating systems have their own implementations due to the differences in the operating systems I/O management, security models and interfaces.

## Background

- .NET was originally called NGWS(Next Generation Windows Services).
- .NET does not run IN any browser. It is a RUNTIME language (Common Language Runtime) like the Java runtime. [Silverlight](#) does run in a browser, but has nothing to do with .NET.
- .NET is based on the newest Web standards.
- .NET is built on the following Internet standards
  - HTTP, the communication protocol between Internet Applications
  - XML, the format for exchanging data between Internet Applications
  - SOAP, the standard format for requesting Web Services
  - UDDI, the standard to search and discover Web Services

# Inheritance

## Inheritance

Inheritance is the ability to create a class from another class, the "parent" class, extending the functionality and state of the parent in the derived, or "child", class.

Inheritance in C# also allows derived classes to overload methods from their parent class.

Inheritance(by Mine)

## Subtyping Inheritance

The code sample below shows two classes, Employee and Executive. Employee has the following methods, GetPayCheck and Work.

We want the Executive class to have the same methods, but differently implemented and one extra method, AdministerEmployee.

Below is the creation of the first class to be derived from, Employee.

```
public class Employee
{
    // we declare one method virtual so that the Executive class can
    // override it.
    public virtual void GetPayCheck()
    {
        //get paycheck logic here.
    }

    //Employee's and Executives both work, so no virtual here needed.
    public void Work()
    {
        //do work logic here.
    }
}
```

```
}  
}
```

Now, we create an Executive class that will override the GetPayCheck method.

```
public class Executive : Employee  
{  
    // the override keyword indicates we want new logic behind the GetPay-  
    Check method.  
    public override void GetPayCheck()  
    {  
        //new getpaycheck logic here.  
    }  
  
    // the extra method is implemented.  
    public void AdministerEmployee()  
    {  
        // manage employee logic here  
    }  
}
```

You'll notice that there is no Work method in the Executive class, it is not required, since that method is automatically added to the Executive class, because it derives its methods from Employee, which has the Work method.

```
static void Main(string[] args)  
{  
    Employee emp = new Employee;  
    Executive exec = new Executive;  
  
    emp.Work();  
    exec.Work();  
    emp.GetPayCheck();  
    exec.GetPayCheck();  
    exec.AdministerEmployee();  
}
```

# Inheritance keywords

How C# inherits from another class syntactically is using the ":" character.

Example. `public class Executive : Employee`

To indicate a method that can be overridden, you mark the method with `virtual`.

```
public virtual void Write(string text)
{
    System.Console.WriteLine("Text:{0}", text);
}
```

To override a method use the `override` keyword

```
public override void Write(string text)
{
    System.Console.WriteLine(text);
}
```



# Interfaces

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

An interface in C# is type definition similar to a class except that it purely represents a contract between an object and a user of the object. An interface cannot be directly instantiated as an object. No data members can be defined in an interface. Methods and properties can only be declared, not defined. For example, the following defines a simple interface:

```
interface IShape
{
    void Draw();
    double X { get; set; }
    double Y { get; set; }
}
```

A convention used in the .NET Framework (and likewise by many C# programmers) is to place an "I" at the beginning of an interface name to distinguish it from a class name. Another common interface naming convention is used when an interface declares only one key method, such `Draw()` in the above example. The interface name is then formed by adding the suffix "able" to the method name. So, in the above example, the interface name would be `IDrawable`. This convention is also used throughout the .NET Framework.

Implementing an interface is simply done by inheriting off the interface and then defining all the methods and properties declared by the interface. For example:

```
class Square : IShape
{
    private double mX, mY;
```

## C Sharp Programming

---

```
public void Draw() { ... }

public double X
{
    set { mX = value; }
    get { return mX; }
}

public double Y
{
    set { mY = value; }
    get { return mY; }
}
}
```

Although a class can only inherit from one other class, it can inherit from any number of interfaces. This is simplified form of multiple inheritance supported by C#. When inheriting from a class and one or more interfaces, the base class should be provided first in the inheritance list followed by any interfaces to be implemented. For example:

```
class MyClass : Class1, Interface1, Interface2 { ... }
```

Object references can be declared using an interface type. For example, using the previous examples:

```
class MyClass
{
    static void Main()
    {
        IShape shape = new Square();
        shape.Draw();
    }
}
```

Interfaces can inherit off of any number of other interfaces but cannot inherit from classes. For example:

```
interface IRotateable
```

```
{  
    void Rotate(double theta);  
}
```

```
interface IDrawable : IRotateable  
{  
    void Draw();  
}
```

## Additional Details

Access specifiers (i.e. private, internal, etc) cannot be provided for interface members. All members are public. A class implementing an interface must define all the members declared by the interface as public. The implementing class has the option of making an implemented method virtual if it is expected to be overridden in a child class.

In addition to methods and properties, interfaces can declare events and indexers as well.

# Delegates and Events

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

## Introduction

Events and delegates are fundamental to any Windows or Web Application. These allow the developer to "subscribe" to particular actions carried out by the user. Therefore instead of expecting everything and filtering out what you want, you choose what you want to be notified of and react to that action.

A delegate is a way of telling C# which method to call when an event is triggered. For example, if you click a Button on a form, the program would call a specific method. It is this pointer which is a delegate. Delegates are good because you can notify several methods that an event has occurred, if you so wish.

An event is a notification by the .NET framework that an action has occurred. Each event contains information about the specific event, e.g., a mouse click would say which mouse button was clicked and where on the form it was clicked.

Lets say you write a program that *only reacts to a Button click*, here is the sequence of events that occurs:

- User presses the mouse down on a button
  - The .NET framework raises a MouseDown event
- User releases the mouse button
  - The .NET framework raises a MouseUp event
  - The .NET framework raises a MouseClick event
  - The .NET framework raises a Clicked event on the Button

Since the button's click event has been subscribed, the rest of the events are ignored by the program and your *delegate* tells the .NET framework which method to call, now that the event has been raised.

## Delegates

Delegates are a construct for abstracting and creating objects that reference methods and can be used to call those methods. Delegates form the basis of event handling in C#. A delegate declaration specifies a particular method signature. References to one or more methods can be added to a delegate instance. The delegate instance can then be "called" which effectively calls all the methods that have been added to the delegate instance. A simple example:

```
delegate void Procedure();

class DelegateDemo
{
    static void Method1()
    {
        Console.WriteLine("Method 1");
    }

    static void Method2()
    {
        Console.WriteLine("Method 2");
    }

    void Method3()
    {
        Console.WriteLine("Method 3");
    }

    static void Main()
```

```
{
    Procedure someProcs = null;
    someProcs += new Procedure(DelegateDemo.Method1);
    someProcs += new Procedure(DelegateDemo.Method2);
    DelegateDemo demo = new DelegateDemo();
    someProcs += new Procedure(demo.Method3);
    someProcs();
}
}
```

In this example, the delegate is declared by the line `delegate void Procedure();` This statement is a complete abstraction. It does not result in executable code that does any work. It merely declares a delegate type called `Procedure` which takes no arguments and returns nothing. Next, in the `Main()` method, the statement `Procedure someProcs = null;` instantiates a delegate. Something concrete has now been created. The assignment of `null` to `someProcs` means that the delegate is not initially referencing any methods. The statements `someProcs += new Procedure(DelegateDemo.Method1);` and `someProcs += new Procedure(DelegateDemo.Method2);` add two static methods to the delegate instance. (Note: the class name could have been left off of `DelegateDemo.Method1` and `DelegateDemo.Method2` because the statement is occurring in the `DelegateDemo` class.) The statement `someProcs += new Procedure(demo.Method3);` adds a non-static method to the delegate instance. For a non-static method, the method name is preceded by an object reference. When the delegate instance is called, `Method3()` is called on the object that was supplied when the method was added to the delegate instance. Finally, the statement `someProcs();` calls the delegate instance. All the methods that were added to the delegate instance are now called in the order that they were added.

Methods that have been added to a delegate instance can be removed with the `-=` operator:

```
someProcess -= new Procedure(DelegateDemo.Method1);
```

In C# 2.0, adding or removing a method reference to a delegate instance can be shortened as follows:

```
someProcess += DelegateDemo.Method1;
someProcess -= DelegateDemo.Method1;
```

Invoking a delegate instance that presently contains no method references results in a `NullReferenceException`.

Note that if a delegate declaration specifies a return type and multiple methods are added to a delegate instance, then an invocation of the delegate instance returns the return value of the last method referenced. The return values of the other methods cannot be retrieved (unless explicitly stored somewhere in addition to being returned).

## Events

An event is a special kind of delegate that facilitates event-driven programming. Events are class members which cannot be called outside of the class regardless of its access specifier. So, for example, an event declared to be public would allow other classes the use of `+=` and `-=` on the event, but firing the event (i.e. invoking the delegate) is only allowed in the class containing the event. A simple example:

```
delegate void ButtonClickedHandler();

class Button
{
    public event ButtonClickedHandler ButtonClicked;

    public void SimulateClick()
    {
        if (ButtonClicked != null)
        {
            ButtonClicked();
        }
    }

    ...
}
```

A method in another class can then subscribe to the event by adding one of its methods to the event delegate:

```
Button b = new Button();
```

## C Sharp Programming

---

```
b.ButtonClicked += MyHandler;
```

Even though the event is declared public, it cannot be directly fired anywhere except in the class containing the event.



# Abstract classes

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

In general terms, an interface is the set of public members of a component. Of course, a [C# interface](#) is an interface that defines a set of public members. A [C# class](#) also defines an interface because it has a set of public members. A nonabstract [C# class](#) also defines the implementation of each member.

In [C#](#) it is possible to have a type that is intermediate between a pure interface that does not define any implementation, and a type that defines a complete implementation. This is called an *abstract class*.

You define an abstract class by including the [abstract](#) keyword on the class definition.

An abstract class is somewhere between a [C# interface](#) and a nonabstract class. Of the public members defined by an abstract class, any number of those members may include an implementation.

For example, an abstract class might provide an implementation for *none* of its members.

```
public abstract class AbstractShape
{
    public abstract void Draw(Graphics g);
    public abstract double X {get; set;}
    public abstract double Y {get; set;}
}
```

## C Sharp Programming

---

This class is equivalent to an interface in many respects. (One difference is that a class that derives from this class cannot derive from any other class.)

An abstract class may also define *all* of its members.

```
public abstract class AbstractShape
{
    private double x;
    private double y;
    //
    // ... (other members)
    //
    public void Draw(Graphics g) {g.DrawRectangle(Pens.Black, g_rect);}
    public double X {get{return x;}}
    public double Y {get{return y;}}
}
```

And an abstract class may define some of its members but leave others undefined.

```
public abstract class AbstractShape
{
    private double x;
    private double y;
    //
    // ... (other members)
    //
    public abstract void Draw(Graphics g);
    public double X {get{return x;}}
    public double Y {get{return y;}}
}
```

An abstract class is similar to a nonabstract class, but there are some important differences.

For one thing, you cannot create an instance of an abstract class with the `new` keyword. For example, the following statement will raise a compiler error:

```
AbstractShape shape = new AbstractShape();
```

Of course, assuming the concrete class `Square` derives from `AbstractShape`, the following would be correct:

```
AbstractShape shape = new Square();
```

A second difference is that an abstract class can contain abstract members. As was shown above, it does not *have* to contain abstract members. The point is that a nonabstract class may *not* contain abstract members. That is, you must include the `abstract` keyword on the class if you include even one abstract member.

The third difference is that an abstract class cannot be sealed. That is, you cannot use both the `abstract` keyword and the `sealed` keyword on the same class.

# Partial classes

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

## Partial Classes

As the name indicates, partial class definitions can be split up across multiple physical files. To the compiler, this does not make a difference as all the fragments of the partial class are grouped and the compiler treats it as a single class. One common usage of partial classes is the separation of automatically generated code from programmer written code.

Below is the example of a partial class.

**Listing 1:** Entire class definition in one file (file1.cs)

```
public class Node
{
    public bool Delete()
    {
    }

    public bool Create()
    {
    }
}
```

**Listing 2:** Class split across multiple files

(file1.cs)

```
public partial class Node
{
    public bool Delete()
    {
    }
}
```

(file2.cs)

```
public partial class Node
{
    public bool Create()
    {
    }
}
```

# Collections

## Lists

It is one type of collection A list is a dynamic array which resizes itself as needed if more data is inserted than it can hold at the time of insertion. Items can be inserted at any index, deleted at any index and accessed at any index. The C# non-generic list class is the `ArrayList`, while the generic one is `List<T>`.

## LinkedLists

Items in a linked list can be accessed directly only one after the other. Of course an item at any index can be accessed, but the list must iterate to the item from the first one, which is much slower than accessing items by index in an array or a list. There is no non-generic list in C#, while the generic one is `LinkedList<T>`.

## Queues

A queue is a FIFO (first in - first out) collection. The item first pushed in the queue gets taken first with the pop function. Only the first item is accessible at any time, and items can only be put to the end. The non-generic queue class is called `Queue`, while the generic one is `Queue<T>`.

## Stacks

A stack is a LIFO (last in - first out) collection. The item pushed in first will be the last to be taken by pop. Only the last item is accessible at any time, and items can only be put at the top. The non-generic stack class is `Stack`, while the generic one is `Stack<T>`.

## Dictionaries

A dictionary is a collection of values with keys. The values can be very complex, yet searching the keys is still fast. The non-generic class is `Hashtable`, while the generic one is `Dictionary<T>`.

# Generics

Generics is essentially the ability to have type parameters on your type. They are also called parameterized types or parametric polymorphism. The classic example is a List collection class. A List is a convenient growable array. It has a sort method, you can index into it, and so on.

## Generic Interfaces

[MSDN2 Entry for Generic Interfaces](#)

## Generic Classes

There are cases when you need to create a class to manage objects of some type, without modifying them. Without Generics, the usual approach (highly simplified) to make such class would be like this:

```
public class SomeObjectContainer
{
    private object obj;

    public SomeObjectContainer(object obj)
    {
        this.obj = obj;
    }

    public object getObject()
    {
        return this.obj;
    }
}
```

And its usage would be:



```
class Program
{
    static void Main(string[] args)
    {
        SomeObjectContainer container = new SomeObjectContainer(25);
        SomeObjectContainer container2 = new SomeObjectContainer(5);
        Console.WriteLine((int)container.GetObject() + (int)container2.GetObject());

        Console.ReadKey(); // wait for user to press any key, so we could see results
    }
}
```

Notice that we have to cast back to original data type we have chosen (in this case - `int`) every time we want to get an object from such a container. In such small programs like this everything is clear. But in more complicated cases with more containers in different parts of the program, we would have to take care that the container is supposed to have `int` type in it, would not have a `string` or any other data type. If that happens, `InvalidCastException` is thrown.

Additionally, if the original data type we have chosen is a `struct` type, such as `int`, we will incur a performance penalty every time we access the elements of the collection, due to the Auto-boxing feature of C#.

However, we could surround every unsafe area with `try - catch` block, or we could create a separate "container" for every data type we need, just to avoid casting. While both ways could work (and worked for many years), it is unnecessary now, because Generics offers a much more elegant solution.

To make our "container" class to support any object and avoid casting, we replace every previous `object` type with some new name, in this case - `T`, and add `<T>` mark immediately after the class name to indicate that this "T" type is Generic / any type.

Note: You can choose any name and use more than one generic type for class, i.e `<genKey, genVal>`

```
public class GenericObjectContainer<T>
{
    private T obj;

    public GenericObjectContainer(T obj)
    {
        this.obj = obj;
    }
}
```

## C Sharp Programming

---

```
}

public T getObject()
{
    return this.obj;
}
}
```

Not a big difference, which results in simple and safe usage:

```
class Program
{
    static void Main(string[] args)
    {
        GenericObjectContainer<int> container = new GenericObjectContain-
er<int>(25);
        GenericObjectContainer<int> container2 = new GenericObjectContain-
er<int>(5);
        Console.WriteLine(container.getObject() + container2.getObject());

        Console.ReadKey(); // wait for user to press any key, so we could see
results
    }
}
```

Generics ensures that you specify the type for a "container" only when creating it, and after that you will be able to use only the type you specified. But now you can create containers for different object types, and avoid the previously mentioned problems. In addition, this avoids the Autoboxing for `struct` types.

While this example is far from practical, it does illustrate some situations where generics are useful:

- You need to keep objects of a single type in a class
- You don't need to modify objects
- You need to manipulate objects in some way
- You wish to store a "value type" (such as `int`, `short`, `string`, or any custom `struct`) in a collection class without incurring the performance penalty of Autoboxing every time you manipulate the stored elements.

## Generic lists

A generic list is an indexed list, so any of its items can be directly accessed by its index.

Many of its methods and properties are demonstrated in the following example:

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

namespace csharp_generic_list
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("List<T> demo");
            // creating an instance which accepts strings
            List<string> foods = new List<string>();

            // adding some items one by one with Add()
            foods.Add("bread");
            foods.Add("butter");
            foods.Add("chocolate");

            // adding a simple string array with AddRange()
            string[] subList1 = {"orange", "apple", "strawberry", "grapes",
"kiwi", "banana"};
            foods.AddRange(subList1);

            // adding another List<string> with AddRange()
            List<string> anotherFoodList = new List<string>();
            anotherFoodList.Add("yoghurt");
            anotherFoodList.Add("tomato");
            anotherFoodList.Add("roast beef");
            anotherFoodList.Add("vanilla cake");
            foods.AddRange(anotherFoodList);

            // removing "orange" with Remove()
            foods.Remove("orange");

            // removing the 5th (index = 4) item ("strawberry") with Re-
moveAt()
            foods.RemoveAt(4);

            // removing a range (4-7: all fruits) with RemoveRange(int index,
```

## C Sharp Programming

---

```
int count)
    foods.RemoveRange(3, 4);

    // sorting the list
    foods.Sort();

    // printing the sorted foods
    foreach (string item in foods)
    {
        Console.WriteLine("| " + item + " ");
    }
    Console.WriteLine("|");

    // removing all items from foods
    foods.Clear();

    // printing the current item count in foods
    Console.WriteLine("The list now has {0} items.", foods.Count);
}
}
```

The terminal output is:

```
List<T> demo
| bread | butter | chocolate | roast beef | tomato | vanilla cake | yoghurt
|
The list now has 0 items.
```

Generic linked lists

Generic queues

Generic stacks

Generic dictionaries

Generic Methods

Generic Delegates

Generic Events

# Object Lifetime

## Introduction

All computer programs use up memory, whether that is a variable in memory, opening a file or connecting to a database. The question is how can the runtime environment reclaim any memory when it is not being used? There are 3 answers to this question:

- If you are using a *managed* resource, this is automatically released by the Garbage Collector
- If you are using an *unmanaged* resource, you must use the `IDisposable` interface to assist with the cleanup
- If you are calling the Garbage Collector directly, by using `System.GC.Collect()` method, it will be forced to tidy up resources immediately.

Before discussing managed and unmanaged resources, it would be interesting to know what the garbage collector actually does.

## Garbage Collector

The garbage collector is a background process running within your program. It is always present within all .NET applications. Its job is to look for objects (i.e. reference types) which are no longer being used by your program. If the object is assigned to null, or the object goes out of scope, the garbage collector will mark the object to be cleaned up at some point in the future, and not necessarily have its resources released immediately!

Why? The garbage collector will have a hard time keeping up with every de-allocation you make, especially at the speed the program runs and therefore only runs when resources become limited. Therefore, the garbage collector has 3 "generations".

- Generation 0 - the most recently created objects
- Generation 1 - the mid-life objects
- Generation 2 - the long term objects.

All reference types will exist in one of these 3 generations. They will firstly be allocated to Gen 0, then moved to Gen 1 and Gen 2 depending on their lifetime. The garbage collector works by removing only what is needed and so will only scan Gen 0 for a quick-fix solution. This is because most if not all local variables are placed in this area.

For more in-depth information, visit the [MSDN Article](#) for a better explanation.

Now you know about the garbage collector, lets discuss the resources that it is managing.

## Managed Resources

Managed resources are objects which run totally within the .NET framework. All memory is reclaimed for you automatically, all resources closed and you are in most cases /guaranteed/ to have all the memory released after the application closes, or when the garbage collector runs.

You do not have to do anything with them with regards to closing connections or anything, it is a self-tidying object.

## Unmanaged Resources

There are circumstances where the .NET framework world will not release resources. This may be because the object references resources outside of the .NET framework, like the operating system, or internally references another unmanaged component, or that the resources accesses a component that uses COM, COM+ or DCOM.

Whatever the reason, if you are using an object that implements the IDisposable interface at a class level, then you too need to implement the IDisposable interface too.

```
public interface IDisposable
{
```

```
public void Dispose();  
}
```

This interface exposes a method called `Dispose()`. This alone will *not* help tidy up resources, as it is only an interface, so the developer must use it correctly in order to ensure the resources are released. The two steps are:

1. Always call `Dispose()` on any object that implements `IDisposable` as soon as you are finished using it. (This can be made easier with the `using` keyword)
2. Use the finalizer method to call `Dispose()`, so that if anyone has not closed your resources, your code will do it for them.

See the `IDisposable` section for a full implementation.

## Applications

If you are coming to C# from [Visual Basic Classic](#) you will have seen code like this:

```
Public Function Read(ByRef FileName) As String  
  
    Dim oFSO As FileSystemObject  
    Set oFSO = New FileSystemObject  
  
    Dim oFile As TextStream  
    Set oFile = oFSO.OpenTextFile(FileName, ForReading, False)  
    Read = oFile.ReadLine  
  
End Function
```

Note that neither `oFSO` nor `oFile` are explicitly disposed of. In Visual Basic Classic this is not necessary because both objects are declared locally. This means that the reference count goes to zero as soon as the function ends which results in calls to the *Terminate* event handlers of both objects. Those event handlers close the file and release the associated resources.

In C# this doesn't happen because the objects are not reference counted. The finalizers will not be called until the garbage collector decides to dispose of the objects. If the program uses very little memory this could be a long time.



This causes a problem because the file is held open which might prevent other processes from accessing it.

In many languages the solution is to explicitly close the file and dispose of the objects and many C# programmers do just that. However, there is a better way: use the *using* statement:

```
public read(string fileName)
{
    using (TextReader textReader = new StreamReader(filename))
    {
        return textReader.ReadLine();
    }
}
```

Behind the scenes the compiler turns the using statement into *try..finally* and produces this intermediate language (IL) code:

```
.method public hidebysig static string Read(string FileName) cil managed
{
    // Code size          39 (0x27)
    .maxstack 5
    .locals init (class [mscorlib]System.IO.TextReader V_0,
                 string V_1)
    IL_0000: ldarg.0
    IL_0001: newobj      instance void [mscorlib]System.IO.StreamReader::.ctor(string)
    IL_0006: stloc.0
    .try
    {
        IL_0007: ldloc.0
        IL_0008: callvirt   instance string [mscorlib]System.IO.TextReader::ReadLine()
        IL_000d: stloc.1
        IL_000e: leave     IL_0025
        IL_0013: leave     IL_0025
    } // end .try
    finally
    {
        IL_0018: ldloc.0
        IL_0019: brfalse  IL_0024
        IL_001e: ldloc.0
        IL_001f: callvirt   instance void [mscorlib]System.IDisposable::Dispose()
        IL_0024: endfinally
    }
}
```

```
} // end handler
IL_0025: ldloc.1
IL_0026: ret
} // end of method Using::Read
```

Notice that the body of the *Read* function has been split into three parts: initialisation, try, and finally. The *finally* block includes code that was never explicitly specified in the original C# source code, namely a call to the *destructor* of the *StreamReader* instance.

See [Understanding the 'using' statement in C# By TiNgZ aBrAhAm](#).

See the following sections for more applications of this technique.

## Resource Acquisition Is Initialisation

The application of the *using* statement in the introduction is an example of an idiom called *Resource Acquisition Is Initialisation* (RAII).

RAII is a natural technique in languages like Visual Basic Classic and C++ that have deterministic finalization but usually requires extra work to include in programs written in garbage collected languages like C# and VB.NET. The *using* statement makes it just as easy. Of course you could write the *try..finally* code out explicitly and in some cases that will still be necessary. For a thorough discussion of the RAII technique see [HackCraft: The RAII Programming Idiom](#). Wikipedia has a brief note on the subject as well: [Resource Acquisition Is Initialization](#).

Work in progress: add C# versions showing incorrect and correct methods with and without using. Add notes on RAII, memoization and cacheing (see OOP wikibook).



*The C Sharp Programming/Object Lifetime module or this section of C Sharp Programming is a **stub**.*

*You can help Wikibooks by [expanding it](#).*

# Design Patterns

Design Patterns are common building blocks designed to solve everyday software issues. Some basic terms and example of such patterns include what we see in everyday life. Key patterns are the singleton pattern, the factory pattern, and chain of responsibility patterns.

## Table Of Contents (TOC)

### Factory Pattern

The factory pattern is a method call that uses abstract classes and its implementations, to give the developer the most appropriate class for the job.

Lets create a couple of classes first to demonstrate how this can be used. Here we take the example of a bank system.

```
public abstract Transaction
{
    private string _sourceAccount;

    //May not be needed in most cases, but may on transfers, closures and
    corrections.
    private string _destinationAccount;

    private decimal _amount;
    public decimal Amount { get { return _amount; } }

    private DateTime _transactionDate;
    private DateTime _effectiveDate;

    public Transaction(string source, string destination, decimal amount)
    {
        _sourceAccount = source;
        _destinationAccount = destination;
        _amount = amount;
    }
}
```

## C Sharp Programming

---

```
        _transactionDate = DateTime.Now;
    }

    public Transaction(string source, string destination, decimal amount,
DateTime effectiveDate) : this(source, destination, amount)
    {
        _effectiveDate = effectiveDate;
    }

    protected decimal AdjustBalance(string accountNumber, decimal amount)
    {
        decimal newBalance = decimal.MinValue;

        using(Mainframe.ICOMInterface mf = new Mainframe.COMInterface-
Class())
        {
            string dateFormat = DateTime.Now.ToString("yyyyMMdd HH:mm:ss");

            mf.Credit(dateFormat, accountNumber, amount);
            newBalance = mf.GetBalance( DateTime.Now.AddSeconds(1), account-
Number);
        }

        return newBalance;
    }

    public abstract bool Complete();
}
```

This Transaction class is incomplete, as there are many types of transactions:

- Opening
- Credits
- Withdrawals
- Transfers
- Penalty
- Correction
- Closure

For this example, we will take credit and withdrawal portions, and create classes for them.

```
public Credit : Transaction
{
    //Implementations hidden for simplicity
```

```
public bool Complete()
{
    this.AdjustBalance( _sourceAccount, amount);
}
}

public Withdrawal : Transaction
{
    //Implementations hidden for simplicity

    public bool Complete()
    {
        this.AdjustBalance( _sourceAccount, -amount);
    }
}
```

The problem is that these classes do much of the same thing, so it would be helpful if we could just give it the values, and it will work out what class type we require. Therefore, we could come up with some ways to distinguish between the different types of transactions:

- Positive values indicate a credit.
- Negative values indicate a withdrawal.
- Having 2 account numbers and a positive value would indicate a transfer.
- Having 2 account numbers and a negative value would indicate a closure.
- etc

So, let us write a new class with a static method that will do this logic for us, ending the name Factory:

```
public class TransactionFactory
{
    public static Transaction Create( string source, string destination,
    decimal amount )
    {
        if( string.IsNullOrEmpty(destination) )
        {
            if(amount >= 0)
                return new Credit( source, null, amount);
            else
                return new Withdrawal( source, null, amount);
        }
        else
        {
            //Other implementations here
        }
    }
}
```

## C Sharp Programming

---

```
}  
}
```

Now, you can use this class to do all of the logic and processing, and be assured that the type you are returned is correct.

```
public class MyProgram  
{  
    static void Main()  
    {  
        decimal randomAmount = new Random().Next() * 1000000;  
        Transaction t = TransactionFactory.Create("123456", "", randomAmount);  
  
        //t.Complete(); <-- This would carry out the requested transaction.  
  
        Console.WriteLine("{0}: {1:C}", t.GetType().Name, t.Amount);  
    }  
}
```

## Singleton

The singleton pattern instantiates only 1 object, and reuses this object for the entire lifetime of the process. This is useful if you wish the object to maintain state, or if it takes lots of resources to set the object up. Below is a basic implementation:

```
public class MySingletonExample  
{  
    private Hashtable sharedHt;  
  
    public Hashtable Singleton  
    {  
        get  
        {  
            if(sharedHt == null)  
                sharedHt = new Hashtable();  
  
            return sharedHt;  
        }  
  
        set { ; } //Not implemented as this would invalidate the pattern  
    }  
}
```

```
}  
    //Class implementation here..  
}
```

The Singleton property will expose the same instance to all callers. Upon the first call, the object is initialised and on subsequent calls this is used.

Examples of this pattern include:

- `HttpApplication` (Application object in ASP .NET)
- `HttpServerUtility` (Server object in ASP .NET)
- `HttpCacheUtility` (Cache object in ASP .NET)
- `ConfigurationSettings` (Generic settings reader)

# abstract

Abstract Classes are those which contain only the declaration of other classes and methods. Those methods which can be defined inside the class are defined except for which we can't create the definition. Those methods which are declared in a ABSTRACT Class can be defined outside of the ABSTRACT Class as a individual method.



# as

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **as** keyword casts an object to a different type. It is therefore similar to the `TypeA varA = (TypeA) varB` syntax. The difference is that this keyword returns null if the object was of an incompatible type, while the former method throws a type-cast exception in that case.

## See also

- [is](#)

# base

The keyword **base** describes that you would like to refer to the base class for the requested information, not in the current instantiated class.

A *base* class is the class in which the currently implemented class inherits from. When creating a class with no defined base class, the compiler automatically uses the System.Object base class.

Therefore the 2 declarations below are equivalent.

```
public class MyClass
{
}

public class MyClass : System.Object
{
}
```

Some of the reasons the base keyword is used is:

- Passing information to the base class's constructor

```
public class MyCustomException : System.Exception
{
    public MyCustomException() : base() { }

    public MyCustomerException(string message, Exception innerException) :
base(message,innerException) { }

    //.....
}
```

- Recalling variables in the base class, where the newly implemented class is overriding its behaviour

```
public class MyBaseClass
{
    protected string className = "MyBaseClass";
}

public class MyNewClass : MyBaseClass
{
    protected new string className = "MyNewClass";

    public override string BaseClassName
    {
        get { return base.className; }
    }
}
```

- Recalling methods in the base class. This is useful when you want to add to a method, but still keep the underlying implementation.

```
//Necessary using's here

public class _Default : System.Web.UI.Page
{
    protected void InitializeCulture()
    {
        System.Threading.Thread.CurrentThread.CurrentUICulture = CultureIn-
        fo.GetSpecificCulture(Page.UICulture);

        base.InitializeCulture();
    }
}
```

# bool

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **bool** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure System.Boolean. That is, it represents a value of *true* or *false*. Unlike in C++, whose *boolean* is actually an *integer*, a *bool* in C# is its own data type and cannot be cast to any other primitive type.

# break

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The keyword **break** is used to exit out of a loop.

```
int x;

while( x < 20 ){

    if( x > 10 ) break;

    x++;

}
```

The while loop would increment x as long as it was less than twenty. However when x is incremented to ten the condition in the if statement becomes true, so the break statement causes the while loop to be broken and execution would continue after the closing parentheses.

# byte

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **byte** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Byte`. That is, it represents an 8-bit unsigned integer whose value ranges from 0 to 255.

# case

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The keyword **case** is often used in a [switch](#) statement.

# catch

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The keyword **catch** is used to identify a *statement* or *statement block* for execution if an exception occurs in the body of the enclosing **try** block. The catch clause may optionally be followed by a **finally** clause.



# char

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **char** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Char`. That is, it represents a Unicode character whose from 0 to 65,535.

# class

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **class** keyword is used to declare a *class*.

# const

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **const** keyword is used in field and local variable declarations to make the variable *constant*. It is thus associated with its declaring class or assembly instead of with an instance of the class or with a method call. It is syntactically invalid to assign a value to such a variable anywhere other than its declaration.

# continue

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The keyword **continue** can be used inside any loop in a method. Its affect is to end the current loop iteration and proceed to the next one. If executed inside a for, end-of-loop statement is executed (just like normal loop termination).

# decimal

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **decimal** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Decimal`. That is, it represents a signed, 128-bit decimal number whose value is 0 or a decimal number with 28 or 29 digits of precision ranging either from  $-1.0 \times 10^{-28}$  to  $-7.9 \times 10^{28}$  or from  $1.0 \times 10^{-28}$  to  $7.9 \times 10^{28}$ .

# default

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **default** keyword can be used in the switch statement or in generic code:

- **The switch statement:** Specifies the default label.
- **Generic code:** Specifies the default value of the type parameter. This will be null for reference types and zero for value types.

Note\* From [MSDN](#)

# delegate

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **delegate** keyword is used to declare a *delegate*. A delegate is a programming construct that is used to obtain a callable reference to a method of a class.

# do

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **do** keyword identifies the beginning of a **do...while** loop.



# double

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **double** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure System.Double. That is, it represents an IEEE 754, 64-bit signed binary floating point number whose value is *negative 0*, *positive 0*, *negative infinity*, *positive infinity*, *not a number*, or a number ranging either from  $-5.0 \times 10^{-324}$  to  $-1.79 \times 10^{308}$  or from  $5.0 \times 10^{-324}$  to  $1.79 \times 10^{308}$ .

# else

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **else** keyword identifies a **else clause** of an if statement with the following syntax:

*if-statement ::= "if" "(" condition ")" if-body "else" else-body*

*condition ::= boolean-expression*

*if-body ::= statement-or-statement-block*

*else-body ::= statement-or-statement-block*

An else clause immediately follows an *if-body*. It provides code to execute when the *condition* is *false*. Making the *else-body* another if statement creates the common *cascade* of if, else if, else if, else if, else if, else statements:

```
using System;

public class IfStatementSample
{
    public void IfMyNumberIs()
    {
        int myNumber = 5;
        if ( myNumber == 4 )
            Console.WriteLine("This will not be shown because myNumber is not
4.");
        if( myNumber < 0 )
        {
            Console.WriteLine("This will not be shown because myNumber is not
negative.");
        }
        if( myNumber % 2 == 0 )
            Console.WriteLine("This will not be shown because myNumber is not
```

```
even.");  
  
    {  
        Console.WriteLine("myNumber does not match the coded conditions,  
so this sentence will be shown!");  
    }  
}  
}
```

The above example only checks whether myNumber is less than 0 if myNumber is not 4. It in turn only checks whether myNumber % 2 is 0 if myNumber is not less than 0. Since none of the conditions are true, it executes the body of the final else clause.

# enum

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **enum** keyword identifies an **enumeration**.

# event

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

C #



C sharp musical note

The **event** keyword is used to declare a *event*.

# explicit

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

## General

When values are cast implicitly, the runtime does not need any casting in code by the developer in order for the value to be converted to its new type.

Here is an example, where the developer is casting *explicitly*:

```
//Example of explicit casting.  
float fNumber = 100.00f;  
int iNumber = (int)fNumber;
```

The developer has told the runtime, "I know what I'm doing, force this conversion."

Implicit casting means that runtime doesn't need any prompting in order to do the conversion. Here is an example of this.

```
//Example of implicit casting.  
byte bNumber = 10;  
int iNumber = bNumber;
```

## Keyword

Notice that no casting was necessary by the developer. What is special about implicit, is that the context that the type is converted to is totally lossless i.e. converting to this type loses no information, so it can be converted back without worry.

The **explicit** keyword is used to create type conversion operators which can only be used by specifying an explicit type cast.

This construct is useful to help software developers write more readable code. Having an explicit cast name makes it clear that a conversion is taking place.

```
class Something
{
    public static explicit operator Something(string s)
    {
        // convert the string to Something
    }
}

string x = "hello";

// Implicit conversion (string to Something) generates a compile time error
Something s = x;

// This statement is correct (explicit type name conversion)
Something s = (Something) x;
```

# extern

The keyword **extern** indicates that the method being called exists in a DLL.

A tool called "tlbimp.exe" can create a wrapper assembly that allows C# to interact with the DLL like it was a .NET assembly i.e. use constructors to instantiate it, call its methods.

Older DLLs will not work with this method. Instead, you have to explicitly tell the compiler what DLL to call, what method to call and what parameters to pass. Since parameter type is very important, you can also explicitly define what type the parameter should be passed to the method as.

Here is an example:

```
using System;
using System.Runtime.InteropServices;

namespace ExternKeyword
{
    public class Program
    {
        static void Main()
        {
            NativeMethods.MessageBoxEx(IntPtr.Zero, "Hello there", "Caption
here", 0, 0);
        }
    }

    public class NativeMethods
    {
        [DllImport("user32.dll")]
        public static extern MessageBoxEx(IntPtr hWnd, string lpText, string
lpCaption, uint uType, short wLanguageId);
    }
}
```



The `[DllImport("user32.dll")]` tells the compiler which DLL to reference. Windows searches for files as defined by the PATH environment variable, and therefore will search those paths before failing.

The method is also static because the DLL may not understand how to be "created", as DLLs can be created in different languages. This allows the method to be called directly, instead of being instantiated and then used.

false

**C# Programming**

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

C#



C sharp musical note

The **true** keyword is a **boolean** constant value.

# finally

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The keyword **finally** is used to identify a *statement* or *statement block* after a [try-catch](#) block for execution regardless of whether the associated try block encountered an exception. The finally block is used to perform cleanup activities.

# fixed

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **fixed** keyword is used to prevent the garbage collector from relocating a variable. You may only use this in an **unsafe** context.

```
fixed (int *c = &shape.color) {  
    *c = Color.White;  
}
```

If you are using C# 2.0 or greater, the **fixed** may also be used to declare a fixed-size array. This is useful when creating code that works with a DLL or COM project.

Your array must be composed of one of the primitive types: **bool**, **byte**, **char**, **short**, **int**, **long**, **sbyte**, **ushort**, **ulong**, **float**, or **double**.

```
protected fixed int monthdays[12];
```

# float

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **float** keyword is used in field, **method**, property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Single`. That is, it represents a IEEE 754, 32-bit signed binary floating point number whose value is *negative 0*, *positive 0*, *negative infinity*, *positive infinity*, *not a number*, or a number ranging either from  $-1.5 \times 10^{-45}$  to  $-3.4 \times 10^{38}$  or from  $1.5 \times 10^{-45}$  to  $3.4 \times 10^{38}$ .

# for

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

C#



C sharp musical note

The **for** keyword identifies a **for loop**.

# foreach

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **foreach** keyword identifies a [foreach loop](#).

```
// example of foreach to iterate over an array
public static void Main() {
    int[] scores = new int [] { 54, 78, 34, 88, 98, 12 };

    foreach (int score in scores) {
        total += score;
    }
    int averageScore = total / scores.length;
}
```

# goto

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

## C#



C sharp musical note

The **goto** keyword returns the flow of operation to the label which follows it. Labels can be created by putting a colon after any word. e.g.

```
thelabel:    // This is a label
System.Console.WriteLine("Blah blah blah");
goto thelabel; // Program flow returns to thelabel
```

The use of **goto** is very controversial, because, when used frivolously, it creates code that jumps from place to place and is disorganized and hard to read. It is rarely even necessary because the same thing can often be accomplished with a more organized **for** loop or **while** loop.



# if

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **if** keyword identifies an **if statement** with the following syntax:

*if-statement* ::= "if" "(" *condition* ")" *if-body* ["else" *else-body*]

*condition* ::= *boolean-expression*

*if-body* ::= *statement-or-statement-block*

*else-body* ::= *statement-or-statement-block*

If the *condition* evaluates to **true**, the *if-body* executes. Curly braces ("{" and "}") allow the *if-body* to contain more than one statement. Optionally, an else clause can immediately follow the *if-body*, providing code to execute when the *condition* is *false*. Making the *else-body* another if statement creates the common *cascade* of if, else if, else if, else if, else statements:

```
using System;

public class IfStatementSample
{
    public void IfMyNumberIs()
    {
        int myNumber = 5;
        ( myNumber == 4 )
        Console.WriteLine("This will not be shown because myNumber is not
4.");
        else ( myNumber < 0 )
        {
            Console.WriteLine("This will not be shown because myNumber is not
negative.");
        }
        else ( myNumber % 2 == 0 )
```

## C Sharp Programming

---

```
        Console.WriteLine("This will not be shown because myNumber is not  
even.");  
        else  
        {  
            Console.WriteLine("myNumber does not match the coded conditions,  
so this sentence will be shown!");  
        }  
    }  
}
```

The boolean expression used in an if statement typically contains one of the following operators:

<b>Operator</b>	<b>Meaning</b>
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
&&	and
	or
!	not

See also [else](#).

# implicit

## General

When values are cast implicitly, the runtime does not need any casting in code by the developer in order for the value to be converted to its new type.

Here is an example, where the developer is casting *explicitly*:

```
//Example of explicit casting.  
float fNumber = 100.00f;  
int iNumber = (int)fNumber;
```

The developer has told the runtime, "I know what I'm doing, force this conversion."

Implicit casting means that runtime doesn't need any prompting in order to do the conversion. Here is an example of this.

```
//Example of implicit casting.  
byte bNumber = 10;  
int iNumber = bNumber;
```

Notice that no casting was necessary by the developer. What is special about implicit, is that the context that the type is converted to is totally lossless i.e. converting to this type loses no information, so it can be converted back without worry.

## Keyword

The keyword **implicit** is used for a type to define how to can be converted implicitly. It is used to define what types can be converted to without the need for explicit casting.

## C Sharp Programming

---

As an example, let us take a Fraction class, that will hold a nominator (the number at the top of the division), and a denominator (the number at the bottom of the division). We will add a property so that the value can be converted to a float.

```
public class Fraction
{
    private int _nominator;
    private int _denominator;

    public Fraction(int nominator, int denominator)
    {
        _nominator = nominator;
        _denominator = denominator;
    }

    public float Value { get { return (float)_nominator / (float)_denominator; } }

    public static implicit operator float(Fraction f)
    {
        return f.Value;
    }

    public override string ToString()
    {
        return _nominator + " / " _denominator;
    }
}

public class Program
{
    [STAThread]
    public static void Main(string[] args)
    {
        Fraction fractionClass = new Fraction( 1,2);
        float number = fractionClass;

        Console.WriteLine("{0} = {1}", fractionClass, number);
    }
}
```

To re-iterate, the value it implicitly casts to *must* hold data in the form that the original class can be converted back to. If this is not possible, and the range is narrowed (like converting *double* to *int*, use the explicit operator.

# in

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **in** keyword identifies the collection to enumerate in a **foreach** loop.

# int

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **int** keyword is used in field, **method**, property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Int32`. That is, it represents a 32-bit signed integer whose value ranges from -2,147,483,648 to 2,147,483,647.

# interface

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C #



C sharp musical note

The **interface** keyword is used to declare a *interface*. Interfaces provide a construct for a programmer to create types that can have methods, properties, delegates, events, and indexers declared, but not implemented.

# internal

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **internal** keyword is an *access modifier* used in field, **method**, and property declarations to make the field, method, or property *internal* to its enclosing assembly. That is, it is only **visible** within the assembly that implements it.



# is

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **is** keyword compares an object to a type, and if they're the same or of the same "kind" (the object **inherits** the type), returns **true**. The keyword is therefore used to check for type compatibility, usually before *casting* (converting) a source type to a destination type in order to ensure that won't cause a type-cast exception to be thrown. Using **is** on a **null** variable always returns **false**.

This code snippet shows a sample usage:

```
System.IO.StreamReader reader = new StreamReader("readme.txt");
bool b = reader is System.IO.TextReader;

// b is now set to true, because StreamReader inherits TextReader
```

# long

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **long** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Int64`. That is, it represents a 64-bit signed integer whose value ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

# namespace

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The "**namespace**" keyword is used to supply a *namespace* for class, structure, and type declarations.

# new

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **new** keyword is an operator that requests a new instance of the class identified by its argument.

# null

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **null** keyword represents an empty value for a *reference* type variable, i.e. for a variable of any type derived from System.Object. In C# 2.0, null also represents the empty value for nullable *value* type variables.

# object

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **object** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure System.Object. That is, it represents the base class from which all other *reference types* derive. On some platforms, the size of the reference is 32 bits, while on other platforms it is 64 bits.

# out

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **out** keyword explicitly specifies that a variable should be passed *by reference* to a method, and set in that method. A variable using this keyword must *not* be initialized before the method call to ensure the developer understand its intended effects. Using this keyword requires the called method to set the variable using this modifier before returning. Using `out` also requires the developer to specify the keyword even in the calling code, to ensure that it is easily visible to developers reading the code that the variable will have its value changed elsewhere, which is useful when analyzing the program flow.

An example of passing a variable with `out` follows:

```
void CallingMethod()
{
    int i;
    SetDependingOnTime( i);
    // i is now 10 before/at 12 am, or 20 after
}

void SetDependingOnTime( int iValue)
{
    iValue = DateTime.Now.Hour <= 12 ? 10 : 20;
}
```

# override

The keyword **override** is use in declaring an overridden function, which extends a base class function of the same name.



# params

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The keyword **params** is used to describe when a grouping of parameters are passed to a method, but the number of parameters are not important, as they may vary. Since the number isn't important, The *params* keyword must be the last variable in a method signature so that the compiler can deal with the parameters which have been defined first, before dealing with the params.

Here are examples of where it will, and will not work:

```
//This works
public static void AddToShoppingBasket(decimal total, params string[] items)
{
    //....
}

//This works
public static void AddToShoppingBasket(decimal total, int totalQuantity,
params string[] items)
{
    //....
}

//THIS DOES NOT WORK
public static void AddToShoppingBasket(params string[] items, decimal total,
int totalQuantity)
{
    //....
}
```

## C Sharp Programming

---

A good example of this is the *String.Format* method. The *String.Format* method allows a user to pass in a string formatted to their requirements, and then lots of parameters for the values to insert into the string. Here is an example:

```
public static string FormatMyString(string format, params string[] values)
{
    string myFormat = "Date: {0}, Time: {1}, WeekDay: {1}";
    return String.Format(myFormat, DateTime.Now.ToShortDateString(), Date-
Time.Now.ToShortTimeString(), DateTime.Now.DayOfWeek);
}

//Output will be something like:
//
//Date: 7/8/2007, Time: 13:00, WeekDay: Tuesday;
//
```

The *String.Format* method has taken a string, and replaced the {0},{1},{2} with the 1st, 2nd and 3rd parameters. If the *params* keyword did not exist, then the *String.Format()* could have an infinite number of overloads to cater for each case.

```
public string Format(string format, string param1)
{
    //.....
}

public string Format(string format, string param1, string param2)
{
    //.....
}

public string Format(string format, string param1, string param2, string
param3)
{
    //.....
}

public string Format(string format, string param1, string param2, string
param3, string param4)
{
    //.....
}

public string Format(string format, string param1, string param2, string
param3, string param4, string param5)
{
```

```
    //.....  
}  
//To infinitum
```

# private

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **private** keyword is used in field, [method](#), and property declarations to make the field, method, or property *private* to its enclosing class. That is, it is not [visible](#) outside of its class.

# protected

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **protected** keyword is used in field, [method](#), and property declarations to make the field, method, or property *protected* to its enclosing class. That is, it is [visible](#) only to its class and the classes that derive from it.

# public

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C #



C sharp musical note

The **public** keyword is used in field, **method**, and property declarations to make the field, method, or property *public* to its enclosing class. That is, it is **visible** from any class.

# readonly

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **readonly** keyword is closely related to the **const** keyword at a glance, with the exception of allowing a variable with this modifier to be initialized in a constructor, along with being associated with a class instance (object) rather than the class itself.

The primary use for this keyword is to allow the variable to take on different values depending on which constructor was called, in case the class has many, while still ensuring the developer that it can never intentionally or unintentionally be changed in the code once the object has been created.

This is a sample usage, assumed to be in a class called `SampleClass`:

```
string s;  
  
SampleClass()  
{  
    s = "Hello!";  
}
```

# ref

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **ref** keyword explicitly specifies that a variable should be passed *by reference* rather than *by value*.

A developer may wish to pass a variable by reference particularly in case of **value types**. If a variable is passed by reference, only a pointer is sent to a function in reality, reducing the cost of a method call in case it would involve copying large amounts of data, something C# does when normally passing value types.

Another common reason to pass a variable by reference is to let the called method modify its value. Because this is allowed, C# always enforces specifying that a value is passed by reference even in the method call, something many other programming languages don't. This let developers reading the code easily spot places that can imply a type has had its value changed in a method, which is useful when analyzing the program flow.

Passing a value by reference does not imply that the called method *has* to modify the value; see the **out** keyword for this.

Passing by reference requires the passed variable to be initialized.

An example of passing a variable by reference follows:

```
void CallingMethod()
{
    int i = 24;
    if (DoubleIfEven( i))
        Console.WriteLine("i was doubled to {0}", i); // outputs "i was doubled
to 48"
}
```



```
bool DoubleIfEven( int iValue)
{
    if (iValue % 2 == 0)
    {
        iValue *= 2;
        return true;
    }
    return false;
}
```

# return

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **return** keyword is used to return execution from a *method* or from a *property* accessor. If the *method* or *property* accessor has a return type, the return keyword is followed by the value to return.

# sbyte

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **sbyte** keyword is used in field, **method**, property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure System.SByte. That is, it represents an 8-bit signed integer whose value ranges from -128 to 127.

# sealed

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **sealed** keyword is used to specify that a class cannot be inherited from. The following example shows the context in which it may be used:

```
public class  
{  
    ...  
}
```

# short

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **short** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Int16`. That is, it represents a 16-bit signed integer whose value ranges from -32,768 to 32,767.

# sizeof

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **sizeof** keyword returns how many bytes an object requires to be stored.

An example usage:

```
int i = 123456;

Console.WriteLine("Storing i, a {0}, requires {1} bytes, or {2} bits.",
    i.GetType(), (i), (i) * 8);

// outputs "Storing i, a System.Int32, requires 4 bytes, or 32 bits."
```

# stackalloc

The keyword **stackalloc** is used in an unsafe code context to allocate a block of memory on the stack.

```
int* fib = stackalloc int[100];
```

In the example above, a block of memory of sufficient size to contain 100 elements of type `int` is allocated on the stack, not the heap; the address of the block is stored in the pointer `fib`. This memory is not subject to garbage collection and therefore does not have to be pinned (via `fixed`). The lifetime of the memory block is limited to the lifetime of the method in which it is defined (there is no way to free the memory before the method returns).

`stackalloc` is only valid in local variable initializers.

Because Pointer types are involved, `stackalloc` requires unsafe context. See [Unsafe Code and Pointers](#).

`stackalloc` is similar to `_alloca` in the C run-time library.

Note\* - From MSDN

# static

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **static** keyword is used to declare a *class* or a class member (*method, property, field, or variable*) as *static*. A *class* that is declared *static* has only *static* members. A class member that is declared *static* is associated with the entire class instead of class *instances*.



# string

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **string** keyword is used in field, **method**, property, and variable declarations and in *cast* and *typeof* operations as an alias for System.String. That is, it indicates an immutable sequence of characters.

# struct

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **struct** keyword declares a **structure**, i.e. a *value type* that functions as a light-weight *class*.

# switch

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **switch** statement is a control statement that handles multiple selections and enumerations by passing control to one of the case statements within its body.

This is an example of a switch statement:

```
int currentAge = 18;

currentAge
{
    case 16:
        Console.WriteLine("You can drive!")
        break;
    case 18:
        Console.WriteLine("You're finally an adult!");
        break;
    default:
        Console.WriteLine("Nothing exciting happened this year.");
        break;
}
```

### Console Output

```
You're finally an adult!
```

# this

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **this** keyword is used in an *instance method* or *instance property* to reference the *current* instance of class. That is, this refers to the object through which its containing method or property was invoked.

# throw

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The throw keyword is used *throw* an exception object.

# true

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **true** keyword is a **boolean** constant value. Therefore

```
while(true)
```

would create an infinite loop.

# try

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The try keyword is used to identify a *statement* or *statement block* as the body of an exception handling sequence. The body of the exception handling sequence must be followed by a catch clause, a finally clause, or both.

# typeof

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **typeof** keyword returns an instance of the System.Type class when passed a name of a class. It is similar to the [sizeof](#) keyword in that it returns a value instead of starting a section (block) of code (see [if](#), [while](#), [try](#)).

An example:

```
using System;

namespace MyNamespace
{
    class MyClass
    {
        static void Main(string[] args)
        {
            Type t = typeof(int);
            Console.Out.WriteLine(t.ToString());
            Console.In.Read();
        }
    }
}
```

The output will be:

```
System.Int32
```



It should be noted that unlike sizeof, only class names themselves and not variables can be passed to typeof, as shown here:

```
using System;

namespace MyNamespace
{
    class MyClass2
    {
        static void Main(string[] args)
        {
            char ch;

            // This line will cause compilation to fail
            Type t = typeof(ch);
            Console.Out.WriteLine(t.ToString());
            Console.In.Read();
        }
    }
}
```

Sometimes, classes will include their own GetType() method that will be similar, if not identical, to typeof.

# uint

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **uint** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.UInt32`. That is, it represents a 32-bit unsigned integer whose value ranges from 0 to 4,294,967,295.

# ulong

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **ulong** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure System.UInt64. That is, it represents a 64-bit unsigned integer whose value ranges from 0 to 18,446,744,073,709,551,615.

# unchecked

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **unchecked** keyword prevents overflow-checking when doing integer arithmetic. It may be used as an *operator* on a single expression or as a statement on a whole block of code.

```
int x, y, z;
x = 1222111000;
y = 1222111000;

// used as an operator
z = unchecked( x * y );

// used as a statement
unchecked {
    z = x * y;
    x = z * z;
}
```

# unsafe

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **unsafe** keyword may be used to modify a procedure or define a block of code which uses unsafe code. Code is unsafe if it uses the "address of" operator(&) or if it uses a pointer operator (\*).

In order for the compiler to compile code containing this keyword, you must use the **/unsafe** option when using the Microsoft C-Sharp Compiler.

```
// example of unsafe to modify a procedure
class MyClass {
    unsafe static void(string *msg) {
        Console.WriteLine(*msg)
    }
}

// example of unsafe to modify a code block
string s = "hello";
unsafe {
    char *cp = &s[2];
    *cp = 'a';
}
```

# ushort

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **ushort** keyword is used in field, [method](#), property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure System.UInt16. That is, it represents a 16-bit unsigned integer whose value ranges from 0 to 65,535.

# using

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **using** keyword has two completely unrelated meanings in C#, depending on if it is used as a directive or a statement.

## The directive

**using** as a *directive* resolves unqualified type references so a developer doesn't have to specify the complete namespace.

Example:

```
using System;

// A developer can now type Console.WriteLine(); rather than System.Console.WriteLine().
```

**using** can also provide a *namespace alias* for referencing types.

Example:

```
using Utils = Company.Application.Utilities;
```

# The statement

**using** as a *statement* with a scope specifies an object's lifetime. At the end of the scope, the object's destructor will be run and the C# garbage collector will free its allocated resources.

Example:

```
using (System.IO.StreamReader reader = new StreamReader("readme.txt"))
{
    // read from the file
}

// reader is now destroyed, its file handle freed, and an unknown variable
again
```



# virtual

The keyword **virtual** is applied to a method declaration to indicate that the method may be overridden in a subclass. If the virtual keyword is not applied and a method is defined in a subclass with the same signature as the one in the parent class, the method in the parent class is hidden by the subclass implementation.

# void

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The **void** keyword is used in [method](#) signatures to declare a method that does not return a value. A method declared with the void return type cannot provide any arguments to any [return](#) statements they contain.

Example:

```
public void WorkRepeatedly(int numberOfTimes)
{
    for(int i=0; i<numberOfTimes; i++)
        if(EarlyTerminationIsRequested)
            return;
        else
            DoWork();
}
```

# volatile

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **volatile** keyword is used to declare a variable which may change its value over time due to modification by an outside process, the system hardware, or another concurrently running thread.

You should use this modifier in your member variable declaration to ensure that whenever the value is read, you are always getting the most recent (up-to-date) value of the variable.

```
class MyClass
{
    public volatile long systemclock;
}
```

This keyword has been part of the C# programming language since .NET Framework 1.1 (Visual Studio 2003).

# while

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **while** keyword identifies a [while](#) loop.

# alias

It can sometimes be necessary to reference two versions of assemblies that have the same fully-qualified type names, for example when you need to use two or more versions of an assembly in the same application. By using an external assembly alias, the namespaces from each assembly can be wrapped inside root-level namespaces named by the alias, allowing them to be used in the same file.

To reference two assemblies with the same fully-qualified type names, an alias must be specified on the command line, as follows:

```
/r:GridV1=grid.dll  
  
/r:GridV2=grid20.dll
```

This creates the external aliases GridV1 and GridV2. To use these aliases from within a program, reference them using the **extern** keyword. For example:

```
extern alias GridV1;  
  
extern alias GridV2;
```

Each extern alias declaration introduces an additional root-level namespace that parallels (but does not lie within) the global namespace. Thus types from each assembly can be referred to without ambiguity using their fully qualified name, rooted in the appropriate namespace-alias

In the above example, GridV1::Grid would be the grid control from grid.dll, and GridV2::Grid would be the grid control from grid20.dll.

# get

### C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The special identifier **get** is used to declare the *read accessor* for a *property*.

# partial

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The special identifier **partial** is used to allow *developers* to build *classes* from different files and have the compiler generate one class (combining all the partial classes). This is mostly useful for separating classes into separate blocks. For example, Visual Studio 2005 separates the UI code for forms into a separate partial class which allows you to work on the business logic separately.

# set

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The special identifier **set** is used to declare the *write accessor* for a *property*.



# value

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)

# C#



C sharp musical note

The special identifier **value** is used in a property's *write accessor* to represent the value requested for assignment to the *property*.

# yield

## C# Programming

[Cover](#) | [Introduction](#) | [Basics](#) | [Classes](#) | [The .NET Framework](#) | [Advanced Topics](#) | [Index](#)



C sharp musical note

The **yield** keyword returns the next value from an iterator or ends an iteration. See [Using yield](#).

# GNU Free Documentation License

Version 1.2, November 2002

```
Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.
```

## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

## C Sharp Programming

---

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D.** Preserve all the copyright notices of the Document.
- E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H.** Include an unaltered copy of this License.
- I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.



## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

## C Sharp Programming

---

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.