Contents

Statements, Expressions, and Operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The C# code that comprises an application consists of statements made up of keywords, expressions and operators. This section contains information regarding these fundamental elements of a C# program.

For more information, see:

- Statements
- Expressions
  - Expression-bodied members
- Operators
- Anonymous Functions
- Overloadable Operators
- Conversion Operators
  - Using Conversion Operators
  - How to: Implement User-Defined Conversions Between Structs
- Equality Comparisons

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Casting and Type Conversions

Statements (C# Programming Guide)

1/23/2019 • 6 minutes to read • Edit Online

The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment statement:
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                     counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;

        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types of Statements

The following table lists the various types of statements in C# and their associated keywords, with links to topics that include more information:

CATEGORY	C# KEYWORDS / NOTES
Declaration statements	A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.
Expression statements	Expression statements that calculate a value must store the value in a variable. For more information, see Expression Statements.
Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics: if, else, switch, case
Iteration statements	Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics: do, for, foreach, in, while
Jump statements	Jump statements transfer control to another section of code. For more information, see the following topics: break, continue, default, goto, return, yield
Exception handling statements	Exception handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics: throw, try-catch, try-finally, try-catch-finally
Checked and unchecked	Checked and unchecked statements enable you to specify whether numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value. For more information, see checked and unchecked.
The `await` statement	If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an `await` expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For a simple example, see the "Async Methods" section of Methods. For more information, see Asynchronous Programming with async and await.

CATEGORY	C# KEYWORDS / NOTES
The `yield return` statement	An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time. For more information, see Iterators.
The `fixed` statement	The fixed statement prevents the garbage collector from relocating a movable variable. For more information, see fixed.
The `lock` statement	The lock statement enables you to limit access to blocks of code to only one thread at a time. For more information, see lock.
Labeled statements	You can give a statement a label and then use the goto keyword to jump to the labeled statement. (See the example in the following row.)
The empty statement	The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed.

Declaration statements

The following code shows examples of variable declarations with and without an initial assignment, and a constant declaration with the necessary initialization.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Expression statements

The following code shows examples of expression statements, including assignment, object creation with assignment, and method invocation.

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

The empty statement

The following examples show two uses for an empty statement:

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Embedded Statements

Some statements, including do, while, for, and foreach, always have an embedded statement that follows them. This embedded statement may be either a single statement or multiple statements enclosed by {} brackets in a statement block. Even single-line embedded statements can be enclosed in {} brackets, as shown in the following example:

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

An embedded statement that is not enclosed in {} brackets cannot be a declaration statement or a labeled statement. This is shown in the following example:

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Put the embedded statement in a block to fix the error:

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Nested Statement Blocks

Statement blocks can be nested, as shown in the following code:

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }

}
return "Not found.";
```

Unreachable Statements

If the compiler determines that the flow of control can never reach a particular statement under any circumstances, it will produce warning CS0162, as shown in the following example:

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

Related Sections

- Statement Keywords
- Expressions
- Operators

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Expressions (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

An *expression* is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a *simple name*. Simple names can be the name of a variable, type member, method parameter, namespace or type.

Expressions can use operators that in turn use other expressions as parameters, or method calls whose parameters are in turn other method calls, so expressions can range from simple to very complex. Following are two examples of expressions:

```
((x < 10) && ( x > 5)) || ((x > 20) && (x < 25));

System.Convert.ToInt32("35");
```

Expression values

In most of the contexts in which expressions are used, for example in statements or method parameters, the expression is expected to evaluate to some value. If x and y are integers, the expression `x + y` evaluates to a numeric value. The expression `new MyClass()` evaluates to a reference to a new instance of a `MyClass` class. The expression `myClass.ToString()` evaluates to a string because that is the return type of the method. However, although a namespace name is classified as an expression, it does not evaluate to a value and therefore can never be the final result of any expression. You cannot pass a namespace name to a method parameter, or use it in a new expression, or assign it to a variable. You can only use it as a sub-expression in a larger expression. The same is true for types (as distinct from System.Type objects), method group names (as distinct from specific methods), and event add and remove accessors.

Every value has an associated type. For example, if x and y are both variables of type `int`, the value of the expression `x + y` is also typed as `int`. If the value is assigned to a variable of a different type, or if x and y are different types, the rules of type conversion are applied. For more information about how such conversions work, see Casting and Type Conversions.

Overflows

Numeric expressions may cause overflows if the value is larger than the maximum value of the value's type. For more information, see Checked and Unchecked and Explicit Numeric Conversions Table.

Operator precedence and associativity

The manner in which an expression is evaluated is governed by the rules of associativity and operator precedence. For more information, see Operators.

Most expressions, except assignment expressions and method invocation expressions, must be embedded in a statement. For more information, see Statements.

Literals and simple names

The two simplest types of expressions are literals and simple names. A literal is a constant value that has no name. For example, in the following code example, both `5` and `"Hello World"` are literal values:

```
// Expression statements.
int i = 5;
string s = "Hello World";
```

For more information on literals, see Types.

In the preceding example, both `i` and `s` are simple names that identify local variables. When those variables are used in an expression, the variable name evaluates to the value that is currently stored in the variable's location in memory. This is shown in the following example:

```
int num = 5;
System.Console.WriteLine(num); // Output: 5
num = 6;
System.Console.WriteLine(num); // Output: 6
```

Invocation expressions

In the following code example, the call to `DoWork` is an invocation expression.

```
DoWork();
```

A method invocation requires the name of the method, either as a name as in the previous example, or as the result of another expression, followed by parenthesis and any method parameters. For more information, see Methods. A delegate invocation uses the name of a delegate and method parameters in parenthesis. For more information, see Delegates. Method invocations and delegate invocations evaluate to the return value of the method, if the method returns a value. Methods that return void cannot be used in place of a value in an expression.

Query expressions

The same rules for expressions in general apply to query expressions. For more information, see LINQ Query Expressions.

Lambda expressions

Lambda expressions represent "inline methods" that have no name but can have input parameters and multiple statements. They are used extensively in LINQ to pass arguments to methods. Lambda expressions are compiled to either delegates or expression trees depending on the context in which they are used. For more information, see Lambda Expressions.

Expression trees

Expression trees enable expressions to be represented as data structures. They are used extensively by LINQ providers to translate query expressions into code that is meaningful in some other context, such as a SQL database. For more information, see Expression Trees (C#).

Expression body definitions

C# supports *expression-bodied members*, which allow you to supply a concise expression body definition for methods, constructors, finalizers, properties, and indexers. For more information, see Expression-bodied members.

Remarks

Whenever a variable, object property, or object indexer access is identified from an expression, the value of that item is used as the value of the expression. An expression can be placed anywhere in C# where a value or object is required, as long as the expression ultimately evaluates to the required type.

See also

- C# Programming Guide
- Methods
- Delegates
- Operators
- Types
- LINQ Query Expressions

Expression-bodied members (C# programming guide)

2/7/2019 • 3 minutes to read • Edit Online

Expression body definitions let you provide a member's implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression. An expression body definition has the following general syntax:

```
member => expression;
```

where *expression* is a valid expression.

Support for expression body definitions was introduced for methods and read-only properties in C# 6 and was expanded in C# 7.0. Expression body definitions can be used with the type members listed in the following table:

MEMBER	SUPPORTED AS OF...
Method	C# 6
Read-only property	C# 6
Property	C# 7.0
Constructor	C# 7.0
Finalizer	C# 7.0
Indexer	C# 7.0

Methods

An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for methods that return `void`, that performs some operation. For example, types that override the ToString method typically include a single expression that returns the string representation of the current object.

The following example defines a `Person` class that overrides the ToString method with an expression body definition. It also defines a `DisplayName` method that displays a name to the console. Note that the `return` keyword is not used in the `ToString` expression body definition.

```
using System;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

For more information, see Methods (C# Programming Guide).

Read-only properties

Starting with C# 6, you can use expression body definition to implement a read-only property. To do that, use the following syntax:

```
PropertyType PropertyName => expression;
```

The following example defines a `Location` class whose read-only `Name` property is implemented as an expression body definition that returns the value of the private `locationName` field:

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

For more information about properties, see Properties (C# Programming Guide).

Properties

Starting with C# 7.0, you can use expression body definitions to implement property `get` and `set` accessors. The following example demonstrates how to do that:

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

For more information about properties, see Properties (C# Programming Guide).

Constructors

An expression body definition for a constructor typically consists of a single assignment expression or a method call that handles the constructor's arguments or initializes instance state.

The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `Name` property.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

For more information, see Constructors (C# Programming Guide).

Finalizers

An expression body definition for a finalizer typically contains cleanup statements, such as statements that release unmanaged resources.

The following example defines a finalizer that uses an expression body definition to indicate that the finalizer has been called.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

For more information, see Finalizers (C# Programming Guide).

Indexers

Like properties, an indexer's get and set accessors consist of expression body definitions if the get accessor consists of a single statement that returns a value or the set accessor performs a simple assignment.

The following example defines a class named `Sports` that includes an internal String array that contains the names of a number of sports. Both the indexer's get and set accessors are implemented as expression body definitions.

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

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

For more information, see Indexers (C# Programming Guide).

Operators (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

In C#, an *operator* is a program element that is applied to one or more *operands* in an expression or statement. Operators that take one operand, such as the increment operator (`++`) or `new`, are referred to as *unary* operators. Operators that take two operands, such as arithmetic operators (`+`, `-`, `*`, `/`), are referred to as *binary* operators. One operator, the conditional operator (`?:`), takes three operands and is the sole ternary operator in C#.

The following C# statement contains a single unary operator and a single operand. The increment operator, `++`, modifies the value of the operand `y`.

```
y++;
```

The following C# statement contains two binary operators, each with two operands. The assignment operator, `=`, has the integer variable `y` and the expression `2 + 3` as operands. The expression `2 + 3` itself consists of the addition operator and two operands, `2` and `3`.

```
y = 2 + 3;
```

Operators, evaluation, and operator precedence

An operand can be a valid expression that is composed of any length of code, and it can comprise any number of sub expressions. In an expression that contains multiple operators, the order in which the operators are applied is determined by *operator precedence*, *associativity*, and parentheses.

Each operator has a defined precedence. In an expression that contains multiple operators that have different precedence levels, the precedence of the operators determines the order in which the operators are evaluated. For example, the following statement assigns 3 to `n1`.

```
n1 = 11 - 2 * 4;
```

The multiplication is executed first because multiplication takes precedence over subtraction.

The following table separates the operators into categories based on the type of operation they perform. The categories are listed in order of precedence.

Primary Operators

EXPRESSION	DESCRIPTION
x.y	Member access
x?.y	Conditional member access
f(x)	Method and delegate invocation
a[x]	Array and indexer access
a?[x]	Conditional array and indexer access

EXPRESSION	DESCRIPTION
x++	Post-increment
x--	Post-decrement
new T(...)	Object and delegate creation
`new` T(...){...}	Object creation with initializer. See Object and Collection Initializers.
`new` {...}	Anonymous object initializer. See Anonymous Types.
`new` T[...]	Array creation. See Arrays.
typeof(T)	Obtain System.Type object for T
checked(x)	Evaluate expression in checked context
unchecked(x)	Evaluate expression in unchecked context
default (T)	Obtain default value of type T
delegate {}	Anonymous function (anonymous method)

Unary Operators

EXPRESSION	DESCRIPTION
+x	Identity
-x	Negation
!x	Logical negation
~x	Bitwise negation
++x	Pre-increment
--x	Pre-decrement
(T)x	Explicitly convert x to type T

Multiplicative Operators

EXPRESSION	DESCRIPTION
*	Multiplication
/	Division
%	Remainder

Additive Operators

EXPRESSION	DESCRIPTION
x + y	Addition, string concatenation, delegate combination
x - y	Subtraction, delegate removal

Shift Operators

EXPRESSION	DESCRIPTION
x << y	Shift left
x >> y	Shift right

Relational and Type Operators

EXPRESSION	DESCRIPTION
x < y	Less than
x > y	Greater than
x <= y	Less than or equal
x >= y	Greater than or equal
x is T	Return true if x is a T, false otherwise
x as T	Return x typed as T, or null if x is not a T

Equality Operators

EXPRESSION	DESCRIPTION
x == y	Equal
x != y	Not equal

Logical, Conditional, and Null Operators

CATEGORY	EXPRESSION	DESCRIPTION
Logical AND	x & y	Integer bitwise AND, Boolean logical AND
Logical XOR	x ^ y	Integer bitwise XOR, Boolean logical XOR
Logical OR	x | y	Integer bitwise OR, Boolean logical OR
Conditional AND	x && y	Evaluates y only if x is true

CATEGORY	EXPRESSION	DESCRIPTION
Conditional OR	x || y	Evaluates y only if x is false
Null coalescing	x ?? y	Evaluates to y if x is null, to x otherwise
Conditional	x ? y : z	Evaluates to y if x is true, z if x is false

Assignment and Anonymous Operators

EXPRESSION	DESCRIPTION
=	Assignment
x op= y	Compound assignment. Supports these operators: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
(T x) => y	Anonymous function (lambda expression)

Associativity

When two or more operators that have the same precedence are present in an expression, they are evaluated based on associativity. Left-associative operators are evaluated in order from left to right. For example, `x * y / z` is evaluated as `(x * y) / z`. Right-associative operators are evaluated in order from right to left. For example, the assignment operator is right associative. If it were not, the following code would result in an error.

```
int a, b, c;
c = 1;
// The following two lines are equivalent.
a = b = c;
a = (b = c);

// The following line, which forces left associativity, causes an error.
//(a = b) = c;
```

As another example the ternary operator (?:) is right associative. Most binary operators are left associative.

Whether the operators in an expression are left associative or right associative, the operands of each expression are evaluated first, from left to right. The following examples illustrate the order of evaluation of operators and operands.

STATEMENT	ORDER OF EVALUATION
`a = b`	a, b, =
`a = b + c`	a, b, c, +, =
`a = b + c * d`	a, b, c, d, *, +, =
`a = b * c + d`	a, b, c, *, d, +, =
`a = b - c + d`	a, b, c, -, d, +, =

STATEMENT	ORDER OF EVALUATION
`a += b -= c`	a, b, c, -=, +=

Adding parentheses

You can change the order imposed by operator precedence and associativity by using parentheses. For example, `2 + 3 * 2` ordinarily evaluates to 8, because multiplicative operators take precedence over additive operators. However, if you write the expression as `(2 + 3) * 2`, the addition is evaluated before the multiplication, and the result is 10. The following examples illustrate the order of evaluation in parenthesized expressions. As in previous examples, the operands are evaluated before the operator is applied.

STATEMENT	ORDER OF EVALUATION
`a = (b + c) * d`	a, b, c, +, d, *, =
`a = b - (c + d)`	a, b, c, d, +, -, =
`a = (b + c) * (d - e)`	a, b, c, +, d, e, -, *, =

Operator overloading

You can change the behavior of operators for custom classes and structs. This process is referred to as *operator overloading*. For more information, see Overloadable Operators and the operator keyword article.

Related sections

For more information, see Operator Keywords and C# Operators.

See also

- C# Programming Guide
- Statements, Expressions, and Operators

Anonymous Functions (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

An anonymous function is an "inline" statement or expression that can be used wherever a delegate type is expected. You can use it to initialize a named delegate or pass it instead of a named delegate type as a method parameter.

There are two kinds of anonymous functions, which are discussed individually in the following topics:

- Lambda Expressions.
- Anonymous Methods

> **NOTE**
> Lambda expressions can be bound to expression trees and also to delegates.

The Evolution of Delegates in C#

In C# 1.0, you created an instance of a delegate by explicitly initializing it with a method that was defined elsewhere in the code. C# 2.0 introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 introduced lambda expressions, which are similar in concept to anonymous methods but more expressive and concise. These two features are known collectively as *anonymous functions*. In general, applications that target version 3.5 and later of the .NET Framework should use lambda expressions.

The following example demonstrates the evolution of delegate creation from C# 1.0 to C# 3.0:

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Hello. My name is M and I write lines.
    That's nothing. I'm anonymous and
    I'm a famous author.
    Press any key to exit.
 */
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Statements, Expressions, and Operators
- Lambda Expressions
- Delegates
- Expression Trees (C#)

Lambda expressions (C# Programming Guide)

1/23/2019 • 9 minutes to read • Edit Online

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator `=>`, and you put the expression or statement block on the other side. For example, the lambda expression `x => x * x` specifies a parameter that's named `x` and returns the value of `x` squared. You can assign this expression to a delegate type, as the following example shows:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

To create an expression tree type:

```
using System.Linq.Expressions;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Expression<del> myET = x => x * x;
        }
    }
}
```

The `=>` operator has the same precedence as assignment (`=`) and is right associative (see "Associativity" section of the Operators article).

Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as Where.

When you use method-based syntax to call the Where method in the Enumerable class (as you do in LINQ to Objects and LINQ to XML) the parameter is a delegate type System.Func<T,TResult>. A lambda expression is the most convenient way to create that delegate. When you call the same method in, for example, the System.Linq.Queryable class (as you do in LINQ to SQL) then the parameter type is an System.Linq.Expressions.Expression<Func> where Func is any of the Func delegates with up to sixteen input parameters. Again, a lambda expression is just a very concise way to construct that expression tree. The lambdas allow the `Where` calls to look similar although in fact the type of object created from the lambda is different.

In the previous example, notice that the delegate signature has one implicitly-typed input parameter of type `int`, and returns an `int`. The lambda expression can be converted to a delegate of that type because it also has one input parameter (`x`) and a return value that the compiler can implicitly convert to type `int`. (Type inference is discussed in more detail in the following sections.) When the delegate is invoked by using an input parameter of 5, it returns a result of 25.

Lambdas are not allowed on the left side of the is or as operator.

All restrictions that apply to anonymous methods also apply to lambda expressions. For more information, see Anonymous Methods.

Expression lambdas

A lambda expression with an expression on the right side of the => operator is called an *expression lambda*. Expression lambdas are used extensively in the construction of Expression Trees. An expression lambda returns the result of the expression and takes the following basic form:

```
(input-parameters) => expression
```

The parentheses are optional only if the lambda has one input parameter; otherwise they are required. Two or more input parameters are separated by commas enclosed in parentheses:

```
(x, y) => x == y
```

Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:

```
(int x, string s) => s.Length > x
```

Input parameter types must be all explicit or all implicit; otherwise, C# generates a CS0748 compiler error.

Specify zero input parameters with empty parentheses:

```
() => SomeMethod()
```

Note in the previous example that the body of an expression lambda can consist of a method call. However, if you are creating expression trees that are evaluated outside of the .NET Framework, such as in SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET common language runtime.

Statement lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

(input-parameters) => { statement; }

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
delegate void TestDelegate(string s);
```

```
TestDelegate del = n => { string s = n + " World";
                          Console.WriteLine(s); };
```

Statement lambdas, like anonymous methods, cannot be used to create expression trees.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the async and await keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

You can add the same event handler by using an async lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\nControl returned to Click event handler.\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

For more information about how to create and use async methods, see Asynchronous Programming with async and await.

Lambdas with the standard query operators

Many standard query operators have an input parameter whose type is one of the Func<T,TResult> family of generic delegates. These delegates use type parameters to define the number and types of input parameters, and the return type of the delegate. `Func` delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the following delegate type:

```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

The delegate can be instantiated as `Func<int,bool> myFunc` where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when it is invoked, will return true or false to indicate whether the input parameter is equal to 5:

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4); // returns false of course
```

You can also supply a lambda expression when the argument type is an `Expression<Func>`, for example in the standard query operators that are defined in System.Linq.Queryable. When you specify an `Expression<Func>` argument, the lambda will be compiled to an expression tree.

A standard query operator, the Count method, is shown here:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following line of code produces a sequence that contains all elements in the `numbers` array that are to the left side of the 9 because that's the first number in the sequence that doesn't meet the condition:

```
var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);
```

This example shows how to specify multiple input parameters by enclosing them in parentheses. The method returns all the elements in the numbers array until a number is encountered whose value is less than its position. Do not confuse the lambda operator (`=>`) with the greater than or equal operator (`>=`).

```
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
```

Type inference in lambdas

When writing lambdas, you often do not have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter's delegate type, and other factors as described in the C# Language Specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. So if you are querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Note that lambda expressions in themselves do not have a type because the common type system has no intrinsic concept of "lambda expression." However, it is sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or Expression type to which the lambda

expression is converted.

Variable scope in lambda expressions

Lambdas can refer to *outer variables* (see Anonymous Methods) that are in scope in the method that defines the lambda function, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```
delegate bool D();
delegate bool D2(int i);

class Test
{
    D del;
    D2 del2;
    public void TestMethod(int input)
    {
        int j = 0;
        // Initialize the delegates with lambda expressions.
        // Note access to 2 outer variables.
        // del will be invoked within this method.
        del = () => { j = 10;  return j > input; };

        // del2 will be invoked after TestMethod goes out of scope.
        del2 = (x) => {return x == j; };

        // Demonstrate value of j:
        // Output: j = 0
        // The delegate has not been invoked yet.
        Console.WriteLine("j = {0}", j);        // Invoke the delegate.
        bool boolResult = del();

        // Output: j = 10 b = True
        Console.WriteLine("j = {0}. b = {1}", j, boolResult);
    }

    static void Main()
    {
        Test test = new Test();
        test.TestMethod(5);

        // Prove that del2 still has a copy of
        // local variable j from TestMethod.
        bool result = test.del2(10);

        // Output: True
        Console.WriteLine(result);

        Console.ReadKey();
    }
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression are not visible in the outer method.
- A lambda expression cannot directly capture an `in`, `ref`, or `out` parameter from an enclosing method.
- A return statement in a lambda expression does not cause the enclosing method to return.

- A lambda expression cannot contain a `goto` statement, `break` statement, or `continue` statement that is inside the lambda function if the jump statement's target is outside the block. It is also an error to have a jump statement outside the lambda function block if the target is inside the block.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured book chapter

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

See also

- C# Programming Guide
- LINQ (Language-Integrated Query)
- Anonymous Methods
- is
- Expression Trees
- Visual Studio 2008 C# Samples (see LINQ Sample Queries files and XQuery program)
- Recursive lambda expressions

How to: Use Lambda Expressions in a Query (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You do not use lambda expressions directly in query syntax, but you do use them in method calls, and query expressions can contain method calls. In fact, some query operations can only be expressed in method syntax. For more information about the difference between query syntax and method syntax, see Query Syntax and Method Syntax in LINQ.

Example

The following example demonstrates how to use a lambda expression in a method-based query by using the Enumerable.Where standard query operator. Note that the Where method in this example has an input parameter of the delegate type Func<TResult> and that delegate takes an integer as input and returns a Boolean. The lambda expression can be converted to that delegate. If this were a LINQ to SQL query that used the Queryable.Where method, the parameter type would be an `Expression<Func<int,bool>>` but the lambda expression would look exactly the same. For more information on the Expression type, see System.Linq.Expressions.Expression.

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

Example

The following example demonstrates how to use a lambda expression in a method call of a query expression. The lambda is necessary because the Sum standard query operator cannot be invoked by using query syntax.

The query first groups the students according to their grade level, as defined in the `GradeLevel` enum. Then for each group it adds the total scores for each student. This requires two `Sum` operations. The inner `Sum` calculates the total score for each student, and the outer `Sum` keeps a running, combined total for all students in the group.

```
private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
    from student in students
    group student by student.Year into studentGroup
    select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
     Outputs:
     Key = SecondYear Sum = 1014
     Key = ThirdYear Sum = 964
     Key = FirstYear Sum = 1058
     Key = FourthYear Sum = 974
*/
```

Compiling the Code

To run this code, copy and paste the method into the `StudentClass` that is provided in How to: Query a Collection of Objects and call it from the `Main` method.

See also

- Lambda Expressions
- Expression Trees (C#)

How to: Use Lambda Expressions Outside LINQ (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Lambda expressions are not limited to LINQ queries. You can use them anywhere a delegate value is expected, that is, wherever an anonymous method can be used. The following example shows how to use a lambda expression in a Windows Forms event handler. Notice that the types of the inputs (Object and MouseEventArgs) are inferred by the compiler and do not have to be explicitly given in the lambda input parameters.

Example

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        // Use a lambda expression to define an event handler.
       this.Click += (s, e) => { MessageBox.Show(((MouseEventArgs)e).Location.ToString());};
    }
}
```

See also

- Lambda Expressions
- Anonymous Methods
- Language Integrated Query (LINQ))

Anonymous Methods (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

In versions of C# before 2.0, the only way to declare a delegate was to use named methods. C# 2.0 introduced anonymous methods and in C# 3.0 and later, lambda expressions supersede anonymous methods as the preferred way to write inline code. However, the information about anonymous methods in this topic also applies to lambda expressions. There is one case in which an anonymous method provides functionality not found in lambda expressions. Anonymous methods enable you to omit the parameter list. This means that an anonymous method can be converted to delegates with a variety of signatures. This is not possible with lambda expressions. For more information specifically about lambda expressions, see Lambda Expressions.

Creating anonymous methods is essentially a way to pass a code block as a delegate parameter. Here are two examples:

```
// Create a handler for a click event.
button1.Click += delegate(System.Object o, System.EventArgs e)
                  { System.Windows.Forms.MessageBox.Show("Click!"); };
```

```
// Create a delegate.
delegate void Del(int x);

// Instantiate the delegate using an anonymous method.
Del d = delegate(int k) { /* ... */ };
```

By using anonymous methods, you reduce the coding overhead in instantiating delegates because you do not have to create a separate method.

For example, specifying a code block instead of a delegate can be useful in a situation when having to create a method might seem an unnecessary overhead. A good example would be when you start a new thread. This class creates a thread and also contains the code that the thread executes without creating an additional method for the delegate.

```
void StartThread()
{
    System.Threading.Thread t1 = new System.Threading.Thread
      (delegate()
            {
                System.Console.Write("Hello, ");
                System.Console.WriteLine("World!");
            });
    t1.Start();
}
```

Remarks

The scope of the parameters of an anonymous method is the *anonymous-method-block*.

It is an error to have a jump statement, such as goto, break, or continue, inside the anonymous method block if the target is outside the block. It is also an error to have a jump statement, such as `goto`, `break`, or `continue`, outside the anonymous method block if the target is inside the block.

The local variables and parameters whose scope contains an anonymous method declaration are called *outer*

variables of the anonymous method. For example, in the following code segment, `n` is an outer variable:

```
int n = 0;
Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };
```

A reference to the outer variable `n` is said to be *captured* when the delegate is created. Unlike local variables, the lifetime of a captured variable extends until the delegates that reference the anonymous methods are eligible for garbage collection.

An anonymous method cannot access the in, ref or out parameters of an outer scope.

No unsafe code can be accessed within the *anonymous-method-block*.

Anonymous methods are not allowed on the left side of the is operator.

Example

The following example demonstrates two ways of instantiating a delegate:

- Associating the delegate with an anonymous method.
- Associating the delegate with a named method (`DoWork`).

In each case, a message is displayed when the delegate is invoked.

```
// Declare a delegate.
delegate void Printer(string s);

class TestClass
{
    static void Main()
    {
        // Instantiate the delegate type using an anonymous method.
        Printer p = delegate(string j)
        {
            System.Console.WriteLine(j);
        };

        // Results from the anonymous delegate call.
        p("The delegate using the anonymous method is called.");

        // The delegate instantiation using a named method "DoWork".
        p = DoWork;

        // Results from the old style delegate call.
        p("The delegate using the named method is called.");
    }

    // The method associated with the named delegate.
    static void DoWork(string k)
    {
        System.Console.WriteLine(k);
    }
}
/* Output:
    The delegate using the anonymous method is called.
    The delegate using the named method is called.
*/
```

See also

- C# Reference
- C# Programming Guide
- Delegates
- Lambda Expressions
- Unsafe Code and Pointers
- Methods
- Delegates with Named vs. Anonymous Methods

Overloadable operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

C# allows user-defined types to overload operators by defining static member functions using the operator keyword. Not all operators can be overloaded, however, and others have restrictions, as listed in this table:

OPERATORS	OVERLOADABILITY
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded (but see the note that follows this table).
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using `&` and `|`, which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
(T)x	The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment operators cannot be explicitly overloaded. However, when you overload a binary operator, the corresponding assignment operator, if any, is also implicitly overloaded. For example, `+=` is evaluated using `+`, which can be overloaded.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	These operators cannot be overloaded.

NOTE

The comparison operators, if overloaded, must be overloaded in pairs; that is, if `==` is overloaded, `!=` must also be overloaded. The reverse is also true, where overloading `!=` requires an overload for `==`. The same is true for comparison operators `<` and `>` and for `<=` and `>=`.

For information about how to overload an operator, see the operator keyword article.

See also

- C# Programming Guide
- Statements, Expressions, and Operators
- Operators
- C# Operators
- Why are overloaded operators always static in C#?

Conversion operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

C# enables programmers to declare conversions on classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types. Conversions are defined like operators and are named for the type to which they convert. Either the type of the argument to be converted, or the type of the result of the conversion, but not both, must be the containing type.

```
class SampleClass
{
    public static explicit operator SampleClass(int i)
    {
        SampleClass temp = new SampleClass();
        // code to convert from int to SampleClass...

        return temp;
    }
}
```

Conversion operators overview

Conversion operators have the following properties:

- Conversions declared as `implicit` occur automatically when it is required.
- Conversions declared as `explicit` require a cast to be called.
- All conversions must be declared as `static`.

Related sections

For more information:

- Using Conversion Operators
- Casting and Type Conversions
- How to: Implement User-Defined Conversions Between Structs
- explicit
- implicit
- static

See also

- Convert
- C# Programming Guide
- Chained user-defined explicit conversions in C#

Using Conversion Operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can use `implicit` conversion operators, which are easier to use, or `explicit` conversion operators, which clearly indicate to anyone reading the code that you're converting a type. This topic demonstrates both types of conversion operator.

> **NOTE**
>
> For information about simple type conversions, see How to: Convert a String to a Number, How to: Convert a byte Array to an int, How to: Convert Between Hexadecimal Strings and Numeric Types, or Convert.

Example

This is an example of an explicit conversion operator. This operator converts from the type Byte to a value type called `Digit`. Because not all bytes can be converted to a digit, the conversion is explicit, meaning that a cast must be used, as shown in the `Main` method.

```
struct Digit
{
    byte value;

    public Digit(byte value)  //constructor
    {
        if (value > 9)
        {
            throw new System.ArgumentException();
        }
        this.value = value;
    }

    public static explicit operator Digit(byte b)  // explicit byte to digit conversion operator
    {
        Digit d = new Digit(b);  // explicit conversion

        System.Console.WriteLine("Conversion occurred.");
        return d;
    }
}

class TestExplicitConversion
{
    static void Main()
    {
        try
        {
            byte b = 3;
            Digit d = (Digit)b;  // explicit conversion
        }
        catch (System.Exception e)
        {
            System.Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
// Output: Conversion occurred.
```

Example

This example demonstrates an implicit conversion operator by defining a conversion operator that undoes what the previous example did: it converts from a value class called `Digit` to the integral Byte type. Because any digit can be converted to a Byte, there's no need to force users to be explicit about the conversion.

```
struct Digit
{
    byte value;

    public Digit(byte value)  //constructor
    {
        if (value > 9)
        {
            throw new System.ArgumentException();
        }
        this.value = value;
    }

    public static implicit operator byte(Digit d)  // implicit digit to byte conversion operator
    {
        System.Console.WriteLine("conversion occurred");
        return d.value;  // implicit conversion
    }
}

class TestImplicitConversion
{
    static void Main()
    {
        Digit d = new Digit(3);
        byte b = d;  // implicit conversion -- no cast needed
    }
}
// Output: Conversion occurred.
```

See also

- C# Reference
- C# Programming Guide
- Conversion Operators
- is

How to: Implement User-Defined Conversions Between Structs (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example defines two structs, `RomanNumeral` and `BinaryNumeral`, and demonstrates conversions between them.

Example

```
struct RomanNumeral
{
    private int value;

    public RomanNumeral(int value)  //constructor
    {
        this.value = value;
    }

    static public implicit operator RomanNumeral(int value)
    {
        return new RomanNumeral(value);
    }

    static public implicit operator RomanNumeral(BinaryNumeral binary)
    {
        return new RomanNumeral((int)binary);
    }

    static public explicit operator int(RomanNumeral roman)
    {
        return roman.value;
    }

    static public implicit operator string(RomanNumeral roman)
    {
        return ("Conversion to string is not implemented");
    }
}

struct BinaryNumeral
{
    private int value;

    public BinaryNumeral(int value)  //constructor
    {
        this.value = value;
    }

    static public implicit operator BinaryNumeral(int value)
    {
        return new BinaryNumeral(value);
    }

    static public explicit operator int(BinaryNumeral binary)
    {
        return (binary.value);
    }

    static public implicit operator string(BinaryNumeral binary)
    {
        return ("Conversion to string is not implemented");
```

```
    }
}

class TestConversions
{
    static void Main()
    {
        RomanNumeral roman;
        BinaryNumeral binary;

        roman = 10;

        // Perform a conversion from a RomanNumeral to a BinaryNumeral:
        binary = (BinaryNumeral)(int)roman;

        // Perform a conversion from a BinaryNumeral to a RomanNumeral:
        // No cast is required:
        roman = binary;

        System.Console.WriteLine((int)binary);
        System.Console.WriteLine(binary);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    10
    Conversion to string is not implemented
*/
```

Robust Programming

- In the previous example, the statement:

  ```
  binary = (BinaryNumeral)(int)roman;
  ```

  performs a conversion from a `RomanNumeral` to a `BinaryNumeral`. Because there is no direct conversion from `RomanNumeral` to `BinaryNumeral`, a cast is used to convert from a `RomanNumeral` to an `int`, and another cast to convert from an `int` to a `BinaryNumeral`.

- Also the statement

  ```
  roman = binary;
  ```

  performs a conversion from a `BinaryNumeral` to a `RomanNumeral`. Because `RomanNumeral` defines an implicit conversion from `BinaryNumeral`, no cast is required.

See also

- C# Reference
- C# Programming Guide
- Conversion Operators

default value expressions (C# programming guide)

1/23/2019 • 4 minutes to read • Edit Online

A default value expression `default(T)` produces the default value of a type `T`. The following table shows which values are produced for various types:

TYPE	DEFAULT VALUE
Any reference type	`null`
Numeric value type	Zero
bool	`false`
char	`'\0'`
enum	The value produced by the expression `(E)0`, where `E` is the enum identifier.
struct	The value produced by setting all value type fields to their default value and all reference type fields to `null`.
Nullable type	An instance for which the HasValue property is `false` and the Value property is undefined.

Default value expressions are particularly useful in generic classes and methods. One issue that arises using generics is how to assign a default value of a parameterized type `T` when you don't know the following in advance:

- Whether `T` is a reference type or a value type.
- If `T` is a value type, whether it's a numeric value or a struct.

Given a variable `t` of a parameterized type `T`, the statement `t = null` is only valid if `T` is a reference type. The assignment `t = 0` only works for numeric value types but not for structs. To solve that, use a default value expression:

```
T t = default(T);
```

The `default(T)` expression is not limited to generic classes and methods. Default value expressions can be used with any managed type. Any of these expressions are valid:

```
var s = default(string);
var d = default(dynamic);
var i = default(int);
var n = default(int?); // n is a Nullable int where HasValue is false.
```

The following example from the `GenericList<T>` class shows how to use the `default(T)` operator in a generic class. For more information, see Introduction to Generics.

```
namespace ConsoleApplication1
```

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Test with a non-empty list of integers.
            GenericList<int> gll = new GenericList<int>();
            gll.AddNode(5);
            gll.AddNode(4);
            gll.AddNode(3);
            int intVal = gll.GetLast();
            // The following line displays 5.
            System.Console.WriteLine(intVal);

            // Test with an empty list of integers.
            GenericList<int> gll2 = new GenericList<int>();
            intVal = gll2.GetLast();
            // The following line displays 0.
            System.Console.WriteLine(intVal);

            // Test with a non-empty list of strings.
            GenericList<string> gll3 = new GenericList<string>();
            gll3.AddNode("five");
            gll3.AddNode("four");
            string sVal = gll3.GetLast();
            // The following line displays five.
            System.Console.WriteLine(sVal);

            // Test with an empty list of strings.
            GenericList<string> gll4 = new GenericList<string>();
            sVal = gll4.GetLast();
            // The following line displays a blank line.
            System.Console.WriteLine(sVal);
        }
    }

    // T is the type of data stored in a particular instance of GenericList.
    public class GenericList<T>
    {
        private class Node
        {
            // Each node has a reference to the next node in the list.
            public Node Next;
            // Each node holds a value of type T.
            public T Data;
        }

        // The list is initially empty.
        private Node head = null;

        // Add a node at the beginning of the list with t as its data value.
        public void AddNode(T t)
        {
            Node newNode = new Node();
            newNode.Next = head;
            newNode.Data = t;
            head = newNode;
        }

        // The following method returns the data value stored in the last node in
        // the list. If the list is empty, the default value for type T is
        // returned.
        public T GetLast()
        {
            // The value of temp is returned as the value of the method.
            // The following declaration initializes temp to the appropriate
            // default value for type T. The default value is returned if the
            // list is empty.
```

```
            T temp = default(T);

            Node current = head;
            while (current != null)
            {
                temp = current.Data;
                current = current.Next;
            }
            return temp;
        }
    }
}
```

default literal and type inference

Beginning with C# 7.1, the `default` literal can be used for default value expressions when the compiler can infer the type of the expression. The `default` literal produces the same value as the equivalent `default(T)` where `T` is the inferred type. This can make code more concise by reducing the redundancy of declaring a type more than once. The `default` literal can be used in any of the following locations:

- variable initializer
- variable assignment
- declaring the default value for an optional parameter
- providing the value for a method call argument
- return statement (or expression in an expression bodied member)

The following example shows many usages of the `default` literal in a default value expression:

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
}

public class LabeledPoint
{
    public double X { get; private set; }
    public double Y { get; private set; }
    public string Label { get; set; }

    // Providing the value for a default argument:
    public LabeledPoint(double x, double y, string label = default)
    {
        X = x;
        Y = y;
        Label = label;
    }

    public static LabeledPoint MovePoint(LabeledPoint source,
        double xDistance, double yDistance)
    {
        // return a default value:
        if (source == null)
            return default;

        return new LabeledPoint(source.X + xDistance, source.Y + yDistance,
        source.Label);
    }

    public static LabeledPoint FindClosestLocation(IEnumerable<LabeledPoint> sequence,
        Point location)
    {
        // initialize variable:
        LabeledPoint rVal = default;
        double distance = double.MaxValue;

        foreach (var pt in sequence)
        {
            var thisDistance = Math.Sqrt((pt.X - location.X) * (pt.X - location.X) +
                (pt.Y - location.Y) * (pt.Y - location.Y));
            if (thisDistance < distance)
            {
                distance = thisDistance;
                rVal = pt;
            }
        }

        return rVal;
    }

    public static LabeledPoint ClosestToOrigin(IEnumerable<LabeledPoint> sequence)
        // Pass default value of an argument.
        => FindClosestLocation(sequence, default);
}
```

See also

- System.Collections.Generic
- C# Programming Guide
- Generics (C# Programming Guide)
- Generic Methods
- Generics in .NET
- Default values table

Equality Comparisons (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

It is sometimes necessary to compare two values for equality. In some cases, you are testing for *value equality*, also known as *equivalence*, which means that the values that are contained by the two variables are equal. In other cases, you have to determine whether two variables refer to the same underlying object in memory. This type of equality is called *reference equality*, or *identity*. This topic describes these two kinds of equality and provides links to other topics for more information.

Reference Equality

Reference equality means that two object references refer to the same underlying object. This can occur through simple assignment, as shown in the following example.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

In this code, two objects are created, but after the assignment statement, both references refer to the same object. Therefore they have reference equality. Use the ReferenceEquals method to determine whether two references refer to the same object.

The concept of reference equality applies only to reference types. Value type objects cannot have reference equality because when an instance of a value type is assigned to a variable, a copy of the value is made. Therefore you can never have two unboxed structs that refer to the same location in memory. Furthermore, if you use ReferenceEquals to compare two value types, the result will always be `false`, even if the values that are contained in the objects are all identical. This is because each variable is boxed into a separate object instance. For more information, see How to: Test for Reference Equality (Identity).

Value Equality

Value equality means that two objects contain the same value or values. For primitive value types such as int or bool, tests for value equality are straightforward. You can use the == operator, as shown in the following example.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if( b == a)
{
    // The two integers are equal.
}
```

For most other types, testing for value equality is more complex because it requires that you understand how the type defines it. For classes and structs that have multiple fields or properties, value equality is often defined to mean that all fields or properties have the same value. For example, two `Point` objects might be defined to be equivalent if pointA.X is equal to pointB.X and pointA.Y is equal to pointB.Y.

However, there is no requirement that equivalence be based on all the fields in a type. It can be based on a subset. When you compare types that you do not own, you should make sure to understand specifically how equivalence is defined for that type. For more information about how to define value equality in your own classes and structs, see How to: Define Value Equality for a Type.

Value Equality for Floating Point Values

Equality comparisons of floating point values (double and float) are problematic because of the imprecision of floating point arithmetic on binary computers. For more information, see the remarks in the topic System.Double.

Related Topics

TITLE	DESCRIPTION
How to: Test for Reference Equality (Identity)	Describes how to determine whether two variables have reference equality.
How to: Define Value Equality for a Type	Describes how to provide a custom definition of value equality for a type.
C# Programming Guide	Provides links to detailed information about important C# language features and features that are available to C# through the .NET Framework.
Types	Provides information about the C# type system and links to additional information.

See also

- C# Programming Guide

How to: Define Value Equality for a Type (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

When you define a class or struct, you decide whether it makes sense to create a custom definition of value equality (or equivalence) for the type. Typically, you implement value equality when objects of the type are expected to be added to a collection of some sort, or when their primary purpose is to store a set of fields or properties. You can base your definition of value equality on a comparison of all the fields and properties in the type, or you can base the definition on a subset. But in either case, and in both classes and structs, your implementation should follow the five guarantees of equivalence:

1. `x.Equals(x)` returns `true`. This is called the reflexive property.
2. `x.Equals(y)` returns the same value as `y.Equals(x)`. This is called the symmetric property.
3. if `(x.Equals(y) && y.Equals(z))` returns `true`, then `x.Equals(z)` returns `true`. This is called the transitive property.
4. Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by x and y are not modified.
5. `x.Equals(null)` returns `false`. However, `null.Equals(null)` throws an exception; it does not obey rule number two above.

Any struct that you define already has a default implementation of value equality that it inherits from the System.ValueType override of the Object.Equals(Object) method. This implementation uses reflection to examine all the fields and properties in the type. Although this implementation produces correct results, it is relatively slow compared to a custom implementation that you write specifically for the type.

The implementation details for value equality are different for classes and structs. However, both classes and structs require the same basic steps for implementing equality:

1. Override the virtual Object.Equals(Object) method. In most cases, your implementation of `bool Equals( object obj )` should just call into the type-specific `Equals` method that is the implementation of the System.IEquatable<T> interface. (See step 2.)
2. Implement the System.IEquatable<T> interface by providing a type-specific `Equals` method. This is where the actual equivalence comparison is performed. For example, you might decide to define equality by comparing only one or two fields in your type. Do not throw exceptions from `Equals`. For classes only: This method should examine only fields that are declared in the class. It should call `base.Equals` to examine fields that are in the base class. (Do not do this if the type inherits directly from Object, because the Object implementation of Object.Equals(Object) performs a reference equality check.)
3. Optional but recommended: Overload the == and != operators.
4. Override Object.GetHashCode so that two objects that have value equality produce the same hash code.
5. Optional: To support definitions for "greater than" or "less than," implement the IComparable<T> interface for your type, and also overload the <= and >= operators.

The first example that follows shows a class implementation. The second example shows a struct implementation.

Example

The following example shows how to implement value equality in a class (reference type).

```
namespace ValueEquality
{
    using System;
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        // Readonly auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        // Set the properties in the constructor.
        public TwoDPoint(int x, int y)
        {
            if ((x < 1) || (x > 2000) || (y < 1) || (y > 2000))
            {
                throw new System.ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj)
        {
            return this.Equals(obj as TwoDPoint);
        }

        public bool Equals(TwoDPoint p)
        {
            // If parameter is null, return false.
            if (Object.ReferenceEquals(p, null))
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }

        public override int GetHashCode()
        {
            return X * 0x00010000 + Y;
        }

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
        {
            // Check for null on left side.
            if (Object.ReferenceEquals(lhs, null))
            {
```

```
            if (Object.ReferenceEquals(rhs, null))
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs == rhs);
    }
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new System.ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj)
    {
        return this.Equals(obj as ThreeDPoint);
    }

    public bool Equals(ThreeDPoint p)
    {
        // If parameter is null, return false.
        if (Object.ReferenceEquals(p, null))
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode()
    {
        return (X * 0x100000) + (Y * 0x1000) + Z;
```

```
        }

        public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            // Check for null.
            if (Object.ReferenceEquals(lhs, null))
            {
                if (Object.ReferenceEquals(rhs, null))
                {
                    // null == null = true.
                    return true;
                }

                // Only the left side is null.
                return false;
            }
            // Equals handles the case of null on right side.
            return lhs.Equals(rhs);
        }

        public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            return !(lhs == rhs);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
            ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
            ThreeDPoint pointC = null;
            int i = 5;

            Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
            Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

            TwoDPoint pointD = null;
            TwoDPoint pointE = null;



            Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

            pointE = new TwoDPoint(3, 4);
            Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
            Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
            Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

            System.Collections.ArrayList list = new System.Collections.ArrayList();
            list.Add(new ThreeDPoint(3, 4, 5));
            Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        null comparison = False
        Compare to some other type = False
        Two null TwoDPoints are equal: True
        (pointE == pointA) = False
```

```
        (pointA == pointE) = False
        (pointA != pointE) = True
        pointE.Equals(list[0]): False
    */
}
```

On classes (reference types), the default implementation of both Object.Equals(Object) methods performs a reference equality comparison, not a value equality check. When an implementer overrides the virtual method, the purpose is to give it value equality semantics.

The `==` and `!=` operators can be used with classes even if the class does not overload them. However, the default behavior is to perform a reference equality check. In a class, if you overload the `Equals` method, you should overload the `==` and `!=` operators, but it is not required.

Example

The following example shows how to implement value equality in a struct (value type):

```
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        // Read/write auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            X = x;
            Y = x;
        }

        public override bool Equals(object obj)
        {
            if (obj is TwoDPoint)
            {
                return this.Equals((TwoDPoint)obj);
            }
            return false;
        }

        public bool Equals(TwoDPoint p)
        {
            return (X == p.X) && (Y == p.Y);
        }

        public override int GetHashCode()
        {
            return X ^ Y;
        }

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
        {
            return lhs.Equals(rhs);
        }

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
        {
            return !(lhs.Equals(rhs));
        }
    }


    class Program
    {
        static void Main(string[] args)
```

```
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // Compare using virtual Equals, static Equals, and == and != operators.
            // True:
            Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
            // True:
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            // True:
            Console.WriteLine("Object.Equals(pointA, pointB) = {0}", Object.Equals(pointA, pointB));
            // False:
            Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
            // False:
            Console.WriteLine("(pointA == null) = {0}", pointA == null);
            // True:
            Console.WriteLine("(pointA != null) = {0}", pointA != null);
            // False:
            Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
            // CS0019:
            // Console.WriteLine("pointA == i = {0}", pointA == i);

            // Compare unboxed to boxed.
            System.Collections.ArrayList list = new System.Collections.ArrayList();
            list.Add(new TwoDPoint(3, 4));
            // True:
            Console.WriteLine("pointE.Equals(list[0]): {0}", pointA.Equals(list[0]));


            // Compare nullable to nullable and to non-nullable.
            TwoDPoint? pointC = null;
            TwoDPoint? pointD = null;
            // False:
            Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
            // True:
            Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

            TwoDPoint temp = new TwoDPoint(3, 4);
            pointC = temp;
            // True:
            Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

            pointD = temp;
            // True:
            Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        Object.Equals(pointA, pointB) = True
        pointA.Equals(null) = False
        (pointA == null) = False
        (pointA != null) = True
        pointA.Equals(i) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

For structs, the default implementation of Object.Equals(Object) (which is the overridden version in System.ValueType) performs a value equality check by using reflection to compare the values of every field in the type. When an implementer overrides the virtual `Equals` method in a struct, the purpose is to provide a more efficient means of performing the value equality check and optionally to base the comparison on some subset of the struct's field or properties.

The == and != operators cannot operate on a struct unless the struct explicitly overloads them.

See also

- Equality Comparisons
- C# Programming Guide

How to: Test for Reference Equality (Identity) (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

You do not have to implement any custom logic to support reference equality comparisons in your types. This functionality is provided for all types by the static Object.ReferenceEquals method.

The following example shows how to determine whether two variables have *reference equality*, which means that they refer to the same object in memory.

The example also shows why Object.ReferenceEquals always returns `false` for value types and why you should not use ReferenceEquals to determine string equality.

Example

```
namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After asignment: ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
```

```
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After asignment: ReferenceEquals(tsC, tsD) = {0}",
                              Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned by the runtime.
            // This means they are stored in the same location in memory. Therefore,
            // the two strings have reference equality although no assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                            Object.ReferenceEquals(strA, strB)); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
                           Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                           Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After asignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After asignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
*/
```

The implementation of `Equals` in the System.Object universal base class also performs a reference equality check, but it is best not to use this because, if a class happens to override the method, the results might not be what you expect. The same is true for the `==` and `!=` operators. When they are operating on reference types, the default behavior of `==` and `!=` is to perform a reference equality check. However, derived classes can overload the operator to perform a value equality check. To minimize the potential for error, it is best to always use

ReferenceEquals when you have to determine whether two objects have reference equality.

Constant strings within the same assembly are always interned by the runtime. That is, only one instance of each unique literal string is maintained. However, the runtime does not guarantee that strings created at runtime are interned, nor does it guarantee that two equal constant strings in different assemblies are interned.

See also

- Equality Comparisons

Contents

Types (C# Programming Guide)

2/5/2019 • 11 minutes to read • Edit Online

Types, Variables, and Values

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> C and C++ developers, notice that in C#, bool is not convertible to int.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying Types in Variable Declarations

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an int as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an int and then assign it a Boolean value of true. However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see Casting and Type Conversions.

Built-in Types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in `string` and `object` types. These are available for you to use in any C# program. For more information about the built-in types, see Reference Tables for Types.

Custom Types

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see .NET Class Library.

The Common Type System

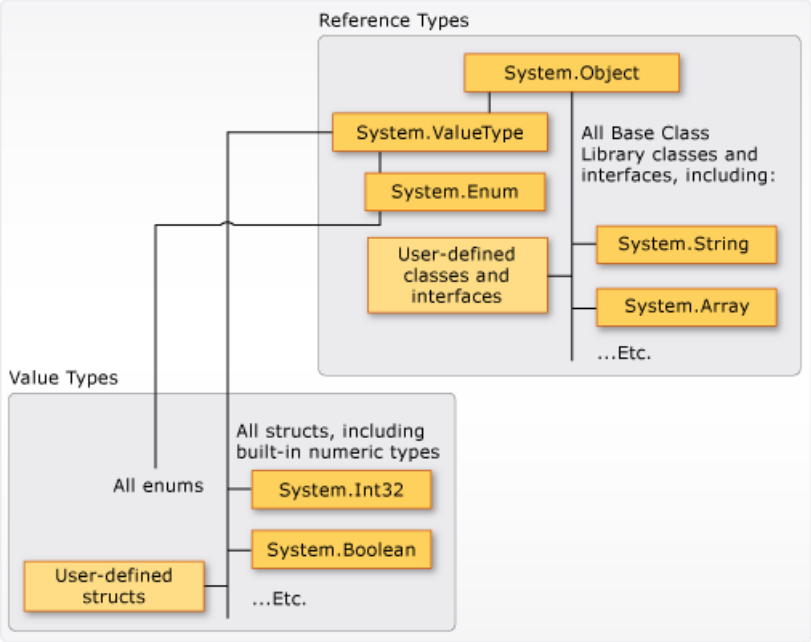
It is important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of

both base types in its inheritance hierarchy. All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS). For more information about inheritance in C#, see Inheritance.

- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



Value types and reference types in the CTS

> **NOTE**
>
> You can see that the most commonly used types are all organized in the System namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value Types

Value types derive from System.ValueType, which derives from System.Object. Types that derive from System.ValueType have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There is no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: struct and enum.

The built-in numeric types are structs, and they have properties and methods that you can access:

```
// Static method on type byte.
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they were simple non-aggregate types:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Value types are *sealed*, which means, for example, that you cannot derive a type from System.Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from System.ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a System.Object or any interface type as an input parameter. For more information, see Boxing and Unboxing.

You use the struct keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see Structs. For more information about value types in .NET, see Value Types.

The other category of value types is enum. An enum defines a set of named integral constants. For example, the System.IO.FileMode enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It is defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it is better to use enumerations instead of constant literal numbers. For more information, see System.IO.FileMode.

All enums inherit from System.Enum, which inherits from System.ValueType. All the rules that apply to structs also apply to enums. For more information about enums, see Enumeration Types.

Reference Types

A type that is defined as a class, delegate, array, or interface is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an object by using the new operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it does not create a performance issue. For more information about garbage collection, see Automatic Memory Management.

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the System.Array class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see Classes and Structs. For more information about inheritance and virtual methods, see Inheritance.

Types of Literal Values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see the reference pages for individual types in Value Types.

Because literals are typed, and all types derive ultimately from System.Object, you can write and compile code such as the following:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Generic Types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type System.Collections.Generic.List<T> has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to object. Generic collection classes are called *strongly-typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see Generics.

Implicit Types, Anonymous Types, and Nullable Types

As stated previously, you can implicitly type a local variable (but not class members) by using the var keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see Implicitly Typed Local Variables.

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see Anonymous Types.

Ordinary value types cannot have a value of null. However, you can create nullable value types by affixing a `?` after the type. For example, `int?` is an `int` type that can also have the value null. In the CTS, nullable types are instances of the generic struct type System.Nullable<T>. Nullable types are especially useful when you are passing data to and from databases in which numeric values might be null. For more information, see Nullable Types.

Related Sections

For more information, see the following topics:

- Casting and Type Conversions
- Boxing and Unboxing
- Using Type dynamic
- Value Types
- Reference Types
- Classes and Structs
- Anonymous Types
- Generics

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Conversion of XML Data Types

- Integral Types Table

Casting and type conversions (C# Programming Guide)

2/3/2019 • 4 minutes to read • Edit Online

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or assigned a value of another type unless that type is implicitly convertible to the variable's type. For example, the `string` cannot be implicitly converted to `int`. Therefore, after you declare `i` as an `int`, you cannot assign the string "Hello" to it, as the following code shows:

```
int i;
i = "Hello"; // error CS0029: Cannot implicitly convert type 'string' to 'int'
```

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as `double`. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called *type conversions*. In C#, you can perform the following kinds of conversions:

- **Implicit conversions**: No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts)**: Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined conversions**: User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship. For more information, see Conversion Operators.
- **Conversions with helper classes**: To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the `Parse` methods of the built-in numeric types, such as Int32.Parse. For more information, see How to: Convert a byte Array to an int, How to: Convert a String to a Number, and How to: Convert Between Hexadecimal Strings and Numeric Types.

Implicit conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For example, a variable of type long (64-bit integer) can store any value that an int (32-bit integer) can store. In the following example, the compiler implicitly converts the value of `num` on the right to a type `long` before assigning it to `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

For a complete list of all implicit numeric conversions, see Implicit Numeric Conversions Table.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

```
Derived d = new Derived();
Base b = d; // Always OK.
```

Explicit conversions

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a double to an int. The program will not compile without the cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

For a list of the explicit numeric conversions that are allowed, see Explicit Numeric Conversions Table.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see Polymorphism.

Type conversion exceptions at run time

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an InvalidCastException to be thrown.

```
using System;

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
    }

}
```

C# provides the is and as operators to enable you to test for compatibility before actually performing a cast. For more information, see How to: Safely cast using pattern matching, as and is Operators.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Types
- () Operator
- explicit
- implicit
- Conversion Operators
- Generalized Type Conversion
- How to: Convert a String to a Number

Boxing and Unboxing (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

Boxing is the process of converting a value type to the type `object` or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

In the following example, the integer variable `i` is *boxed* and assigned to object `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

The object `o` can then be unboxed and assigned to integer variable `i`:

```
o = 123;
i = (int)o;  // unboxing
```

The following examples illustrate how boxing is used in C#.

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));


// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
```

```
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

Performance

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see Performance.

Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type `object` or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

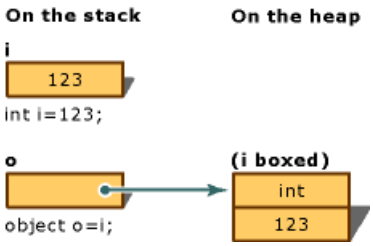
Consider the following declaration of a value-type variable:

```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable `i`:

```
// Boxing copies the value of i into object o.
object o = i;
```

The result of this statement is creating an object reference `o`, on the stack, that references a value of the type `int`, on the heap. This value is a copy of the value-type value assigned to the variable `i`. The difference between the two variables, `i` and `o`, is illustrated in the following figure.



Boxing Conversion

It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Description

This example converts an integer variable `i` to an object `o` by using boxing. Then, the value stored in the variable `i` is changed from `123` to `456`. The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

Example

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Unboxing

Unboxing is an explicit conversion from the type `object` to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

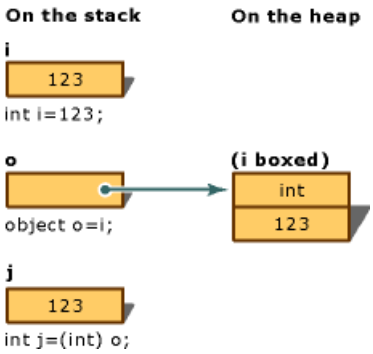
- Checking the object instance to make sure that it is a boxed value of the given value type.

- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

The following figure demonstrates the result of the previous statements.



Unboxing Conversion

For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox `null` causes a NullReferenceException. Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.

Example

The following example demonstrates a case of invalid unboxing and the resulting `InvalidCastException`. Using `try` and `catch`, an error message is displayed when the error occurs.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

This program outputs:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

If you change the statement:

```
int j = (short) o;
```

to:

```
int j = (int) o;
```

the conversion will be performed, and you will get the output:

```
Unboxing OK.
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Related Sections

For more information:

- Reference Types
- Value Types

See also

- C# Programming Guide

Using type dynamic (C# Programming Guide)

12/11/2018 • 5 minutes to read • Edit Online

C# 4 introduces a new type, `dynamic`. The type is a static type, but an object of type `dynamic` bypasses static type checking. In most cases, it functions like it has type `object`. At compile time, an element that is typed as `dynamic` is assumed to support any operation. Therefore, you do not have to be concerned about whether the object gets its value from a COM API, from a dynamic language such as IronPython, from the HTML Document Object Model (DOM), from reflection, or from somewhere else in the program. However, if the code is not valid, errors are caught at run time.

For example, if instance method `exampleMethod1` in the following code has only one parameter, the compiler recognizes that the first call to the method, `ec.exampleMethod1(10, 4)`, is not valid because it contains two arguments. The call causes a compiler error. The second call to the method, `dynamic_ec.exampleMethod1(10, 4)`, is not checked by the compiler because the type of `dynamic_ec` is `dynamic`. Therefore, no compiler error is reported. However, the error does not escape notice indefinitely. It is caught at run time and causes a run-time exception.

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

The role of the compiler in these examples is to package together information about what each statement is proposing to do to the object or expression that is typed as `dynamic`. At run time, the stored information is examined, and any statement that is not valid causes a run-time exception.

The result of most dynamic operations is itself `dynamic`. For example, if you rest the mouse pointer over the use of `testSum` in the following example, IntelliSense displays the type **(local variable) dynamic testSum**.

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Operations in which the result is not `dynamic` include:

- Conversions from `dynamic` to another type.
- Constructor calls that include arguments of type `dynamic`.

For example, the type of `testInstance` in the following declaration is `ExampleClass`, not `dynamic`:

```
var testInstance = new ExampleClass(d);
```

Conversion examples are shown in the following section, "Conversions."

Conversions

Conversions between dynamic objects and other types are easy. This enables the developer to switch between dynamic and non-dynamic behavior.

Any object can be converted to dynamic type implicitly, as shown in the following examples.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Conversely, an implicit conversion can be dynamically applied to any expression of type `dynamic`.

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Overload resolution with arguments of type dynamic

Overload resolution occurs at run time instead of at compile time if one or more of the arguments in a method call have the type `dynamic`, or if the receiver of the method call is of type `dynamic`. In the following example, if the only accessible `exampleMethod2` method is defined to take a string argument, sending `d1` as the argument does not cause a compiler error, but it does cause a run-time exception. Overload resolution fails at run time because the run-time type of `d1` is `int`, and `exampleMethod2` requires a string.

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic language runtime

The dynamic language runtime (DLR) is a new API in .NET Framework 4. It provides the infrastructure that supports the `dynamic` type in C#, and also the implementation of dynamic programming languages such as IronPython and IronRuby. For more information about the DLR, see Dynamic Language Runtime Overview.

COM interop

C# 4 includes several features that improve the experience of interoperating with COM APIs such as the Office Automation APIs. Among the improvements are the use of the `dynamic` type, and of named and optional arguments.

Many COM methods allow for variation in argument types and return type by designating the types as `object`. This has necessitated explicit casting of the values to coordinate with strongly typed variables in C#. If you compile by using the /link (C# Compiler Options) option, the introduction of the `dynamic` type enables you to treat the occurrences of `object` in COM signatures as if they were of type `dynamic`, and thereby to avoid much of the casting. For example, the following statements contrast how you access a cell in a Microsoft Office Excel spreadsheet with the `dynamic` type and without the `dynamic` type.

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Related topics

TITLE	DESCRIPTION
dynamic	Describes the usage of the `dynamic` keyword.
Dynamic Language Runtime Overview	Provides an overview of the DLR, which is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR).
Walkthrough: Creating and Using Dynamic Objects	Provides step-by-step instructions for creating a custom dynamic object and for creating a project that accesses an `IronPython` library.
How to: Access Office Interop Objects by Using Visual C# Features	Demonstrates how to create a project that uses named and optional arguments, the `dynamic` type, and other enhancements that simplify access to Office API objects.

Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

2/12/2019 • 11 minutes to read • Edit Online

Dynamic objects expose members such as properties and methods at run time, instead of at compile time. This enables you to create objects to work with structures that do not match a static type or format. For example, you can use a dynamic object to reference the HTML Document Object Model (DOM), which can contain any combination of valid HTML markup elements and attributes. Because each HTML document is unique, the members for a particular HTML document are determined at run time. A common method to reference an attribute of an HTML element is to pass the name of the attribute to the `GetProperty` method of the element. To reference the `id` attribute of the HTML element `<div id="Div1">`, you first obtain a reference to the `<div>` element, and then use `divElement.GetProperty("id")`. If you use a dynamic object, you can reference the `id` attribute as `divElement.id`.

Dynamic objects also provide convenient access to dynamic languages such as IronPython and IronRuby. You can use a dynamic object to refer to a dynamic script that is interpreted at run time.

You reference a dynamic object by using late binding. In C#, you specify the type of a late-bound object as `dynamic`. In Visual Basic, you specify the type of a late-bound object as `Object`. For more information, see dynamic and Early and Late Binding.

You can create custom dynamic objects by using the classes in the System.Dynamic namespace. For example, you can create an ExpandoObject and specify the members of that object at run time. You can also create your own type that inherits the DynamicObject class. You can then override the members of the DynamicObject class to provide run-time dynamic functionality.

In this walkthrough you will perform the following tasks:

- Create a custom object that dynamically exposes the contents of a text file as properties of an object.
- Create a project that uses an `IronPython` library.

Prerequisites

You need IronPython for .NET to complete this walkthrough. Go to their Download page to obtain the latest version.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Creating a Custom Dynamic Object

The first project that you create in this walkthrough defines a custom dynamic object that searches the contents of a text file. Text to search for is specified by the name of a dynamic property. For example, if calling code specifies `dynamicFile.Sample`, the dynamic class returns a generic list of strings that contains all of the lines from the file that begin with "Sample". The search is case-insensitive. The dynamic class also supports two optional arguments. The first argument is a search option enum value that specifies that the dynamic class should search for matches at the

start of the line, the end of the line, or anywhere in the line. The second argument specifies that the dynamic class should trim leading and trailing spaces from each line before searching. For example, if calling code specifies `dynamicFile.Sample(StringSearchOption.Contains)`, the dynamic class searches for "Sample" anywhere in a line. If calling code specifies `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, the dynamic class searches for "Sample" at the start of each line, and does not remove leading and trailing spaces. The default behavior of the dynamic class is to search for a match at the start of each line and to remove leading and trailing spaces.

To create a custom dynamic class

1. Start Visual Studio.
2. On the **File** menu, point to **New** and then click **Project**.
3. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type `DynamicSample`, and then click **OK**. The new project is created.
4. Right-click the DynamicSample project and point to **Add**, and then click **Class**. In the **Name** box, type `ReadOnlyFile`, and then click **OK**. A new file is added that contains the ReadOnlyFile class.
5. At the top of the ReadOnlyFile.cs or ReadOnlyFile.vb file, add the following code to import the System.IO and System.Dynamic namespaces.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

6. The custom dynamic object uses an enum to determine the search criteria. Before the class statement, add the following enum definition.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

7. Update the class statement to inherit the `DynamicObject` class, as shown in the following code example.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

8. Add the following code to the `ReadOnlyFile` class to define a private field for the file path and a constructor for the `ReadOnlyFile` class.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

9. Add the following `GetPropertyValue` method to the `ReadOnlyFile` class. The `GetPropertyValue` method takes, as input, search criteria and returns the lines from a text file that match that search criteria. The dynamic methods provided by the `ReadOnlyFile` class call the `GetPropertyValue` method to retrieve their respective results.

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

```
Public Function GetPropertyValue(ByVal propertyName As String,
                                 Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                                 Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function
```

10. After the `GetPropertyValue` method, add the following code to override the TryGetMember method of the DynamicObject class. The TryGetMember method is called when a member of a dynamic class is requested and no arguments are specified. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The TryGetMember method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

```
' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function
```

11. After the `TryGetMember` method, add the following code to override the TryInvokeMember method of the DynamicObject class. The TryInvokeMember method is called when a member of a dynamic class is requested with arguments. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `args` argument contains an array of the arguments that are passed to the member. The TryInvokeMember method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

    The custom version of the `TryInvokeMember` method expects the first argument to be a value from the `StringSearchOption` enum that you defined in a previous step. The `TryInvokeMember` method expects the second argument to be a Boolean value. If one or both arguments are valid values, they are passed to the `GetPropertyValue` method to retrieve the results.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```
' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                          ByVal args() As Object,
                                          ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function
```

12. Save and close the file.

To create a sample text file

1. Right-click the DynamicSample project and point to **Add**, and then click **New Item**. In the **Installed Templates** pane, select **General**, and then select the **Text File** template. Leave the default name of TextFile1.txt in the **Name** box, and then click **Add**. A new text file is added to the project.

2. Copy the following text to the TextFile1.txt file.

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

3. Save and close the file.

To create a sample application that uses the custom dynamic object

1. In **Solution Explorer**, double-click the Module1.vb file if you are using Visual Basic or the Program.cs file if you are using Visual C#.

2. Add the following code to the Main procedure to create an instance of the `ReadOnlyFile` class for the TextFile1.txt file. The code uses late binding to call dynamic members and retrieve lines of text that contain the string "Customer".

```
dynamic rFile = new ReadOnlyFile(@"..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

```
Dim rFile As Object = New ReadOnlyFile("..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("----------------------------")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Save the file and press CTRL+F5 to build and run the application.

Calling a Dynamic Language Library

The next project that you create in this walkthrough accesses a library that is written in the dynamic language IronPython.

To create a custom dynamic class

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type `DynamicIronPythonSample`, and then click **OK**. The new project is created.
3. If you are using Visual Basic, right-click the DynamicIronPythonSample project and then click **Properties**. Click the **References** tab. Click the **Add** button. If you are using Visual C#, in **Solution Explorer**, right-click the **References** folder and then click **Add Reference**.
4. On the **Browse** tab, browse to the folder where the IronPython libraries are installed. For example, C:\Program Files\IronPython 2.6 for .NET 4.0. Select the **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll**, and **Microsoft.Dynamic.dll** libraries. Click **OK**.
5. If you are using Visual Basic, edit the Module1.vb file. If you are using Visual C#, edit the Program.cs file.
6. At the top of the file, add the following code to import the `Microsoft.Scripting.Hosting` and `IronPython.Hosting` namespaces from the IronPython libraries.

```
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
```

7. In the Main method, add the following code to create a new `Microsoft.Scripting.Hosting.ScriptRuntime` object to host the IronPython libraries. The `ScriptRuntime` object loads the IronPython library module random.py.

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.6 for .NET 4.0\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.
My.Computer.FileSystem.CurrentDirectory =
   My.Computer.FileSystem.SpecialDirectories.ProgramFiles &
   "\IronPython 2.6 for .NET 4.0\Lib"

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

8. After the code to load the random.py module, add the following code to create an array of integers. The array is passed to the `shuffle` method of the random.py module, which randomly sorts the values in the array.

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("--------------------");
}
```

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("--------------------")
Next
```

9. Save the file and press CTRL+F5 to build and run the application.

See also

- System.Dynamic
- System.Dynamic.DynamicObject

- Using Type dynamic
- Early and Late Binding
- dynamic
- Implementing Dynamic Interfaces (downloadable PDF from Microsoft TechNet)

How to: Convert a byte Array to an int (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example shows you how to use the BitConverter class to convert an array of bytes to an int and back to an array of bytes. You may have to convert from bytes to a built-in data type after you read bytes off the network, for example. In addition to the ToInt32(Byte[], Int32) method in the example, the following table lists methods in the BitConverter class that convert bytes (from an array of bytes) to other built-in types.

TYPE RETURNED	METHOD
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Example

This example initializes an array of bytes, reverses the array if the computer architecture is little-endian (that is, the least significant byte is stored first), and then calls the ToInt32(Byte[], Int32) method to convert four bytes in the array to an `int`. The second argument to ToInt32(Byte[], Int32) specifies the start index of the array of bytes.

> **NOTE**
>
> The output may differ depending on the endianess of your computer's architecture.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Example

In this example, the GetBytes(Int32) method of the BitConverter class is called to convert an `int` to an array of bytes.

> **NOTE**
> The output may differ depending on the endianess of your computer's architecture.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

See also

- BitConverter
- IsLittleEndian
- Types

How to: Convert a String to a Number (C# Programming Guide)

2/13/2019 • 4 minutes to read • Edit Online

You can convert a string to a number by calling the `Parse` or `TryParse` method found on the various numeric types (`int` , `long` , `double` , etc.), or by using methods in the System.Convert class.

If you have a string, it is slightly more efficient and straightforward to call a `TryParse` method (for example, `int.TryParse("11", out number)`) or `Parse` method (for example, `var number = int.Parse("11")`). Using a Convert method is more useful for general objects that implement IConvertible.

You can use `Parse` or `TryParse` methods on the numeric type you expect the string contains, such as the System.Int32 type. The Convert.ToInt32 method uses Parse internally. The `Parse` method returns the converted number; the `TryParse` method returns a Boolean value that indicates whether the conversion succeeded, and returns the converted number in an `out` parameter. If the string is not in a valid format, `Parse` throws an exception, whereas `TryParse` returns false. When calling a `Parse` method, you should always use exception handling to catch a FormatException in the event that the parse operation fails.

Calling the Parse and TryParse methods

The `Parse` and `TryParse` methods ignore white space at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (`int` , `long` , `ulong` , `float` , `decimal` , etc.). Any white space within the string that forms the number causes an error. For example, you can use `decimal.TryParse` to parse "10", "10.3", or " 10 ", but you cannot use this method to parse 10 from "10X", "1 0" (note the embedded space), "10 .3" (note the embedded space), "10e1" (`float.TryParse` works here), and so on. In addition, a string whose value is `null` or String.Empty fails to parse successfully. You can check for a null or empty string before attempting to parse it by calling the String.IsNullOrEmpty method.

The following example demonstrates both successful and unsuccessful calls to `Parse` and `TryParse` .

```
using System;

public class StringConversion
{
    public static void Main()
    {
       string input = String.Empty;
       try
       {
           int result = Int32.Parse(input);
           Console.WriteLine(result);
       }
       catch (FormatException)
       {
           Console.WriteLine($"Unable to parse '{input}'");
       }
       // Output: Unable to parse ''

       try
       {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
       }
       catch (FormatException e)
       {
           Console.WriteLine(e.Message);
       }
       // Output: -105

        if (Int32.TryParse("-105", out int j))
            Console.WriteLine(j);
        else
            Console.WriteLine("String could not be parsed.");
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
            Console.WriteLine(inputString);
        else
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        // Output: Int32.TryParse could not parse 'abc' to an int.
     }
}
```

The following example illustrates one a approach to parsing a string that is expected to include leading numeric characters (including hexadecimal characters) and trailing non-numeric characters. It assigns valid characters from the beginning of a string to a new string before calling the TryParse method. Because the strings to be parsed contain a small number of characters, the example calls the String.Concat method to assign valid characters to a new string. For a larger string, the StringBuilder class can be used instead.

```
using System;

public class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = String.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (Char.ToUpperInvariant(c) >= 'A' && Char.ToUpperInvariant(c) <= 'F')
|| c == ' ') {
                numericString = String.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }
        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
            // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str) {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = String.Concat(numericString, c);
            } else
                break;
        }
        if (int.TryParse(numericString, out int j))
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
            // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Calling the Convert methods

The following table lists some of the methods from the Convert class that you can use to convert a string to a number.

NUMERIC TYPE	METHOD
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)

NUMERIC TYPE	METHOD
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

The following example calls the Convert.ToInt32(String) method to convert an input string to an int. The example catches the two most common exceptions that can be thrown by this method, FormatException and OverflowException. If the resulting number can be incremented without exceeding Int32.MaxValue, the example adds 1 to the result and displays the output.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between –2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
           }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
//   The new value is -999
//   Go again? Y/N: n
```

See also

- Types
- How to: Determine Whether a String Represents a Numeric Value
- .NET Framework 4 Formatting Utility

How to: Convert Between Hexadecimal Strings and Numeric Types (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

These examples show you how to perform the following tasks:

- Obtain the hexadecimal value of each character in a string.
- Obtain the char that corresponds to each value in a hexadecimal string.
- Convert a hexadecimal `string` to an int.
- Convert a hexadecimal `string` to a float.
- Convert a byte array to a hexadecimal `string`.

Example

This example outputs the hexadecimal value of each character in a `string`. First it parses the `string` to an array of characters. Then it calls ToInt32(Char) on each character to obtain its numeric value. Finally, it formats the number as its hexadecimal representation in a `string`.

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

Example

This example parses a `string` of hexadecimal values and outputs the character corresponding to each hexadecimal value. First it calls the Split(Char[]) method to obtain each hexadecimal value as an individual `string` in an array. Then it calls ToInt32(String, Int32) to convert the hexadecimal value to a decimal value represented as an int. It shows two different ways to obtain the character corresponding to that character code. The first technique uses ConvertFromUtf32(Int32), which returns the character corresponding to the integer argument as a `string`. The second technique explicitly casts the `int` to a char.

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

Example

This example shows another way to convert a hexadecimal `string` to an integer, by calling the Parse(String, NumberStyles) method.

```
string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

Example

The following example shows how to convert a hexadecimal `string` to a float by using the System.BitConverter class and the UInt32.Parse method.

```
string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

Example

The following example shows how to convert a byte array to a hexadecimal string by using the System.BitConverter class.

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

See also

- Standard Numeric Format Strings
- Types
- How to: Determine Whether a String Represents a Numeric Value

Contents

Classes and Structs (C# Programming Guide)

1/23/2019 • 6 minutes to read • Edit Online

Classes and structs are two of the basic constructs of the common type system in the .NET Framework. Each is essentially a data structure that encapsulates a set of data and behaviors that belong together as a logical unit. The data and behaviors are the *members* of the class or struct, and they include its methods, properties, and events, and so on, as listed later in this topic.

A class or struct declaration is like a blueprint that is used to create instances or objects at run time. If you define a class or struct called `Person`, `Person` is the name of the type. If you declare and initialize a variable `p` of type `Person`, `p` is said to be an object or instance of `Person`. Multiple instances of the same `Person` type can be created, and each instance can have different values in its properties and fields.

A class is a reference type. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

A struct is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

In general, classes are used to model more complex behavior, or data that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that is not intended to be modified after the struct is created.

For more information, see Classes, Objects, and Structs.

Example

In the following example, `CustomClass` in the `ProgrammingGuide` namespace has three members: an instance constructor, a property named `Number`, and a method named `Multiply`. The `Main` method in the `Program` class creates an instance (object) of `CustomClass`, and the object's method and property are accessed by using dot notation.

```
using System;

namespace ProgrammingGuide
{
   // Class definition.
   public class CustomClass
   {
      // Class members.
      //
      // Property.
      public int Number { get; set; }

      // Method.
      public int Multiply(int num)
      {
          return num * Number;
      }

      // Instance Constructor.
      public CustomClass()
      {
          Number = 0;
      }
   }

   // Another class definition that contains Main, the program entry point.
   class Program
   {
      static void Main(string[] args)
      {
         // Create an object of type CustomClass.
         CustomClass custClass = new CustomClass();

         // Set the value of the public property.
         custClass.Number = 27;

         // Call the public method.
         int result = custClass.Multiply(4);
         Console.WriteLine($"The result is {result}.");
      }
   }
}
// The example displays the following output:
//      The result is 108.
```

Encapsulation

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. According to the principle of encapsulation, a class or struct can specify how accessible each of its members is to code outside of the class or struct. Methods and variables that are not intended to be used from outside of the class or assembly can be hidden to limit the potential for coding errors or malicious exploits.

For more information about classes, see Classes and Objects.

Members

All methods, fields, constants, properties, and events must be declared within a type; these are called the *members* of the type. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the `Main` method, must be declared within a class or struct. The following list includes all the various kinds of members that may be declared in a class or struct.

- Fields
- Constants

- Properties
- Methods
- Constructors
- Events
- Finalizers
- Indexers
- Operators
- Nested Types

Accessibility

Some methods and properties are meant to be called or accessed from code outside your class or struct, known as *client code*. Other methods and properties might be only for use in the class or struct itself. It is important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the access modifiers public, protected, internal, protected internal, private and private protected. The default accessibility is `private`. For more information, see Access Modifiers.

Inheritance

Classes (but not structs) support the concept of inheritance. A class that derives from another class (the *base class*) automatically contains all the public, protected, and internal members of the base class except its constructors and finalizers. For more information, see Inheritance and Polymorphism.

Classes may be declared as abstract, which means that one or more of their methods have no implementation. Although abstract classes cannot be instantiated directly, they can serve as base classes for other classes that provide the missing implementation. Classes can also be declared as sealed to prevent other classes from inheriting from them. For more information, see Abstract and Sealed Classes and Class Members.

Interfaces

Classes and structs can inherit multiple interfaces. To inherit from an interface means that the type implements all the methods defined in the interface. For more information, see Interfaces.

Generic Types

Classes and structs can be defined with one or more type parameters. Client code supplies the type when it creates an instance of the type. For example The List<T> class in the System.Collections.Generic namespace is defined with one type parameter. Client code creates an instance of a `List<string>` or `List<int>` to specify the type that the list will hold. For more information, see Generics.

Static Types

Classes (but not structs) can be declared as static. A static class can contain only static members and cannot be instantiated with the new keyword. One copy of the class is loaded into memory when the program loads, and its members are accessed through the class name. Both classes and structs can contain static members. For more information, see Static Classes and Static Class Members.

Nested Types

A class or struct can be nested within another class or struct. For more information, see Nested Types.

Partial Types

You can define part of a class, struct or method in one code file and another part in a separate code file. For more information, see Partial Classes and Methods.

Object Initializers

You can instantiate and initialize class or struct objects, and collections of objects, without explicitly calling their constructor. For more information, see Object and Collection Initializers.

Anonymous Types

In situations where it is not convenient or necessary to create a named class, for example when you are populating a list with data structures that you do not have to persist or pass to another method, you use anonymous types. For more information, see Anonymous Types.

Extension Methods

You can "extend" a class without creating a derived class by creating a separate type whose methods can be called as if they belonged to the original type. For more information, see Extension Methods.

Implicitly Typed Local Variables

Within a class or struct method, you can use implicit typing to instruct the compiler to determine the correct type at compile time. For more information, see Implicitly Typed Local Variables.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Classes (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

Reference types

A type that is defined as a class is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an instance of the class by using the new operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized and in most scenarios, it does not create a performance issue. For more information about garbage collection, see Automatic memory management and garbage collection.

Declaring Classes

Classes are declared by using the class keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

The `class` keyword is preceded by the access level. Because public is used in this case, anyone can create instances of this class. The name of the class follows the `class` keyword. The name of the class must be a valid C# identifier name. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous

example, `object1` is a reference to an object that is based on `Customer`. This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

We don't recommend creating object references such as this one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it to an existing object, such as this:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through `object3` are reflected in subsequent uses of `object4`. Because objects that are based on classes are referred to by reference, classes are known as reference types.

Class inheritance

Classes fully support *inheritance*, a fundamental characteristic of object-oriented programming. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override class virtual methods.

Inheritance is accomplished by using a *derivation*, which means a class is declared by using a *base class* from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see Inheritance.

Unlike C++, a class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement more than one interface. For more information, see Interfaces.

A class can be declared abstract. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a sealed class does not allow other classes to derive from it. For more information, see Abstract and Sealed Classes and Class Members.

Class definitions can be split between different source files. For more information, see Partial Classes and Methods.

Example

The following example defines a public class that contains an auto-implemented property, a method, and a special method called a constructor. For more information, see Properties, Methods, and Constructors topics. The instances of the class are then instantiated with the `new` keyword.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Object-Oriented Programming
- Polymorphism
- Identifier names
- Members
- Methods

- Constructors
- Finalizers
- Objects

Objects (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically.

> **NOTE**
>
> Static types behave differently than what is described here. For more information, see Static Classes and Static Class Members.

Struct Instances vs. Class Instances

Because classes are reference types, a variable of a class object holds a reference to the address of the object on the managed heap. If a second object of the same type is assigned to the first object, then both variables refer to the object at that address. This point is discussed in more detail later in this topic.

Instances of classes are created by using the new operator. In the following example, `Person` is the type and `person1` and `person 2` are instances, or objects, of that type.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();

    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

Because structs are value types, a variable of a struct object holds a copy of the entire object. Instances of structs can also be created by using the `new` operator, but this is not required, as shown in the following example:

```
public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
  Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

The memory for both `p1` and `p2` is allocated on the thread stack. That memory is reclaimed along with the type or method in which it is declared. This is one reason why structs are copied on assignment. By contrast, the memory that is allocated for a class instance is automatically reclaimed (garbage collected) by the common language runtime when all references to the object have gone out of scope. It is not possible to deterministically destroy a class object like you can in C++. For more information about garbage collection in the .NET Framework, see Garbage Collection.

> **NOTE**
> The allocation and deallocation of memory on the managed heap is highly optimized in the common language runtime. In most cases there is no significant difference in the performance cost of allocating a class instance on the heap versus allocating a struct instance on the stack.

Object Identity vs. Value Equality

When you compare two objects for equality, you must first distinguish whether you want to know whether the two variables represent the same object in memory, or whether the values of one or more of their fields are equivalent.

If you are intending to compare values, you must consider whether the objects are instances of value types (structs) or reference types (classes, delegates, arrays).

- To determine whether two class instances refer to the same location in memory (which means that they have the same *identity*), use the static Equals method. (System.Object is the implicit base class for all value types and reference types, including user-defined structs and classes.)
- To determine whether the instance fields in two struct instances have the same values, use the ValueType.Equals method. Because all structs implicitly inherit from System.ValueType, you call the method directly on your object as shown in the following example:

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

The System.ValueType implementation of `Equals` uses reflection because it must be able to determine what the fields are in any struct. When creating your own structs, override the `Equals` method to provide an efficient equality algorithm that is specific to your type.

- To determine whether the values of the fields in two class instances are equal, you might be able to use the Equals method or the == operator. However, only use them if the class has overridden or overloaded them to provide a custom definition of what "equality" means for objects of that type. The class might also implement the IEquatable<T> interface or the IEqualityComparer<T> interface. Both interfaces provide methods that can be used to test value equality. When designing your own classes that override `Equals`, make sure to follow the guidelines stated in How to: Define Value Equality for a Type and Object.Equals(Object).

Related Sections

For more information:

- Classes
- Structs
- Constructors
- Finalizers
- Events

See also

- C# Programming Guide
- object
- Inheritance
- class
- struct
- new Operator
- Common Type System

Structs (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Structs are defined by using the struct keyword, for example:

```
public struct PostalAddress
{
    // Fields, properties, methods and events go here...
}
```

Structs share most of the same syntax as classes. The name of the struct must be a valid C# identifier name. Structs are more limited than classes in the following ways:

- Within a struct declaration, fields cannot be initialized unless they are declared as const or static.
- A struct cannot declare a default constructor (a constructor without parameters) or a finalizer.
- Structs are copied on assignment. When a struct is assigned to a new variable, all the data is copied, and any modification to the new copy does not change the data for the original copy. This is important to remember when working with collections of value types such as `Dictionary<string, myStruct>`.
- Structs are value types, unlike classes, which are reference types.
- Unlike classes, structs can be instantiated without using a `new` operator.
- Structs can declare constructors that have parameters.
- A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from ValueType, which inherits from Object.
- A struct can implement interfaces.
- A struct cannot be `null`, and a struct variable cannot be assigned `null` unless the variable is declared as a nullable type.

Related sections

For more information:

- Using Structs
- Constructors
- Nullable Types
- How to: Know the Difference Between Passing a Struct and Passing a Class Reference to a Method
- How to: Implement User-Defined Conversions Between Structs

See also

- C# Programming Guide
- Classes and Structs
- Classes
- Identifier names

Using Structs (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The `struct` type is suitable for representing lightweight objects such as `Point`, `Rectangle`, and `Color`. Although it is just as convenient to represent a point as a class with Auto-Implemented Properties, a struct might be more efficient in some scenarios. For example, if you declare an array of 1000 `Point` objects, you will allocate additional memory for referencing each object; in this case, a struct would be less expensive. Because the .NET Framework contains an object called Point, the struct in this example is named "Coords" instead.

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

It is an error to define a default (parameterless) constructor for a struct. It is also an error to initialize an instance field in a struct body. You can initialize externally accessible struct members only by using a parameterized constructor, the implicit, default constructor, an object initializer, or by accessing the members individually after the struct is declared. Any private or otherwise inaccessible members require the use of constructors exclusively.

When you create a struct object using the new operator, it gets created and the appropriate constructor is called according to the constructor's signature. Unlike classes, structs can be instantiated without using the `new` operator. In such a case, there is no constructor call, which makes the allocation more efficient. However, the fields will remain unassigned and the object cannot be used until all of the fields are initialized. This includes the inability to get or set values through auto-implemented properties.

If you instantiate a struct object using the default, parameterless constructor, all members are assigned according to their default values.

When writing a constructor with parameters for a struct, you must explicitly initialize all members; otherwise one or more members remain unassigned and the struct cannot be used, producing compiler error CS0171.

There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class. Structs, however, inherit from the base class Object. A struct can implement interfaces, and it does that exactly as classes do.

You cannot declare a class using the keyword `struct`. In C#, classes and structs are semantically different. A struct is a value type, while a class is a reference type. For more information, see Value Types.

Unless you need reference-type semantics, a small class may be more efficiently handled by the system if you declare it as a struct instead.

Example 1

Description

This example demonstrates `struct` initialization using both default and parameterized constructors.

Code

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

```
// Declare and initialize struct objects.
class TestCoords
{
    static void Main()
    {
        // Initialize:
        Coords coords1 = new Coords();
        Coords coords2 = new Coords(10, 10);

        // Display results:
        Console.Write("Coords 1: ");
        Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y);

        Console.Write("Coords 2: ");
        Console.WriteLine("x = {0}, y = {1}", coords2.x, coords2.y);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Coords 1: x = 0, y = 0
    Coords 2: x = 10, y = 10
*/
```

Example 2

Description

This example demonstrates a feature that is unique to structs. It creates a Coords object without using the `new` operator. If you replace the word `struct` with the word `class`, the program will not compile.

Code

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

```
// Declare a struct object without "new."
class TestCoordsNoNew
{
    static void Main()
    {
        // Declare an object:
        Coords coords1;

        // Initialize:
        coords1.x = 10;
        coords1.y = 20;

        // Display results:
        Console.Write("Coords 1: ");
        Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords 1: x = 10, y = 20
```

See also

- C# Programming Guide
- Classes and Structs
- Structs

Inheritance (C# Programming Guide)

1/23/2019 • 7 minutes to read • Edit Online

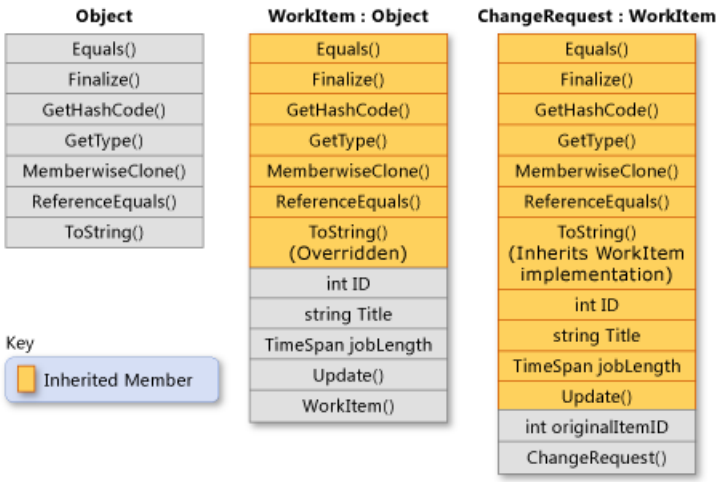
Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. A derived class can have only one direct base class. However, inheritance is transitive. If ClassC is derived from ClassB, and ClassB is derived from ClassA, ClassC inherits the members declared in ClassB and ClassA.

> **NOTE**
>
> Structs do not support inheritance, but they can implement interfaces. For more information, see Interfaces.

Conceptually, a derived class is a specialization of the base class. For example, if you have a base class `Animal`, you might have one derived class that is named `Mammal` and another derived class that is named `Reptile`. A `Mammal` is an `Animal`, and a `Reptile` is an `Animal`, but each derived class represents different specializations of the base class.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class can thereby reuse the code in the base class without having to re-implement it. In the derived class, you can add more members. In this manner, the derived class extends the functionality of the base class.

The following illustration shows a class `WorkItem` that represents an item of work in some business process. Like all classes, it derives from System.Object and inherits all its methods. `WorkItem` adds five members of its own. These include a constructor, because constructors are not inherited. Class `ChangeRequest` inherits from `WorkItem` and represents a particular kind of work item. `ChangeRequest` adds two more members to the members that it inherits from `WorkItem` and from Object. It must add its own constructor, and it also adds `originalItemID`. Property `originalItemID` enables the `ChangeRequest` instance to be associated with the original `WorkItem` to which the change request applies.



Class inheritance

The following example shows how the class relationships demonstrated in the previous illustration are expressed in C#. The example also shows how `WorkItem` overrides the virtual method Object.ToString, and how the `ChangeRequest` class inherits the `WorkItem` implementation of the method.

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem()
    {
        currentID = 0;
    }


    protected int GetNextID()
    {
        // currentID is a static field. It is incremented each time a new
        // instance of WorkItem is created.
        return ++currentID;
    }

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString()
    {
        return $"{this.ID} - {this.Title}";
    }
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
```

```
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of WorkItem by using the constructor in the
        // base class that takes three arguments.
        WorkItem item = new WorkItem("Fix Bugs",
                                     "Fix all bugs in my code branch",
                                     new TimeSpan(3, 4, 0, 0));

        // Create an instance of ChangeRequest by using the constructor in
        // the derived class that takes four arguments.
        ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                                 "Add members to the class",
                                                 new TimeSpan(4, 0, 0),
                                                 1);

        // Use the ToString method defined in WorkItem.
        Console.WriteLine(item.ToString());

        // Use the inherited Update method to change the title of the
        // ChangeRequest object.
        change.Update("Change the Design of the Base Class",
            new TimeSpan(4, 0, 0));

        // ChangeRequest inherits WorkItem's override of ToString.
        Console.WriteLine(change.ToString());

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Abstract and Virtual Methods

When a base class declares a method as virtual, a derived class can override the method with its own implementation. If a base class declares a member as abstract, that method must be overridden in any non-abstract class that directly inherits from that class. If a derived class is itself abstract, it inherits abstract members without implementing them. Abstract and virtual members are the basis for polymorphism, which is the second primary characteristic of object-oriented programming. For more information, see Polymorphism.

Abstract Base Classes

You can declare a class as abstract if you want to prevent direct instantiation by using the new keyword. If you do this, the class can be used only if a new class is derived from it. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract class does not have to contain abstract members; however, if a class does contain an abstract member, the class itself must be declared as abstract. Derived classes that are not abstract themselves must provide the implementation for any abstract methods from an abstract base class. For more information, see Abstract and Sealed Classes and Class Members.

Interfaces

An *interface* is a reference type that is somewhat similar to an abstract base class that consists of only abstract members. When a class implements an interface, it must provide an implementation for all the members of the interface. A class can implement multiple interfaces even though it can derive from only a single direct base class.

Interfaces are used to define specific capabilities for classes that do not necessarily have an "is a" relationship. For example, the System.IEquatable<T> interface can be implemented by any class or struct that has to enable client code to determine whether two objects of the type are equivalent (however the type defines equivalence). IEquatable<T> does not imply the same kind of "is a" relationship that exists between a base class and a derived class (for example, a `Mammal` is an `Animal`). For more information, see Interfaces.

Preventing Further Derivation

A class can prevent other classes from inheriting from it, or from any of its members, by declaring itself or the member as sealed. For more information, see Abstract and Sealed Classes and Class Members.

Derived Class Hiding of Base Class Members

A derived class can hide base class members by declaring members with the same name and signature. The new modifier can be used to explicitly indicate that the member is not intended to be an override of the base member. The use of new is not required, but a compiler warning will be generated if new is not used. For more information, see Versioning with the Override and New Keywords and Knowing When to Use Override and New Keywords.

See also

- C# Programming Guide
- Classes and Structs
- class
- struct

Polymorphism (C# Programming Guide)

1/23/2019 • 7 minutes to read • Edit Online

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called `Shape`, and derived classes such as `Rectangle`, `Circle`, and `Triangle`. Give the `Shape` class a virtual method called `Draw`, and override it in each derived class to draw the particular shape that the class represents. Create a `List<Shape>` object and add a Circle, Triangle and Rectangle to it. To update the drawing surface, use a foreach loop to iterate through the list and call the `Draw` method on each `Shape` object in the list. Even though each object in the list has a declared type of `Shape`, it is the run-time type (the overridden version of the method in each derived class) that will be invoked.

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

public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
```

```
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        var shapes = new List<Shape>
        {
            new Rectangle(),
            new Triangle(),
            new Circle()
        };

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (var shape in shapes)
        {
            shape.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

In C#, every type is polymorphic because all types, including user-defined types, inherit from Object.

Polymorphism Overview

Virtual Members

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. The designer of the derived class can choose whether to

- override virtual members in the base class,
- inherit the closest base class method without overriding it
- define new non-virtual implementation of those members that hide the base class implementations

A derived class can override a base class member only if the base class member is declared as virtual or abstract. The derived member must use the override keyword to explicitly indicate that the method is intended to participate in virtual invocation. The following code provides an example:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Fields cannot be virtual; only methods, properties, events and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. The following code provides an example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Virtual methods and properties enable derived classes to extend a base class without needing to use the base class implementation of a method. For more information, see Versioning with the Override and New Keywords. An interface provides another way to define a method or set of methods whose implementation is left to derived classes. For more information, see Interfaces.

Hiding Base Class Members with New Members

If you want your derived member to have the same name as a member in a base class, but you do not want it to participate in virtual invocation, you can use the new keyword. The `new` keyword is put before the return type of a class member that is being replaced. The following code provides an example:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Hidden base class members can still be accessed from client code by casting the instance of the derived class to an instance of the base class. For example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Preventing Derived Classes from Overriding Virtual Members

Virtual members remain virtual indefinitely, regardless of how many classes have been declared between the virtual member and the class that originally declared it. If class A declares a virtual member, and class B derives from A, and class C derives from B, class C inherits the virtual member, and has the option to override it, regardless of whether class B declared an override for that member. The following code provides an example:

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as sealed. This requires putting the `sealed` keyword before the `override` keyword in the class member declaration. The following code provides an example:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

In the previous example, the method `DoWork` is no longer virtual to any class derived from C. It is still virtual for instances of C, even if they are cast to type B or type A. Sealed methods can be replaced by derived classes by using the `new` keyword, as the following example shows:

```
public class D : C
{
    public new void DoWork() { }
}
```

In this case, if `DoWork` is called on D using a variable of type D, the new `DoWork` is called. If a variable of type C, B, or A is used to access an instance of D, a call to `DoWork` will follow the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class C.

Accessing Base Class Virtual Members from Derived Classes

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the base keyword. The following code provides an example:

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

For more information, see base.

> **NOTE**
>
> It is recommended that virtual members use `base` to call the base class implementation of that member in their own implementation. Letting the base class behavior occur enables the derived class to concentrate on implementing behavior specific to the derived class. If the base class implementation is not called, it is up to the derived class to make their behavior compatible with the behavior of the base class.

In This Section

- Versioning with the Override and New Keywords
- Knowing When to Use Override and New Keywords
- How to: Override the ToString Method

See also

- C# Programming Guide
- Inheritance
- Abstract and Sealed Classes and Class Members
- Methods
- Events
- Properties
- Indexers
- Types

Versioning with the Override and New Keywords (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

The C# language is designed so that versioning between base and derived classes in different libraries can evolve and maintain backward compatibility. This means, for example, that the introduction of a new member in a base class with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that hides a similarly named inherited method.

In C#, derived classes can contain methods with the same name as base class methods.

- The base class method must be defined virtual.
- If the method in the derived class is not preceded by new or override keywords, the compiler will issue a warning and the method will behave as if the `new` keyword were present.
- If the method in the derived class is preceded with the `new` keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the `override` keyword, objects of the derived class will call that method instead of the base class method.
- The base class method can be called from within the derived class using the `base` keyword.
- The `override`, `virtual`, and `new` keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the `virtual` modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the `override` keyword or hide the virtual method in the base class by using the `new` keyword. If neither the `override` keyword nor the `new` keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.

To demonstrate this in practice, assume for a moment that Company A has created a class named `GraphicsClass`, which your program uses. The following is `GraphicsClass`:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Your company uses this class, and you use it to derive your own class, adding a new method:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Your application is used without problems, until Company A releases a new version of `GraphicsClass`, which resembles the following code:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

The new version of `GraphicsClass` now contains a method named `DrawRectangle`. Initially, nothing occurs. The new version is still binary compatible with the old version. Any software that you have deployed will continue to work, even if the new class is installed on those computer systems. Any existing calls to the method `DrawRectangle` will continue to reference your version, in your derived class.

However, as soon as you recompile your application by using the new version of `GraphicsClass`, you will receive a warning from the compiler, CS0108. This warning informs you that you have to consider how you want your `DrawRectangle` method to behave in your application.

If you want your method to override the new base class method, use the `override` keyword:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

The `override` keyword makes sure that any objects derived from `YourDerivedGraphicsClass` will use the derived class version of `DrawRectangle`. Objects derived from `YourDerivedGraphicsClass` can still access the base class version of `DrawRectangle` by using the base keyword:

```
base.DrawRectangle();
```

If you do not want your method to override the new base class method, the following considerations apply. To avoid confusion between the two methods, you can rename your method. This can be time-consuming and error-prone, and just not practical in some cases. However, if your project is relatively small, you can use Visual Studio's Refactoring options to rename the method. For more information, see Refactoring Classes and Types (Class Designer).

Alternatively, you can prevent the warning by using the keyword `new` in your derived class definition:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Using the `new` keyword tells the compiler that your definition hides the definition that is contained in the base class. This is the default behavior.

Override and Method Selection

When a method is named on a class, the C# compiler selects the best method to call if more than one method is compatible with the call, such as when there are two methods with the same name, and parameters that are compatible with the parameter passed. The following methods would be compatible:

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

When `DoWork` is called on an instance of `Derived`, the C# compiler will first try to make the call compatible with the versions of `DoWork` declared originally on `Derived`. Override methods are not considered as declared on a class, they are new implementations of a method declared on a base class. Only if the C# compiler cannot match the method call to an original method on `Derived` will it try to match the call to an overridden method with the same name and compatible parameters. For example:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Because the variable `val` can be converted to a double implicitly, the C# compiler calls `DoWork(double)` instead of `DoWork(int)`. There are two ways to avoid this. First, avoid declaring new methods with the same name as virtual methods. Second, you can instruct the C# compiler to call the virtual method by making it search the base class method list by casting the instance of `Derived` to `Base`. Because the method is virtual, the implementation of `DoWork(int)` on `Derived` will be called. For example:

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

For more examples of `new` and `override`, see Knowing When to Use Override and New Keywords.

See also

- C# Programming Guide
- Classes and Structs
- Methods
- Inheritance

Knowing When to Use Override and New Keywords (C# Programming Guide)

1/23/2019 • 10 minutes to read • Edit Online

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the new and override keywords. The `override` modifier *extends* the base class method, and the `new` modifier *hides* it. The difference is illustrated in the examples in this topic.

In a console application, declare the following two classes, `BaseClass` and `DerivedClass`. `DerivedClass` inherits from `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the `Main` method, declare variables `bc`, `dc`, and `bcdc`.

- `bc` is of type `BaseClass`, and its value is of type `BaseClass`.
- `dc` is of type `DerivedClass`, and its value is of type `DerivedClass`.
- `bcdc` is of type `BaseClass`, and its value is of type `DerivedClass`. This is the variable to pay attention to.

Because `bc` and `bcdc` have type `BaseClass`, they can only directly access `Method1`, unless you use casting. Variable `dc` can access both `Method1` and `Method2`. These relationships are shown in the following code.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Next, add the following `Method2` method to `BaseClass`. The signature of this method matches the signature of the `Method2` method in `DerivedClass`.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Because `BaseClass` now has a `Method2` method, a second calling statement can be added for `BaseClass` variables `bc` and `bcdc`, as shown in the following code.

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

When you build the project, you see that the addition of the `Method2` method in `BaseClass` causes a warning. The warning says that the `Method2` method in `DerivedClass` hides the `Method2` method in `BaseClass`. You are advised to use the `new` keyword in the `Method2` definition if you intend to cause that result. Alternatively, you could rename one of the `Method2` methods to resolve the warning, but that is not always practical.

Before adding `new`, run the program to see the output produced by the additional calling statements. The following results are displayed.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

The `new` keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type `BaseClass` continue to access the members of `BaseClass`, and the variable that has type `DerivedClass` continues to access members in `DerivedClass` first, and then to consider members inherited from

`BaseClass`.

To suppress the warning, add the `new` modifier to the definition of `Method2` in `DerivedClass`, as shown in the following code. The modifier can be added before or after `public`.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using `new`, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance, see new Modifier.

To contrast this behavior to the effects of using `override`, add the following method to `DerivedClass`. The `override` modifier can be added before or after `public`.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the `virtual` modifier to the definition of `Method1` in `BaseClass`. The `virtual` modifier can be added before or after `public`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the `override` modifier enables `bcdc` to access the `Method1` method that is defined in `DerivedClass`. Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using `override` to extend the base class method.

The following code contains the full example.

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

The following example illustrates similar behavior in a different context. The example defines three classes: a base class named `Car` and two classes that are derived from it, `ConvertibleCar` and `Minivan`. The base class contains a `DescribeCar` method. The method displays a basic description of a car, and then calls `ShowDetails` to provide additional information. Each of the three classes defines a `ShowDetails` method. The `new` modifier is used to define `ShowDetails` in the `ConvertibleCar` class. The `override` modifier is used to define `ShowDetails` in the `Minivan` class.

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

The example tests which version of `ShowDetails` is called. The following method, `TestCars1`, declares an instance of each class, and then calls `DescribeCar` on each instance.

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` produces the following output. Notice especially the results for `car2`, which probably are not what you expected. The type of the object is `ConvertibleCar`, but `DescribeCar` does not access the version of `ShowDetails` that is defined in the `ConvertibleCar` class because that method is declared with the `new` modifier, not the `override` modifier. As a result, a `ConvertibleCar` object displays the same description as a `Car` object. Contrast the results for `car3`, which is a `Minivan` object. In this case, the `ShowDetails` method that is declared in the `Minivan` class overrides the `ShowDetails` method that is declared in the `Car` class, and the description that is displayed describes a minivan.

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` creates a list of objects that have type `Car`. The values of the objects are instantiated from the `Car`, `ConvertibleCar`, and `Minivan` classes. `DescribeCar` is called on each element of the list. The following code shows the definition of `TestCars2`.

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

The following output is displayed. Notice that it is the same as the output that is displayed by `TestCars1`. The `ShowDetails` method of the `ConvertibleCar` class is not called, regardless of whether the type of the object is `ConvertibleCar`, as in `TestCars1`, or `Car`, as in `TestCars2`. Conversely, `car3` calls the `ShowDetails` method from the `Minivan` class in both cases, whether it has type `Minivan` or type `Car`.

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Methods `TestCars3` and `TestCars4` complete the example. These methods call `ShowDetails` directly, first from objects declared to have type `ConvertibleCar` and `Minivan` (`TestCars3`), then from objects declared to have type `Car` (`TestCars4`). The following code defines these two methods.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

The methods produce the following output, which corresponds to the results from the first example in this topic.

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

The following code shows the complete project and its output.

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

namespace OverrideAndNew2
```

```
namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
```

```
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
```

```
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

See also

- C# Programming Guide
- Classes and Structs
- Versioning with the Override and New Keywords
- base
- abstract

How to: Override the ToString Method (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Every class or struct in C# implicitly inherits the Object class. Therefore, every object in C# gets the ToString method, which returns a string representation of that object. For example, all variables of type `int` have a `ToString` method, which enables them to return their contents as a string:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

When you create a custom class or struct, you should override the ToString method in order to provide information about your type to client code.

For information about how to use format strings and other types of custom formatting with the `ToString` method, see Formatting Types.

> **IMPORTANT**
>
> When you decide what information to provide through this method, consider whether your class or struct will ever be used by untrusted code. Be careful to ensure that you do not provide any information that could be exploited by malicious code.

To override the ToString method in your class or struct

1. Declare a `ToString` method with the following modifiers and return type:

   ```
   public override string ToString(){}
   ```

2. Implement the method so that it returns a string.

   The following example returns the name of the class in addition to the data specific to a particular instance of the class.

   ```
   class Person
   {
       public string Name { get; set; }
       public int Age { get; set; }

       public override string ToString()
       {
           return "Person: " + Name + " " + Age;
       }
   }
   ```

   You can test the `ToString` method as shown in the following code example:

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

See also

- IFormattable
- C# Programming Guide
- Classes and Structs
- Strings
- string
- new
- override
- virtual
- Formatting Types

Abstract and Sealed Classes and Class Members (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The abstract keyword enables you to create classes and class members that are incomplete and must be implemented in a derived class.

The sealed keyword enables you to prevent the inheritance of a class or certain class members that were previously marked virtual.

Abstract Classes and Class Members

Classes can be declared as abstract by putting the keyword `abstract` before the class definition. For example:

```
public abstract class A
{
    // Class members here.
}
```

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword `abstract` before the return type of the method. For example:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

If a `virtual` method is declared `abstract`, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, `DoWork` on class F cannot call `DoWork` on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Class Members

Classes can be declared as sealed by putting the keyword `sealed` before the class definition. For example:

```
public sealed class D
{
    // Class members here.
}
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the `sealed` keyword before the override keyword in the class member declaration. For example:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

See also

- C# Programming Guide
- Classes and Structs
- Inheritance
- Methods

- Fields
- How to: Define Abstract Properties

How to: Define Abstract Properties (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following example shows how to define abstract properties. An abstract property declaration does not provide an implementation of the property accessors -- it declares that the class supports properties, but leaves the accessor implementation to derived classes. The following example demonstrates how to implement the abstract properties inherited from a base class.

This sample consists of three files, each of which is compiled individually and its resulting assembly is referenced by the next compilation:

- abstractshape.cs: the `Shape` class that contains an abstract `Area` property.
- shapes.cs: The subclasses of the `Shape` class.
- shapetest.cs: A test program to display the areas of some `Shape`-derived objects.

To compile the example, use the following command:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

This will create the executable file shapetest.exe.

Example

This file declares the `Shape` class that contains the `Area` property of the type `double`.

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Modifiers on the property are placed on the property declaration itself. For example:

  ```
  public abstract double Area
  ```

- When declaring an abstract property (such as `Area` in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a get accessor is available, so the property is read-only.

Example

The following code shows three subclasses of `Shape` and how they override the `Area` property to provide their own implementation.

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

Example

The following code shows a test program that creates a number of `Shape`-derived objects and prints out their areas.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

See also

- C# Programming Guide
- Classes and Structs
- Abstract and Sealed Classes and Class Members
- Properties
- How to: Create and Use Assemblies Using the Command Line

Static Classes and Static Class Members (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new keyword to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself. For example, if you have a static class that is named `UtilityClass` that has a public static method named `MethodA`, you call the method as shown in the following example:

```
UtilityClass.MethodA();
```

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields. For example, in the .NET Framework Class Library, the static System.Math class contains methods that perform mathematical operations, without any requirement to store or retrieve data that is unique to a particular instance of the Math class. That is, you apply the members of the class by specifying the class name and the method name, as shown in the following example.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

As is the case with all class types, the type information for a static class is loaded by the .NET Framework common language runtime (CLR) when the program that references the class is loaded. The program cannot specify exactly when the class is loaded. However, it is guaranteed to be loaded and to have its fields initialized and its static constructor called before the class is referenced for the first time in your program. A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

> **NOTE**
> To create a non-static class that allows only one instance of itself to be created, see Implementing Singleton in C#.

The following list provides the main features of a static class:

- Contains only static members.
- Cannot be instantiated.
- Is sealed.
- Cannot contain Instance Constructors.

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a

static class is that the compiler can check to make sure that no instance members are accidentally added. The compiler will guarantee that instances of this class cannot be created.

Static classes are sealed and therefore cannot be inherited. They cannot inherit from any class except Object. Static classes cannot contain an instance constructor; however, they can contain a static constructor. Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization. For more information, see Static Constructors.

Example

Here is an example of a static class that contains two methods that convert temperature from Celsius to Fahrenheit and from Fahrenheit to Celsius:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
```

```
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Static Members

A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it is explicitly passed in a method parameter.

It is more typical to declare a non-static class with some static members, than to declare an entire class as static. Two common uses of static fields are to keep a count of the number of objects that have been instantiated, or to store a value that must be shared among all instances.

Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.

Although a field cannot be declared as `static const`, a const field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, const fields can be accessed by using the same `ClassName.MemberName` notation that is used for static fields. No object instance is required.

C# does not support static local variables (variables that are declared in method scope).

You declare static class members by using the `static` keyword before the return type of the member, as shown in the following example:

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

Static members are initialized before the static member is accessed for the first time and before the static constructor, if there is one, is called. To access a static class member, use the name of the class instead of a variable

name to specify the location of the member, as shown in the following example:

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

If your class contains static fields, provide a static constructor that initializes them when the class is loaded.

A call to a static method generates a call instruction in Microsoft intermediate language (MSIL), whereas a call to an instance method generates a `callvirt` instruction, which also checks for a null object references. However, most of the time the performance difference between the two is not significant.

C# Language Specification

For more information, see Static classes and Static and instance members in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- static
- Classes
- class
- Static Constructors
- Instance Constructors

Members (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Classes and structs have members that represent their data and behavior. A class's members include all the members declared in the class, along with all members (except constructors and finalizers) declared in all classes in its inheritance hierarchy. Private members in base classes are inherited but are not accessible from derived classes.

The following table lists the kinds of members a class or struct may contain:

MEMBER	DESCRIPTION
Fields	Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class. For example, a calendar class may have a field that contains the current date.
Constants	Constants are fields or properties whose value is set at compile time and cannot be changed.
Properties	Properties are methods on a class that are accessed as if they were fields on that class. A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
Methods	Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter.
Events	Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates.
Operators	Overloaded operators are considered class members. When you overload an operator, you define it as a public static method in a class. The predefined operators (`+`, `*`, `<`, and so on) are not considered members. For more information, see Overloadable Operators.
Indexers	Indexers enable an object to be indexed in a manner similar to arrays.
Constructors	Constructors are methods that are called when the object is first created. They are often used to initialize the data of an object.
Finalizers	Finalizers are used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately.

MEMBER	DESCRIPTION
Nested Types	Nested types are types declared within another type. Nested types are often used to describe objects that are used only by the types that contain them.

See also

- C# Programming Guide
- Classes
- Methods
- Constructors
- Finalizers
- Properties
- Fields
- Indexers
- Events
- Nested Types
- Operators
- Overloadable Operators

Access Modifiers (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

All types and type members have an accessibility level, which controls whether they can be used from other code in your assembly or other assemblies. You can use the following access modifiers to specify the accessibility of a type or member when you declare it:

public
The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private
The type or member can be accessed only by code in the same class or struct.

protected
The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
internal
The type or member can be accessed by any code in the same assembly, but not from another assembly.

protected internal The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.

private protected The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

The following examples demonstrate how to specify access modifiers on a type and member:

```
public class Bicycle
{
    public void Pedal() { }
}
```

Not all access modifiers can be used by all types or members in all contexts, and in some cases the accessibility of a type member is constrained by the accessibility of its containing type. The following sections provide more details about accessibility.

Class and Struct Accessibility

Classes and structs that are declared directly within a namespace (in other words, that are not nested within other classes or structs) can be either public or internal. Internal is the default if no access modifier is specified.

Struct members, including nested classes and structs, can be declared as public, internal, or private. Class members, including nested classes and structs, can be public, protected internal, protected, internal, private protected or private. The access level for class members and struct members, including nested classes and structs, is private by default. Private nested types are not accessible from outside the containing type.

Derived classes cannot have greater accessibility than their base types. In other words, you cannot have a public class `B` that derives from an internal class `A`. If this were allowed, it would have the effect of making `A` public, because all protected or internal members of `A` are accessible from the derived class.

You can enable specific other assemblies to access your internal types by using the InternalsVisibleToAttribute. For more information, see Friend Assemblies.

Class and Struct Member Accessibility

Class members (including nested classes and structs) can be declared with any of the six types of access. Struct members cannot be declared as protected because structs do not support inheritance.

Normally, the accessibility of a member is not greater than the accessibility of the type that contains it. However, a public member of an internal class might be accessible from outside the assembly if the member implements interface methods or overrides virtual methods that are defined in a public base class.

The type of any member that is a field, property, or event must be at least as accessible as the member itself. Similarly, the return type and the parameter types of any member that is a method, indexer, or delegate must be at least as accessible as the member itself. For example, you cannot have a public method `M` that returns a class `C` unless `C` is also public. Likewise, you cannot have a protected property of type `A` if `A` is declared as private.

User-defined operators must always be declared as public. For more information, see operator (C# Reference).

Finalizers cannot have accessibility modifiers.

To set the access level for a class or struct member, add the appropriate keyword to the member declaration, as shown in the following example.

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

> **NOTE**
>
> The protected internal accessibility level means protected OR internal, not protected AND internal. In other words, a protected internal member can be accessed from any class in the same assembly, including derived classes. To limit accessibility to only derived classes in the same assembly, declare the class itself internal, and declare its members as protected. Also, starting with C# 7.2, you can use the private protected access modifier to achieve the same result without need to make the containing class internal.

Other Types

Interfaces declared directly within a namespace can be declared as public or internal and, just like classes and structs, interfaces default to internal access. Interface members are always public because the purpose of an interface is to enable other types to access a class or struct. No access modifiers can be applied to interface members.

Enumeration members are always public, and no access modifiers can be applied.

Delegates behave like classes and structs. By default, they have internal access when declared directly within a namespace, and private access when nested.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Classes and Structs
- Interfaces
- private
- public
- internal
- protected
- protected internal
- private protected
- class
- struct
- interface

Fields (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

A *field* is a variable of any type that is declared directly in a class or struct. Fields are *members* of their containing type.

A class or struct may have instance fields or static fields or both. Instance fields are specific to an instance of a type. If you have a class T, with an instance field F, you can create two objects of type T, and modify the value of F in each object without affecting the value in the other object. By contrast, a static field belongs to the class itself, and is shared among all instances of that class. Changes made from instance A will be visibly immediately to instances B and C if they access the field.

Generally, you should use fields only for variables that have private or protected accessibility. Data that your class exposes to client code should be provided through methods, properties and indexers. By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a *backing store* or *backing field*.

Fields typically store the data that must be accessible to more than one class method and must be stored for longer than the lifetime of any single method. For example, a class that represents a calendar date might have three integer fields: one for the month, one for the day, and one for the year. Variables that are not used outside the scope of a single method should be declared as *local variables* within the method body itself.

Fields are declared in the class block by specifying the access level of the field, followed by the type of the field, followed by the name of the field. For example:

```
public class CalendarEntry
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // Public property exposes date field safely.
    public DateTime Date
    {
        get
        {
            return date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                date = value;
            }
            else
                throw new ArgumentOutOfRangeException();
        }

    }

    // Public method also exposes date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            date = dt;
        }
        else
            throw new ArgumentOutOfRangeException();
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt != null && dt.Ticks < date.Ticks)
        {
            return date - dt;
        }
        else
            throw new ArgumentOutOfRangeException();

    }
}
```

To access a field in an object, add a period after the object name, followed by the name of the field, as in `objectname.fieldname`. For example:

```
CalendarEntry birthday = new CalendarEntry();
birthday.day = "Saturday";
```

A field can be given an initial value by using the assignment operator when the field is declared. To automatically

assign the `day` field to `"Monday"`, for example, you would declare `day` as in the following example:

```
public class CalendarDateWithInitialization
{
    public string day = "Monday";
    //...
}
```

Fields are initialized immediately before the constructor for the object instance is called. If the constructor assigns the value of a field, it will overwrite any value given during field declaration. For more information, see Using Constructors.

> **NOTE**
> A field initializer cannot refer to other instance fields.

Fields can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the fields. For more information, see Access Modifiers.

A field can optionally be declared static. This makes the field available to callers at any time, even if no instance of the class exists. For more information, see Static Classes and Static Class Members.

A field can be declared readonly. A read-only field can only be assigned a value during initialization or in a constructor. A `static readonly` field is very similar to a constant, except that the C# compiler does not have access to the value of a static read-only field at compile time, only at run time. For more information, see Constants.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Classes and Structs
- Using Constructors
- Inheritance
- Access Modifiers
- Abstract and Sealed Classes and Class Members

Constants (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the const modifier. Only the C# built-in types (excluding System.Object) may be declared as `const`. For a list of the built-in types, see Built-In Types Table. User-defined types, including classes, structs, and arrays, cannot be `const`. Use the readonly modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

C# does not support `const` methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example `int`, `uint`, `long`, and so on). For more information, see enum.

Constants must be initialized as they are declared. For example:

```
class Calendar1
{
    public const int months = 12;
}
```

In this example, the constant `months` is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, `months`), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, `const` fields cannot be passed by reference and cannot appear as an l-value in an expression.

> **NOTE**
> Use caution when you refer to constant values defined in other code such as DLLs. If a new version of the DLL defines a new value for the constant, your program will still hold the old literal value until it is recompiled against the new version.

Multiple constants of the same type can be declared at the same time, for example:

```
class Calendar2
{
    const int months = 12, weeks = 52, days = 365;
}
```

The expression that is used to initialize a constant can refer to another constant if it does not create a circular reference. For example:

```
class Calendar3
{
    const int months = 12;
    const int weeks = 52;
    const int days = 365;

    const double daysPerWeek = (double) days / (double) weeks;
    const double daysPerMonth = (double) days / (double) months;
}
```

Constants can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the constant. For more information, see Access Modifiers.

Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the `static` keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example:

```
int birthstones = Calendar.months;
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Classes and Structs
- Properties
- Types
- readonly
- Immutability in C# Part One: Kinds of Immutability

How to: Define Constants in C#

1/23/2019 • 2 minutes to read • Edit Online

Constants are fields whose values are set at compile time and can never be changed. Use constants to provide meaningful names instead of numeric literals ("magic numbers") for special values.

> **NOTE**
> In C# the #define preprocessor directive cannot be used to define constants in the way that is typically used in C and C++.

To define constant values of integral types (`int` , `byte` , and so on) use an enumerated type. For more information, see enum.

To define non-integral constants, one approach is to group them in a single static class named `Constants` . This will require that all references to the constants be prefaced with the class name, as shown in the following example.

Example

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}
class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
    }
}
```

The use of the class name qualifier helps ensure that you and others who use the constant understand that it is constant and cannot be modified.

See also

- Classes and Structs

Properties (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
- The value keyword is used to define the value being assigned by the `set` accessor.
- Properties can be *read-write* (they have both a `get` and a `set` accessor), *read-only* (they have a `get` accessor but no `set` accessor), or *write-only* (they have a `set` accessor, but no `get` accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.

Properties with backing fields

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The `get` accessor returns the value of the private field, and the `set` accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

The following example illustrates this pattern. In this example, the `TimePeriod` class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named `_seconds`. A read-write property named `Hours` allows the customer to specify the time interval in hours. Both the `get` and the `set` accessors perform the necessary conversion between hours and seconds. In addition, the `set` accessor validates the data and throws an ArgumentOutOfRangeException if the number of hours is invalid.

```
using System;

class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }
}

class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Expression body definitions

Property accessors often consist of single-line statements that just assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the `=>` symbol followed by the expression to assign to or retrieve from the property.

Starting with C# 6, read-only properties can implement the `get` accessor as an expression-bodied member. In this case, neither the `get` accessor keyword nor the `return` keyword is used. The following example implements the read-only `Name` property as an expression-bodied member.

```
using System;

public class Person
{
   private string _firstName;
   private string _lastName;

   public Person(string first, string last)
   {
      _firstName = first;
      _lastName = last;
   }

   public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
   public static void Main()
   {
      var person = new Person("Isabelle", "Butts");
      Console.WriteLine(person.Name);
   }
}
// The example displays the following output:
//      Isabelle Butts
```

Starting with C# 7.0, both the `get` and the `set` accessor can be implemented as expression-bodied members. In this case, the `get` and `set` keywords must be present. The following example illustrates the use of expression body definitions for both accessors. Note that the `return` keyword is not used with the `get` accessor.

```
using System;

public class SaleItem
{
   string _name;
   decimal _cost;

   public SaleItem(string name, decimal cost)
   {
      _name = name;
      _cost = cost;
   }

   public string Name
   {
      get => _name;
      set => _name = value;
   }

   public decimal Price
   {
      get => _cost;
      set => _cost = value;
   }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem("Shoes", 19.95m);
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Auto-implemented properties

In some cases, property `get` and `set` accessors just assign a value to or retrieve a value from a backing field without including any additional logic. By using auto-implemented properties, you can simplify your code while having the C# compiler transparently provide the backing field for you.

If a property has both a `get` and a `set` accessor, both must be auto-implemented. You define an auto-implemented property by using the `get` and `set` keywords without providing any implementation. The following example repeats the previous one, except that `Name` and `Price` are auto-implemented properties. Note that the example also removes the parameterized constructor, so that `SaleItem` objects are now initialized with a call to the default constructor and an object initializer.

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Related sections

- Using Properties
- Interface Properties
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility
- Auto-Implemented Properties

C# Language Specification

For more information, see Properties in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Using Properties
- Indexers
- get keyword
- set keyword

Using Properties (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a get accessor and/or a set accessor. The code block for the `get` accessor is executed when the property is read; the code block for the `set` accessor is executed when the property is assigned a new value. A property without a `set` accessor is considered read-only. A property without a `get` accessor is considered write-only. A property that has both accessors is read-write.

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a ref or out parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a `get`-accessor and/or a `set` accessor. For example:

```
public class Date
{
    private int month = 7;  // Backing store

    public int Month
    {
        get
        {
            return month;
        }
        set
        {
            if ((value > 0) && (value < 13))
            {
                month = value;
            }
        }
    }
}
```

In this example, `Month` is declared as a property so that the `set` accessor can make sure that the `Month` value is set between 1 and 12. The `Month` property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see Access Modifiers.

Auto-implemented properties provide simplified syntax for simple property declarations. For more information, see Auto-Implemented Properties.

The get Accessor

The body of the `get` accessor resembles that of a method. It must return a value of the property type. The execution of the `get` accessor is equivalent to reading the value of the field. For example, when you are returning

the private variable from the `get` accessor and optimizations are enabled, the call to the `get` accessor method is inlined by the compiler so there is no method-call overhead. However, a virtual `get` accessor method cannot be inlined because the compiler does not know at compile-time which method may actually be called at run time. The following is a `get` accessor that returns the value of a private field `name`:

```
class Person
{
    private string name;  // the name field
    public string Name    // the Name property
    {
        get
        {
            return name;
        }
    }
}
```

When you reference the property, except as the target of an assignment, the `get` accessor is invoked to read the value of the property. For example:

```
Person person = new Person();
//...

System.Console.Write(person.Name);  // the get accessor is invoked here
```

The `get` accessor must end in a return or throw statement, and control cannot flow off the accessor body.

It is a bad programming style to change the state of the object by using the `get` accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the `number` field is accessed.

```
private int number;
public int Number
{
    get
    {
        return number++;   // Don't do this
    }
}
```

The `get` accessor can be used to return the field value or to compute it and return it. For example:

```
class Employee
{
    private string name;
    public string Name
    {
        get
        {
            return name != null ? name : "NA";
        }
    }
}
```

In the previous code segment, if you do not assign a value to the `Name` property, it will return the value NA.

The set Accessor

The `set` accessor resembles a method whose return type is void. It uses an implicit parameter called `value`, whose type is the type of the property. In the following example, a `set` accessor is added to the `Name` property:

```
class Person
{
    private string name;  // the name field
    public string Name    // the Name property
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

When you assign a value to the property, the `set` accessor is invoked by using an argument that provides the new value. For example:

```
Person person = new Person();
person.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(person.Name);  // the get accessor is invoked here
```

It is an error to use the implicit parameter name, `value`, for a local variable declaration in a `set` accessor.

Remarks

Properties can be marked as `public`, `private`, `protected`, `internal`, `protected internal` or `private protected`. These access modifiers define how users of the class can access the property. The `get` and `set` accessors for the same property may have different access modifiers. For example, the `get` may be `public` to allow read-only access from outside the type, and the `set` may be `private` or `protected`. For more information, see Access Modifiers.

A property may be declared as a static property by using the `static` keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information, see Static Classes and Static Class Members.

A property may be marked as a virtual property by using the virtual keyword. This enables derived classes to override the property behavior by using the override keyword. For more information about these options, see Inheritance.

A property overriding a virtual property can also be sealed, specifying that for derived classes it is no longer virtual. Lastly, a property can be declared abstract. This means that there is no implementation in the class, and derived classes must write their own implementation. For more information about these options, see Abstract and Sealed Classes and Class Members.

> **NOTE**
> It is an error to use a virtual, abstract, or override modifier on an accessor of a static property.

Example

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `NumberOfEmployees` by 1, and displays the Employee name and number.

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int counter;
    private string name;

    // A read-write instance property:
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // A read-only static property:
    public static int Counter
    {
        get { return counter; }
    }

    // A Constructor:
    public Employee()
    {
        // Calculate the employee's number:
        counter = ++NumberOfEmployees;
    }
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
    Employee number: 108
    Employee name: Claude Vige
*/
```

Example

This example demonstrates how to access a property in a base class that is hidden by another property that has the same name in a derived class.

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

public class Manager : Employee
{
    private string name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get { return name; }
        set { name = value + ", Manager"; }
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

The following are important points in the previous example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the `new` modifier is used in the declaration of the property in the derived class:

  ```
  public new string Name
  ```

- The cast `(Employee)` is used to access the hidden property in the base class:

  ```
  ((Employee)m1).Name = "Mary";
  ```

  For more information about hiding members, see the new Modifier.

Example

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the override modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the

corresponding side for the square and cube.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    public Square(double s)  //constructor
    {
        side = s;
    }

    public override double Area
    {
        get
        {
            return side * side;
        }
        set
        {
            side = System.Math.Sqrt(value);
        }
    }
}

class Cube : Shape
{
    public double side;

    public Cube(double s)
    {
        side = s;
    }

    public override double Area
    {
        get
        {
            return 6 * side * side;
        }
        set
        {
            side = System.Math.Sqrt(value / 6);
        }
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
```

```
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

See also

- C# Programming Guide
- Properties
- Interface Properties
- Auto-Implemented Properties

Interface Properties (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Properties can be declared on an interface. The following is an example of an interface property accessor:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

The accessor of an interface property does not have a body. Thus, the purpose of the accessors is to indicate whether the property is read-write, read-only, or write-only.

Example

In this example, the interface `IEmployee` has a read-write property, `Name`, and a read-only property, `Counter`. The class `Employee` implements the `IEmployee` interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

You could use the fully qualified name of the property, which references the interface in which the member is declared. For example:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

This is called Explicit Interface Implementation. For example, if the class `Employee` is implementing two interfaces `ICitizen` and `IEmployee` and both interfaces have the `Name` property, the explicit interface member implementation will be necessary. That is, the following property declaration:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implements the `Name` property on the `IEmployee` interface, while the following declaration:

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implements the `Name` property on the `ICitizen` interface.

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string name;
    public string Name  // read-write instance property
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    private int counter;
    public int Counter  // read-only instance property
    {
        get
        {
            return counter;
        }
    }

    public Employee()  // constructor
    {
        counter = ++numberOfEmployees;
    }
}

class TestEmployee
{
    static void Main()
    {
        System.Console.Write("Enter number of employees: ");
        Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

        Employee e1 = new Employee();
        System.Console.Write("Enter the name of the new employee: ");
        e1.Name = System.Console.ReadLine();

        System.Console.WriteLine("The employee information:");
        System.Console.WriteLine("Employee number: {0}", e1.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
```

Sample Output

```
Enter number of employees: 210
```

```
Enter the name of the new employee: Hazem Abolrous
```

```
The employee information:
```

```
Employee number: 211
```

```
Employee name: Hazem Abolrous
```

See also

- C# Programming Guide
- Properties
- Using Properties
- Comparison Between Properties and Indexers
- Indexers
- Interfaces

Restricting Accessor Accessibility (C# Programming Guide)

2/13/2019 • 4 minutes to read • Edit Online

The get and set portions of a property or indexer are called *accessors*. By default these accessors have the same visibility or access level of the property or indexer to which they belong. For more information, see accessibility levels. However, it is sometimes useful to restrict access to one of these accessors. Typically, this involves restricting the accessibility of the `set` accessor, while keeping the `get` accessor publicly accessible. For example:

```
private string name = "Hello";

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

In this example, a property called `Name` defines a `get` and `set` accessor. The `get` accessor receives the accessibility level of the property itself, `public` in this case, while the `set` accessor is explicitly restricted by applying the protected access modifier to the accessor itself.

Restrictions on Access Modifiers on Accessors

Using the accessor modifiers on properties or indexers is subject to these conditions:

- You cannot use accessor modifiers on an interface or an explicit interface member implementation.
- You can use accessor modifiers only if the property or indexer has both `set` and `get` accessors. In this case, the modifier is permitted on only one of the two accessors.
- If the property or indexer has an override modifier, the accessor modifier must match the accessor of the overridden accessor, if any.
- The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.

Access Modifiers on Overriding Accessors

When you override a property or indexer, the overridden accessors must be accessible to the overriding code. Also, the accessibility of both the property/indexer and its accessors must match the corresponding overridden property/indexer and its accessors. For example:

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Implementing Interfaces

When you use an accessor to implement an interface, the accessor may not have an access modifier. However, if you implement the interface using one accessor, such as `get`, the other accessor can have an access modifier, as in the following example:

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Accessor Accessibility Domain

If you use an access modifier on the accessor, the accessibility domain of the accessor is determined by this modifier.

If you did not use an access modifier on the accessor, the accessibility domain of the accessor is determined by the accessibility level of the property or indexer.

Example

The following example contains three classes, `BaseClass`, `DerivedClass`, and `MainClass`. There are two properties on the `BaseClass`, `Name` and `Id` on both classes. The example demonstrates how the property `Id` on `DerivedClass` can be hidden by the property `Id` on `BaseClass` when you use a restrictive access modifier such as protected or private. Therefore, when you assign values to this property, the property on the `BaseClass` class is called instead. Replacing the access modifier by public will make the property accessible.

The example also demonstrates that a restrictive access modifier, such as `private` or `protected`, on the `set` accessor of the `Name` property in `DerivedClass` prevents access to the accessor and generates an error when you assign to it.

```
public class BaseClass
{
    private string name = "Name-BaseClass";
    private string id = "ID-BaseClass";

    public string Name
    {
        get { return name; }
        set { }
    }

    public string Id
    {
        get { return id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string name = "Name-DerivedClass";
    private string id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return id;
        }
        set
        {
            id = value;
        }
    }
}

class MainClass
```

```
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Comments

Notice that if you replace the declaration `new private string Id` by `new public string Id`, you get the output:

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass
```

```
Name and ID in the derived class: John, John123
```

See also

- C# Programming Guide
- Properties
- Indexers
- Access Modifiers

How to: Declare and Use Read Write Properties (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. This is accomplished through *accessors*: special methods that assign and retrieve values from the underlying data member. The set accessor enables data members to be assigned, and the get accessor retrieves data member values.

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties provide `get` and `set` accessors, so they are considered read/write properties.

Example

```
class Person
{
    private string name = "N/A";
    private int age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return age;
        }

        set
        {
            age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();
```

```
        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Robust Programming

In the previous example, the `Name` and `Age` properties are public and include both a `get` and a `set` accessor. This allows any object to read and write these properties. It is sometimes desirable, however, to exclude one of the accessors. Omitting the `set` accessor, for example, makes the property read-only:

```
public string Name
{
    get
    {
        return name;
    }
}
```

Alternatively, you can expose one accessor publicly but make the other private or protected. For more information, see Asymmetric Accessor Accessibility.

Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

```
person.Name = "Joe";
person.Age = 99;
```

Note that in a property `set` method a special `value` variable is available. This variable contains the value that the user specified, for example:

```
name = value;
```

Notice the clean syntax for incrementing the `Age` property on a `Person` object:

```
person.Age += 1;
```

If separate `set` and `get` methods were used to model properties, the equivalent code might look like this:

```
person.SetAge(person.GetAge() + 1);
```

The `ToString` method is overridden in this example:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Notice that `ToString` is not explicitly used in the program. It is invoked by default by the `WriteLine` calls.

See also

- C# Programming Guide
- Properties
- Classes and Structs

Auto-Implemented Properties (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's `get` and `set` accessors.

Example

The following example shows a simple class that has some auto-implemented properties:

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerID { get; set; }

    // Constructor
    public Customer(double purchases, string name, int ID)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerID = ID;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

In C# 6 and later, you can initialize auto-implemented properties similarly to fields:

```
public string FirstName { get; set; } = "Jane";
```

The class that is shown in the previous example is mutable. Client code can change the values in objects after they are created. In complex classes that contain significant behavior (methods) as well as data, it is often necessary to

have public properties. However, for small classes or structs that just encapsulate a set of values (data) and have little or no behaviors, you should either make the objects immutable by declaring the set accessor as private (immutable to consumers) or by declaring only a get accessor (immutable everywhere except the constructor). For more information, see How to: Implement a Lightweight Class with Auto-Implemented Properties.

See also

- Properties
- Modifiers

How to: Implement a Lightweight Class with Auto-Implemented Properties (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to create an immutable lightweight class that serves only to encapsulate a set of auto-implemented properties. Use this kind of construct instead of a struct when you must use reference type semantics.

You can make an immutable property in two ways. You can declare the set accessor to be private. The property is only settable within the type, but it is immutable to consumers. You can instead declare only the get accessor, which makes the property immutable everywhere except in the type's constructor.

When you declare a private `set` accessor, you cannot use an object initializer to initialize the property. You must use a constructor or a factory method.

Example

The following example shows two ways to implement an immutable class that has auto-implemented properties. Each way declares one of the properties with a private `set` and one of the properties with a `get` only. The first class uses a constructor only to initialize the properties, and the second class uses a static factory method that calls a constructor.

```
// This class is immutable. After an object is created,
    // it cannot be modified from outside the class. It uses a
    // constructor to initialize its properties.
    class Contact
    {
        // Read-only properties.
        public string Name { get; }
        public string Address { get; private set; }

        // Public constructor.
        public Contact(string contactName, string contactAddress)
        {
            Name = contactName;
            Address = contactAddress;
        }
    }

    // This class is immutable. After an object is created,
    // it cannot be modified from outside the class. It uses a
    // static method and private constructor to initialize its properties.
    public class Contact2
    {
        // Read-only properties.
        public string Name { get; private set; }
        public string Address { get; }

        // Private constructor.
        private Contact2(string contactName, string contactAddress)
        {
            Name = contactName;
            Address = contactAddress;
        }

        // Public factory method.
        public static Contact2 CreateContact(string name, string address)
```

```
        {
            return new Contact2(name, address);
        }
    }

    public class Program
    {
        static void Main()
        {
            // Some simple data sources.
            string[] names = {"Terry Adams","Fadi Fakhouri", "Hanying Feng",
                              "Cesar Garcia", "Debra Garcia"};
            string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
                                  "12 108th St.", "89 E. 42nd St."};

            // Simple query to demonstrate object creation in select clause.
            // Create Contact objects by using a constructor.
            var query1 = from i in Enumerable.Range(0, 5)
                         select new Contact(names[i], addresses[i]);

            // List elements cannot be modified by client code.
            var list = query1.ToList();
            foreach (var contact in list)
            {
                Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
            }

            // Create Contact2 objects by using a static factory method.
            var query2 = from i in Enumerable.Range(0, 5)
                         select Contact2.CreateContact(names[i], addresses[i]);

            // Console output is identical to query1.
            var list2 = query2.ToList();

            // List elements cannot be modified by client code.
            // CS0272:
            // list2[0].Name = "Eugene Zabokritski";

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

The compiler creates backing fields for each auto-implemented property. The fields are not accessible directly from source code.

See also

- Properties
- struct
- Object and Collection Initializers

Methods (C# Programming Guide)

1/23/2019 • 10 minutes to read • Edit Online

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The Main method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

> **NOTE**
>
> This topic discusses named methods. For information about anonymous functions, see Anonymous Functions.

Method Signatures

Methods are declared in a class or struct by specifying the access level such as `public` or `private`, optional modifiers such as `abstract` or `sealed`, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

> **NOTE**
>
> A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains four methods:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Method Access

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the `Motorcycle` class can therefore be called as in the following example:

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Method Parameters vs. Arguments

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code does not have to be the same as the parameter named defined in the method. For example:

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaulates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Passing by Reference vs. Passing by Value

By default, when a value type is passed to a method, a copy is passed instead of the object itself. Therefore, changes to the argument have no effect on the original copy in the calling method. You can pass a value-type by reference by using the ref keyword. For more information, see Passing Value-Type Parameters. For a list of built-in value types, see Value Types Table.

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the method

receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

You create a reference type by using the `class` keyword, as the following example shows.

```
public class SampleRefType
{
    public int value;
}
```

Now, if you pass an object that is based on this type to a method, a reference to the object is passed. The following example passes an object of type `SampleRefType` to method `ModifyObject`.

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}
static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

The example does essentially the same thing as the previous example in that it passes an argument by value to a method. But, because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `value` field of the parameter, `obj`, also changes the `value` field of the argument, `rt`, in the `TestRefType` method. The `TestRefType` method displays 33 as the output.

For more information about how to pass reference types by reference and by value, see Passing Reference-Type Parameters and Reference Types.

Return Values

Methods can return a value to the caller. If the return type, the type listed before the method name, is not `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a value that matches the return type will return that value to the method caller.

The value can be returned to the caller by value or, starting with C# 7.0, by reference. Values are returned to the caller by reference if the `ref` keyword is used in the method signature and it follows each `return` keyword. For example, the following method signature and return statement indicate that the method returns a variable names `estDistance` by reference to the caller.

```
public ref double GetEstimatedDistance()
{
   return ref estDistance;
}
```

The `return` keyword also stops the execution of the method. If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the `return` keyword to return a value. For example, these two methods use the `return` keyword to return integers:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

To use a value returned by reference from a method, you must declare a ref local variable if you intend to modify its value. For example, if the `Planet.GetEstimatedDistance` method returns a Double value by reference, you can define it as a ref local variable with code like the following:

```
ref int distance = plant
```

Returning a multi-dimensional array from a method, `M`, that modifies the array's contents is not necessary if the calling function passed the array into `M`. You may return the resulting array from `M` for good style or functional flow of values, but it is not necessary because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the method `M`, any changes to the array's contents are observable by any code that has a reference to the array, as shown in the following example.

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

For more information, see return.

Async Methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

> **NOTE**
> An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method can have a return type of Task<TResult>, Task, or void. The void return type is used primarily to define event handlers, where a void return type is required. An async method that returns void can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws.

In the following example, `DelayAsync` is an async method that has a return type of Task<TResult>. `DelayAsync` has a `return` statement that returns an integer. Therefore the method declaration of `DelayAsync` must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer as the following statement demonstrates: `int result = await delayTask`.

The `startButton_Click` method is an example of an async method that has a return type of void. Because `DoSomethingAsync` is an async method, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `await DoSomethingAsync();`. The `startButton_Click` method must be defined with the `async` modifier because the method has an `await` expression.

```
// using System.Diagnostics;
// using System.Threading.Tasks;

// This Click event is marked with the async modifier.
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    await DoSomethingAsync();
}

private async Task DoSomethingAsync()
{
    Task<int> delayTask = DelayAsync();
    int result = await delayTask;

    // The previous two statements may be combined into
    // the following statement.
    //int result = await DelayAsync();

    Debug.WriteLine("Result: " + result);
}

private async Task<int> DelayAsync()
{
    await Task.Delay(100);
    return 5;
}

// Output:
//  Result: 5
```

An async method can't declare any ref or out parameters, but it can call methods that have such parameters.

For more information about async methods, see Asynchronous Programming with async and await, Control Flow in Async Programs, and Async Return Types.

Expression Body Definitions

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, then the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.

You call an iterator from client code by using a foreach statement.

The return type of an iterator can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Classes and Structs
- Access Modifiers
- Static Classes and Static Class Members
- Inheritance
- Abstract and Sealed Classes and Class Members
- params
- return
- out
- ref
- Passing Parameters

Local functions (C# Programming Guide)

1/23/2019 • 6 minutes to read • Edit Online

Starting with C# 7.0, C# supports *local functions*. Local functions are private methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:

- Methods, especially iterator methods and async methods
- Constructors
- Property accessors
- Event accessors
- Anonymous methods
- Lambda expressions
- Finalizers
- Other local functions

However, local functions can't be declared inside an expression-bodied member.

> **NOTE**
> In some cases, you can use a lambda expression to implement functionality also supported by a local function. For a comparison, see Local functions compared to Lambda expressions.

Local functions make the intent of your code clear. Anyone reading you code can see that the method is not callable except by the containing method. For team projects, they also make it impossible for another developer to mistakenly call the method directly from elsewhere in the class or struct.

Local function syntax

A local function is defined as a nested method inside a containing member. Its definition has the following syntax:

```
<modifiers: async | unsafe> <return-type> <method-name> <parameter-list>
```

Local functions can use the async and unsafe modifiers.

Note that all local variables that are defined in the containing member, including its method parameters, are accessible in the local function.

Unlike a method definition, a local function definition cannot include the following elements:

- The member access modifier. Because all local functions are private, including an access modifier, such as the `private` keyword, generates compiler error CS0106, "The modifier 'private' is not valid for this item."
- The static keyword. Including the `static` keyword generates compiler error CS0106, "The modifier 'static' is not valid for this item."

In addition, attributes can't be applied to the local function or to its parameters and type parameters.

The following example defines a local function named `AppendPathSeparator` that is private to a method named `GetText`:

```
using System;
using System.IO;

class Example
{
    static void Main()
    {
        string contents = GetText(@"C:\temp", "example.txt");
        Console.WriteLine("Contents of the file:\n" + contents);
    }

    private static string GetText(string path, string filename)
    {
         var sr = File.OpenText(AppendPathSeparator(path) + filename);
         var text = sr.ReadToEnd();
         return text;

         // Declare a local function.
         string AppendPathSeparator(string filepath)
         {
            if (! filepath.EndsWith(@"\"))
               filepath += @"\";

            return filepath;
         }
    }
}
```

Local functions and exceptions

One of the useful features of local functions is that they can allow exceptions to surface immediately. For method iterators, exceptions are surfaced only when the returned sequence is enumerated, and not when the iterator is retrieved. For async methods, any exceptions thrown in an async method are observed when the returned task is awaited.

The following example defines an `OddSequence` method that enumerates odd numbers between a specified range. Because it passes a number greater than 100 to the `OddSequence` enumerator method, the method throws an ArgumentOutOfRangeException. As the output from the example shows, the exception surfaces only when you iterate the numbers, and not when you retrieve the enumerator.

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

class Example
{
   static void Main()
   {
      IEnumerable<int> ienum = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var i in ienum)
      {
         Console.Write($"{i} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException("start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException("end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the following output:
//    Retrieved enumerator...
//
//    Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valid
values.
//    Parameter name: end must be less than or equal to 100.
//       at Sequence.<GetNumericRange>d__1.MoveNext() in Program.cs:line 23
//       at Example.Main() in Program.cs:line 43
```

Instead, you can throw an exception when performing validation and before retrieving the iterator by returning the iterator from a local function, as the following example shows.

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

class Example
{
   static void Main()
   {
      IEnumerable<int> ienum = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var i in ienum)
      {
         Console.Write($"{i} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException("start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException("end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the following output:
//    Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valid
values.
//    Parameter name: end must be less than or equal to 100.
//       at Sequence.<GetNumericRange>d__1.MoveNext() in Program.cs:line 23
//       at Example.Main() in Program.cs:line 43
```

Local functions can be used in a similar way to handle exceptions outside of the asynchronous operation. Ordinarily, exceptions thrown in async method require that you examine the inner exceptions of an AggregateException. Local functions allow your code to fail fast and allow your exception to be both thrown and observed synchronously.

The following example uses an asynchronous method named `GetMultipleAsync` to pause for a specified number of seconds and return a value that is a random multiple of that number of seconds. The maximum delay is 5 seconds; an ArgumentOutOfRangeException results if the value is greater than 5. As the following example shows, the exception that is thrown when a value of 6 is passed to the `GetMultipleAsync` method is wrapped in an AggregateException after the `GetMultipleAsync` method begins execution.

```
using System;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      int result = GetMultipleAsync(6).Result;
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static async Task<int> GetMultipleAsync(int secondsDelay)
   {
      Console.WriteLine("Executing GetMultipleAsync...");
      if (secondsDelay < 0 || secondsDelay > 5)
         throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5.");

      await Task.Delay(secondsDelay * 1000);
      return secondsDelay * new Random().Next(2,10);
   }
}
// The example displays the following output:
//    Executing GetMultipleAsync...
//
//    Unhandled Exception: System.AggregateException:
//         One or more errors occurred. (Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.) --->
//         System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.
//       at Example.<GetMultiple>d__1.MoveNext() in Program.cs:line 17
//       --- End of inner exception stack trace ---
//       at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
//       at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)
//       at Example.Main() in C:\Users\ronpet\Documents\Visual Studio 2017\Projects\local-
functions\async1\Program.cs:line 8
```

As we did with the method iterator, we can refactor the code from this example to perform the validation before calling the asynchronous method. As the output from the following example shows, the ArgumentOutOfRangeException is not wrapped in a AggregateException.

```
using System;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      int result = GetMultiple(6).Result;
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static Task<int> GetMultiple(int secondsDelay)
   {
      if (secondsDelay < 0 || secondsDelay > 5)
         throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5.");

      return GetValueAsync();

      async Task<int> GetValueAsync()
      {
         Console.WriteLine("Executing GetValueAsync...");
         await Task.Delay(secondsDelay * 1000);
         return secondsDelay * new Random().Next(2,10);
      }
   }
}
// The example displays the following output:
//    Unhandled Exception: System.ArgumentOutOfRangeException:
//       Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.
//       at Example.GetMultiple(Int32 secondsDelay) in Program.cs:line 17
//       at Example.Main() in Program.cs:line 8
```

See also

- Methods

Ref returns and ref locals

1/23/2019 • 6 minutes to read • Edit Online

Starting with C# 7.0, C# supports reference return values (ref returns). A reference return value allows a method to return a reference to a variable, rather than a value, back to a caller. The caller can then choose to treat the returned variable as if it were returned by value or by reference. The caller can create a new variable that is itself a reference to the returned value, called a ref local.

What is a reference return value?

Most developers are familiar with passing an argument to a called method *by reference*. A called method's argument list includes a variable passed by reference. Any changes made to its value by the called method are observed by the caller. A *reference return value* means that a method returns a *reference* (or an alias) to some variable. That variable's scope must include the method. That variable's lifetime must extend beyond the return of the method. Modifications to the method's return value by the caller are made to the variable that is returned by the method.

Declaring that a method returns a *reference return value* indicates that the method returns an alias to a variable. The design intent is often that the calling code should have access to that variable through the alias, including to modify it. It follows that methods returning by reference can't have the return type `void`.

There are some restrictions on the expression that a method can return as a reference return value. Restrictions include:

- The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method. Attempting to return a local variable generates compiler error CS8168, "Cannot return local 'obj' by reference because it is not a ref local."
- The return value cannot be the literal `null`. Returning `null` generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

  A method with a ref return can return an alias to a variable whose value is currently the null (uninstantiated) value or a nullable type for a value type.
- The return value cannot be a constant, an enumeration member, the by-value return value from a property, or a method of a `class` or `struct`. Violating this rule generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

In addition, reference return values are not allowed on async methods. An asynchronous method may return before it has finished execution, while its return value is still unknown.

Defining a ref return value

A method that returns a *reference return value* must satisfy the following two conditions:

- The method signature includes the ref keyword in front of the return type.
- Each return statement in the method body includes the ref keyword in front of the name of the returned instance.

The following example shows a method that satisfies those conditions and returns a reference to a `Person` object named `p`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Consuming a ref return value

The ref return value is an alias to another variable in the called method's scope. You can interpret any use of the ref return as using the variable it aliases:

- When you assign its value, you are assigning a value to the variable it aliases.
- When you read its value, you are reading the value of the variable it aliases.
- If you return it *by reference*, you are returning an alias to that same variable.
- If you pass it to another method *by reference*, you are passing a reference to the variable it aliases.
- When you make a ref local alias, you make a new alias to the same variable.

Ref locals

Assume the `GetContactInformation` method is declared as a ref return:

```
public ref Person GetContactInformation(string fname, string lname)
```

A by-value assignment reads the value of a variable and assigns it to a new variable:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

The preceding assignment declares `p` as a local variable. Its initial value is copied from reading the value returned by `GetContactInformation`. Any future assignments to `p` will not change the value of the variable returned by `GetContactInformation`. The variable `p` is no longer an alias to the variable returned.

You declare a *ref local* variable to copy the alias to the original value. In the following assignment, `p` is an alias to the variable returned from `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

Subsequent usage of `p` is the same as using the variable returned by `GetContactInformation` because `p` is an alias for that variable. Changes to `p` also change the variable returned from `GetContactInformation`.

The `ref` keyword is used both before the local variable declaration *and* before the method call.

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how one can define a ref local value that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

The `ref` keyword is used both before the local variable declaration *and* before the value in the second example. Failure to include both `ref` keywords in the variable declaration and assignment in both examples results in compiler error CS8172, "Cannot initialize a by-reference variable with a value."

Prior to C# 7.3, ref local variables couldn't be reassigned to refer to different storage after being initialized. That

restriction has been removed. The following example shows a reassignment:

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Ref local variables must still be initialized when they are declared.

Ref returns and ref locals: an example

The following example defines a `NumberStore` class that stores an array of integer values. The `FindNumber` method returns by reference the first number that is greater than or equal to the number passed as an argument. If no number is greater than or equal to the argument, the method returns the number in index 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

The following example calls the `NumberStore.FindNumber` method to retrieve the first value that is greater than or equal to 16. The caller then doubles the value returned by the method. The output from the example shows the change reflected in the value of the array elements of the `NumberStore` instance.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//       Original sequence: 1 3 7 15 31 63 127 255 511 1023
//       New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Without support for reference return values, such an operation is performed by returning the index of the array element along with its value. The caller can then use this index to modify the value in a separate method call. However, the caller can also modify the index to access and possibly modify other array values.

The following example shows how the `FindNumber` method could be rewritten after C# 7.3 to use ref local reassignment:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr > 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

This second version is more efficient with longer sequences in scenarios where the number sought is closer to the end of the array.

See also

- ref keyword
- Write safe efficient code

Passing Parameters (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C#, arguments can be passed to parameters either by value or by reference. Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment. To pass a parameter by reference with the intent of changing the value, use the `ref`, or `out` keyword. To pass by reference with the intent of avoiding copying but not changing the value, use the `in` modifier. For simplicity, only the `ref` keyword is used in the examples in this topic. For more information about the difference between `in`, `ref`, and `out`, see in, ref, and out.

The following example illustrates the difference between value and reference parameters.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

For more information, see the following topics:

- Passing Value-Type Parameters
- Passing Reference-Type Parameters

C# Language Specification

For more information, see Argument lists in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Methods

Passing Value-Type Parameters (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

A value-type variable contains its data directly as opposed to a reference-type variable, which contains a reference to its data. Passing a value-type variable to a method by value means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the argument variable. If you want the called method to change the value of the parameter, you must pass it by reference, using the ref or out keyword. You may also use the in keyword to pass a value parameter by reference to avoid the copy while guaranteeing that the value will not be changed. For simplicity, the following examples use `ref`.

Passing Value Types by Value

The following example demonstrates passing value-type parameters by value. The variable `n` is passed by value to the method `SquareIt`. Any changes that take place inside the method have no affect on the original value of the variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

The variable `n` is a value type. It contains its data, the value `5`. When `SquareIt` is invoked, the contents of `n` are copied into the parameter `x`, which is squared inside the method. In `Main`, however, the value of `n` is the same after calling the `SquareIt` method as it was before. The change that takes place inside the method only affects the local variable `x`.

Passing Value Types by Reference

The following example is the same as the previous example, except that the argument is passed as a `ref` parameter. The value of the underlying argument, `n`, is changed when `x` is changed in the method.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

In this example, it is not the value of `n` that is passed; rather, a reference to `n` is passed. The parameter `x` is not an int; it is a reference to an `int`, in this case, a reference to `n`. Therefore, when `x` is squared inside the method, what actually is squared is what `x` refers to, `n`.

Swapping Value Types

A common example of changing the values of arguments is a swap method, where you pass two variables to the method, and the method swaps their contents. You must pass the arguments to the swap method by reference. Otherwise, you swap local copies of the parameters inside the method, and no change occurs in the calling method. The following example swaps integer values.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

When you call the `SwapByRef` method, use the `ref` keyword in the call, as shown in the following example.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
    i = 2  j = 3
    i = 3  j = 2
*/
```

See also

- C# Programming Guide
- Passing Parameters
- Passing Reference-Type Parameters

Passing Reference-Type Parameters (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

A variable of a reference type does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data belonging to the referenced object, such as the value of a class member. However, you cannot change the value of the reference itself; for example, you cannot use the same reference to allocate memory for a new class and have it persist outside the method. To do that, pass the parameter using the ref or out keyword. For simplicity, the following examples use `ref`.

Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, `arr`, by value, to a method, `Change`. Because the parameter is a reference to `arr`, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr
[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr [0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

In the preceding example, the array, `arr`, which is a reference type, is passed to the method without the `ref` parameter. In such a case, a copy of the reference, which points to `arr`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element, in this case from `1` to `888`. However, allocating a new portion of memory by using the new operator inside the `Change` method makes the variable `pArray` reference a new array. Thus, any changes after that will not affect the original array, `arr`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

Passing Reference Types by Reference

The following example is the same as the previous example, except that the `ref` keyword is added to the method

header and call. Any changes that take place in the method affect the original variable in the calling program.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

All of the changes that take place inside the method affect the original array in `Main`. In fact, the original array is reallocated using the `new` operator. Thus, after calling the `Change` method, any reference to `arr` points to the five-element array, which is created in the `Change` method.

Swapping Two Strings

Swapping strings is a good example of passing reference-type parameters by reference. In the example, two strings, `str1` and `str2`, are initialized in `Main` and passed to the `SwapStrings` method as parameters modified by the `ref` keyword. The two strings are swapped inside the method and inside `Main` as well.

```
class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);   // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
    Inside Main, before swapping: John Smith
    Inside the method: Smith John
    Inside Main, after swapping: Smith John
*/
```

In this example, the parameters need to be passed by reference to affect the variables in the calling program. If you remove the `ref` keyword from both the method header and the method call, no changes will take place in the calling program.

For more information about strings, see string.

See also

- C# Programming Guide
- Passing Parameters
- ref
- in
- out
- Reference Types

How to: Know the Difference Between Passing a Struct and Passing a Class Reference to a Method (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following example demonstrates how passing a struct to a method differs from passing a class instance to a method. In the example, both of the arguments (struct and class instance) are passed by value, and both methods change the value of one field of the argument. However, the results of the two methods are not the same because what is passed when you pass a struct differs from what is passed when you pass an instance of a class.

Because a struct is a value type, when you pass a struct by value to a method, the method receives and operates on a copy of the struct argument. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a reference type, not a value type. When a reference type is passed by value to a method, the method receives a copy of the reference to the class instance. That is, the method receives a copy of the address of the instance, not a copy of the instance itself. The class instance in the calling method has an address, the parameter in the called method has a copy of the address, and both addresses refer to the same object. Because the parameter contains only a copy of the address, the called method cannot change the address of the class instance in the calling method. However, the called method can use the address to access the class members that both the original address and the copy reference. If the called method changes a class member, the original class instance in the calling method also changes.

The output of the following example illustrates the difference. The value of the `willIChange` field of the class instance is changed by the call to method `ClassTaker` because the method uses the address in the parameter to find the specified field of the class instance. The `willIChange` field of the struct in the calling method is not changed by the call to method `StructTaker` because the value of the argument is a copy of the struct itself, not a copy of its address. `StructTaker` changes the copy, and the copy is lost when the call to `StructTaker` is completed.

Example

```
class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

See also

- C# Programming Guide
- Classes
- Structs
- Passing Parameters

Implicitly typed local variables (C# Programming Guide)

1/11/2019 • 4 minutes to read • Edit Online

Local variables can be declared without giving an explicit type. The `var` keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET Framework class library. For more information about how to initialize arrays with `var`, see Implicitly Typed Arrays.

The following examples show various ways in which local variables can be declared with `var`:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

It is important to understand that the `var` keyword does not mean "variant" and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The `var` keyword may be used in the following contexts:

- On local variables (variables declared at method scope) as shown in the previous example.
- In a for initialization statement.

  ```
  for(var x = 1; x < 10; x++)
  ```

- In a foreach initialization statement.

  ```
  foreach(var item in list){...}
  ```

- In a using statement.

  ```
  using (var file = new StreamReader("C:\\myfile.txt")) {...}
  ```

For more information, see How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression.

var and anonymous types

In many cases the use of `var` is optional and is just a syntactic convenience. However, when a variable is initialized with an anonymous type you must declare the variable as `var` if you need to access the properties of the object at a later point. This is a common scenario in LINQ query expressions. For more information, see Anonymous Types.

From the perspective of your source code, an anonymous type has no name. Therefore, if a query variable has been initialized with `var`, then the only way to access the properties in the returned sequence of objects is to use `var` as the type of the iteration variable in the `foreach` statement.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Remarks

The following restrictions apply to implicitly-typed variable declarations:

- `var` can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
- `var` cannot be used on fields at class scope.
- Variables declared by using `var` cannot be used in the initialization expression. In other words, this expression is legal `: int i = (i = 20);` but this expression produces a compile-time error: `var i = (i = 20);`
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- If a type named `var` is in scope, then the `var` keyword will resolve to that type name and will not be treated as part of an implicitly typed local variable declaration.

You may find that `var` can also be useful with query expressions in which the exact constructed type of the query variable is difficult to determine. This can occur with grouping and ordering operations.

The `var` keyword can also be useful when the specific type of the variable is tedious to type on the keyboard, or is

obvious, or does not add to the readability of the code. One example where `var` is helpful in this manner is with nested generic types such as those used with group operations. In the following query, the type of the query variable is `IEnumerable<IGrouping<string, Student>>`. As long as you and others who must maintain your code understand this, there is no problem with using implicit typing for convenience and brevity.

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

However, the use of `var` does have at least the potential to make your code more difficult to understand for other developers. For that reason, the C# documentation generally uses `var` only when it is required.

See also

- C# Reference
- Implicitly Typed Arrays
- How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression
- Anonymous Types
- Object and Collection Initializers
- var
- LINQ Query Expressions
- LINQ (Language-Integrated Query)
- for
- foreach, in
- using Statement

How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can use implicitly typed local variables whenever you want the compiler to determine the type of a local variable. You must use implicitly typed local variables to store anonymous types, which are often used in query expressions. The following examples illustrate both optional and required uses of implicitly typed local variables in queries.

Implicitly typed local variables are declared by using the var contextual keyword. For more information, see Implicitly Typed Local Variables and Implicitly Typed Arrays.

Example

The following example shows a common scenario in which the `var` keyword is required: a query expression that produces a sequence of anonymous types. In this scenario, both the query variable and the iteration variable in the `foreach` statement must be implicitly typed by using `var` because you do not have access to a type name for the anonymous type. For more information about anonymous types, see Anonymous Types.

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

Example

The following example uses the `var` keyword in a situation that is similar, but in which the use of `var` is optional. Because `student.LastName` is a string, execution of the query returns a sequence of strings. Therefore, the type of `queryID` could be declared as `System.Collections.Generic.IEnumerable<string>` instead of `var`. Keyword `var` is used for convenience. In the example, the iteration variable in the `foreach` statement is explicitly typed as a string, but it could instead be declared by using `var`. Because the type of the iteration variable is not an anonymous type, the use of `var` is an option, not a requirement. Remember, `var` itself is not a type, but an instruction to the compiler to infer and assign the type.

```
// Variable queryID could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryID =
    from student in students
    where student.ID > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryID)
{
    Console.WriteLine("Last name: {0}", str);
}
```

See also

- C# Programming Guide
- Extension Methods
- LINQ (Language-Integrated Query)
- var
- LINQ Query Expressions

Extension Methods (C# Programming Guide)

1/23/2019 • 7 minutes to read • Edit Online

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

The most common extension methods are the LINQ standard query operators that add query functionality to the existing System.Collections.IEnumerable and System.Collections.Generic.IEnumerable<T> types. To use the standard query operators, first bring them into scope with a `using System.Linq` directive. Then any type that implements IEnumerable<T> appears to have instance methods such as GroupBy, OrderBy, Average, and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an IEnumerable<T> type such as List<T> or Array.

The following example shows how to call the standard query operator `OrderBy` method on an array of integers. The expression in parentheses is a lambda expression. Many standard query operators take lambda expressions as parameters, but this is not a requirement for extension methods. For more information, see Lambda Expressions.

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on, and the parameter is preceded by the this modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

The following example shows an extension method defined for the System.String class. Note that it is defined inside a non-nested, non-generic static class:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

The `WordCount` extension method can be brought into scope with this `using` directive:

```
using ExtensionMethods;
```

And it can be called from an application by using this syntax:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

In your code you invoke the extension method with instance method syntax. However, the intermediate language (IL) generated by the compiler translates your code into a call on the static method. Therefore, the principle of encapsulation is not really being violated. In fact, extension methods cannot access private variables in the type they are extending.

For more information, see How to: Implement and Call a Custom Extension Method.

In general, you will probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code. To enable extension methods for a particular type, just add a `using` directive for the namespace in which the methods are defined. For example, to use the standard query operators, add this `using` directive to your code:

```
using System.Linq;
```

(You may also have to add a reference to System.Core.dll.) You will notice that the standard query operators now appear in IntelliSense as additional methods available for most IEnumerable<T> types.

Binding Extension Methods at Compile Time

You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than instance methods defined in the type itself. In other words, if a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.

Example

The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class `Extensions` contains extension methods defined for any type that implements `IMyInterface`. Classes `A`, `B`, and `C` all implement the interface.

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes.

When the compiler cannot find an instance method with a matching signature, it will bind to a matching extension method if one exists.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
```

```
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}


// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}


// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
```

```
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

General Guidelines

In general, we recommend that you implement extension methods sparingly and only when you have to. Whenever possible, client code that must extend an existing type should do so by creating a new type derived from the existing type. For more information, see Inheritance.

When using an extension method to extend a type whose source code you cannot change, you run the risk that a change in the implementation of the type will cause your extension method to break.

If you do implement extension methods for a given type, remember the following points:

- An extension method will never be called if it has the same signature as a method defined in the type.
- Extension methods are brought into scope at the namespace level. For example, if you have multiple static classes that contain extension methods in a single namespace named `Extensions`, they will all be brought into scope by the `using Extensions;` directive.

For a class library that you implemented, you shouldn't use extension methods to avoid incrementing the version number of an assembly. If you want to add significant functionality to a library for which you own the source code, you should follow the standard .NET Framework guidelines for assembly versioning. For more information, see Assembly Versioning.

See also

- C# Programming Guide
- Parallel Programming Samples (these include many example extension methods)
- Lambda Expressions
- Standard Query Operators Overview
- Conversion rules for Instance parameters and their impact
- Extension methods Interoperability between languages
- Extension methods and Curried Delegates
- Extension method Binding and Error reporting

How to: Implement and Call a Custom Extension Method (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This topic shows how to implement your own extension methods for any .NET type. Client code can use your extension methods by adding a reference to the DLL that contains them, and adding a using directive that specifies the namespace in which the extension methods are defined.

To define and call the extension method

1. Define a static class to contain the extension method.

   The class must be visible to client code. For more information about accessibility rules, see Access Modifiers.

2. Implement the extension method as a static method with at least the same visibility as the containing class.

3. The first parameter of the method specifies the type that the method operates on; it must be preceded with the this modifier.

4. In the calling code, add a `using` directive to specify the namespace that contains the extension method class.

5. Call the methods as if they were instance methods on the type.

   Note that the first parameter is not specified by calling code because it represents the type on which the operator is being applied, and the compiler already knows the type of your object. You only have to provide arguments for parameters 2 through `n`.

Example

The following example implements an extension method named `WordCount` in the `CustomExtensions.StringExtension` class. The method operates on the String class, which is specified as the first method parameter. The `CustomExtensions` namespace is imported into the application namespace, and the method is called inside the `Main` method.

```
using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.','?'}, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}
```

Compiling the Code

To run this code, copy and paste it into a Visual C# console application project that has been created in Visual Studio. By default, this project targets version 3.5 of the .NET Framework, and it has a reference to System.Core.dll and a `using` directive for System.Linq. If one or more of these requirements are missing from the project, you can add them manually.

.NET Framework Security

Extension methods present no specific security vulnerabilities. They can never be used to impersonate existing methods on a type, because all name collisions are resolved in favor of the instance or static method defined by the type itself. Extension methods cannot access any private data in the extended class.

See also

- C# Programming Guide
- Extension Methods
- LINQ (Language-Integrated Query)
- Static Classes and Static Class Members
- protected
- internal
- public

- this
- namespace

How to: Create a New Method for an Enumeration (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can use extension methods to add functionality specific to a particular enum type.

Example

In the following example, the `Grades` enumeration represents the possible letter grades that a student may receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not.

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

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
  }
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
 */
```

Note that the `Extensions` class also contains a static variable that is updated dynamically and that the return value

of the extension method reflects the current value of that variable. This demonstrates that, behind the scenes, extension methods are invoked directly on the static class in which they are defined.

Compiling the Code

To run this code, copy and paste it into a Visual C# console application project that has been created in Visual Studio. By default, this project targets version 3.5 of the .NET Framework, and it has a reference to System.Core.dll and a `using` directive for System.Linq. If one or more of these requirements are missing from the project, you can add them manually.

See also

- C# Programming Guide
- Extension Methods

Named and Optional Arguments (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

C# 4 introduces named and optional arguments. *Named arguments* enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than with the parameter's position in the parameter list. *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters, when used together, enable you to supply arguments for only a few parameters from a list of optional parameters. This capability greatly facilitates calls to COM interfaces such as the Microsoft Office Automation APIs.

Named Arguments

Named arguments free you from the need to remember or to look up the order of parameters in the parameter lists of called methods. The parameter for each argument can be specified by parameter name. For example, a function that prints order details (such as, seller name, order number & product name) can be called in the standard way by sending arguments by position, in the order defined by the function.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

If you do not remember the order of the parameters but know their names, you can send the arguments in any order.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
```

```
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Named arguments also improve the readability of your code by identifying what each argument represents. In the example method below, the `sellerName` cannot be null or white space. As both `sellerName` and `productName` are string types, instead of sending arguments by position, it makes sense to use named arguments to disambiguate the two and reduce confusion for anyone reading the code.

Named arguments, when used with positional arguments, are valid as long as

- they're not followed by any positional arguments, or

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- *starting with C# 7.2*, they're used in the correct position. In the example below, the parameter `orderNum` is in the correct position but isn't explicitly named.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

However, out-of-order named arguments are invalid if they're followed by positional arguments.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Example

The following code implements the examples from this section along with some additional ones.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                   // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Optional Arguments

The definition of a method, constructor, indexer, or delegate can specify that its parameters are required or that they are optional. Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.

Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. A default value must be one of the following types of expressions:

- a constant expression;
- an expression of the form `new ValType()`, where `ValType` is a value type, such as an enum or a struct;
- an expression of the form default(ValType), where `ValType` is a value type.

Optional parameters are defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters. Comma-separated gaps in the argument list are not supported. For example, in the following code, instance method `ExampleMethod` is defined with one required and two optional parameters.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

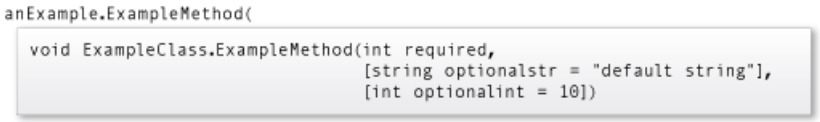
The following call to `ExampleMethod` causes a compiler error, because an argument is provided for the third parameter but not for the second.

```
//anExample.ExampleMethod(3, ,4);
```

However, if you know the name of the third parameter, you can use a named argument to accomplish the task.

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense uses brackets to indicate optional parameters, as shown in the following illustration.



Optional parameters in ExampleMethod

> **NOTE**
>
> You can also declare optional parameters by using the .NET OptionalAttribute class. `OptionalAttribute` parameters do not require a default value.

Example

In the following example, the constructor for `ExampleClass` has one parameter, which is optional. Instance method `ExampleMethod` has one required parameter, `required`, and two optional parameters, `optionalstr` and `optionalint`. The code in `Main` shows the different ways in which the constructor and method can be invoked.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);
```

```
            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default string",
            int optionalint = 10)
        {
            Console.WriteLine("{0}: {1}, {2}, and {3}.", _name, required, optionalstr,
                optionalint);
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.

}
```

COM Interfaces

Named and optional arguments, along with support for dynamic objects and other enhancements, greatly improve interoperability with COM APIs, such as Office Automation APIs.

For example, the AutoFormat method in the Microsoft Office Excel Range interface has seven parameters, all of which are optional. These parameters are shown in the following illustration.

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

AutoFormat parameters

In C# 3.0 and earlier versions, an argument is required for each parameter, as shown in the following example.

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

However, you can greatly simplify the call to `AutoFormat` by using named and optional arguments, introduced in C# 4.0. Named and optional arguments enable you to omit the argument for an optional parameter if you do not want to change the parameter's default value. In the following call, a value is specified for only one of the seven parameters.

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

For more information and examples, see How to: Use Named and Optional Arguments in Office Programming and How to: Access Office Interop Objects by Using Visual C# Features.

Overload Resolution

Use of named and optional arguments affects overload resolution in the following ways:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that does not have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- How to: Use Named and Optional Arguments in Office Programming
- Using Type dynamic
- Using Constructors
- Using Indexers

How to: Use Named and Optional Arguments in Office Programming (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

Named arguments and optional arguments, introduced in C# 4, enhance convenience, flexibility, and readability in C# programming. In addition, these features greatly facilitate access to COM interfaces such as the Microsoft Office automation APIs.

In the following example, method ConvertToTable has sixteen parameters that represent characteristics of a table, such as number of columns and rows, formatting, borders, fonts, and colors. All sixteen parameters are optional, because most of the time you do not want to specify particular values for all of them. However, without named and optional arguments, a value or a placeholder value has to be provided for each parameter. With named and optional arguments, you specify values only for the parameters that are required for your project.

You must have Microsoft Office Word installed on your computer to complete these procedures.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Templates Categories** pane, expand **Visual C#**, and then click **Windows**.
4. Look in the top of the **Templates** pane to make sure that **.NET Framework 4** appears in the **Target Framework** box.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

   The new project appears in **Solution Explorer**.

To add a reference

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **.NET** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list.
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the **Program.cs** file and then click **View Code**.
2. Add the following `using` directives to the top of the code file.

```
using Word = Microsoft.Office.Interop.Word;
```

To display text in a Word document

1. In the `Program` class in Program.cs, add the following method to create a Word application and a Word document. The Add method has four optional parameters. This example uses their default values. Therefore, no arguments are necessary in the calling statement.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Add the following code at the end of the method to define where to display text in the document, and what text to display.

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

To run the application

1. Add the following statement to Main.

```
DisplayInWord();
```

2. Press CTRL+F5 to run the project. A Word document appears that contains the specified text.

To change the text to a table

1. Use the `ConvertToTable` method to enclose the text in a table. The method has sixteen optional parameters. IntelliSense encloses optional parameters in brackets, as shown in the following illustration.

```
range.ConvertToTable(
```

```
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

ConvertToTable parameters

Named and optional arguments enable you to specify values for only the parameters that you want to change. Add the following code to the end of method `DisplayInWord` to create a simple table. The argument specifies that the commas in the text string in `range` separate the cells of the table.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

In earlier versions of C#, the call to `ConvertToTable` requires a reference argument for each parameter, as shown in the following code.

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing);
```

2. Press CTRL+F5 to run the project.

To experiment with other parameters

1. To change the table so that it has one column and three rows, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. To specify a predefined format for the table, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5. The format can be any of the WdTableFormat constants.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Example

The following code includes the full example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;


namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

See also

- Named and Optional Arguments

Constructors (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Whenever a class or struct is created, its constructor is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read. For more information and examples, see Using Constructors and Instance Constructors.

Default constructors

If you don't provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values as listed in the Default Values Table. If you don't provide a constructor for your struct, C# relies on an *implicit default constructor* to automatically initialize each field of a value type to its default value as listed in the Default Values Table. For more information and examples, see Instance Constructors.

Constructor syntax

A constructor is a method whose name is the same as the name of its type. Its method signature includes only the method name and its parameter list; it does not include a return type. The following example shows the constructor for a class named `Person`.

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

If a constructor can be implemented as a single statement, you can use an expression body definition. The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `locationName` field.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Static constructors

The previous examples have all shown instance constructors, which create a new object. A class or struct can also have a static constructor, which initializes static members of the type. Static constructors are parameterless. If you don't provide a static constructor to initialize static fields, the C# compiler will supply a default static constructor that initializes static fields to their default value as listed in the Default Values Table.

The following example uses a static constructor to initialize a static field.

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

You can also define a static constructor with an expression body definition, as the following example shows.

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

For more information and examples, see Static Constructors.

In This Section

Using Constructors

Instance Constructors

Private Constructors

Static Constructors

How to: Write a Copy Constructor

See also

- C# Programming Guide
- Classes and Structs
- Finalizers
- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One

Using Constructors (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

When a class or struct is created, its constructor is called. Constructors have the same name as the class or struct, and they usually initialize the data members of the new object.

In the following example, a class named `Taxi` is defined by using a simple constructor. This class is then instantiated with the new operator. The `Taxi` constructor is invoked by the `new` operator immediately after memory is allocated for the new object.

```
public class Taxi
{
    public bool IsInitialized;
    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

A constructor that takes no parameters is called a *default constructor*. Default constructors are invoked whenever an object is instantiated by using the `new` operator and no arguments are provided to `new`. For more information, see Instance Constructors.

Unless the class is static, classes without constructors are given a public default constructor by the C# compiler in order to enable class instantiation. For more information, see Static Classes and Static Class Members.

You can prevent a class from being instantiated by making the constructor private, as follows:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double E = Math.E;  //2.71828...
}
```

For more information, see Private Constructors.

Constructors for struct types resemble class constructors, but `structs` cannot contain an explicit default constructor because one is provided automatically by the compiler. This constructor initializes each field in the `struct` to the default values. For more information, see Default Values Table. However, this default constructor is only invoked if the `struct` is instantiated with `new`. For example, this code uses the default constructor for Int32, so that you are assured that the integer is initialized:

```
int i = new int();
Console.WriteLine(i);
```

The following code, however, causes a compiler error because it does not use `new`, and because it tries to use an object that has not been initialized:

```
int i;
Console.WriteLine(i);
```

Alternatively, objects based on `structs` (including all built-in numeric types) can be initialized or assigned and then used as in the following example:

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

So calling the default constructor for a value type is not required.

Both classes and `structs` can define constructors that take parameters. Constructors that take parameters must be called through a `new` statement or a base statement. Classes and `structs` can also define multiple constructors, and neither is required to define a default constructor. For example:

```
public class Employee
{
    public int Salary;

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

This class can be created by using either of the following statements:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

A constructor can use the `base` keyword to call the constructor of a base class. For example:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

In this example, the constructor for the base class is called before the block for the constructor is executed. The

`base` keyword can be used with or without parameters. Any parameters to the constructor can be used as parameters to `base`, or as part of an expression. For more information, see base.

In a derived class, if a base-class constructor is not called explicitly by using the `base` keyword, the default constructor, if there is one, is called implicitly. This means that the following constructor declarations are effectively the same:

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

If a base class does not offer a default constructor, the derived class must make an explicit call to a base constructor by using `base`.

A constructor can invoke another constructor in the same object by using the this keyword. Like `base`, `this` can be used with or without parameters, and any parameters in the constructor are available as parameters to `this`, or as part of an expression. For example, the second constructor in the previous example can be rewritten using `this`:

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

The use of the `this` keyword in the previous example causes this constructor to be called:

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Constructors can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can construct the class. For more information, see Access Modifiers.

A constructor can be declared static by using the static keyword. Static constructors are called automatically, immediately before any static fields are accessed, and are generally used to initialize static class members. For more information, see Static Constructors.

C# Language Specification

For more information, see Instance constructors and Static constructors in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

- Classes and Structs
- Constructors
- Finalizers

Instance Constructors (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

Instance constructors are used to create and initialize any instance member variables when you use the new expression to create an object of a class. To initialize a static class, or static variables in a non-static class, you must define a static constructor. For more information, see Static Constructors.

The following example shows an instance constructor:

```
class Coords
{
    public int x, y;

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

NOTE

For clarity, this class contains public fields. The use of public fields is not a recommended programming practice because it allows any method anywhere in a program unrestricted and unverified access to an object's inner workings. Data members should generally be private, and should be accessed only through class methods and properties.

This instance constructor is called whenever an object based on the `Coords` class is created. A constructor like this one, which takes no arguments, is called a *default constructor*. However, it is often useful to provide additional constructors. For example, we can add a constructor to the `Coords` class that allows us to specify the initial values for the data members:

```
// A constructor with two arguments:
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

This allows `CoOrd` objects to be created with default or specific initial values, like this:

```
Coords p1 = new Coords();
Coords p2 = new Coords(5, 3);
```

If a class does not have a constructor, a default constructor is automatically generated and default values are used to initialize the object fields. For example, an int is initialized to 0. For more information on default values, see Default Values Table. Therefore, because the `Coords` class default constructor initializes all data members to zero, it can be removed altogether without changing how the class works. A complete example using multiple constructors is provided in Example 1 later in this topic, and an example of an automatically generated constructor is provided in Example 2.

Instance constructors can also be used to call the instance constructors of base classes. The class constructor can invoke the constructor of the base class through the initializer, as follows:

```
class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}
```

In this example, the `Circle` class passes values representing radius and height to the constructor provided by `Shape` from which `Circle` is derived. A complete example using `Shape` and `Circle` appears in this topic as Example 3.

Example 1

The following example demonstrates a class with two class constructors, one without arguments and one with two arguments.

```
class Coords
{
    public int x, y;

    // Default constructor:
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments:
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method:
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        Coords p1 = new Coords();
        Coords p2 = new Coords(5, 3);

        // Display the results using the overriden ToString method:
        Console.WriteLine("Coords #1 at {0}", p1);
        Console.WriteLine("Coords #2 at {0}", p2);
        Console.ReadKey();
    }
}
/* Output:
 Coords #1 at (0,0)
 Coords #2 at (5,3)
*/
```

Example 2

In this example, the class `Person` does not have any constructors, in which case, a default constructor is automatically provided and the fields are initialized to their default values.

```
public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        Person person = new Person();

        Console.WriteLine("Name: {0}, Age: {1}", person.name, person.age);
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:  Name: , Age: 0
```

Notice that the default value of `age` is `0` and the default value of `name` is `null`. For more information on default values, see Default Values Table.

Example 3

The following example demonstrates using the base class initializer. The `Circle` class is derived from the general class `Shape`, and the `Cylinder` class is derived from the `Circle` class. The constructor on each derived class is using its base class initializer.

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
    public override double Area()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area of the circle = 19.63
    Area of the cylinder = 86.39
*/
```

For more examples on invoking the base class constructors, see virtual, override, and base.

See also

- C# Programming Guide
- Classes and Structs
- Constructors
- Finalizers
- static

Private Constructors (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class. For example:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

The declaration of the empty constructor prevents the automatic generation of a default constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the private modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the Math class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static. For more information see Static Classes and Static Class Members.

Example

The following is an example of a class using a private constructor.

```
public class Counter
{
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Notice that if you uncomment the following statement from the example, it will generate an error because the constructor is inaccessible because of its protection level:

```
// Counter aCounter = new Counter();   // Error
```

C# Language Specification

For more information, see Private constructors in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Classes and Structs
- Constructors
- Finalizers
- private
- public

Static Constructors (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Static constructors have the following properties:

- A static constructor does not take access modifiers or have parameters.
- A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
- A static constructor cannot be called directly.
- The user has no control on when the static constructor is executed in the program.
- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the `LoadLibrary` method.
- If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.

Example

In this example, class `Bus` has a static constructor. When the first instance of `Bus` is created (`bus1`), the static constructor is invoked to initialize the class. The sample output verifies that the static constructor runs only one time, even though two instances of `Bus` are created, and that it runs before the instance constructor runs.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
```

```
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to simulate
        // actual bus times. Do not do this in your actual bus schedule program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

See also

- C# Programming Guide

- Classes and Structs
- Constructors
- Static Classes and Static Class Members
- Finalizers

How to: Write a Copy Constructor (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

C# doesn't provide a copy constructor for objects, but you can write one yourself.

Example

In the following example, the `Person` class defines a copy constructor that takes, as its argument, an instance of `Person`. The values of the properties of the argument are assigned to the properties of the new instance of `Person`. The code contains an alternative copy constructor that sends the `Name` and `Age` properties of the instance that you want to copy to the instance constructor of the class.

```
class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

See also

- ICloneable

- C# Programming Guide
- Classes and Structs
- Constructors
- Finalizers

Finalizers (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

Finalizers (which are also called **destructors**) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the `Car` class.

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

A finalizer can also be implemented as an expression body definition, as the following example shows.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

The finalizer implicitly calls Finalize on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

This means that the `Finalize` method is called recursively for all instances in the inheritance chain, from the

most-derived to the least-derived.

> **NOTE**
> Empty finalizers should not be used. When a class contains a finalizer, an entry is created in the `Finalize` queue. When the finalizer is called, the garbage collector is invoked to process the queue. An empty finalizer just causes a needless loss of performance.

The programmer has no control over when the finalizer is called because this is determined by the garbage collector. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object.

In .NET Framework applications (but not in .NET Core applications), finalizers are also called when the program exits.

It is possible to force garbage collection by calling Collect, but most of the time, this should be avoided because it may create performance issues.

Using finalizers to release resources

In general, C# does not require as much memory management as is needed when you develop with a language that does not target a runtime with garbage collection. This is because the .NET Framework garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the `Finalize` method of the object.

Explicit release of resources

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. You do this by implementing a `Dispose` method from the IDisposable interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the `Dispose` method failed.

For more details about cleaning up resources, see the following topics:

- Cleaning Up Unmanaged Resources
- Implementing a Dispose Method
- using Statement

Example

The following example creates three classes that make a chain of inheritance. The class `First` is the base class, `Second` is derived from `First`, and `Third` is derived from `Second`. All three have finalizers. In `Main`, an instance of the most-derived class is created. When the program runs, notice that the finalizers for the three classes are called automatically, and in order, from the most-derived to the least-derived.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

class TestFinalizers
{
    static void Main()
    {
        Third t = new Third();
    }

}
/* Output (to VS Output Window):
    Third's finalizer is called.
    Second's finalizer is called.
    First's finalizer is called.
*/
```

C# language specification

For more information, see the Destructors section of the C# language specification.

See also

- IDisposable
- C# Programming Guide
- Constructors
- Garbage Collection

Object and Collection Initializers (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

C# lets you instantiate an object or collection and perform member assignments in a single statement.

Object initializers

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments (and parentheses syntax). The following example shows how to use an object initializer with a named type, `Cat` and how to invoke the default constructor. Note the use of auto-implemented properties in the `Cat` class. For more information, see Auto-Implemented Properties.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

The object initializers syntax allows you to create an instance, and after that it assigns the newly created object, with its assigned properties, to the variable in the assignment.

Starting with C# 6, object initializers can set indexers, in addition to assigning fields and properties. Consider this basic `Matrix` class:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

You could initialize the identity matrix with the following code:

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Any accessible indexer that contains an accessible setter can be used as one of the expressions in an object initializer, regardless of the number or types of arguments. The index arguments form the left side of the assignment, and the value is the right side of the expression. For example, these are all valid if `IndexersExample` has the appropriate indexers:

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Baz = Math.PI,
    ['C',4] = "Middle C"
}
```

For the preceding code to compile, the `IndexersExample` type must have the following members:

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
}
```

Object Initializers with anonymous types

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. Query expressions make frequent use of anonymous types, which can only be initialized by using an object initializer, as shown in the following declaration.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Anonymous types enable the `select` clause in a LINQ query expression to transform objects of the original sequence into objects whose value and shape may differ from the original. This is useful if you want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (`p`) contains many fields and methods, and that you are only interested in creating a sequence of objects that contain the product name and the unit price.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

When this query is executed, the `productInfos` variable will contain a sequence of objects that can be accessed in a `foreach` statement as shown in this example:

```
foreach(var p in productInfos){...}
```

Each object in the new anonymous type has two public properties that receive the same names as the properties or fields in the original object. You can also rename a field when you are creating an anonymous type; the following example renames the `UnitPrice` field to `Price`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

Collection initializers

Collection initializers let you specify one or more element initializers when you initialize a collection type that implements IEnumerable and has `Add` with the appropriate signature as an instance method or an extension method. The element initializers can be a simple value, an expression, or an object initializer. By using a collection initializer, you do not have to specify multiple calls; the compiler adds the calls automatically.

The following example shows two simple collection initializers:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

The following collection initializer uses object initializers to initialize objects of the `Cat` class defined in a previous example. Note that the individual object initializers are enclosed in braces and separated by commas.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

You can specify null as an element in a collection initializer if the collection's `Add` method allows it.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

You can specify indexed elements if the collection supports read / write indexing.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

The preceding sample generates code that calls the Item[TKey] to set the values. Beginning with C# 6, you can initialize dictionaries and other associative containers using the following syntax. Notice that instead of indexer syntax, with parentheses and an assignment, it uses an object with multiple values:

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

This initializer example calls Add(TKey, TValue) to add the three items into the dictionary. These two different ways to initialize associative collections have slightly different behavior because of the method calls the compiler generates. Both variants work with the `Dictionary` class. Other types may only support one or the other based on their public API.

Examples

The following example combines the concepts of object and collection initializers.

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat{ Name = "Sylvester", Age=8 },
            new Cat{ Name = "Whiskers", Age=2 },
            new Cat{ Name = "Sasha", Age=14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat{ Name = "Furrytail", Age=5 },
            new Cat{ Name = "Peaches", Age=4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

The following example shows an object that implements IEnumerable and contains an `Add` method with multiple parameters, It uses a collection initializer with multiple elements per item in the list that correspond to the signature of the `Add` method.

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

`Add` methods can use the `params` keyword to take a variable number of arguments, as shown in the following example. This example also demonstrates the custom implementation of an indexer to initialize a collection using indexes.

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();
```

```
        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue> storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");
```

```
            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
     */
}
```

See also

- C# Programming Guide
- LINQ Query Expressions
- Anonymous Types

How to: Initialize Objects by Using an Object Initializer (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type.

The following examples show how to use object initializers with named objects. The compiler processes object initializers by first accessing the default instance constructor and then processing the member initializations. Therefore, if the default constructor is declared as `private` in the class, object initializers that require public access will fail.

You must use an object initializer if you're defining an anonymous type. For more information, see How to: Return Subsets of Element Properties in a Query.

Example

The following example shows how to initialize a new `StudentName` type by using object initializers. This example sets properties in the `StudentName` type:

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The default constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the default constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
```

```
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // The default constructor has no parameters. The default constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Object initializers can be used to set indexers in an object. The following example defines a `BaseballTeam` class that uses an indexer to get and set players at different positions. The initializer can assign players, based on the abbreviation for the position, or the number used for each position baseball scorecards:

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

See also

- C# Programming Guide
- Object and Collection Initializers

How to initialize a dictionary with a collection initializer (C# Programming Guide)

1/8/2019 • 2 minutes to read • Edit Online

A Dictionary<TKey,TValue> contains a collection of key/value pairs. Its Add method takes two parameters, one for the key and one for the value. One way to initialize a Dictionary<TKey,TValue>, or any collection whose `Add` method takes multiple parameters, is to enclose each set of parameters in braces as shown in the following example. Another option is to use an index initializer, also shown in the following example.

Example

In the following code example, a Dictionary<TKey,TValue> is initialized with instances of type `StudentName`. The first initialization uses the `Add` method with two arguments. The compiler generates a call to `Add` for each of the pairs of `int` keys and `StudentName` values. The second uses a public read / write indexer method of the `Dictionary` class:

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        Dictionary<int, StudentName> students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }

        Dictionary<int, StudentName> students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
    }
}
```

Note the two pairs of braces in each element of the collection in the first declaration. The innermost braces enclose the object initializer for the `StudentName`, and the outermost braces enclose the initializer for the key/value pair that

will be added to the `students` Dictionary<TKey,TValue>. Finally, the whole collection initializer for the dictionary is enclosed in braces. In the second initialization, the left side of the assignment is the key and the right side is the value, using an object initializer for `StudentName`.

See also

- C# Programming Guide
- Object and Collection Initializers

Nested Types (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

A type defined within a class or struct is called a nested type. For example:

```
class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Regardless of whether the outer type is a class or a struct, nested types default to private; they are accessible only from their containing type. In the previous example, the `Nested` class is inaccessible to external types.

You can also specify an access modifier to define the accessibility of a nested type, as follows:

- Nested types of a **class** can be public, protected, internal, protected internal, private or private protected.

  However, defining a `protected`, `protected internal` or `private protected` nested class inside a sealed class generates compiler warning CS0628, "new protected member declared in sealed class."

- Nested types of a **struct** can be public, internal, or private.

The following example makes the `Nested` class public:

```
class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

The nested, or inner, type can access the containing, or outer, type. To access the containing type, pass it as an argument to the constructor of the nested type. For example:

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

A nested type has access to all of the members that are accessible to its containing type. It can access private and

protected members of the containing type, including any inherited protected members.

In the previous declaration, the full name of class `Nested` is `Container.Nested`. This is the name used to create a new instance of the nested class, as follows:

```
Container.Nested nest = new Container.Nested();
```

See also

- C# Programming Guide
- Classes and Structs
- Access Modifiers
- Constructors

Partial Classes and Methods (C# Programming Guide)

1/23/2019 • 6 minutes to read • Edit Online

It is possible to split the definition of a class, a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- To split a class definition, use the partial keyword modifier, as shown here:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

> **NOTE**
>
> The `partial` modifier is not available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They are equivalent to the following declarations:

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They are equivalent to the following declarations:

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

```
public partial class A { }
//public class A { }  // Error, must also be marked partial
```

- The `partial` modifier can only appear immediately before the keywords `class`, `struct`, or `interface`.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - base class
  - new modifier (nested parts)
  - generic constraints

    For more information, see Constraints on Type Parameters.

Example 1

Description

In the following example, the fields and the constructor of the class, `CoOrds`, are declared in one partial class definition, and the member, `PrintCoOrds`, is declared in another partial class definition.

Code

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }

}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

Example 2

Description

The following example shows that you can also develop partial structs and interfaces.

Code

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Partial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

- Partial method declarations must begin with the contextual keyword partial and the method must return void.
- Partial methods can have in or ref but not out parameters.
- Partial methods are implicitly private, and therefore they cannot be virtual.
- Partial methods cannot be extern, because the presence of the body determines whether they are defining or implementing.
- Partial methods can have static and unsafe modifiers.
- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
- You can make a delegate to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

C# Language Specification

For more information, see Partial types in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

- Classes
- Structs
- Interfaces
- partial (Type)

Anonymous Types (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the new operator together with an object initializer. For more information about object initializers, see Object and Collection Initializers.

The following example shows an anonymous type that is initialized with two properties named `Amount` and `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Anonymous types typically are used in the select clause of a query expression to return a subset of the properties from each object in the source sequence. For more information about queries, see LINQ Query Expressions.

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be `null`, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named `Product`. Class `Product` includes `Color` and `Price` properties, together with other properties that you are not interested in. Variable `products` is a collection of `Product` objects. The anonymous type declaration starts with the `new` keyword. The declaration initializes a new type that uses only two properties from `Product`. This causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You must provide a name for a property that is being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are `Color` and `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using var. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type. For more information about `var`, see Implicitly Typed Local Variables.

You can create an array of anonymously typed elements by combining an implicitly typed local variable and an implicitly typed array, as shown in the following example.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Remarks

Anonymous types are class types that derive directly from object, and that cannot be cast to any type except object. The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

You cannot declare a field, a property, an event, or the return type of a method as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type object. However, doing this defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Because the Equals and GetHashCode methods on anonymous types are defined in terms of the `Equals` and `GetHashCode` methods of the properties, two instances of the same anonymous type are equal only if all their properties are equal.

See also

- C# Programming Guide
- Object and Collection Initializers
- Getting Started with LINQ in C#
- LINQ Query Expressions

How to: Return Subsets of Element Properties in a Query (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Use an anonymous type in a query expression when both of these conditions apply:

- You want to return only some of the properties of each source element.
- You do not have to store the query results outside the scope of the method in which the query is executed.

If you only want to return one property or field from each source element, then you can just use the dot operator in the `select` clause. For example, to return only the `ID` of each `student`, write the `select` clause as follows:

```
select student.ID;
```

Example

The following example shows how to use an anonymous type to return only a subset of the properties of each source element that matches the specified condition.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Note that the anonymous type uses the source element's names for its properties if no names are specified. To give new names to the properties in the anonymous type, write the `select` statement as follows:

```
select new { First = student.FirstName, Last = student.LastName };
```

If you try this in the previous example, then the `Console.WriteLine` statement must also change:

```
Console.WriteLine(student.First + " " + student.Last);
```

Compiling the Code

- To run this code, copy and paste the class into a Visual C# console application project that has been created in Visual Studio. By default, this project targets version 3.5 of the .NET Framework, and it will have a reference to System.Core.dll and a `using` directive for System.Linq. If one or more of these requirements are missing from the project, you can add them manually.

See also

- C# Programming Guide
- Anonymous Types
- LINQ Query Expressions

Contents

Interfaces (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

An interface contains definitions for a group of related functionalities that a class or a struct can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the interface keyword. as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of the struct must be a valid C# identifier name. By convention, interface names begin with a capital `I`.

Any class or struct that implements the IEquatable<T> interface must contain a definition for an Equals method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. The interface defines only the signature. In that way, an interface in C# is similar to an abstract class in which all the methods are abstract. However, a class or struct can implement multiple interfaces, but a class can inherit only a single class, abstract or not. Therefore, by using interfaces, you can include behavior from multiple sources in a class.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. For links to examples, see Related Sections. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be static.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the IEquatable<T> interface. The implementing class, `Car`, must provide an implementation of the Equals method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a get accessor. The class that implements the interface can declare the same property with both a `get` and set accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see Explicit Interface Implementation and Interface Properties.

Interfaces can inherit from other interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see Polymorphism.

Interfaces summary

An interface has the following properties:

- An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

In this section

Explicit Interface Implementation
Explains how to create a class member that's specific to an interface.

How to: Explicitly Implement Interface Members
Provides an example of how to explicitly implement members of interfaces.

How to: Explicitly Implement Members of Two Interfaces
Provides an example of how to explicitly implement members of interfaces with inheritance.

Related Sections

- Interface Properties
- Indexers in Interfaces
- How to: Implement Interface Events
- Classes and Structs
- Inheritance
- Methods
- Polymorphism
- Abstract and Sealed Classes and Class Members
- Properties
- Events
- Indexers

featured book chapter

Interfaces in Learning C# 3.0: Master the Fundamentals of C# 3.0

See also

- C# Programming Guide
- Inheritance
- Identifier names

Explicit Interface Implementation (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

If a class implements two interfaces that contain a member with the same signature, then implementing that member on the class will cause both interfaces to use that member as their implementation. In the following example, all the calls to `Paint` invoke the same method.

```
class Test
{
    static void Main()
    {
        SampleClass sc = new SampleClass();
        IControl ctrl = sc;
        ISurface srfc = sc;

        // The following lines all call the same method.
        sc.Paint();
        ctrl.Paint();
        srfc.Paint();
    }
}


interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

If the two interface members do not perform the same function, however, this can lead to an incorrect implementation of one or both of the interfaces. It is possible to implement an interface member explicitly—creating a class member that is only called through the interface, and is specific to that interface. This is accomplished by naming the class member with the name of the interface and a period. For example:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

The class member `IControl.Paint` is only available through the `IControl` interface, and `ISurface.Paint` is only available through `ISurface`. Both method implementations are separate, and neither is available directly on the class. For example:

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Explicit implementation is also used to resolve cases where two interfaces each declare different members of the same name such as a property and a method:

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}
```

To implement both interfaces, a class has to use explicit implementation either for the property P, or the method P, or both, to avoid a compiler error. For example:

```
class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

See also

- C# Programming Guide
- Classes and Structs
- Interfaces

- Inheritance

How to: Explicitly Implement Interface Members (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example declares an interface, `IDimensions`, and a class, `Box`, which explicitly implements the interface members `getLength` and `getWidth`. The members are accessed through the interface instance `dimensions`.

Example

```
interface IDimensions
{
    float getLength();
    float getWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.getLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.getWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.getLength());
        //System.Console.WriteLine("Width: {0}", box1.getWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.getLength());
        System.Console.WriteLine("Width: {0}", dimensions.getWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Robust Programming

- Notice that the following lines, in the `Main` method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a class instance:

  ```
  //System.Console.WriteLine("Length: {0}", box1.getLength());
  //System.Console.WriteLine("Width: {0}", box1.getWidth());
  ```

- Notice also that the following lines, in the `Main` method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

```
System.Console.WriteLine("Length: {0}", dimensions.getLength());
System.Console.WriteLine("Width: {0}", dimensions.getWidth());
```

See also

- C# Programming Guide
- Classes and Structs
- Interfaces
- How to: Explicitly Implement Members of Two Interfaces

How to: Explicitly Implement Members of Two Interfaces (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Explicit interface implementation also allows the programmer to implement two interfaces that have the same member names and give each interface member a separate implementation. This example displays the dimensions of a box in both metric and English units. The Box class implements two interfaces IEnglishDimensions and IMetricDimensions, which represent the different measurement systems. Both interfaces have identical member names, Length and Width.

Example

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Robust Programming

If you want to make the default measurements in English units, implement the methods Length and Width normally, and explicitly implement the Length and Width methods from the IMetricDimensions interface:

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

In this case, you can access the English units from the class instance and access the metric units from the interface instance:

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

See also

- C# Programming Guide
- Classes and Structs
- Interfaces
- How to: Explicitly Implement Interface Members

Contents

C# programming guide

1/23/2019 • 2 minutes to read • Edit Online

This section provides detailed information on key C# language features and features accessible to C# through the .NET Framework.

Most of this section assumes that you already know something about C# and general programming concepts. If you are a complete beginner with programming or with C#, you might want to visit the Introduction to C# Tutorials or Getting Started with C# interactive tutorial, where no prior programming knowledge is required.

For information about specific keywords, operators and preprocessor directives, see C# Reference. For information about the C# Language Specification, see C# Language Specification.

Program sections

Inside a C# Program

Main() and Command-Line Arguments

Language Sections

Statements, Expressions, and Operators

Types

Classes and Structs

Interfaces

Enumeration Types

Delegates

Arrays

Strings

Properties

Indexers

Events

Generics

Iterators

LINQ Query Expressions

Lambda Expressions

Namespaces

Nullable Types

Unsafe Code and Pointers

XML Documentation Comments

Platform Sections

Application Domains

Assemblies and the Global Assembly Cache

Attributes

Collections

Exceptions and Exception Handling

File System and the Registry (C# Programming Guide)

Interoperability

Reflection

See also

- C# Reference
- C#

Inside a C# program

1/11/2019 • 2 minutes to read • Edit Online

The section discusses the general structure of a C# program, and includes the standard "Hello, World!" example.

In this section

- Hello World -- Your First Program
- General Structure of a C# Program
- Identifier names
- C# Coding Conventions

Related sections

- Getting Started with C#
- C# Programming Guide
- C# Reference
- Samples and tutorials

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Main() and command-line arguments (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The `Main` method is the entry point of a C# application. (Libraries and services do not require a `Main` method as an entry point.) When the application is started, the `Main` method is the first method that is invoked.

There can only be one entry point in a C# program. If you have more than one class that has a `Main` method, you must compile your program with the **/main** compiler option to specify which `Main` method to use as the entry point. For more information, see /main (C# Compiler Options).

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

Overview

- The `Main` method is the entry point of an executable program; it is where the program control starts and ends.
- `Main` is declared inside a class or struct. `Main` must be static and it need not be public. (In the earlier example, it receives the default access of private.) The enclosing class or struct is not required to be static.
- `Main` can either have a `void`, `int`, or, starting with C# 7.1, `Task`, or `Task<int>` return type.
- If and only if `Main` returns a `Task` or `Task<int>`, the declaration of `Main` may include the `async` modifier. Note that this specifically excludes an `async void Main` method.
- The `Main` method can be declared with or without a `string[]` parameter that contains command-line arguments. When using Visual Studio to create Windows applications, you can add the parameter manually or else use the Environment class to obtain the command-line arguments. Parameters are read as zero-indexed command-line arguments. Unlike C and C++, the name of the program is not treated as the first command-line argument.

The addition of `async` and `Task`, `Task<int>` return types simplifies program code when console applications need to start and `await` asynchronous operations in `Main`.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Command-line Building With csc.exe
- C# Programming Guide
- Methods
- Inside a C# Program

Programming Concepts (C#)

11/9/2018 • 2 minutes to read • Edit Online

This section explains programming concepts in the C# language.

In This Section

TITLE	DESCRIPTION
Assemblies and the Global Assembly Cache (C#)	Describes how to create and use assemblies.
Asynchronous Programming with async and await (C#)	Describes how to write asynchronous solutions by using the async and await keywords in C#. Includes a walkthrough.
Attributes (C#)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Caller Information (C#)	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
Collections (C#)	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (C#)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (C#)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (C#)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (C#)	Discusses the powerful query capabilities in the language syntax of C#, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Object-Oriented Programming (C#)	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.
Reflection (C#)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (C#)	Describes key concepts in binary, XML, and SOAP serialization.

Related Sections

Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

Statements, Expressions, and Operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The C# code that comprises an application consists of statements made up of keywords, expressions and operators. This section contains information regarding these fundamental elements of a C# program.

For more information, see:

- Statements
- Expressions
  - Expression-bodied members
- Operators
- Anonymous Functions
- Overloadable Operators
- Conversion Operators
  - Using Conversion Operators
  - How to: Implement User-Defined Conversions Between Structs
- Equality Comparisons

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Casting and Type Conversions

Types (C# Programming Guide)

2/5/2019 • 11 minutes to read • Edit Online

Types, Variables, and Values

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> C and C++ developers, notice that in C#, bool is not convertible to int.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying Types in Variable Declarations

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an int as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an int and then assign it a Boolean value of true. However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see Casting and Type Conversions.

Built-in Types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in `string` and `object` types. These are available for you to use in any C# program. For more information about the built-in types, see Reference Tables for Types.

Custom Types

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see .NET Class Library.

The Common Type System

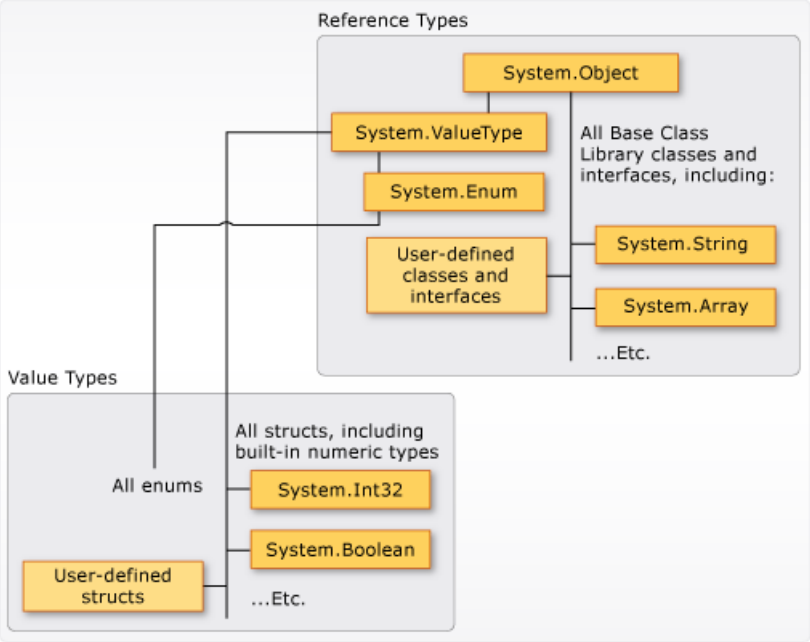
It is important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of

both base types in its inheritance hierarchy. All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS). For more information about inheritance in C#, see Inheritance.

- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



Value types and reference types in the CTS

> **NOTE**
>
> You can see that the most commonly used types are all organized in the System namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value Types

Value types derive from System.ValueType, which derives from System.Object. Types that derive from System.ValueType have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There is no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: struct and enum.

The built-in numeric types are structs, and they have properties and methods that you can access:

```
// Static method on type byte.
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they were simple non-aggregate types:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Value types are *sealed*, which means, for example, that you cannot derive a type from System.Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from System.ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a System.Object or any interface type as an input parameter. For more information, see Boxing and Unboxing.

You use the struct keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see Structs. For more information about value types in .NET, see Value Types.

The other category of value types is enum. An enum defines a set of named integral constants. For example, the System.IO.FileMode enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It is defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it is better to use enumerations instead of constant literal numbers. For more information, see System.IO.FileMode.

All enums inherit from System.Enum, which inherits from System.ValueType. All the rules that apply to structs also apply to enums. For more information about enums, see Enumeration Types.

Reference Types

A type that is defined as a class, delegate, array, or interface is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an object by using the new operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it does not create a performance issue. For more information about garbage collection, see Automatic Memory Management.

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the System.Array class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see Classes and Structs. For more information about inheritance and virtual methods, see Inheritance.

Types of Literal Values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see the reference pages for individual types in Value Types.

Because literals are typed, and all types derive ultimately from System.Object, you can write and compile code such as the following:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Generic Types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type System.Collections.Generic.List<T> has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to object. Generic collection classes are called *strongly-typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see Generics.

Implicit Types, Anonymous Types, and Nullable Types

As stated previously, you can implicitly type a local variable (but not class members) by using the var keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see Implicitly Typed Local Variables.

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see Anonymous Types.

Ordinary value types cannot have a value of null. However, you can create nullable value types by affixing a `?` after the type. For example, `int?` is an `int` type that can also have the value null. In the CTS, nullable types are instances of the generic struct type System.Nullable<T>. Nullable types are especially useful when you are passing data to and from databases in which numeric values might be null. For more information, see Nullable Types.

Related Sections

For more information, see the following topics:

- Casting and Type Conversions
- Boxing and Unboxing
- Using Type dynamic
- Value Types
- Reference Types
- Classes and Structs
- Anonymous Types
- Generics

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Conversion of XML Data Types

- Integral Types Table

Classes and Structs (C# Programming Guide)

1/23/2019 • 6 minutes to read • Edit Online

Classes and structs are two of the basic constructs of the common type system in the .NET Framework. Each is essentially a data structure that encapsulates a set of data and behaviors that belong together as a logical unit. The data and behaviors are the *members* of the class or struct, and they include its methods, properties, and events, and so on, as listed later in this topic.

A class or struct declaration is like a blueprint that is used to create instances or objects at run time. If you define a class or struct called `Person`, `Person` is the name of the type. If you declare and initialize a variable `p` of type `Person`, `p` is said to be an object or instance of `Person`. Multiple instances of the same `Person` type can be created, and each instance can have different values in its properties and fields.

A class is a reference type. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

A struct is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

In general, classes are used to model more complex behavior, or data that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that is not intended to be modified after the struct is created.

For more information, see Classes, Objects, and Structs.

Example

In the following example, `CustomClass` in the `ProgrammingGuide` namespace has three members: an instance constructor, a property named `Number`, and a method named `Multiply`. The `Main` method in the `Program` class creates an instance (object) of `CustomClass`, and the object's method and property are accessed by using dot notation.

```
using System;

namespace ProgrammingGuide
{
   // Class definition.
   public class CustomClass
   {
      // Class members.
      //
      // Property.
      public int Number { get; set; }

      // Method.
      public int Multiply(int num)
      {
          return num * Number;
      }

      // Instance Constructor.
      public CustomClass()
      {
          Number = 0;
      }
   }

   // Another class definition that contains Main, the program entry point.
   class Program
   {
      static void Main(string[] args)
      {
         // Create an object of type CustomClass.
         CustomClass custClass = new CustomClass();

         // Set the value of the public property.
         custClass.Number = 27;

         // Call the public method.
         int result = custClass.Multiply(4);
         Console.WriteLine($"The result is {result}.");
      }
   }
}
// The example displays the following output:
//      The result is 108.
```

Encapsulation

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. According to the principle of encapsulation, a class or struct can specify how accessible each of its members is to code outside of the class or struct. Methods and variables that are not intended to be used from outside of the class or assembly can be hidden to limit the potential for coding errors or malicious exploits.

For more information about classes, see Classes and Objects.

Members

All methods, fields, constants, properties, and events must be declared within a type; these are called the *members* of the type. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the `Main` method, must be declared within a class or struct. The following list includes all the various kinds of members that may be declared in a class or struct.

- Fields
- Constants

- Properties
- Methods
- Constructors
- Events
- Finalizers
- Indexers
- Operators
- Nested Types

Accessibility

Some methods and properties are meant to be called or accessed from code outside your class or struct, known as *client code*. Other methods and properties might be only for use in the class or struct itself. It is important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the access modifiers public, protected, internal, protected internal, private and private protected. The default accessibility is `private`. For more information, see Access Modifiers.

Inheritance

Classes (but not structs) support the concept of inheritance. A class that derives from another class (the *base class*) automatically contains all the public, protected, and internal members of the base class except its constructors and finalizers. For more information, see Inheritance and Polymorphism.

Classes may be declared as abstract, which means that one or more of their methods have no implementation. Although abstract classes cannot be instantiated directly, they can serve as base classes for other classes that provide the missing implementation. Classes can also be declared as sealed to prevent other classes from inheriting from them. For more information, see Abstract and Sealed Classes and Class Members.

Interfaces

Classes and structs can inherit multiple interfaces. To inherit from an interface means that the type implements all the methods defined in the interface. For more information, see Interfaces.

Generic Types

Classes and structs can be defined with one or more type parameters. Client code supplies the type when it creates an instance of the type. For example The List<T> class in the System.Collections.Generic namespace is defined with one type parameter. Client code creates an instance of a `List<string>` or `List<int>` to specify the type that the list will hold. For more information, see Generics.

Static Types

Classes (but not structs) can be declared as static. A static class can contain only static members and cannot be instantiated with the new keyword. One copy of the class is loaded into memory when the program loads, and its members are accessed through the class name. Both classes and structs can contain static members. For more information, see Static Classes and Static Class Members.

Nested Types

A class or struct can be nested within another class or struct. For more information, see Nested Types.

Partial Types

You can define part of a class, struct or method in one code file and another part in a separate code file. For more information, see Partial Classes and Methods.

Object Initializers

You can instantiate and initialize class or struct objects, and collections of objects, without explicitly calling their constructor. For more information, see Object and Collection Initializers.

Anonymous Types

In situations where it is not convenient or necessary to create a named class, for example when you are populating a list with data structures that you do not have to persist or pass to another method, you use anonymous types. For more information, see Anonymous Types.

Extension Methods

You can "extend" a class without creating a derived class by creating a separate type whose methods can be called as if they belonged to the original type. For more information, see Extension Methods.

Implicitly Typed Local Variables

Within a class or struct method, you can use implicit typing to instruct the compiler to determine the correct type at compile time. For more information, see Implicitly Typed Local Variables.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Interfaces (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

An interface contains definitions for a group of related functionalities that a class or a struct can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the interface keyword. as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of the struct must be a valid C# identifier name. By convention, interface names begin with a capital `I`.

Any class or struct that implements the IEquatable<T> interface must contain a definition for an Equals method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. The interface defines only the signature. In that way, an interface in C# is similar to an abstract class in which all the methods are abstract. However, a class or struct can implement multiple interfaces, but a class can inherit only a single class, abstract or not. Therefore, by using interfaces, you can include behavior from multiple sources in a class.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. For links to examples, see Related Sections. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be static.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the IEquatable<T> interface. The implementing class, `Car`, must provide an implementation of the Equals method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a get accessor. The class that implements the interface can declare the same property with both a `get` and set accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see Explicit Interface Implementation and Interface Properties.

Interfaces can inherit from other interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see Polymorphism.

Interfaces summary

An interface has the following properties:

- An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

In this section

Explicit Interface Implementation
Explains how to create a class member that's specific to an interface.

How to: Explicitly Implement Interface Members
Provides an example of how to explicitly implement members of interfaces.

How to: Explicitly Implement Members of Two Interfaces
Provides an example of how to explicitly implement members of interfaces with inheritance.

Related Sections

- Interface Properties
- Indexers in Interfaces
- How to: Implement Interface Events
- Classes and Structs
- Inheritance
- Methods
- Polymorphism
- Abstract and Sealed Classes and Class Members
- Properties
- Events
- Indexers

featured book chapter

Interfaces in Learning C# 3.0: Master the Fundamentals of C# 3.0

See also

- C# Programming Guide
- Inheritance
- Identifier names

Enumeration types (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

An enumeration type (also named an enumeration or an enum) provides an efficient way to define a set of named integral constants that may be assigned to a variable. For example, assume that you have to define a variable whose value will represent a day of the week. There are only seven meaningful values which that variable will ever store. To define those values, you can use an enumeration type, which is declared by using the enum keyword.

```
enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum Month : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
```

By default the underlying type of each element in the enum is int. You can specify another integral numeric type by using a colon, as shown in the previous example. For a full list of possible types, see enum (C# Reference).

You can verify the underlying numeric values by casting to the underlying type, as the following example shows.

```
Day today = Day.Monday;
int dayNumber =(int)today;
Console.WriteLine("{0} is day number #{1}.", today, dayNumber);

Month thisMonth = Month.Dec;
byte monthNumber = (byte)thisMonth;
Console.WriteLine("{0} is month number #{1}.", thisMonth, monthNumber);

// Output:
// Monday is day number #1.
// Dec is month number #11.
```

The following are advantages of using an enum instead of a numeric type:

- You clearly specify for client code which values are valid for the variable.
- In Visual Studio, IntelliSense lists the defined values.

When you do not specify values for the elements in the enumerator list, the values are automatically incremented by 1. In the previous example, `Day.Sunday` has a value of 0, `Day.Monday` has a value of 1, and so on. When you create a new `Day` object, it will have a default value of `Day.Sunday` (0) if you do not explicitly assign it a value. When you create an enum, select the most logical default value and give it a value of zero. That will cause all enums to have that default value if they are not explicitly assigned a value when they are created.

If the variable `meetingDay` is of type `Day`, then (without an explicit cast) you can only assign it one of the values defined by `Day`. And if the meeting day changes, you can assign a new value from `Day` to `meetingDay`:

```
Day meetingDay = Day.Monday;
//...
meetingDay = Day.Friday;
```

NOTE

It's possible to assign any arbitrary integer value to `meetingDay`. For example, this line of code does not produce an error: `meetingDay = (Day) 42`. However, you should not do this because the implicit expectation is that an enum variable will only hold one of the values defined by the enum. To assign an arbitrary value to a variable of an enumeration type is to introduce a high risk for errors.

You can assign any values to the elements in the enumerator list of an enumeration type, and you can also use computed values:

```
enum MachineState
{
    PowerOff = 0,
    Running = 5,
    Sleeping = 10,
    Hibernating = Sleeping + 5
}
```

Enumeration types as bit flags

You can use an enumeration type to define bit flags, which enables an instance of the enumeration type to store any combination of the values that are defined in the enumerator list. (Of course, some combinations may not be meaningful or allowed in your program code.)

You create a bit flags enum by applying the System.FlagsAttribute attribute and defining the values appropriately so that `AND`, `OR`, `NOT` and `XOR` bitwise operations can be performed on them. In a bit flags enum, include a named constant with a value of zero that means "no flags are set." Do not give a flag a value of zero if it does not mean "no flags are set".

In the following example, another version of the `Day` enum, which is named `Days`, is defined. `Days` has the `Flags` attribute, and each value is assigned the next greater power of 2. This enables you to create a `Days` variable whose value is `Days.Tuesday | Days.Thursday`.

```
[Flags]
enum Days
{
    None = 0x0,
    Sunday = 0x1,
    Monday = 0x2,
    Tuesday = 0x4,
    Wednesday = 0x8,
    Thursday = 0x10,
    Friday = 0x20,
    Saturday = 0x40
}
class MyClass
{
    Days meetingDays = Days.Tuesday | Days.Thursday;
}
```

To set a flag on an enum, use the bitwise `OR` operator as shown in the following example:

```
// Initialize with two flags using bitwise OR.
meetingDays = Days.Tuesday | Days.Thursday;

// Set an additional flag using bitwise OR.
meetingDays = meetingDays | Days.Friday;

Console.WriteLine("Meeting days are {0}", meetingDays);
// Output: Meeting days are Tuesday, Thursday, Friday

// Remove a flag using bitwise XOR.
meetingDays = meetingDays ^ Days.Tuesday;
Console.WriteLine("Meeting days are {0}", meetingDays);
// Output: Meeting days are Thursday, Friday
```

To determine whether a specific flag is set, use a bitwise `AND` operation, as shown in the following example:

```
// Test value of flags using bitwise AND.
bool test = (meetingDays & Days.Thursday) == Days.Thursday;
Console.WriteLine("Thursday {0} a meeting day.", test == true ? "is" : "is not");
// Output: Thursday is a meeting day.
```

For more information about what to consider when you define enumeration types with the System.FlagsAttribute attribute, see System.Enum.

Using the System.Enum methods to discover and manipulate enum values

All enums are instances of the System.Enum type. You cannot derive new classes from System.Enum, but you can use its methods to discover information about and manipulate values in an enum instance.

```
string s = Enum.GetName(typeof(Day), 4);
Console.WriteLine(s);

Console.WriteLine("The values of the Day Enum are:");
foreach (int i in Enum.GetValues(typeof(Day)))
    Console.WriteLine(i);

Console.WriteLine("The names of the Day Enum are:");
foreach (string str in Enum.GetNames(typeof(Day)))
    Console.WriteLine(str);
```

For more information, see System.Enum.

You can also create a new method for an enum by using an extension method. For more information, see How to: Create a New Method for an Enumeration.

See also

- System.Enum
- C# Programming Guide
- enum

Delegates (C# Programming Guide)

2/3/2019 • 2 minutes to read • Edit Online

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

```
public delegate int PerformCalculation(int x, int y);
```

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

> **NOTE**
> In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

Delegates Overview

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods do not have to match the delegate type exactly. For more information, see Using Variance in Delegates.
- C# version 2.0 introduced the concept of Anonymous Methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions. For more information about lambda expressions, see Anonymous Functions.

In This Section

- Using Delegates
- When to Use Delegates Instead of Interfaces (C# Programming Guide)
- Delegates with Named vs. Anonymous Methods
- Anonymous Methods
- Using Variance in Delegates
- How to: Combine Delegates (Multicast Delegates)
- How to: Declare, Instantiate, and Use a Delegate

C# Language Specification

For more information, see Delegates in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- Delegate
- C# Programming Guide
- Events

Arrays (C# Programming Guide)

1/24/2019 • 2 minutes to read • Edit Online

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.

```
type[] arrayName;
```

The following example creates single-dimensional, multidimensional, and jagged arrays:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Array Overview

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Related Sections

- Arrays as Objects
- Using foreach with Arrays
- Passing Arrays as Arguments

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Collections

Strings (C# Programming Guide)

12/11/2018 • 12 minutes to read • Edit Online

A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The Length property of a string represents the number of `Char` objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the StringInfo object.

string vs. System.String

In C#, the `string` keyword is an alias for String. Therefore, `String` and `string` are equivalent, and you can use whichever naming convention you prefer. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see string. For more information about the type and its methods, see String.

Declaring and Initializing Strings

You can declare and initialize strings in various ways, as shown in the following example:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

//Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Note that you do not use the new operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the Empty constant value to create a new String object whose string is of zero length. The string literal representation of a zero-length string is "". By initializing strings with the Empty value instead of null, you can reduce the chances of a NullReferenceException occurring. Use the static IsNullOrEmpty(String) method to verify the value of a string before you try to access it.

Immutability of String Objects

String objects are *immutable*: they cannot be changed after they have been created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object that was assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see How to: Modify String Contents.

Regular and Verbatim String Literals

Use regular string literals when you must embed escape characters provided by C#, as shown in the following example:

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

String Escape Sequences

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\U	Unicode escape sequence for surrogate pairs.	\Unnnnnnnn
\u	Unicode escape sequence	\u0041 = "A"
\v	Vertical tab	0x000B
\x	Unicode escape sequence similar to "\u" except with variable length.	\x0041 or \x41 = "A"

> **NOTE**
> At compile time, verbatim strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim version from your source code. For example, the verbatim string @"C:\files.txt" will appear in the watch window as "C:\\files.txt".

Format Strings

A format string is a string whose contents are determined dynamically at runtime. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) will be resolved to a value and output as a formatted string at runtime. There are two methods to create format strings: string interpolation and composite formatting.

String Interpolation

Available in C# 6.0 and later, *interpolated strings* are identified by the `$` special character and include interpolated expressions in braces. If you are new to string interpolation, see the String interpolation - C# interactive tutorial for a quick overview.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but improves ease of use and inline clarity.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Composite Formatting

The String.Format utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information on formatting .NET types see Formatting Types in .NET.

Substrings

A substring is any sequence of characters that is contained in a string. Use the Substring method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the IndexOf method. Use the Replace method to replace all occurrences of a specified substring with a new string. Like the Substring method, Replace actually returns a new string and does not modify the original string. For more information, see How to: search strings and How to: Modify String Contents.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing Individual Characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

If the String methods do not provide the functionality that you must have to modify individual characters in a string, you can use a StringBuilder object to modify the individual chars "in-place", and then create a new string to store the results by using the StringBuilder methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null Strings and Empty Strings

An empty string is an instance of a System.String object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid System.String objects. Empty strings are initialized as follows:

```
string s = String.Empty;
```

By contrast, a null string does not refer to an instance of a System.String object and any attempt to call a method on a null string causes a NullReferenceException. However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Using StringBuilder for Fast String Creation

String operations in .NET are highly optimized and in most cases do not significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The StringBuilder class creates a string buffer that offers better performance if your program performs many string manipulations. The StringBuilder string also enables you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

In this example, a StringBuilder object is used to create a string from a set of numeric types:

```
class TestStringBuilder
{
    static void Main()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();

        // Create a string composed of numbers 0 - 9
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        System.Console.WriteLine(sb);  // displays 0123456789

        // Copy one character of the string (not possible with a System.String)
        sb[0] = sb[9];

        System.Console.WriteLine(sb);  // displays 9123456789
    }
}
```

Strings, Extension Methods and LINQ

Because the String type implements IEnumerable<T>, you can use the extension methods defined in the Enumerable class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the String type, but they are available nevertheless. You can also use LINQ query expressions on strings. For more information, see LINQ and Strings.

Related Topics

TOPIC	DESCRIPTION
How to: Modify String Contents	Illustrates techniques to transform strings and modify the contents of strings.
How to: Compare Strings	Shows how to perform ordinal and culture specific comparisons of strings.
How to: Concatenate Multiple Strings	Demonstrates various ways to join multiple strings into one.
How to: Parse Strings Using String.Split	Contains code examples that illustrate how to use the `String.Split` method to parse strings.
How to: Search Strings	Explains how to use search for specific text or patterns in strings.
How to: Determine Whether a String Represents a Numeric Value	Shows how to safely parse a string to see whether it has a valid numeric value.
String interpolation	Describes the string interpolation feature that provides a convenient syntax to format strings.
Basic String Operations	Provides links to topics that use System.String and System.Text.StringBuilder methods to perform basic string operations.
Parsing Strings	Describes how to convert string representations of .NET base types to instances of the corresponding types.

TOPIC	DESCRIPTION
Parsing Date and Time Strings in .NET	Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
Comparing Strings	Includes information about how to compare strings and provides examples in C# and Visual Basic.
Using the StringBuilder Class	Describes how to create and modify dynamic string objects by using the StringBuilder class.
LINQ and Strings	Provides information about how to perform various string operations by using LINQ queries.
C# Programming Guide	Provides links to topics that explain programming constructs in C#.

Indexers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The `Program` class creates an instance of this class for storing strings.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

For more examples, see Related Sections.

Expression Body Definitions

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. Starting with C# 6, a read-only indexer can be implemented as an expression-bodied member, as the following example shows.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Note that `=>` introduces the expression body, and that the `get` keyword is not used.

Starting with C# 7.0, both the get and set accessor can be an implemented as expression-bodied members. In this case, both `get` and `set` keywords must be used. For example:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The this keyword is used to define the indexer.
- The value keyword is used to define the value being assigned by the `set` indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Related Sections

- Using Indexers
- Indexers in Interfaces
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility

C# Language Specification

For more information, see Indexers in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Properties

Events (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see How to: Subscribe to and Unsubscribe from Events.

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see Calling Synchronous Methods Asynchronously.
- In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.

Related Sections

For more information, see:

- How to: Subscribe to and Unsubscribe from Events
- How to: Publish Events that Conform to .NET Framework Guidelines
- How to: Raise Base Class Events in Derived Classes
- How to: Implement Interface Events
- How to: Use a Dictionary to Store Event Instances
- How to: Implement Custom Event Accessors

C# Language Specification

For more information, see Events in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- EventHandler
- C# Programming Guide
- Delegates
- Creating Event Handlers in Windows Forms

Generics (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Generics were added to version 2.0 of the C# language and the common language runtime (CLR). Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Generics Overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related Sections

For more information:

- Introduction to Generics

- Benefits of Generics
- Generic Type Parameters
- Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generic Delegates
- Differences Between C++ Templates and C# Generics
- Generics and Reflection
- Generics in the Run Time

C# Language Specification

For more information, see the C# Language Specification.

See also

- System.Collections.Generic
- C# Programming Guide
- Types
- <typeparam>
- <typeparamref>
- Generics in .NET

Namespaces (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Namespaces are heavily used in C# programming in two ways. First, the .NET Framework uses namespaces to organize its many classes, as follows:

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

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name is not required, as in the following example:

```
using System;
```

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

For more information, see the using Directive.

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# identifier name.

Namespaces Overview

Namespaces have the following properties:

- They organize large code projects.
- They are delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET System namespace.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Using Namespaces
- How to: Use the Global Namespace Alias
- How to: Use the My Namespace
- C# Programming Guide
- Identifier names
- Namespace Keywords
- using Directive
- :: Operator
- . Operator

Nullable types (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Nullable types are instances of the System.Nullable<T> struct. Nullable types can represent all the values of an underlying type `T`, and an additional null value. The underlying type `T` can be any non-nullable value type. `T` cannot be a reference type.

For example, you can assign `null` or any integer value from Int32.MinValue to Int32.MaxValue to a `Nullable<int>` and true, false, or `null` to a `Nullable<bool>`.

You use a nullable type when you need to represent the undefined value of an underlying type. A Boolean variable can have only two values: true and false. There is no "undefined" value. In many programming applications, most notably database interactions, a variable value can be undefined or missing. For example, a field in a database may contain the values true or false, or it may contain no value at all. You use a `Nullable<bool>` type in that case.

Nullable types have the following characteristics:

- Nullable types represent value-type variables that can be assigned the `null` value. You cannot create a nullable type based on a reference type. (Reference types already support the `null` value.)
- The syntax `T?` is shorthand for `Nullable<T>`. The two forms are interchangeable.
- Assign a value to a nullable type just as you would for an underlying value type: `int? x = 10;` or `double? d = 4.108;`. You also can assign the `null` value: `int? x = null;`.
- Use the Nullable<T>.HasValue and Nullable<T>.Value readonly properties to test for null and retrieve the value, as shown in the following example: `if (x.HasValue) y = x.Value;`
  - The HasValue property returns `true` if the variable contains a value, or `false` if it's `null`.
  - The Value property returns a value if HasValue returns `true`. Otherwise, an InvalidOperationException is thrown.
- You can also use the `==` and `!=` operators with a nullable type, as shown in the following example: `if (x != null) y = x.Value;`. If `a` and `b` are both null, `a == b` evaluates to `true`.
- Beginning with C# 7.0, you can use pattern matching to both examine and get a value of a nullable type: `if (x is int valueOfX) y = valueOfX;`.
- The default value of `T?` is an instance whose HasValue property returns `false`.
- Use the GetValueOrDefault() method to return either the assigned value, or the default value of the underlying value type if the value of the nullable type is `null`.
- Use the GetValueOrDefault(T) method to return either the assigned value, or the provided default value if the value of the nullable type is `null`.
- Use the null-coalescing operator, `??`, to assign a value to an underlying type based on a value of the nullable type: `int? x = null; int y = x ?? -1;`. In the example, since `x` is null, the result value of `y` is `-1`.
- If a user-defined conversion is defined between two data types, the same conversion can also be used with the nullable versions of these data types.
- Nested nullable types are not allowed. The following line doesn't compile: `Nullable<Nullable<int>> n;`

For more information, see the Using nullable types and How to: Identify a nullable type topics.

See also

- System.Nullable<T>
- System.Nullable
- ?? Operator
- C# Programming Guide
- C# Guide
- C# Reference
- Nullable Value Types (Visual Basic)

Unsafe Code and Pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To maintain type safety and security, C# does not support pointer arithmetic, by default. However, by using the unsafe keyword, you can define an unsafe context in which pointers can be used. For more information about pointers, see the topic Pointer types.

> **NOTE**
>
> In the common language runtime (CLR), unsafe code is referred to as unverifiable code. Unsafe code in C# is not necessarily dangerous; it is just code whose safety cannot be verified by the CLR. The CLR will therefore only execute unsafe code if it is in a fully trusted assembly. If you use unsafe code, it is your responsibility to ensure that your code does not introduce security risks or pointer errors.

Unsafe Code Overview

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code may increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- In order for C# to compile unsafe code, the application must be compiled with /unsafe.

Related Sections

For more information, see:

- Pointer types
- Fixed Size Buffers
- How to: Use Pointers to Copy an Array of Bytes
- unsafe

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

XML Documentation Comments (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

In Visual C# you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

When you compile with the /doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as DocFX or Sandcastle.

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List&lt;T&gt;"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

> **NOTE**
> The XML documentation comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

In This Section

- Recommended Tags for Documentation Comments
- Processing the XML File
- Delimiters for Documentation Tags
- How to: Use the XML Documentation Features

Related Sections

For more information, see:

- /doc (Process Documentation Comments)

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Exceptions and Exception Handling (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the `try`, `catch`, and `finally` keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code. Exceptions are created by using the `throw` keyword.

In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When this happens, the CLR will unwind the stack, looking for a method with a `catch` block for the specific exception type, and it will execute the first such `catch` block that if finds. If it finds no appropriate `catch` block anywhere in the call stack, it will terminate the process and display a message to the user.

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from `System.Exception`.
- Use a `try` block around the statements that might throw exceptions.
- Once an exception occurs in the `try` block, the flow of control jumps to the first associated exception

handler that is present anywhere in the call stack. In C#, the `catch` keyword is used to define an exception handler.

- If no exception handler for a given exception is present, the program stops executing with an error message.
- Do not catch an exception unless you can handle it and leave the application in a known state. If you catch `System.Exception`, rethrow it using the `throw` keyword at the end of the `catch` block.
- If a `catch` block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the `throw` keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a `finally` block is executed even if an exception is thrown. Use a `finally` block to release resources, for example to close any streams or files that were opened in the `try` block.
- Managed exceptions in the .NET Framework are implemented on top of the Win32 structured exception handling mechanism. For more information, see Structured Exception Handling (C/C++) and A Crash Course on the Depths of Win32 Structured Exception Handling.

Related Sections

See the following topics for more information about exceptions and exception handling:

- Using Exceptions
- Exception Handling
- Creating and Throwing Exceptions
- Compiler-Generated Exceptions
- How to: Handle an Exception Using try/catch (C# Programming Guide)
- How to: Execute Cleanup Code Using finally

C# Language Specification

For more information, see Exceptions in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- SystemException
- C# Programming Guide
- C# Keywords
- throw
- try-catch
- try-finally
- try-catch-finally
- Exceptions

File System and the Registry (C# Programming Guide)

12/11/2018 • 2 minutes to read • Edit Online

The following topics show how to use C# and the .NET Framework to perform various basic operations on files, folders, and the Registry.

In This Section

TITLE	DESCRIPTION
How to: Iterate Through a Directory Tree	Shows how to manually iterate through a directory tree.
How to: Get Information About Files, Folders, and Drives	Shows how to retrieve information such as creation times and size, about files, folders and drives.
How to: Create a File or Folder	Shows how to create a new file or folder.
How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)	Shows how to copy, delete and move files and folders.
How to: Provide a Progress Dialog Box for File Operations	Shows how to display a standard Windows progress dialog for certain file operations.
How to: Write to a Text File	Shows how to write to a text file.
How to: Read From a Text File	Shows how to read from a text file.
How to: Read a Text File One Line at a Time	Shows how to retrieve text from a file one line at a time.
How to: Create a Key In the Registry	Shows how to write a key to the system registry.

Related Sections

File and Stream I/O

How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)

C# Programming Guide

Files, Folders and Drives

System.IO

Interoperability (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Win32 API are examples of unmanaged code.

The .NET Framework enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

Interoperability Overview
Describes methods to interoperate between C# managed code and unmanaged code.

How to: Access Office Interop Objects by Using Visual C# Features
Describes features that are introduced in Visual C# to facilitate Office programming.

How to: Use Indexed Properties in COM Interop Programming
Describes how to use indexed properties to access COM properties that have parameters.

How to: Use Platform Invoke to Play a Wave File
Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

Walkthrough: Office Programming
Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

Example COM Class
Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see Basic concepts in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Marshal.ReleaseComObject
- C# Programming Guide
- Interoperating with Unmanaged Code
- Walkthrough: Office Programming

Contents

Delegates (C# Programming Guide)

2/3/2019 • 2 minutes to read • Edit Online

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

```
public delegate int PerformCalculation(int x, int y);
```

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

> **NOTE**
> In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

Delegates Overview

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods do not have to match the delegate type exactly. For more information, see Using Variance in Delegates.
- C# version 2.0 introduced the concept of Anonymous Methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions. For more information about lambda expressions, see Anonymous Functions.

In This Section

- Using Delegates
- When to Use Delegates Instead of Interfaces (C# Programming Guide)
- Delegates with Named vs. Anonymous Methods
- Anonymous Methods
- Using Variance in Delegates
- How to: Combine Delegates (Multicast Delegates)
- How to: Declare, Instantiate, and Use a Delegate

C# Language Specification

For more information, see Delegates in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- Delegate
- C# Programming Guide
- Events

Using Delegates (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

A delegate is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate. The following example declares a delegate named `Del` that can encapsulate a method that takes a string as an argument and returns void:

```
public delegate void Del(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an anonymous Method. Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Delegate types are derived from the Delegate class in the .NET Framework. Delegate types are sealed—they cannot be derived from— and it is not possible to derive custom classes from Delegate. Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the `Del` type as a parameter:

```
public void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

You can then pass the delegate created above to that method:

```
MethodWithCallback(1, 2, handler);
```

and receive the following output to the console:

```
The number is: 3
```

Using the delegate as an abstraction, `MethodWithCallback` does not need to call the console directly—it does not have to be designed with a console in mind. What `MethodWithCallback` does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Along with the static `DelegateMethod` shown previously, we now have three methods that can be wrapped by a `Del` instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

```
MethodClass obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

At this point `allMethodsDelegate` contains three methods in its invocation list— `Method1`, `Method2`, and `DelegateMethod`. The original three delegates, `d1`, `d2`, and `d3`, remain unchanged. When `allMethodsDelegate` is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the decrement or decrement assignment operator ('-' or '-='). For example:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Because delegate types are derived from `System.Delegate`, the methods and properties defined by that class can be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from MulticastDelegate, which is a subclass of `System.Delegate`. The above code works in either case because both classes support `GetInvocationList`.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see Events.

Comparing delegates of two different types assigned at compile-time will result in a compilation error. If the delegate instances are statically of the type `System.Delegate`, then the comparison is allowed, but will return false at run time. For example:

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    System.Console.WriteLine(d == f);
}
```

See also

- C# Programming Guide
- Delegates
- Using Variance in Delegates
- Variance in Delegates
- Using Variance for Func and Action Generic Delegates
- Events

Delegates with Named vs. Anonymous Methods (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

A delegate can be associated with a named method. When you instantiate a delegate by using a named method, the method is passed as a parameter, for example:

```
// Declare a delegate:
delegate void Del(int x);

// Define a named method:
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter:
Del d = obj.DoWork;
```

This is called using a named method. Delegates constructed with a named method can encapsulate either a static method or an instance method. Named methods are the only way to instantiate a delegate in earlier versions of C#. However, in a situation where creating a new method is unwanted overhead, C# enables you to instantiate a delegate and immediately specify a code block that the delegate will process when it is called. The block can contain either a lambda expression or an anonymous method. For more information, see Anonymous Functions.

Remarks

The method that you pass as a delegate parameter must have the same signature as the delegate declaration.

A delegate instance may encapsulate either static or instance method.

Although the delegate can use an out parameter, we do not recommend its use with multicast event delegates because you cannot know which delegate will be called.

Example 1

The following is a simple example of declaring and using a delegate. Notice that both the delegate, `Del`, and the associated method, `MultiplyNumbers`, have the same signature

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        System.Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        System.Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

Example 2

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        System.Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        System.Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        SampleClass sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

See also

- C# Programming Guide
- Delegates
- Anonymous Methods
- How to: Combine Delegates (Multicast Delegates)
- Events

How to: Combine Delegates (Multicast Delegates)(C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example demonstrates how to create multicast delegates. A useful property of delegate objects is that multiple objects can be assigned to one delegate instance by using the `+` operator. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

The `-` operator can be used to remove a component delegate from a multicast delegate.

Example

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        System.Console.WriteLine("  Hello, {0}!", s);
    }

    static void Goodbye(string s)
    {
        System.Console.WriteLine("  Goodbye, {0}!", s);
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

See also

- MulticastDelegate
- C# Programming Guide
- Events

How to: Declare, Instantiate, and Use a Delegate (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

In C# 1.0 and later, delegates can be declared as shown in the following example.

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine("Notification received for: {0}", name);
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

C# 2.0 provides a simpler way to write the previous declaration, as shown in the following example.

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

In C# 2.0 and later, it is also possible to use an anonymous method to declare and initialize a delegate, as shown in the following example.

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine("Notification received for: {0}", name); };
```

In C# 3.0 and later, delegates can also be declared and instantiated by using a lambda expression, as shown in the following example.

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine("Notification received for: {0}", name); };
```

For more information, see Lambda Expressions.

The following example illustrates declaring, instantiating, and using a delegate. The `BookDB` class encapsulates a bookstore database that maintains a database of books. It exposes a method, `ProcessPaperbackBooks`, which finds all paperback books in the database and calls a delegate for each one. The `delegate` type that is used is named `ProcessBookDelegate`. The `Test` class uses this class to print the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is performed on the paperback books after it finds them.

Example

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookDelegate(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookDelegate processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}


// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
```

```
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            System.Console.WriteLine("   {0}", b.Title);
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            System.Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            System.Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Robust Programming

- Declaring a delegate.

The following statement declares a new delegate type.

```
public delegate void ProcessBookDelegate(Book book);
```

Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

- Instantiating a delegate.

  After a delegate type has been declared, a delegate object must be created and associated with a particular method. In the previous example, you do this by passing the `PrintTitle` method to the `ProcessPaperbackBooks` method as in the following example:

  ```
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  This creates a new delegate object associated with the static method `Test.PrintTitle`. Similarly, the non-static method `AddBookToTotal` on the object `totaller` is passed as in the following example:

  ```
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  In both cases a new delegate object is passed to the `ProcessPaperbackBooks` method.

  After a delegate is created, the method it is associated with never changes; delegate objects are immutable.

- Calling a delegate.

  After a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. Following is an example of a delegate call:

  ```
  processBook(b);
  ```

  A delegate can be either called synchronously, as in this example, or asynchronously by using `BeginInvoke` and `EndInvoke` methods.

See also

- C# Programming Guide
- Events
- Delegates

Contents

Arrays (C# Programming Guide)

1/24/2019 • 2 minutes to read • Edit Online

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.

```
type[] arrayName;
```

The following example creates single-dimensional, multidimensional, and jagged arrays:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Array Overview

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Related Sections

- Arrays as Objects
- Using foreach with Arrays
- Passing Arrays as Arguments

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Collections

Arrays as Objects (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++. Array is the abstract base type of all array types. You can use the properties, and other class members, that Array has. An example of this would be using the Length property to get the length of an array. The following code assigns the length of the `numbers` array, which is `5`, to a variable called `lengthOfNumbers`:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

The Array class provides many other useful methods and properties for sorting, searching, and copying arrays.

Example

This example uses the Rank property to display the number of dimensions of an array.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array:
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

See also

- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays

Single-Dimensional Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can declare a single-dimensional array of five integers as shown in the following example:

```
int[] array = new int[5];
```

This array contains the elements from `array[0]` to `array[4]`. The new operator is used to create the array and initialize the array elements to their default values. In this example, all the array elements are initialized to zero.

An array that stores string elements can be declared in the same way. For example:

```
string[] stringArray = new string[6];
```

Array Initialization

It is possible to initialize an array upon declaration, in which case, the length specifier is not needed because it is already supplied by the number of elements in the initialization list. For example:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

A string array can be initialized in the same way. The following is a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

When you initialize an array upon declaration, you can use the following shortcuts:

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

It is possible to declare an array variable without initialization, but you must use the `new` operator when you assign an array to this variable. For example:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

C# 3.0 introduces implicitly typed arrays. For more information, see Implicitly Typed Arrays.

Value Type and Reference Type Arrays

Consider the following array declaration:

```
SomeType[] array4 = new SomeType[10];
```

The result of this statement depends on whether `SomeType` is a value type or a reference type. If it is a value type, the statement creates an array of 10 elements, each of which has the type `SomeType`. If `SomeType` is a reference type, the statement creates an array of 10 elements, each of which is initialized to a null reference.

For more information about value types and reference types, see Types.

See also

- Array
- C# Programming Guide
- Arrays
- Multidimensional Arrays
- Jagged Arrays

Multidimensional Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns.

```
int[,] array = new int[4, 2];
```

The following declaration creates an array of three dimensions, 4, 2, and 3.

```
int[, ,] array1 = new int[4, 2, 3];
```

Array Initialization

You can initialize the array upon declaration, as is shown in the following example.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[, ,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                 { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[, ,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                       { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++) {
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

You also can initialize the array without specifying the rank.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

If you choose to declare an array variable without initialization, you must use the `new` operator to assign an array to the variable. The use of `new` is shown in the following example.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

The following example assigns a value to a particular array element.

```
array5[2, 1] = 25;
```

Similarly, the following example gets the value of a particular array element and assigns it to variable

`elementValue`.

```
int elementValue = array5[2, 1];
```

The following code example initializes the array elements to default values (except for jagged arrays).

```
int[,] array6 = new int[10, 10];
```

See also

- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Jagged Arrays

Jagged Arrays (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use `jaggedArray`, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

You can also initialize the array upon declaration like this:

```
int[][] jaggedArray2 = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

You can use the following shorthand form. Notice that you cannot omit the `new` operator from the elements initialization because there is no default initialization for the elements:

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.

You can access individual array elements like these examples:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

It is possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes. For more information about two-dimensional arrays, see Multidimensional Arrays.

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

You can access individual elements as shown in this example, which displays the value of the element `[1,0]` of the first array (value `5`):

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

The method `Length` returns the number of arrays contained in the jagged array. For example, assuming you have declared the previous array, this line:

```
System.Console.WriteLine(jaggedArray4.Length);
```

returns a value of 3.

Example

This example builds an array whose elements are themselves arrays. Each one of the array elements has a different size.

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements:
        int[][] arr = new int[2][];

        // Initialize the elements:
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements:
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

See also

- Array
- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays

Using foreach with arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The foreach statement provides a simple, clean way to iterate through the elements of an array.

For single-dimensional arrays, the `foreach` statement processes elements in increasing index order, starting with index 0 and ending with index `Length - 1`:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

However, with multidimensional arrays, using a nested for loop gives you more control over the order in which to process the array elements.

See also

- Array
- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays

Passing arrays as arguments (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

Arrays can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

Passing single-dimensional arrays as arguments

You can pass an initialized single-dimensional array to a method. For example, the following statement sends an array to a print method.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

The following code shows a partial implementation of the print method.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Example

In the following example, an array of strings is initialized and passed as an argument to a `DisplayArray` method for strings. The method displays the elements of the array. Next, the `ChangeArray` method reverses the array elements, and then the `ChangeArrayElements` method modifies the first three elements of the array. After each method returns, the `DisplayArray` method shows that passing an array by value doesn't prevent changes to the array elements.

```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Passing multidimensional arrays as arguments

You pass an initialized multidimensional array to a method in the same way that you pass a one-dimensional array.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

The following code shows a partial declaration of a print method that accepts a two-dimensional array as its argument.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example:

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Example

In the following example, a two-dimensional array of integers is initialized and passed to the `Print2DArray` method. The method displays the elements of the array.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Element(0,0)=1
        Element(0,1)=2
        Element(1,0)=3
        Element(1,1)=4
        Element(2,0)=5
        Element(2,1)=6
        Element(3,0)=7
        Element(3,1)=8
    */
```

See also

- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays

Implicitly Typed Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly-typed variable also apply to implicitly-typed arrays. For more information, see Implicitly Typed Local Variables.

Implicitly-typed arrays are usually used in query expressions together with anonymous types and object and collection initializers.

The following examples show how to create an implicitly-typed array:

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

In the previous example, notice that with implicitly-typed arrays, no square brackets are used on the left side of the initialization statement. Note also that jagged arrays are initialized by using `new []` just like single-dimension arrays.

Implicitly-typed Arrays in Object Initializers

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, `contacts` is an implicitly-typed array of anonymous types, each of which contains an array named `PhoneNumbers`. Note that the `var` keyword is not used inside the object initializers.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

See also

- C# Programming Guide
- Implicitly Typed Local Variables
- Arrays
- Anonymous Types
- Object and Collection Initializers
- var
- LINQ Query Expressions

Contents

Strings (C# Programming Guide)

12/11/2018 • 12 minutes to read • Edit Online

A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The Length property of a string represents the number of `Char` objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the StringInfo object.

string vs. System.String

In C#, the `string` keyword is an alias for String. Therefore, `String` and `string` are equivalent, and you can use whichever naming convention you prefer. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see string. For more information about the type and its methods, see String.

Declaring and Initializing Strings

You can declare and initialize strings in various ways, as shown in the following example:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

//Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Note that you do not use the new operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the Empty constant value to create a new String object whose string is of zero length. The string literal representation of a zero-length string is "". By initializing strings with the Empty value instead of null, you can reduce the chances of a NullReferenceException occurring. Use the static IsNullOrEmpty(String) method to verify the value of a string before you try to access it.

Immutability of String Objects

String objects are *immutable*: they cannot be changed after they have been created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object that was assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see How to: Modify String Contents.

Regular and Verbatim String Literals

Use regular string literals when you must embed escape characters provided by C#, as shown in the following example:

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

String Escape Sequences

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\U	Unicode escape sequence for surrogate pairs.	\Unnnnnnnn
\u	Unicode escape sequence	\u0041 = "A"
\v	Vertical tab	0x000B
\x	Unicode escape sequence similar to "\u" except with variable length.	\x0041 or \x41 = "A"

> **NOTE**
> At compile time, verbatim strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim version from your source code. For example, the verbatim string @"C:\files.txt" will appear in the watch window as "C:\\files.txt".

Format Strings

A format string is a string whose contents are determined dynamically at runtime. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) will be resolved to a value and output as a formatted string at runtime. There are two methods to create format strings: string interpolation and composite formatting.

String Interpolation

Available in C# 6.0 and later, *interpolated strings* are identified by the `$` special character and include interpolated expressions in braces. If you are new to string interpolation, see the String interpolation - C# interactive tutorial for a quick overview.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but improves ease of use and inline clarity.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Composite Formatting

The String.Format utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information on formatting .NET types see Formatting Types in .NET.

Substrings

A substring is any sequence of characters that is contained in a string. Use the Substring method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the IndexOf method. Use the Replace method to replace all occurrences of a specified substring with a new string. Like the Substring method, Replace actually returns a new string and does not modify the original string. For more information, see How to: search strings and How to: Modify String Contents.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing Individual Characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

If the String methods do not provide the functionality that you must have to modify individual characters in a string, you can use a StringBuilder object to modify the individual chars "in-place", and then create a new string to store the results by using the StringBuilder methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null Strings and Empty Strings

An empty string is an instance of a System.String object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid System.String objects. Empty strings are initialized as follows:

```
string s = String.Empty;
```

By contrast, a null string does not refer to an instance of a System.String object and any attempt to call a method on a null string causes a NullReferenceException. However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Using StringBuilder for Fast String Creation

String operations in .NET are highly optimized and in most cases do not significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The StringBuilder class creates a string buffer that offers better performance if your program performs many string manipulations. The StringBuilder string also enables you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

In this example, a StringBuilder object is used to create a string from a set of numeric types:

```
class TestStringBuilder
{
    static void Main()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();

        // Create a string composed of numbers 0 - 9
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        System.Console.WriteLine(sb);  // displays 0123456789

        // Copy one character of the string (not possible with a System.String)
        sb[0] = sb[9];

        System.Console.WriteLine(sb);  // displays 9123456789
    }
}
```

Strings, Extension Methods and LINQ

Because the String type implements IEnumerable<T>, you can use the extension methods defined in the Enumerable class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the String type, but they are available nevertheless. You can also use LINQ query expressions on strings. For more information, see LINQ and Strings.

Related Topics

TOPIC	DESCRIPTION
How to: Modify String Contents	Illustrates techniques to transform strings and modify the contents of strings.
How to: Compare Strings	Shows how to perform ordinal and culture specific comparisons of strings.
How to: Concatenate Multiple Strings	Demonstrates various ways to join multiple strings into one.
How to: Parse Strings Using String.Split	Contains code examples that illustrate how to use the `String.Split` method to parse strings.
How to: Search Strings	Explains how to use search for specific text or patterns in strings.
How to: Determine Whether a String Represents a Numeric Value	Shows how to safely parse a string to see whether it has a valid numeric value.
String interpolation	Describes the string interpolation feature that provides a convenient syntax to format strings.
Basic String Operations	Provides links to topics that use System.String and System.Text.StringBuilder methods to perform basic string operations.
Parsing Strings	Describes how to convert string representations of .NET base types to instances of the corresponding types.

TOPIC	DESCRIPTION
Parsing Date and Time Strings in .NET	Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
Comparing Strings	Includes information about how to compare strings and provides examples in C# and Visual Basic.
Using the StringBuilder Class	Describes how to create and modify dynamic string objects by using the StringBuilder class.
LINQ and Strings	Provides information about how to perform various string operations by using LINQ queries.
C# Programming Guide	Provides links to topics that explain programming constructs in C#.

How to: Determine Whether a String Represents a Numeric Value (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To determine whether a string is a valid representation of a specified numeric type, use the static `TryParse` method that is implemented by all primitive numeric types and also by types such as DateTime and IPAddress. The following example shows how to determine whether "108" is a valid int.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

If the string contains nonnumeric characters or the numeric value is too large or too small for the particular type you have specified, `TryParse` returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.

> **NOTE**
> A string may contain only numeric characters and still not be valid for the type whose `TryParse` method that you use. For example, "256" is not a valid value for `byte` but it is valid for `int`. "98.6" is not a valid value for `int` but it is a valid `decimal`.

Example

The following examples show how to use `TryParse` with string representations of `long`, `byte`, and `decimal` values.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
  Console.WriteLine("number1 now = {0}", number1);
else
  Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
  Console.WriteLine("number2 now = {0}", number2);
else
  Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
  Console.WriteLine("number3 now = {0}", number3);
else
  Console.WriteLine("number3 is not a valid decimal");
```

Robust Programming

Primitive numeric types also implement the `Parse` static method, which throws an exception if the string is not a valid number. `TryParse` is generally more efficient because it just returns false if the number is not valid.

.NET Framework Security

Always use the `TryParse` or `Parse` methods to validate user input from controls such as text boxes and combo boxes.

See also

- How to: Convert a byte Array to an int
- How to: Convert a String to a Number
- How to: Convert Between Hexadecimal Strings and Numeric Types
- Parsing Numeric Strings
- Formatting Types

Contents

Indexers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The `Program` class creates an instance of this class for storing strings.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

For more examples, see Related Sections.

Expression Body Definitions

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. Starting with C# 6, a read-only indexer can be implemented as an expression-bodied member, as the following example shows.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Note that `=>` introduces the expression body, and that the `get` keyword is not used.

Starting with C# 7.0, both the get and set accessor can be an implemented as expression-bodied members. In this case, both `get` and `set` keywords must be used. For example:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The this keyword is used to define the indexer.
- The value keyword is used to define the value being assigned by the `set` indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Related Sections

- Using Indexers
- Indexers in Interfaces
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility

C# Language Specification

For more information, see Indexers in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Properties

Using indexers (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

Indexers are a syntactic convenience that enable you to create a class, struct, or interface that client applications can access just as an array. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class `TempRecord` that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array `temps` of type `float[]` to store the temperature values. By implementing an indexer in this class, clients can access the temperatures in a `TempRecord` instance as `float temp = tr[4]` instead of as `float temp = tr.temps[4]`. The indexer notation not only simplifies the syntax for client applications; it also makes the class and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the this keyword, as the following example shows:

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

Remarks

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see Access Modifiers.

For more information about how to use indexers with an interface, see Interface Indexers.

The signature of an indexer consists of the number and types of its formal parameters. It doesn't include the indexer type or the names of the formal parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, you cannot pass an indexer value as a ref or out parameter.

To provide the indexer with a name that other languages can use, use System.Runtime.CompilerServices.IndexerNameAttribute, as the following example shows:

```
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]   // Indexer declaration
{
    // get and set accessors
}
```

This indexer will have the name `TheItem`. Not providing the name attribute would make `Item` the default name.

Example 1

The following example shows how to declare a private array field, `temps`, and an indexer. The indexer enables direct access to the instance `tempRecord[i]`. The alternative to using the indexer is to declare the array as a public member and access its members, `tempRecord.temps[i]`, directly.

Notice that when an indexer's access is evaluated, for example, in a `Console.Write` statement, the get accessor is

invoked. Therefore, if no `get` accessor exists, a compile-time error occurs.

```
class TempRecord
{
    // Array of temperature values
    private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
                                            61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length
    {
        get { return temps.Length; }
    }
    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get
        {
            return temps[index];
        }

        set
        {
            temps[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
/* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
```

Indexing using other values

C# doesn't limit the indexer parameter type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can co-exist.

Example 2

The following example declares a class that stores the days of the week. A `get` accessor takes a string, the name of a day, and returns the corresponding integer. For example, "Sunday" returns 0, "Monday" returns 1, and so on.

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // This method finds the day or returns an Exception if the day is not found
    private int GetDay(string testDay)
    {

        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == testDay)
            {
                return j;
            }
        }

        throw new System.ArgumentOutOfRangeException(testDay, "testDay must be in the form \"Sun\", \"Mon\",
etc");
    }

    // The get accessor returns an integer for a given string
    public int this[string day]
    {
        get
        {
            return (GetDay(day));
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        DayCollection week = new DayCollection();
        System.Console.WriteLine(week["Fri"]);

        // Raises ArgumentOutOfRangeException
        System.Console.WriteLine(week["Made-up Day"]);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
// Output: 5
```

Robust programming

There are two main ways in which the security and reliability of indexers can be improved:

- Be sure to incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value. In the first example earlier in this topic, the TempRecord class provides a Length

property that enables the client code to verify the input before passing it to the indexer. You can also put the error handling code inside the indexer itself. Be sure to document for users any exceptions that you throw inside an indexer accessor.

- Set the accessibility of the get and set accessors to be as restrictive as is reasonable. This is important for the `set` accessor in particular. For more information, see Restricting Accessor Accessibility.

See also

- C# Programming Guide
- Indexers
- Properties

Indexers in Interfaces (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Indexers can be declared on an interface. Accessors of interface indexers differ from the accessors of class indexers in the following ways:

- Interface accessors do not use modifiers.
- An interface accessor does not have a body.

Thus, the purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only.

The following is an example of an interface indexer accessor:

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

```
// Indexer on an interface:
public interface ISomeInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : ISomeInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        get
        {
            // The arr object will throw IndexOutOfRange exception.
            return arr[index];
        }
        set
        {
            arr[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        IndexerClass test = new IndexerClass();
        System.Random rand = new System.Random();
        // Call the indexer to initialize its elements.
        for (int i = 0; i < 10; i++)
        {
            test[i] = rand.Next();
        }
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, test[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
 */
```

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example:

```
string ISomeInterface.this[int index]
{
}
```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if an `Employee` class is implementing two interfaces, `ICitizen` and `IEmployee`, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

```
string IEmployee.this[int index]
{
}
```

implements the indexer on the `IEmployee` interface, while the following declaration:

```
string ICitizen.this[int index]
{
}
```

implements the indexer on the `ICitizen` interface.

See also

- C# Programming Guide
- Indexers
- Properties
- Interfaces

Comparison Between Properties and Indexers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Indexers are like properties. Except for the differences shown in the following table, all the rules that are defined for property accessors apply to indexer accessors also.

PROPERTY	INDEXER
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A `get` accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit `value` parameter.	A `set` accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.
Supports shortened syntax with Auto-Implemented Properties.	Does not support shortened syntax.

See also

- C# Programming Guide
- Indexers
- Properties

Contents

Events (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see How to: Subscribe to and Unsubscribe from Events.

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see Calling Synchronous Methods Asynchronously.
- In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.

Related Sections

For more information, see:

- How to: Subscribe to and Unsubscribe from Events
- How to: Publish Events that Conform to .NET Framework Guidelines
- How to: Raise Base Class Events in Derived Classes
- How to: Implement Interface Events
- How to: Use a Dictionary to Store Event Instances
- How to: Implement Custom Event Accessors

C# Language Specification

For more information, see Events in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- EventHandler
- C# Programming Guide
- Delegates
- Creating Event Handlers in Windows Forms

How to: Subscribe to and Unsubscribe from Events (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

You subscribe to an event that is published by another class when you want to write custom code that is called when that event is raised. For example, you might subscribe to a button's `click` event in order to make your application do something useful when the user clicks the button.

To subscribe to events by using the Visual Studio IDE

1. If you cannot see the **Properties** window, in **Design** view, right-click the form or control for which you want to create an event handler, and select **Properties**.
2. On top of the **Properties** window, click the **Events** icon.
3. Double-click the event that you want to create, for example the `Load` event.

   Visual C# creates an empty event handler method and adds it to your code. Alternatively you can add the code manually in **Code** view. For example, the following lines of code declare an event handler method that will be called when the `Form` class raises the `Load` event.

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   The line of code that is required to subscribe to the event is also automatically generated in the `InitializeComponent` method in the Form1.Designer.cs file in your project. It resembles this:

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

To subscribe to events programmatically

1. Define an event handler method whose signature matches the delegate signature for the event. For example, if the event is based on the EventHandler delegate type, the following code represents the method stub:

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Use the addition assignment operator (`+=`) to attach your event handler to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent`. Note that the subscriber class needs a reference to the publisher class in order to subscribe to its events.

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   Note that the previous syntax is new in C# 2.0. It is exactly equivalent to the C# 1.0 syntax in which the encapsulating delegate must be explicitly created by using the `new` keyword:

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

An event handler can also be added by using a lambda expression:

```
public Form1()
{
    InitializeComponent();
    // Use a lambda expression to define an event handler.
    this.Click += (s,e) => { MessageBox.Show(
       ((MouseEventArgs)e).Location.ToString());};
}
```

For more information, see How to: Use Lambda Expressions Outside LINQ.

To subscribe to events by using an anonymous method

- If you will not have to unsubscribe to an event later, you can use the addition assignment operator (`+=`) to attach an anonymous method to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent` and that a `CustomEventArgs` class has also been defined to carry some kind of specialized event information. Note that the subscriber class needs a reference to `publisher` in order to subscribe to its events.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

It is important to notice that you cannot easily unsubscribe from an event if you used an anonymous function to subscribe to it. To unsubscribe in this scenario, it is necessary to go back to the code where you subscribe to the event, store the anonymous method in a delegate variable, and then add the delegate to the event. In general, we recommend that you do not use anonymous functions to subscribe to events if you will have to unsubscribe from the event at some later point in your code. For more information about anonymous functions, see Anonymous Functions.

Unsubscribing

To prevent your event handler from being invoked when the event is raised, unsubscribe from the event. In order to prevent resource leaks, you should unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference to the delegate that encapsulates the subscriber's event handler. As long as the publishing object holds that reference, garbage collection will not delete your subscriber object.

To unsubscribe from an event

- Use the subtraction assignment operator (`-=`) to unsubscribe from an event:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to `null`.

See also

- Events

- event
- How to: Publish Events that Conform to .NET Framework Guidelines
- -= Operator (C# Reference)
- += Operator

How to: Publish Events that Conform to .NET Framework Guidelines (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The following procedure demonstrates how to add events that follow the standard .NET Framework pattern to your classes and structs. All events in the .NET Framework class library are based on the EventHandler delegate, which is defined as follows:

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

> **NOTE**
> The .NET Framework 2.0 introduces a generic version of this delegate, EventHandler<TEventArgs>. The following examples show how to use both versions.

Although events in classes that you define can be based on any valid delegate type, even delegates that return a value, it is generally recommended that you base your events on the .NET Framework pattern by using EventHandler, as shown in the following example.

To publish events based on the EventHandler pattern

1. (Skip this step and go to Step 3a if you do not have to send custom data with your event.) Declare the class for your custom data at a scope that is visible to both your publisher and subscriber classes. Then add the required members to hold your custom event data. In this example, a simple string is returned.

   ```
   public class CustomEventArgs : EventArgs
   {
       public CustomEventArgs(string s)
       {
           msg = s;
       }
       private string msg;
       public string Message
       {
           get { return msg; }
       }
   }
   ```

2. (Skip this step if you are using the generic version of EventHandler<TEventArgs> .) Declare a delegate in your publishing class. Give it a name that ends with *EventHandler*. The second parameter specifies your custom EventArgs type.

   ```
   public delegate void CustomEventHandler(object sender, CustomEventArgs a);
   ```

3. Declare the event in your publishing class by using one of the following steps.

   a. If you have no custom EventArgs class, your Event type will be the non-generic EventHandler delegate. You do not have to declare the delegate because it is already declared in the System namespace that is included when you create your C# project. Add the following code to your publisher class.

```
public event EventHandler RaiseCustomEvent;
```

b. If you are using the non-generic version of EventHandler and you have a custom class derived from EventArgs, declare your event inside your publishing class and use your delegate from step 2 as the type.

```
public event CustomEventHandler RaiseCustomEvent;
```

c. If you are using the generic version, you do not need a custom delegate. Instead, in your publishing class, you specify your event type as `EventHandler<CustomEventArgs>`, substituting the name of your own class between the angle brackets.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Example

The following example demonstrates the previous steps by using a custom EventArgs class and EventHandler<TEventArgs> as the event type.

```
namespace DotNetEvents
{
    using System;
    using System.Collections.Generic;

    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string s)
        {
            message = s;
        }
        private string message;

        public string Message
        {
            get { return message; }
            set { message = value; }
        }
    }

    // Class that publishes an event
    class Publisher
    {

        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Did something"));

        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
```

```
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> handler = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (handler != null)
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";

                // Use the () operator to raise the event.
                handler(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private string id;
        public Subscriber(string ID, Publisher pub)
        {
            id = ID;
            // Subscribe to the event using C# 2.0 syntax
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine(id + " received this message: {0}", e.Message);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Publisher pub = new Publisher();
            Subscriber sub1 = new Subscriber("sub1", pub);
            Subscriber sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            Console.WriteLine("Press Enter to close this window.");
            Console.ReadLine();

        }
    }
}
```

See also

- Delegate
- C# Programming Guide
- Events
- Delegates

How to: Raise Base Class Events in Derived Classes (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The following simple example shows the standard way to declare events in a base class so that they can also be raised from derived classes. This pattern is used extensively in Windows Forms classes in the .NET Framework class library.

When you create a class that can be used as a base class for other classes, you should consider the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Derived classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, most of the time, you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

> **NOTE**
> Do not declare virtual events in a base class and override them in a derived class. The C# compiler does not handle these correctly and it is unpredictable whether a subscriber to the derived event will actually be subscribing to the base class event.

Example

```
namespace BaseClassEvents
{
    using System;
    using System.Collections.Generic;

    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        private double newArea;

        public ShapeEventArgs(double d)
        {
            newArea = d;
        }
        public double NewArea
        {
            get { return newArea; }
        }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double area;

        public double Area
        {
            get { return area; }
            set { area = value; }
        }
        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;
```

```
    public event EventHandler<ShapeEventArgs> ShapeChanged;

    public abstract void Draw();

    //The event-invoking method that derived classes can override.
    protected virtual void OnShapeChanged(ShapeEventArgs e)
    {
        // Make a temporary copy of the event to avoid possibility of
        // a race condition if the last subscriber unsubscribes
        // immediately after the null check and before the event is raised.
        EventHandler<ShapeEventArgs> handler = ShapeChanged;
        if (handler != null)
        {
            handler(this, e);
        }
    }
}

public class Circle : Shape
{
    private double radius;
    public Circle(double d)
    {
        radius = d;
        area = 3.14 * radius * radius;
    }
    public void Update(double d)
    {
        radius = d;
        area = 3.14 * radius * radius;
        OnShapeChanged(new ShapeEventArgs(area));
    }
    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any circle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
{
    private double length;
    private double width;
    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
        area = length * width;
    }
    public void Update(double length, double width)
    {
        this.length = length;
        this.width = width;
        area = length * width;
        OnShapeChanged(new ShapeEventArgs(area));
    }
    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any rectangle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }
```

```
        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }

    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape s)
        {
            _list.Add(s);
            // Subscribe to the base class event.
            s.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            Shape s = (Shape)sender;

            // Diagnostic message for demonstration purposes.
            Console.WriteLine("Received event. Shape area is now {0}", e.NewArea);

            // Redraw the shape here.
            s.Draw();
        }
    }

    class Test
    {

        static void Main(string[] args)
        {
            //Create the event publishers and subscriber
            Circle c1 = new Circle(54);
            Rectangle r1 = new Rectangle(12, 9);
            ShapeContainer sc = new ShapeContainer();

            // Add the shapes to the container.
            sc.AddShape(c1);
            sc.AddShape(r1);

            // Cause some events to be raised.
            c1.Update(57);
            r1.Update(7, 7);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
```

```
*/
```

See also

- C# Programming Guide
- Events
- Delegates
- Access Modifiers
- Creating Event Handlers in Windows Forms

How to: Implement Interface Events (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

An interface can declare an event. The following example shows how to implement interface events in a class. Basically the rules are the same as when you implement any interface method or property.

To implement interface events in a class

Declare the event in your class and then invoke it in the appropriate areas.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Example

The following example shows how to handle the less-common situation in which your class inherits from two or more interfaces and each interface has an event with the same name. In this situation, you must provide an explicit interface implementation for at least one of the events. When you write an explicit interface implementation for an event, you must also write the `add` and `remove` event accessors. Normally these are provided by the compiler, but in this case the compiler cannot provide them.

By providing your own accessors, you can specify whether the two events are represented by the same event in your class, or by different events. For example, if the events should be raised at different times according to the interface specifications, you can associate each event with a separate implementation in your class. In the following example, subscribers determine which `OnDraw` event they will receive by casting the shape reference to either an `IShape` or an `IDrawingObject`.

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }


    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }

        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
```

```
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }


    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }

}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

See also

- C# Programming Guide
- Events
- Delegates
- Explicit Interface Implementation
- How to: Raise Base Class Events in Derived Classes

How to: Use a Dictionary to Store Event Instances (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

One use for `accessor-declarations` is to expose many events without allocating a field for each event, but instead using a Dictionary to store the event instances. This is only useful if you have many events, but you expect most of the events will not be implemented.

Example

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

public delegate void EventHandler1(int i);

public delegate void EventHandler2(string s);

public class PropertyEventsSample
{
    private readonly Dictionary<string, Delegate> _eventTable;
    private readonly List<EventHandler1> _event1List = new List<EventHandler1>();
    private readonly List<EventHandler2> _event2List = new List<EventHandler2>();
    public PropertyEventsSample()
    {
        _eventTable = new Dictionary<string, Delegate>
        {
            {"Event1", null},
            {"Event2", null}
        };
    }

    public event EventHandler1 Event1
    {
        add
        {
            _event1List.Add(value);
            lock (_eventTable)
            {
                _eventTable["Event1"] = (EventHandler1) _eventTable["Event1"] + value;
            }
        }
        remove
        {
            if (!_event1List.Contains(value)) return;
            _event1List.Remove(value);
            lock (_eventTable)
            {
                _eventTable["Event1"] = null;
                foreach (var event1 in _event1List)
                {
                    _eventTable["Event1"] = (EventHandler1) _eventTable["Event1"] + event1;
                }
            }
        }
    }

    public event EventHandler2 Event2
    {
        add
```

```
        {
            _event2List.Add(value);
            lock (_eventTable)
            {
                _eventTable["Event2"] = (EventHandler2) _eventTable["Event2"] + value;
            }
        }
        remove
        {
            if (!_event2List.Contains(value)) return;
            _event2List.Remove(value);
            lock (_eventTable)
            {
                _eventTable["Event2"] = null;
                foreach (var event2 in _event2List)
                {
                    _eventTable["Event2"] = (EventHandler2) _eventTable["Event2"] + event2;
                }
            }
        }
    }

    internal void RaiseEvent1(int i)
    {
        lock (_eventTable)
        {
            var handler1 = (EventHandler1) _eventTable["Event1"];
            handler1?.Invoke(i);
        }
    }

    internal void RaiseEvent2(string s)
    {
        lock (_eventTable)
        {
            var handler2 = (EventHandler2) _eventTable["Event2"];
            handler2?.Invoke(s);
        }
    }
}

public static class TestClass
{
    private static void Delegate1Method(int i)
    {
        Console.WriteLine(i);
    }

    private static void Delegate2Method(string s)
    {
        Console.WriteLine(s);
    }

    private static void Main()
    {
        var p = new PropertyEventsSample();

        p.Event1 += Delegate1Method;
        p.Event1 += Delegate1Method;
        p.Event1 -= Delegate1Method;
        p.RaiseEvent1(2);

        p.Event2 += Delegate2Method;
        p.Event2 += Delegate2Method;
        p.Event2 -= Delegate2Method;
        p.RaiseEvent2("TestString");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
```

```
            Console.ReadKey();
        }
    }
    /* Output:
        2
        TestString
    */
```

See also

- C# Programming Guide
- Events
- Delegates

How to: Implement Custom Event Accessors (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

An event is a special kind of multicast delegate that can only be invoked from within the class that it is declared in. Client code subscribes to the event by providing a reference to a method that should be invoked when the event is fired. These methods are added to the delegate's invocation list through event accessors, which resemble property accessors, except that event accessors are named `add` and `remove`. In most cases, you do not have to supply custom event accessors. When no custom event accessors are supplied in your code, the compiler will add them automatically. However, in some cases you may have to provide custom behavior. One such case is shown in the topic How to: Implement Interface Events.

Example

The following example shows how to implement custom add and remove event accessors. Although you can substitute any code inside the accessors, we recommend that you lock the event before you add or remove a new event handler method.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

See also

- Events
- event

Contents

Generics (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Generics were added to version 2.0 of the C# language and the common language runtime (CLR). Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Generics Overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related Sections

For more information:

- Introduction to Generics

- Benefits of Generics
- Generic Type Parameters
- Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generic Delegates
- Differences Between C++ Templates and C# Generics
- Generics and Reflection
- Generics in the Run Time

C# Language Specification

For more information, see the C# Language Specification.

See also

- System.Collections.Generic
- C# Programming Guide
- Types
- <typeparam>
- <typeparamref>
- Generics in .NET

Introduction to Generics (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Generic classes and methods combine reusability, type safety and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. Version 2.0 of the .NET Framework class library provides a new namespace, System.Collections.Generic, which contains several new generic-based collection classes. It is recommended that all applications that target the .NET Framework 2.0 and later use the new generic collection classes instead of the older non-generic counterparts such as ArrayList. For more information, see Generics in .NET.

Of course, you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the List<T> class provided by the .NET Framework class library instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

Note that T is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

See also

- System.Collections.Generic
- C# Programming Guide
- Generics

Benefits of Generics (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Generics provide the solution to a limitation in earlier versions of the common language runtime and the C# language in which generalization is accomplished by casting types to and from the universal base type Object. By creating a generic class, you can create a collection that is type-safe at compile-time.

The limitations of using non-generic collection classes can be demonstrated by writing a short program that uses the ArrayList collection class from the .NET class library. An instance of the ArrayList class can store any reference or value type.

```
// The .NET Framework 1.1 way to create a list:
System.Collections.ArrayList list1 = new System.Collections.ArrayList();
list1.Add(3);
list1.Add(105);

System.Collections.ArrayList list2 = new System.Collections.ArrayList();
list2.Add("It is raining in Redmond.");
list2.Add("It is snowing in the mountains.");
```

But this convenience comes at a cost. Any reference or value type that is added to an ArrayList is implicitly upcast to Object. If the items are value types, they must be boxed when they are added to the list, and unboxed when they are retrieved. Both the casting and the boxing and unboxing operations decrease performance; the effect of boxing and unboxing can be very significant in scenarios where you must iterate over large collections.

The other limitation is lack of compile-time type checking; because an ArrayList casts everything to Object, there is no way at compile-time to prevent client code from doing something such as this:

```
System.Collections.ArrayList list = new System.Collections.ArrayList();
// Add an integer to the list.
list.Add(3);
// Add a string to the list. This will compile, but may cause an error later.
list.Add("It is raining in Redmond.");

int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list)
{
    t += x;
}
```

Although perfectly acceptable and sometimes intentional if you are creating a heterogeneous collection, combining strings and `ints` in a single ArrayList is more likely to be a programming error, and this error will not be detected until runtime.

In versions 1.0 and 1.1 of the C# language, you could avoid the dangers of generalized code in the .NET Framework base class library collection classes only by writing your own type specific collections. Of course, because such a class is not reusable for more than one data type, you lose the benefits of generalization, and you have to rewrite the class for each type that will be stored.

What ArrayList and other similar classes really need is a way for client code to specify, on a per-instance basis, the particular data type that they intend to use. That would eliminate the need for the upcast to Object and would also make it possible for the compiler to do type checking. In other words, ArrayList needs a type parameter. That is exactly what generics provide. In the generic List<T> collection, in the System.Collections.Generic namespace, the

same operation of adding items to the collection resembles this:

```
// The .NET Framework 2.0 way to create a list
List<int> list1 = new List<int>();

// No boxing, no casting:
list1.Add(3);

// Compile-time error:
// list1.Add("It is raining in Redmond.");
```

For client code, the only added syntax with List<T> compared to ArrayList is the type argument in the declaration and instantiation. In return for this slightly more coding complexity, you can create a list that is not only safer than ArrayList, but also significantly faster, especially when the list items are value types.

See also

- System.Collections.Generic
- C# Programming Guide
- Introduction to Generics
- Boxing and Unboxing
- When to Use Generic Collections
- Guidelines for Collections

Generic Type Parameters (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In a generic type or method definition, a type parameters is a placeholder for a specific type that a client specifies when they instantiate a variable of the generic type. A generic class, such as `GenericList<T>` listed in Introduction to Generics, cannot be used as-is because it is not really a type; it is more like a blueprint for a type. To use `GenericList<T>`, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets. The type argument for this particular class can be any type recognized by the compiler. Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

In each of these instances of `GenericList<T>`, every occurrence of `T` in the class will be substituted at run time with the type argument. By means of this substitution, we have created three separate type-safe and efficient objects using a single class definition. For more information on how this substitution is performed by the CLR, see Generics in the Run Time.

Type Parameter Naming Guidelines

- **Do** name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name would not add value.

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **Consider** using T as the type parameter name for types with one single letter type parameter.

  ```
  public int IComparer<T>() { return 0; }
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Do** prefix descriptive type parameter names with "T".

  ```
  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **Consider** indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` may be called `TSession`.

See also

- System.Collections.Generic
- C# Programming Guide
- Generics

- Differences Between C++ Templates and C# Generics

Constraints on type parameters (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

Constraints inform the compiler about the capabilities a type argument must have. Without any constraints, the type argument could be any type. The compiler can only assume the members of Object, which is the ultimate base class for any .NET type. For more information, see Why use constraints. If client code tries to instantiate your class by using a type that is not allowed by a constraint, the result is a compile-time error. Constraints are specified by using the `where` contextual keyword. The following table lists the seven types of constraints:

CONSTRAINT	DESCRIPTION
`where T : struct`	The type argument must be a value type. Any value type except Nullable<T> can be specified. For more information about nullable types, see Nullable types.
`where T : class`	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type.
`where T : unmanaged`	The type argument must not be a reference type and must not contain any reference type members at any level of nesting.
`where T : new()`	The type argument must have a public parameterless constructor. When used together with other constraints, the `new()` constraint must be specified last.
`where T :` *<base class name>*	The type argument must be or derive from the specified base class.
`where T :` *<interface name>*	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
`where T : U`	The type argument supplied for T must be or derive from the argument supplied for U.

Some of the constraints are mutually exclusive. All value types must have an accessible parameterless constructor. The `struct` constraint implies the `new()` constraint and the `new()` constraint cannot be combined with the `struct` constraint. The `unmanaged` constraint implies the `struct` constraint. The `unmanaged` constraint cannot be combined with either the `struct` or `new()` constraints.

Why use constraints

By constraining the type parameter, you increase the number of allowable operations and method calls to those supported by the constraining type and all types in its inheritance hierarchy. When you design generic classes or methods, if you will be performing any operation on the generic members beyond simple assignment or calling any methods not supported by System.Object, you will have to apply constraints to the type parameter. For example, the base class constraint tells the compiler that only objects of this type or derived from this type will be used as type arguments. Once the compiler has this guarantee, it can allow methods of that type to be called in the

generic class. The following code example demonstrates the functionality you can add to the `GenericList<T>` class (in Introduction to Generics) by applying a base class constraint.

```
public class Employee
{
    public Employee(string s, int i) => (Name, ID) = (s, i);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

The constraint enables the generic class to use the `Employee.Name` property. The constraint specifies that all items of type `T` are guaranteed to be either an `Employee` object or an object that inherits from `Employee`.

Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic

types, as follows:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

When applying the `where T : class` constraint, avoid the `==` and `!=` operators on the type parameter because these operators will test for reference identity only, not for value equality. This behavior occurs even if these operators are overloaded in a type that is used as an argument. The following code illustrates this point; the output is false even though the String class overloads the `==` operator.

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}
private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

The compiler only knows that T is a reference type at compile time and must use the default operators that are valid for all reference types. If you must test for value equality, the recommended way is to also apply the `where T : IEquatable<T>` or `where T : IComparable<T>` constraint and implement the interface in any class that will be used to construct the generic class.

Constraining multiple parameters

You can apply constraints to multiple parameters, and multiple constraints to a single parameter, as shown in the following example:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Unbounded type parameters

Type parameters that have no constraints, such as T in public class `SampleClass<T>{}`, are called unbounded type parameters. Unbounded type parameters have the following rules:

- The `!=` and `==` operators cannot be used because there is no guarantee that the concrete type argument will support these operators.
- They can be converted to and from `System.Object` or explicitly converted to any interface type.
- You can compare them to null. If an unbounded parameter is compared to `null`, the comparison will always return false if the type argument is a value type.

Type parameters as constraints

The use of a generic type parameter as a constraint is useful when a member function with its own type parameter has to constrain that parameter to the type parameter of the containing type, as shown in the following example:

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

In the previous example, `T` is a type constraint in the context of the `Add` method, and an unbounded type parameter in the context of the `List` class.

Type parameters can also be used as constraints in generic class definitions. The type parameter must be declared within the angle brackets together with any other type parameters:

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

The usefulness of type parameters as constraints with generic classes is limited because the compiler can assume nothing about the type parameter except that it derives from `System.Object`. Use type parameters as constraints on generic classes in scenarios in which you want to enforce an inheritance relationship between two type parameters.

Unmanaged constraint

Beginning with C# 7.3, you can use the `unmanaged` constraint to specify that the type parameter must be an **unmanaged type**. An **unmanaged type** is a type that is not a reference type and doesn't contain reference type fields at any level of nesting. The `unmanaged` constraint enables you to write reusable routines to work with types that can be manipulated as blocks of memory, as shown in the following example:

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

The preceding method must be compiled in an `unsafe` context because it uses the `sizeof` operator on a type not known to be a built-in type. Without the `unmanaged` constraint, the `sizeof` operator is unavailable.

Delegate constraints

Also beginning with C# 7.3, you can use System.Delegate or System.MulticastDelegate as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. The `System.Delegate` constraint enables you to write code that works with delegates in a type-safe manner. The following code defines an extension method that combines two delegates provided they are the same type:

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source, TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

You can use the above method to combine delegates that are the same type:

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

If you uncomment the last line, it won't compile. Both `first` and `test` are delegate types, but they are different delegate types.

Enum constraints

Beginning in C# 7.3, you can also specify the System.Enum type as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. Generics using `System.Enum` provide type-safe programming to cache results from using the static methods in `System.Enum`. The following sample finds all the valid values for an enum type, and then builds a dictionary that maps those values to its string representation.

```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

The methods used make use of reflection, which has performance implications. You can call this method to build a collection that is cached and reused rather than repeating the calls that require reflection.

You could use it as shown in the following sample to create an enum and build a dictionary of its values and names:

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

See also

- System.Collections.Generic

- C# Programming Guide
- Introduction to Generics
- Generic Classes
- new Constraint

Generic Classes (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see Generic Collections in .NET.

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

  As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters (See Constraints on Type Parameters).

  A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the `as` operator on `T`, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

  Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

  For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as IComparable<T> where `T` is the type of your class.

For an example of a simple generic class, see Introduction to Generics.

The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class `Node<T>,` client code can reference the class either by specifying a type argument, to create a closed constructed type (`Node<int>`). Alternatively, it can leave the type parameter unspecified, for example when you specify a generic base class, to create an open constructed type (`Node<T>`). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as demonstrated in the following code:

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

Generic types can use multiple type parameters and constraints, as follows:

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Open constructed and closed constructed types can be used as method parameters:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

See also

- System.Collections.Generic
- C# Programming Guide
- Generics
- Saving the State of Enumerators
- An Inheritance Puzzle, Part One

Generic Interfaces (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. The preference for generic classes is to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types. The .NET Framework class library defines several generic interfaces for use with the collection classes in the System.Collections.Generic namespace.

When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used. The following code example shows a `SortedList<T>` class that derives from the `GenericList<T>` class. For more information, see Introduction to Generics. `SortedList<T>` adds the constraint `where T : IComparable<T>`. This enables the `BubbleSort` method in `SortedList<T>` to use the generic CompareTo method on list elements. In this example, list elements are a simple class, `Person`, that implements `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
```

```
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IEnumerable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}
```

```
// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

class Program
{
    static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
```

```
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Multiple interfaces can be specified as constraints on a single type, as follows:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

An interface can define more than one type parameter, as follows:

```
interface IDictionary<K, V>
{
}
```

The rules of inheritance that apply to classes also apply to interfaces:

```
interface IMonth<T> { }

interface IJanuary     : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>    : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

Generic interfaces can inherit from non-generic interfaces if the generic interface is contravariant, which means it only uses its type parameter as a return value. In the .NET Framework class library, IEnumerable<T> inherits from IEnumerable because IEnumerable<T> only uses `T` in the return value of GetEnumerator and in the Current property getter.

Concrete classes can implement closed constructed interfaces, as follows:

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Generic classes can implement generic interfaces or closed constructed interfaces as long as the class parameter list supplies all arguments required by the interface, as follows:

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

The rules that control method overloading are the same for methods within generic classes, generic structs, or generic interfaces. For more information, see Generic Methods.

See also

- C# Programming Guide
- Introduction to Generics
- interface
- Generics

Generic Methods (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

A generic method is a method that is declared with type parameters, as follows:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

The following code example shows one way to call the method by using `int` for the type argument:

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

You can also omit the type argument and the compiler will infer it. The following call to `Swap` is equivalent to the previous call:

```
Swap(ref a, ref b);
```

The same rules for type inference apply to static methods and instance methods. The compiler can infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters only from a constraint or return value. Therefore type inference does not work with methods that have no parameters. Type inference occurs at compile time before the compiler tries to resolve overloaded method signatures. The compiler applies type inference logic to all generic methods that share the same name. In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

If you define a generic method that takes the same type parameters as the containing class, the compiler generates warning CS0693 because within the method scope, the argument supplied for the inner `T` hides the argument supplied for the outer `T`. If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as shown in `GenericList2<T>` in the following example.

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

Use constraints to enable more specialized operations on type parameters in methods. This version of `Swap<T>`, now named `SwapIfGreater<T>`, can only be used with type arguments that implement IComparable<T>.

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Generic methods can be overloaded on several type parameters. For example, the following methods can all be located in the same class:

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

C# Language Specification

For more information, see the C# Language Specification.

See also

- System.Collections.Generic
- C# Programming Guide
- Introduction to Generics
- Methods

Generics and Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C# 2.0 and later, single-dimensional arrays that have a lower bound of zero automatically implement IList<T>. This enables you to create generic methods that can use the same code to iterate through arrays and other collection types. This technique is primarily useful for reading data in collections. The IList<T> interface cannot be used to add or remove elements from an array. An exception will be thrown if you try to call an IList<T> method such as RemoveAt on an array in this context.

The following code example demonstrates how a single generic method that takes an IList<T> input parameter can iterate through both a list and an array, in this case an array of integers.

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

See also

- System.Collections.Generic
- C# Programming Guide
- Generics
- Arrays
- Generics

Generic Delegates (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

A delegate can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
Del<int> m2 = Notify;
```

Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

Code that references the delegate must specify the type argument of the containing class, as follows:

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Generic delegates are especially useful in defining events based on the typical design pattern because the sender argument can be strongly typed and no longer has to be cast to and from Object.

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

See also

- System.Collections.Generic
- C# Programming Guide
- Introduction to Generics
- Generic Methods
- Generic Classes
- Generic Interfaces
- Delegates
- Generics

Differences Between C++ Templates and C# Generics (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

C# Generics and C++ templates are both language features that provide support for parameterized types. However, there are many differences between the two. At the syntax level, C# generics are a simpler approach to parameterized types without the complexity of C++ templates. In addition, C# does not attempt to provide all of the functionality that C++ templates provide. At the implementation level, the primary difference is that C# generic type substitutions are performed at runtime and generic type information is thereby preserved for instantiated objects. For more information, see Generics in the Run Time.

The following are the key differences between C# Generics and C++ templates:

- C# generics do not provide the same amount of flexibility as C++ templates. For example, it is not possible to call arithmetic operators in a C# generic class, although it is possible to call user defined operators.
- C# does not allow non-type template parameters, such as `template C<int i> {}`.
- C# does not support explicit specialization; that is, a custom implementation of a template for a specific type.
- C# does not support partial specialization: a custom implementation for a subset of the type arguments.
- C# does not allow the type parameter to be used as the base class for the generic type.
- C# does not allow type parameters to have default types.
- In C#, a generic type parameter cannot itself be a generic, although constructed types can be used as generics. C++ does allow template parameters.
- C++ allows code that might not be valid for all type parameters in the template, which is then checked for the specific type used as the type parameter. C# requires code in a class to be written in such a way that it will work with any type that satisfies the constraints. For example, in C++ it is possible to write a function that uses the arithmetic operators `+` and `-` on objects of the type parameter, which will produce an error at the time of instantiation of the template with a type that does not support these operators. C# disallows this; the only language constructs allowed are those that can be deduced from the constraints.

See also

- C# Programming Guide
- Introduction to Generics
- Templates

Generics in the Run Time (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

When a generic type or method is compiled into Microsoft intermediate language (MSIL), it contains metadata that identifies it as having type parameters. How the MSIL for a generic type is used differs based on whether the supplied type parameter is a value type or reference type.

When a generic type is first constructed with a value type as a parameter, the runtime creates a specialized generic type with the supplied parameter or parameters substituted in the appropriate locations in the MSIL. Specialized generic types are created one time for each unique value type that is used as a parameter.

For example, suppose your program code declared a stack that is constructed of integers:

```
Stack<int> stack;
```

At this point, the runtime generates a specialized version of the Stack<T> class that has the integer substituted appropriately for its parameter. Now, whenever your program code uses a stack of integers, the runtime reuses the generated specialized Stack<T> class. In the following example, two instances of a stack of integers are created, and they share a single instance of the `Stack<int>` code:

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

However, suppose that another Stack<T> class with a different value type such as a `long` or a user-defined structure as its parameter is created at another point in your code. As a result, the runtime generates another version of the generic type and substitutes a `long` in the appropriate locations in MSIL. Conversions are no longer necessary because each specialized generic class natively contains the value type.

Generics work somewhat differently for reference types. The first time a generic type is constructed with any reference type, the runtime creates a specialized generic type with object references substituted for the parameters in the MSIL. Then, every time that a constructed type is instantiated with a reference type as its parameter, regardless of what type it is, the runtime reuses the previously created specialized version of the generic type. This is possible because all references are the same size.

For example, suppose you had two reference types, a `Customer` class and an `Order` class, and also suppose that you created a stack of `Customer` types:

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

At this point, the runtime generates a specialized version of the Stack<T> class that stores object references that will be filled in later instead of storing data. Suppose the next line of code creates a stack of another reference type, which is named `Order`:

```
Stack<Order> orders = new Stack<Order>();
```

Unlike with value types, another specialized version of the Stack<T> class is not created for the `Order` type. Instead, an instance of the specialized version of the Stack<T> class is created and the `orders` variable is set to reference it. Suppose that you then encountered a line of code to create a stack of a `Customer` type:

```
customers = new Stack<Customer>();
```

As with the previous use of the Stack<T> class created by using the `Order` type, another instance of the specialized Stack<T> class is created. The pointers that are contained therein are set to reference an area of memory the size of a `Customer` type. Because the number of reference types can vary wildly from program to program, the C# implementation of generics greatly reduces the amount of code by reducing to one the number of specialized classes created by the compiler for generic classes of reference types.

Moreover, when a generic C# class is instantiated by using a value type or reference type parameter, reflection can query it at runtime and both its actual type and its type parameter can be ascertained.

See also

- System.Collections.Generic
- C# Programming Guide
- Introduction to Generics
- Generics

Generics and Reflection (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Because the Common Language Runtime (CLR) has access to generic type information at run time, you can use reflection to obtain information about generic types in the same way as for non-generic types. For more information, see Generics in the Run Time.

In the .NET Framework 2.0 several new members are added to the Type class to enable run-time information for generic types. See the documentation on these classes for more information on how to use these methods and properties. The System.Reflection.Emit namespace also contains new members that support generics. See How to: Define a Generic Type with Reflection Emit.

For a list of the invariant conditions for terms used in generic reflection, see the IsGenericType property remarks.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
IsGenericType	Returns true if a type is generic.
GetGenericArguments	Returns an array of `Type` objects that represent the type arguments supplied for a constructed type, or the type parameters of a generic type definition.
GetGenericTypeDefinition	Returns the underlying generic type definition for the current constructed type.
GetGenericParameterConstraints	Returns an array of `Type` objects that represent the constraints on the current generic type parameter.
ContainsGenericParameters	Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types have not been supplied.
GenericParameterAttributes	Gets a combination of `GenericParameterAttributes` flags that describe the special constraints of the current generic type parameter.
GenericParameterPosition	For a `Type` object that represents a type parameter, gets the position of the type parameter in the type parameter list of the generic type definition or generic method definition that declared the type parameter.
IsGenericParameter	Gets a value that indicates whether the current `Type` represents a type parameter of a generic type or method definition.
IsGenericTypeDefinition	Gets a value that indicates whether the current Type represents a generic type definition, from which other generic types can be constructed. Returns true if the type represents the definition of a generic type.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
DeclaringMethod	Returns the generic method that defined the current generic type parameter, or null if the type parameter was not defined by a generic method.
MakeGenericType	Substitutes the elements of an array of types for the type parameters of the current generic type definition, and returns a Type object representing the resulting constructed type.

In addition, members of the MethodInfo class enable run-time information for generic methods. See the IsGenericMethod property remarks for a list of invariant conditions for terms used to reflect on generic methods.

SYSTEM.REFLECTION.MEMBERINFO MEMBER NAME	DESCRIPTION
IsGenericMethod	Returns true if a method is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments of a constructed generic method or the type parameters of a generic method definition.
GetGenericMethodDefinition	Returns the underlying generic method definition for the current constructed method.
ContainsGenericParameters	Returns true if the method or any of its enclosing types contain any type parameters for which specific types have not been supplied.
IsGenericMethodDefinition	Returns true if the current MethodInfo represents the definition of a generic method.
MakeGenericMethod	Substitutes the elements of an array of types for the type parameters of the current generic method definition, and returns a MethodInfo object representing the resulting constructed method.

See also

- C# Programming Guide
- Generics
- Reflection and Generic Types
- Generics

Generics and Attributes (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Attributes can be applied to generic types in the same way as non-generic types. For more information on applying attributes, see Attributes.

Custom attributes are only permitted to reference open generic types, which are generic types for which no type arguments are supplied, and closed constructed generic types, which supply arguments for all type parameters.

The following examples use this custom attribute:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

An attribute can reference an open generic type:

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Specify multiple type parameters using the appropriate number of commas. In this example, `GenericClass2` has two type parameters:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

An attribute can reference a closed constructed generic type:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

An attribute that references a generic type parameter will cause a compile-time error:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
class ClassD<T> { }
```

A generic type cannot inherit from Attribute:

```
//public class CustomAtt<T> : System.Attribute {}  //Error
```

To obtain information about a generic type or type parameter at run time, you can use the methods of System.Reflection. For more information, see Generics and Reflection

See also

- C# Programming Guide
- Generics
- Attributes

Contents

Language Integrated Query (LINQ)

10/13/2018 • 3 minutes to read • Edit Online

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see Type relationships in LINQ query operations.
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see Introduction to LINQ queries.

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see C# language specification and Standard query operators overview .
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see Query syntax and method syntax in LINQ.
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. IEnumerable<T> queries are compiled to delegates. IQueryable and IQueryable<T> queries are compiled to expression trees. For more information, see Expression trees.

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in Query expression basics, and then read the documentation for the LINQ technology in which you are interested:

- XML documents: LINQ to XML
- ADO.NET Entity Framework: LINQ to entities
- .NET collections, files, strings and so on: LINQ to objects

To gain a deeper understanding of LINQ in general, see LINQ in C#.

To start working with LINQ in C#, see the tutorial Working with LINQ.

Query expression basics

1/23/2019 • 12 minutes to read • Edit Online

This article introduces the basic concepts related to query expressions in C#.

What is a query and what does it do?

A *query* is a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have. A query is distinct from the results that it produces.

Generally, the source data is organized logically as a sequence of elements of the same kind. For example, a SQL database table contains a sequence of rows. In an XML file, there is a "sequence" of XML elements (although these are organized hierarchically in a tree structure). An in-memory collection contains a sequence of objects.

From an application's viewpoint, the specific type and structure of the original source data is not important. The application always sees the source data as an IEnumerable<T> or IQueryable<T> collection. For example, in LINQ to XML, the source data is made visible as an `IEnumerable`<XElement>.

Given this source sequence, a query may do one of three things:

- Retrieve a subset of the elements to produce a new sequence without modifying the individual elements. The query may then sort or group the returned sequence in various ways, as shown in the following example (assume `scores` is an `int[]`):

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Retrieve a sequence of elements as in the previous example but transform them to a new type of object. For example, a query may retrieve only the last names from certain customer records in a data source. Or it may retrieve the complete record and then use it to construct another in-memory object type or even XML data before generating the final result sequence. The following example shows a projection from an `int` to a `string`. Note the new type of `highScoresQuery`.

  ```
  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- Retrieve a singleton value about the source data, such as:

  - The number of elements that match a certain condition.
  - The element that has the greatest or least value.
  - The first element that matches a condition, or the sum of particular values in a specified set of elements. For example, the following query returns the number of scores greater than 80 from the `scores` integer array:

```
int highScoreCount =
    (from score in scores
     where score > 80
     select score)
     .Count();
```

In the previous example, note the use of parentheses around the query expression before the call to the `Count` method. You can also express this by using a new variable to store the concrete result. This technique is more readable because it keeps the variable that stores the query separate from the query that stores a result.

```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

In the previous example, the query is executed in the call to `Count`, because `Count` must iterate over the results in order to determine the number of elements returned by `highScoresQuery`.

What is a query expression?

A *query expression* is a query expressed in query syntax. A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid. A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery. Each clause in turn contains one or more C# expressions, and these expressions may themselves be either a query expression or contain a query expression.

A query expression must begin with a from clause and must end with a select or group clause. Between the first `from` clause and the last `select` or `group` clause, it can contain one or more of these optional clauses: where, orderby, join, let and even additional from clauses. You can also use the into keyword to enable the result of a `join` or `group` clause to serve as the source for additional query clauses in the same query expression.

Query variable

In LINQ, a query variable is any variable that stores a *query* instead of the *results* of a query. More specifically, a query variable is always an enumerable type that will produce a sequence of elements when it is iterated over in a `foreach` statement or a direct call to its `IEnumerator.MoveNext` method.

The following code example shows a simple query expression with one data source, one filtering clause, one ordering clause, and no transformation of the source elements. The `select` clause ends the query.

```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

In the previous example, `scoreQuery` is a *query variable,* which is sometimes referred to as just a *query*. The query variable stores no actual result data, which is produced in the `foreach` loop. And when the `foreach` statement executes, the query results are not returned through the query variable `scoreQuery`. Rather, they are returned through the iteration variable `testScore`. The `scoreQuery` variable can be iterated in a second `foreach` loop. It will produce the same results as long as neither it nor the data source has been modified.

A query variable may store a query that is expressed in query syntax or method syntax, or a combination of the two. In the following examples, both `queryMajorCities` and `queryMajorCities2` are query variables:

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;


// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);
```

On the other hand, the following two examples show variables that are not query variables even though each is initialized with a query. They are not query variables because they store results:

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
       .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

For more information about the different ways to express queries, see Query syntax and method syntax in LINQ.

Explicit and implicit typing of query variables

This documentation usually provides the explicit type of the query variable in order to show the type relationship between the query variable and the select clause. However, you can also use the var keyword to instruct the compiler to infer the type of a query variable (or any other local variable) at compile time. For example, the query example that was shown previously in this topic can also be expressed by using implicit typing:

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

For more information, see Implicitly typed local variables and Type relationships in LINQ query operations.

Starting a query expression

A query expression must begin with a `from` clause. It specifies a data source together with a range variable. The range variable represents each successive element in the source sequence as the source sequence is being traversed. The range variable is strongly typed based on the type of elements in the data source. In the following example, because `countries` is an array of `Country` objects, the range variable is also typed as `Country`. Because the range variable is strongly typed, you can use the dot operator to access any available members of the type.

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

The range variable is in scope until the query is exited either with a semicolon or with a *continuation* clause.

A query expression may contain multiple `from` clauses. Use additional `from` clauses when each element in the source sequence is itself a collection or contains a collection. For example, assume that you have a collection of `Country` objects, each of which contains a collection of `City` objects named `Cities`. To query the `City` objects in each `Country`, use two `from` clauses as shown here:

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

For more information, see from clause.

Ending a query expression

A query expression must end with either a `group` clause or a `select` clause.

group clause

Use the `group` clause to produce a sequence of groups organized by a key that you specify. The key can be any data type. For example, the following query creates a sequence of groups that contains one or more `Country` objects and whose key is a `char` value.

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

For more information about grouping, see group clause.

select clause

Use the `select` clause to produce all other types of sequences. A simple `select` clause just produces a sequence of the same type of objects as the objects that are contained in the data source. In this example, the data source contains `Country` objects. The `orderby` clause just sorts the elements into a new order and the `select` clause produces a sequence of the reordered `Country` objects.

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

The `select` clause can be used to transform source data into sequences of new types. This transformation is also named a *projection*. In the following example, the `select` clause *projects* a sequence of anonymous types which contains only a subset of the fields in the original element. Note that the new objects are initialized by using an object initializer.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

For more information about all the ways that a `select` clause can be used to transform source data, see select clause.

Continuations with "into"

You can use the `into` keyword in a `select` or `group` clause to create a temporary identifier that stores a query. Do this when you must perform additional query operations on a query after a grouping or select operation. In the following example `countries` are grouped according to population in ranges of 10 million. After these groups are created, additional clauses filter out some groups, and then to sort the groups in ascending order. To perform those additional operations, the continuation represented by `countryGroup` is required.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

For more information, see into.

Filtering, ordering, and joining

Between the starting `from` clause, and the ending `select` or `group` clause, all other clauses (`where`, `join`, `orderby`, `from`, `let`) are optional. Any of the optional clauses may be used zero times or multiple times in a query body.

where clause

Use the `where` clause to filter out elements from the source data based on one or more predicate expressions. The `where` clause in the following example has one predicate with two conditions.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

For more information, see where clause.

orderby clause

Use the `orderby` clause to sort the results in either ascending or descending order. You can also specify secondary sort orders. The following example performs a primary sort on the `country` objects by using the `Area` property. It then performs a secondary sort by using the `Population` property.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

The `ascending` keyword is optional; it is the default sort order if no order is specified. For more information, see orderby clause.

join clause

Use the `join` clause to associate and/or combine elements from one data source with elements from another data source based on an equality comparison between specified keys in each element. In LINQ, join operations are

performed on sequences of objects whose elements are different types. After you have joined two sequences, you must use a `select` or `group` statement to specify which element to store in the output sequence. You can also use an anonymous type to combine properties from each set of associated elements into a new type for the output sequence. The following example associates `prod` objects whose `Category` property matches one of the categories in the `categories` string array. Products whose `Category` does not match any string in `categories` are filtered out. The `select` statement projects a new type whose properties are taken from both `cat` and `prod`.

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

You can also perform a group join by storing the results of the `join` operation into a temporary variable by using the into keyword. For more information, see join clause.

let clause

Use the `let` clause to store the result of an expression, such as a method call, in a new range variable. In the following example, the range variable `firstName` stores the first element of the array of strings that is returned by `Split`.

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar
```

For more information, see let clause.

Subqueries in a query expression

A query clause may itself contain a query expression, which is sometimes referred to as a *subquery*. Each subquery starts with its own `from` clause that does not necessarily point to the same data source in the first `from` clause. For example, the following query shows a query expression that is used in the select statement to retrieve the results of a grouping operation.

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
             select student2.Scores.Average())
             .Max()
    };
```

For more information, see How to: perform a subquery on a grouping operation.

See also

- C# programming guide
- Language Integrated Query (LINQ)

- Query keywords (LINQ)
- Standard query operators overview

LINQ in C#

7/3/2018 • 2 minutes to read • Edit Online

This section contains links to topics that provide more detailed information about LINQ.

In this section

Introduction to LINQ queries

Describes the three parts of the basic LINQ query operation that are common across all languages and data sources.

LINQ and generic types

Provides a brief introduction to generic types as they are used in LINQ.

Data transformations with LINQ

Describes the various ways that you can transform data retrieved in queries.

Type relationships in LINQ query operations

Describes how types are preserved and/or transformed in the three parts of a LINQ query operation

Query syntax and method syntax in LINQ

Compares method syntax and query syntax as two ways to express a LINQ query.

C# features that support LINQ

Describes the language constructs in C# that support LINQ.

Related sections

LINQ query expressions

Includes an overview of queries in LINQ and provides links to additional resources.

Standard query operators overview

Introduces the standard methods used in LINQ.

Write LINQ queries in C#

1/24/2019 • 4 minutes to read • Edit Online

This article shows the three ways in which you can write a LINQ query in C#:

1. Use query syntax.
2. Use method syntax.
3. Use a combination of query syntax and method syntax.

The following examples demonstrate some simple LINQ queries by using each approach listed previously. In general, the rule is to use (1) whenever possible, and use (2) and (3) whenever necessary.

> **NOTE**
> These queries operate on simple in-memory collections; however, the basic syntax is identical to that used in LINQ to Entities and LINQ to XML.

Example - Query syntax

The recommended way to write most queries is to use *query syntax* to create *query expressions*. The following example shows three query expressions. The first query expression demonstrates how to filter or restrict results by applying conditions with a `where` clause. It returns all elements in the source sequence whose values are greater than 7 or less than 3. The second expression demonstrates how to order the returned results. The third expression demonstrates how to group results according to a key. This query returns two groups based on the first letter of the word.

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Note that the type of the queries is IEnumerable<T>. All of these queries could be written using `var` as shown in the following example:

```
var query = from num in numbers...
```

In each previous example, the queries do not actually execute until you iterate over the query variable in a `foreach` statement or other statement. For more information, see Introduction to LINQ Queries.

Example - Method syntax

Some query operations must be expressed as a method call. The most common such methods are those that return singleton numeric values, such as Sum, Max, Min, Average, and so on. These methods must always be called last in any query because they represent only a single value and cannot serve as the source for an additional query operation. The following example shows a method call in a query expression:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

If the method has Action or Func parameters, these are provided in the form of a lambda expression, as shown in the following example:

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

In the previous queries, only Query #4 executes immediately. This is because it returns a single value, and not a generic IEnumerable<T> collection. The method itself has to use `foreach` in order to compute its value.

Each of the previous queries can be written by using implicit typing with var, as shown in the following example:

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Example - Mixed query and method syntax

This example shows how to use method syntax on the results of a query clause. Just enclose the query expression in parentheses, and then apply the dot operator and call the method. In the following example, query #7 returns a count of the numbers whose value is between 3 and 7. In general, however, it is better to use a second variable to store the result of the method call. In this manner, the query is less likely to be confused with the results of the query.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Because Query #7 returns a single value and not a collection, the query executes immediately.

The previous query can be written by using implicit typing with `var`, as follows:

```
var numCount = (from num in numbers...
```

It can be written in method syntax as follows:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

It can be written by using explicit typing, as follows:

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

See also

- Walkthrough: Writing Queries in C#
- Language Integrated Query (LINQ)
- where clause

Query a collection of objects

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to perform a simple query over a list of `Student` objects. Each `Student` object contains some basic information about the student, and a list that represents the student's scores on four examinations.

This application serves as the framework for many other examples in this section that use the same `students` data source.

Example

The following query returns the students who received a score of 90 or greater on their first exam.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
```

```
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }



    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

This query is intentionally simple to enable you to experiment. For example, you can try more conditions in the `where` clause, or use an `orderby` clause to sort the results.

See also

- Language Integrated Query (LINQ)
- String interpolation

How to: Return a Query from a Method (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to return a query from a method as the return value and as an `out` parameter.

Query objects are composable, meaning that you can return a query from a method. Objects that represent queries do not store the resulting collection, but rather the steps to produce the results when needed. The advantage of returning query objects from methods is that they can be further composed or modified. Therefore any return value or `out` parameter of a method that returns a query must also have that type. If a method materializes a query into a concrete List<T> or Array type, it is considered to be returning the query results instead of the query itself. A query variable that is returned from a method can still be composed or modified.

Example

In the following example, the first method returns a query as a return value, and the second method returns a query as an `out` parameter. Note that in both cases it is a query that is returned, not query results.

```
class MQ
{
    // QueryMethhod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
```

```
        // Rest the mouse pointer over the call to QueryMethod1 to see its
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }


        IEnumerable<string> myQuery2;
        // QueryMethod2 returns a query as the value of its out parameter.
        app.QueryMethod2(ref nums, out myQuery2);

        // Execute the returned query.
        Console.WriteLine("\nResults of executing myQuery2:");
        foreach (string s in myQuery2)
        {
            Console.WriteLine(s);
        }


        // You can modify a query by using query composition. A saved query
        // is nested inside a new query definition that revises the results
        // of the first query.
        myQuery1 = from item in myQuery1
                   orderby item descending
                   select item;

        // Execute the modified query.
        Console.WriteLine("\nResults of executing modified myQuery1:");
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

See also

- Language Integrated Query (LINQ)

Store the results of a query in memory

8/24/2018 • 2 minutes to read • Edit Online

A query is basically a set of instructions for how to retrieve and organize data. Queries are executed lazily, as each subsequent item in the result is requested. When you use `foreach` to iterate the results, items are returned as accessed. To evaluate a query and store its results without executing a `foreach` loop, just call one of the following methods on the query variable:

- ToList
- ToArray
- ToDictionary
- ToLookup

We recommend that when you store the query results, you assign the returned collection object to a new variable as shown in the following example:

Example

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

See also

- Language Integrated Query (LINQ)

Group query results

1/24/2019 • 8 minutes to read • Edit Online

Grouping is one of the most powerful capabilities of LINQ. The following examples show how to group data in various ways:

- By a single property.
- By the first letter of a string property.
- By a computed numeric range.
- By Boolean predicate or other expression.
- By a compound key.

In addition, the last two queries project their results into a new anonymous type that contains only the student's first and last name. For more information, see the group clause.

Example

All the examples in this topic use the following helper classes and data sources.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
```

```
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }



    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Example

The following example shows how to group source elements by using a single property of the element as the group key. In this case the key is a `string`, the student's last name. It is also possible to use a substring for the key. The grouping operation uses the default equality comparer for the type.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySingleProperty()`.

```
public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by a single property in an object:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

Example

The following example shows how to group source elements by using something other than a property of the object for the group key. In this example, the key is the first letter of the student's last name.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySubstring()`.

```
public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by something other than a property of the object:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

Example

The following example shows how to group source elements by using a numeric range as a group key. The query then projects the results into an anonymous type that contains only the first and last name and the percentile range to which the student belongs. An anonymous type is used because it is not necessary to use the complete `Student` object to display the results. `GetPercentile` is a helper function that calculates a percentile based on the student's average score. The method returns an integer between 0 and 10.

```
//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByRange()`.

```
public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
        }
    }
}
/* Output:
    Group by numeric range and project into a new anonymous type:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

Example

The following example shows how to group source elements by using a Boolean comparison expression. In this example, the Boolean expression tests whether a student's average exam score is greater than 75. As in previous examples, the results are projected into an anonymous type because the complete source element is not needed. Note that the properties in the anonymous type become properties on the `Key` member and can be accessed by name when the query is executed.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByBoolean()`.

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                               group new { student.FirstName, student.LastName }
                                    by student.ExamScores.Average() > 75 into studentGroup
                               select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
/* Output:
    Group by a Boolean into two groups with string keys
    "True" and "False" and project into a new anonymous type:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

Example

The following example shows how to use an anonymous type to encapsulate a key that contains multiple values. In this example, the first key value is the first letter of the student's last name. The second key value is a Boolean that specifies whether the student scored over 85 on the first exam. You can order the groups by any property in the key.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByCompositeKey()`.

```
public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"\t{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

See also

- GroupBy
- IGrouping<TKey,TElement>
- Language Integrated Query (LINQ)
- group clause
- Anonymous Types
- Perform a Subquery on a Grouping Operation
- Create a Nested Group
- Grouping Data

Create a nested group

1/24/2019 • 2 minutes to read • Edit Online

The following example shows how to create nested groups in a LINQ query expression. Each group that is created according to student year or grade level is then further subdivided into groups based on the individuals' names.

Example

> **NOTE**
> This example contains references to objects that are defined in the sample code in Query a collection of objects.

```
public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"\t\t{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}
/*
 Output:
DataClass.Student Level = SecondYear
        Names that begin with: Adams
                Adams Terry
        Names that begin with: Garcia
                Garcia Hugo
        Names that begin with: Omelchenko
                Omelchenko Svetlana
DataClass.Student Level = ThirdYear
        Names that begin with: Fakhouri
                Fakhouri Fadi
        Names that begin with: Garcia
                Garcia Debra
        Names that begin with: Tucker
                Tucker Lance
DataClass.Student Level = FirstYear
        Names that begin with: Feng
                Feng Hanying
        Names that begin with: Mortensen
                Mortensen Sven
        Names that begin with: Tucker
                Tucker Michael
DataClass.Student Level = FourthYear
        Names that begin with: Garcia
                Garcia Cesar
        Names that begin with: O'Donnell
                O'Donnell Claire
        Names that begin with: Zabokritski
                Zabokritski Eugene
 */
```

Note that three nested `foreach` loops are required to iterate over the inner elements of a nested group.

See also

- Language Integrated Query (LINQ)

Perform a subquery on a grouping operation

1/24/2019 • 2 minutes to read • Edit Online

This article shows two different ways to create a query that orders the source data into groups, and then performs a subquery over each group individually. The basic technique in each example is to group the source elements by using a *continuation* named `newGroup`, and then generating a new subquery against `newGroup`. This subquery is run against each new group that is created by the outer query. Note that in this particular example the final output is not a group, but a flat sequence of anonymous types.

For more information about how to group, see group clause.

For more information about continuations, see into. The following example uses an in-memory data structure as the data source, but the same principles apply for any kind of LINQ data source.

Example

> **NOTE**
>
> This example contains references to objects that are defined in the sample code in Query a collection of objects.

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
            (from student2 in studentGroup
             select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

The query in the snippet above can also be written using method syntax. The following code snippet has a semantically equivalent query written using method syntax.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
        });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

See also

- Language Integrated Query (LINQ)

Group results by contiguous keys

8/24/2018 • 7 minutes to read • Edit Online

The following example shows how to group elements into chunks that represent subsequences of contiguous keys. For example, assume that you are given the following sequence of key-value pairs:

KEY	VALUE
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

The following groups will be created in this order:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

The solution is implemented as an extension method that is thread-safe and that returns its results in a streaming manner. In other words, it produces its groups as it moves through the source sequence. Unlike the `group` or `orderby` operators, it can begin returning groups to the caller before all of the sequence has been read.

Thread-safety is accomplished by making a copy of each group or chunk as the source sequence is iterated, as explained in the source code comments. If the source sequence has a large sequence of contiguous items, the common language runtime may throw an OutOfMemoryException.

Example

The following example shows both the extension method and the client code that uses it:

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


```

```
namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            //   (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)

            // A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk. Each
```

ChunkItem

```
            // has a reference to the next ChunkItem in the list.
            class ChunkItem
            {
                public ChunkItem(TSource value)
                {
                    Value = value;
                }
                public readonly TSource Value;
                public ChunkItem Next = null;
            }

            // The value that is used to determine matching elements
            private readonly TKey key;

            // Stores a reference to the enumerator for the source sequence
            private IEnumerator<TSource> enumerator;

            // A reference to the predicate that is used to compare keys.
            private Func<TSource, bool> predicate;

            // Stores the contents of the first source element that
            // belongs with this chunk.
            private readonly ChunkItem head;

            // End of the list. It is repositioned each time a new
            // ChunkItem is added.
            private ChunkItem tail;

            // Flag to indicate the source iterator has reached the end of the source sequence.
            internal bool isLastSourceElement = false;

            // Private object for thread syncronization
            private object m_Lock;

            // REQUIRES: enumerator != null && predicate != null
            public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)
            {
                this.key = key;
                this.enumerator = enumerator;
                this.predicate = predicate;

                // A Chunk always contains at least one element.
                head = new ChunkItem(enumerator.Current);

                // The end and beginning are the same until the list contains > 1 elements.
                tail = head;

                m_Lock = new object();
            }

            // Indicates that all chunk elements have been copied to the list of ChunkItems,
            // and the source enumerator is either at the end, or else on an element with a new key.
            // the tail of the linked list is set to null in the CopyNextChunkElement method if the
            // key of the next element does not match the current chunk's key, or there are no more elements in
the source.
            private bool DoneCopyingChunk => tail == null;

            // Adds one ChunkItem to the current group
            // REQUIRES: !DoneCopyingChunk && lock(this)
            private void CopyNextChunkElement()
            {
                // Try to advance the iterator on the source sequence.
                // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
                isLastSourceElement = !enumerator.MoveNext();

                // If we are (a) at the end of the source, or (b) at the end of the current chunk
                // then null out the enumerator and predicate for reuse with the next chunk.
                if (isLastSourceElement || !predicate(enumerator.Current))
```

```
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next;
    }

    // Called after the end of the last chunk was reached. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key => key;

    // Invoked by the inner foreach loop. This method stays just one step ahead
    // of the client requests. It adds the next element of the chunk only after
    // the clients requests the last element in the list so far.
    public IEnumerator<TSource> GetEnumerator()
    {
        //Specify the initial element to enumerate.
        ChunkItem current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }
```

```
            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
        }
    }

    // A simple named type is used for easier viewing in the debugger. Anonymous types
    // work just as well with the ChunkBy operator.
    public class KeyValPair
    {
        public string Key { get; set; }
        public string Value { get; set; }
    }

    class Program
    {
        // The source sequence.
        public static IEnumerable<KeyValPair> list;

        // Query variable declared as class member to be available
        // on different threads.
        static IEnumerable<IGrouping<string, KeyValPair>> query;

        static void Main(string[] args)
        {
            // Initialize the source sequence with an array initializer.
            list = new[]
            {
                new KeyValPair{ Key = "A", Value = "We" },
                new KeyValPair{ Key = "A", Value = "think" },
                new KeyValPair{ Key = "A", Value = "that" },
                new KeyValPair{ Key = "B", Value = "Linq" },
                new KeyValPair{ Key = "C", Value = "is" },
                new KeyValPair{ Key = "A", Value = "really" },
                new KeyValPair{ Key = "B", Value = "cool" },
                new KeyValPair{ Key = "B", Value = "!" }
            };

            // Create the query by using our user-defined query operator.
            query = list.ChunkBy(p => p.Key);


            // ChunkBy returns IGrouping objects, therefore a nested
            // foreach loop is required to access the elements in each "chunk".
            foreach (var item in query)
            {
                Console.WriteLine($"Group key = {item.Key}");
                foreach (var inner in item)
                {
                    Console.WriteLine($"\t{inner.Value}");
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

To use the extension method in your project, copy the `MyExtensions` static class to a new or existing source code file and if it is required, add a `using` directive for the namespace where it is located.

See also

- Language Integrated Query (LINQ)

Dynamically specify predicate filters at runtime

1/24/2019 • 2 minutes to read • Edit Online

In some cases, you don't know until run time how many predicates you have to apply to source elements in the `where` clause. One way to dynamically specify multiple predicate filters is to use the Contains method, as shown in the following example. The example is constructed in two ways. First, the project is run by filtering on values that are provided in the program. Then the project is run again by using input provided at run time.

To filter by using the Contains method

1. Open a new console application and name it `PredicateFilters`.
2. Copy the `StudentClass` class from Query a collection of objects and paste it into namespace `PredicateFilters` underneath class `Program`. `StudentClass` provides a list of `Student` objects.
3. Comment out the `Main` method in `StudentClass`.
4. Replace class `Program` with the following code:

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Add the following line to the `Main` method in class `DynamicPredicates`, under the declaration of `ids`.

```
QueryById(ids);
```

6. Run the project.
7. The following output is displayed in a console window:

   Garcia: 114

   O'Donnell: 112

Omelchenko: 111

8. The next step is to run the project again, this time by using input entered at run time instead of array `ids`. Change `QueryByID(ids)` to `QueryByID(args)` in the `Main` method.

9. Run the project with the command line arguments `122 117 120 115`. When the project is run, those values become elements of `args`, the parameter of the `Main` method.

10. The following output is displayed in a console window:

    Adams: 120

    Feng: 117

    Garcia: 115

    Tucker: 122

To filter by using a switch statement

1. You can use a `switch` statement to select among predetermined alternative queries. In the following example, `studentQuery` uses a different `where` clause depending on which grade level, or year, is specified at run time.

2. Copy the following method and paste it into class `DynamicPredicates`.

```
// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}
```

3. In the `Main` method, replace the call to `QueryByID` with the following call, which sends the first element from the `args` array as its argument: `QueryByYear(args[0])`.

4. Run the project with a command line argument of an integer value between 1 and 4.

See also

- Language Integrated Query (LINQ)
- where clause

Perform inner joins

1/24/2019 • 10 minutes to read • Edit Online

In relational database terms, an *inner join* produces a result set in which each element of the first collection appears one time for every matching element in the second collection. If an element in the first collection has no matching elements, it does not appear in the result set. The Join method, which is called by the `join` clause in C#, implements an inner join.

This article shows you how to perform four variations of an inner join:

- A simple inner join that correlates elements from two data sources based on a simple key.
- An inner join that correlates elements from two data sources based on a *composite* key. A composite key, which is a key that consists of more than one value, enables you to correlate elements based on more than one property.
- A *multiple join* in which successive join operations are appended to each other.
- An inner join that is implemented by using a group join.

Example - Simple key join

The following example creates two collections that contain objects of two user-defined types, `Person` and `Pet`. The query uses the `join` clause in C# to match `Person` objects with `Pet` objects whose `Owner` is that `Person`. The `select` clause in C# defines how the resulting objects will look. In this example the resulting objects are anonymous types that consist of the owner's first name and the pet's name.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"\"{ownerAndPet.PetName}\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui
```

Note that the `Person` object whose `LastName` is "Huff" does not appear in the result set because there is no `Pet` object that has `Pet.Owner` equal to that `Person`.

Example - Composite key join

Instead of correlating elements based on just one property, you can use a composite key to compare elements based on multiple properties. To do this, specify the key selector function for each collection to return an anonymous type that consists of the properties you want to compare. If you label the properties, they must have the same label in each key's anonymous type. The properties must also appear in the same order.

The following example uses a list of `Employee` objects and a list of `Student` objects to determine which employees

are also students. Both of these types have a `FirstName` and a `LastName` property of type String. The functions that create the join keys from each list's elements return an anonymous type that consists of the `FirstName` and `LastName` properties of each element. The join operation compares these composite keys for equality and returns pairs of objects from each list where both the first name and the last name match.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
         new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
         new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
         new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Example - Multiple join

Any number of join operations can be appended to each other to perform a multiple join. Each `join` clause in C# correlates a specified data source with the results of the previous join.

The following example creates three collections: a list of `Person` objects, a list of `Cat` objects, and a list of `Dog` objects.

The first `join` clause in C# matches people and cats based on a `Person` object matching `Cat.Owner`. It returns a sequence of anonymous types that contain the `Person` object and `Cat.Name`.

The second `join` clause in C# correlates the anonymous types returned by the first join with `Dog` objects in the supplied list of dogs, based on a composite key that consists of the `Owner` property of type `Person`, and the first letter of the animal's name. It returns a sequence of anonymous types that contain the `Cat.Name` and `Dog.Name` properties from each matching pair. Because this is an inner join, only those objects from the first data source that have a match in the second data source are returned.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };
```

```
    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name,
            with \"{obj.DogName}\".");
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".
```

Example - Inner join by using grouped join

The following example shows you how to implement an inner join by using a group join.

In `query1`, the list of `Person` objects is group-joined to the list of `Pet` objects based on the `Person` matching the `Pet.Owner` property. The group join creates a collection of intermediate groups, where each group consists of a `Person` object and a sequence of matching `Pet` objects.

By adding a second `from` clause to the query, this sequence of sequences is combined (or flattened) into one longer sequence. The type of the elements of the final sequence is specified by the `select` clause. In this example, that type is an anonymous type that consists of the `Person.FirstName` and `Pet.Name` properties for each matching pair.

The result of `query1` is equivalent to the result set that would have been obtained by using the `join` clause without the `into` clause to perform an inner join. The `query2` variable demonstrates this equivalent query.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                 join pet in pets on person equals pet.Owner into gj
```

```
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj
                select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine($"{v.OwnerName} - {v.PetName}"));
    }

    var query2 = from person in people
                 join pet in pets on person equals pet.Owner
                 select new { OwnerName = person.FirstName, PetName = pet.Name };

    Console.WriteLine("\nThe equivalent operation using Join():");
    foreach (var v in query2)
        Console.WriteLine($"{v.OwnerName} - {v.PetName}"));
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

See also

- Join
- GroupJoin
- Perform grouped joins
- Perform left outer joins
- Anonymous types

Perform grouped joins

1/24/2019 • 4 minutes to read • Edit Online

The group join is useful for producing hierarchical data structures. It pairs each element from the first collection with a set of correlated elements from the second collection.

For example, a class or a relational database table named `Student` might contain two fields: `Id` and `Name`. A second class or relational database table named `Course` might contain two fields: `StudentId` and `CourseTitle`. A group join of these two data sources, based on matching `Student.Id` and `Course.StudentId`, would group each `Student` with a collection of `Course` objects (which might be empty).

> **NOTE**
>
> Each element of the first collection appears in the result set of a group join regardless of whether correlated elements are found in the second collection. In the case where no correlated elements are found, the sequence of correlated elements for that element is empty. The result selector therefore has access to every element of the first collection. This differs from the result selector in a non-group join, which cannot access elements from the first collection that have no match in the second collection.

The first example in this article shows you how to perform a group join. The second example shows you how to use a group join to create XML elements.

Example - Group join

The following example performs a group join of objects of type `Person` and `Pet` based on the `Person` matching the `Pet.Owner` property. Unlike a non-group join, which would produce a pair of elements for each match, the group join produces only one resulting object for each element of the first collection, which in this example is a `Person` object. The corresponding elements from the second collection, which in this example are `Pet` objects, are grouped into a collection. Finally, the result selector function creates an anonymous type for each match that consists of `Person.FirstName` and a collection of `Pet` objects.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:
```

Example - Group join to create XML

Group joins are ideal for creating XML by using LINQ to XML. The following example is similar to the previous example except that instead of creating anonymous types, the result selector function creates XML elements that

represent the joined objects.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

See also

- Join
- GroupJoin
- Perform inner joins
- Perform left outer joins
- Anonymous types

Perform left outer joins

1/24/2019 • 2 minutes to read • Edit Online

A left outer join is a join in which each element of the first collection is returned, regardless of whether it has any correlated elements in the second collection. You can use LINQ to perform a left outer join by calling the DefaultIfEmpty method on the results of a group join.

Example

The following example demonstrates how to use the DefaultIfEmpty method on the results of a group join to perform a left outer join.

The first step in producing a left outer join of two collections is to perform an inner join by using a group join. (See Perform inner joins for an explanation of this process.) In this example, the list of `Person` objects is inner-joined to the list of `Pet` objects based on a `Person` object that matches `Pet.Owner`.

The second step is to include each element of the first (left) collection in the result set even if that element has no matches in the right collection. This is accomplished by calling DefaultIfEmpty on each sequence of matching elements from the group join. In this example, DefaultIfEmpty is called on each sequence of matching `Pet` objects. The method returns a collection that contains a single, default value if the sequence of matching `Pet` objects is empty for any `Person` object, thereby ensuring that each `Person` object is represented in the result collection.

> **NOTE**
>
> The default value for a reference type is `null`; therefore, the example checks for a null reference before accessing each element of each `Pet` collection.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:        Daisy
// Terry:         Barley
// Terry:         Boots
// Terry:         Blue Moon
// Charlotte:     Whiskers
// Arlene:
```

See also

- Join
- GroupJoin
- Perform inner joins
- Perform grouped joins
- Anonymous types

Order the results of a join clause

1/24/2019 • 2 minutes to read • Edit Online

This example shows how to order the results of a join operation. Note that the ordering is performed after the join. Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we do not recommend it. Some LINQ providers might not preserve that ordering after the join.

Example

This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a sub-query orders all the matching elements from the products sequence.

```
    class HowToOrderJoins
    {
        #region Data
        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
{
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() {  Name="Grains", ID=004},
    new Category() {  Name="Fruit", ID=005}
};

        // Specify the second data source.
        List<Product> products = new List<Product>()
{
   new Product{Name="Cola",  CategoryID=001},
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
};
        #endregion
        static void Main()
        {
            HowToOrderJoins app = new HowToOrderJoins();
            app.OrderJoin1();

            // Keep console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();

        }
```

```
        void OrderJoin1()
        {
            var groupJoinQuery2 =
                from category in categories
                join prod in products on category.ID equals prod.CategoryID into prodGroup
                orderby category.Name
                select new
                {
                    Category = category.Name,
                    Products = from prod2 in prodGroup
                               orderby prod2.Name
                               select prod2
                };

            foreach (var productGroup in groupJoinQuery2)
            {
                Console.WriteLine(productGroup.Category);
                foreach (var prodItem in productGroup.Products)
                {
                    Console.WriteLine($"  {prodItem.Name,-10} {prodItem.CategoryID}");
                }
            }
        }
        /* Output:
            Beverages
              Cola       1
              Tea        1
            Condiments
              Mustard    2
              Pickles    2
            Fruit
              Melons     5
              Peaches    5
            Grains
            Vegetables
              Bok Choy   3
              Carrots    3
         */
    }
```

See also

- Language Integrated Query (LINQ)
- orderby clause
- join clause

Join by using composite keys

1/24/2019 • 2 minutes to read • Edit Online

This example shows how to perform join operations in which you want to use more than one key to define a match. This is accomplished by using a composite key. You create a composite key as an anonymous type or named typed with the values that you want to compare. If the query variable will be passed across method boundaries, use a named type that overrides Equals and GetHashCode for the key. The names of the properties, and the order in which they occur, must be identical in each key.

Example

The following example demonstrates how to use a composite key to join data from three tables:

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

Type inference on composite keys depends on the names of the properties in the keys, and the order in which they occur. If the properties in the source sequences don't have the same names, you must assign new names in the keys. For example, if the `Orders` table and `OrderDetails` table each used different names for their columns, you could create composite keys by assigning identical names in the anonymous types:

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Composite keys can be also used in a `group` clause.

See also

- Language Integrated Query (LINQ)
- join clause
- group clause

Perform custom join operations

1/24/2019 • 5 minutes to read • Edit Online

This example shows how to perform join operations that aren't possible with the `join` clause. In a query expression, the `join` clause is limited to, and optimized for, equijoins, which are by far the most common type of join operation. When performing an equijoin, you will probably always get the best performance by using the `join` clause.

However, the `join` clause cannot be used in the following cases:

- When the join is predicated on an expression of inequality (a non-equijoin).
- When the join is predicated on more than one expression of equality or inequality.
- When you have to introduce a temporary range variable for the right side (inner) sequence before the join operation.

To perform joins that aren't equijoins, you can use multiple `from` clauses to introduce each data source independently. You then apply a predicate expression in a `where` clause to the range variable for each source. The expression also can take the form of a method call.

> **NOTE**
>
> Don't confuse this kind of custom join operation with the use of multiple `from` clauses to access inner collections. For more information, see join clause.

Example

The first method in the following example shows a simple cross join. Cross joins must be used with caution because they can produce very large result sets. However, they can be useful in some scenarios for creating source sequences against which additional queries are run.

The second method produces a sequence of all the products whose category ID is listed in the category list on the left side. Note the use of the `let` clause and the `Contains` method to create a temporary array. It also is possible to create the array before the query and eliminate the first `from` clause.

```
class CustomJoins
{

    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
```

```
 {
     new Category(){Name="Beverages", ID=001},
     new Category(){ Name="Condiments", ID=002},
     new Category(){ Name="Vegetables", ID=003},
 };

         // Specify the second data source.
         List<Product> products = new List<Product>()
{
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
   new Product{Name="Ice Cream", CategoryID=007},
   new Product{Name="Mackerel", CategoryID=012},
 };
         #endregion

         static void Main()
         {
             CustomJoins app = new CustomJoins();
             app.CrossJoin();
             app.NonEquijoin();

             Console.WriteLine("Press any key to exit.");
             Console.ReadKey();
         }

         void CrossJoin()
         {
             var crossJoinQuery =
                 from c in categories
                 from p in products
                 select new { c.ID, p.Name };

             Console.WriteLine("Cross Join Query:");
             foreach (var v in crossJoinQuery)
             {
                 Console.WriteLine($"{v.ID:-5}{v.Name}");
             }
         }

         void NonEquijoin()
         {
             var nonEquijoinQuery =
                 from p in products
                 let catIds = from c in categories
                              select c.ID
                 where catIds.Contains(p.CategoryID) == true
                 select new { Product = p.Name, CategoryID = p.CategoryID };

             Console.WriteLine("Non-equijoin query:");
             foreach (var v in nonEquijoinQuery)
             {
                 Console.WriteLine($"{v.CategoryID:-5}{v.Product}");
             }
         }
     }
     /* Output:
 Cross Join Query:
 1    Tea
 1    Mustard
 1    Pickles
 1    Carrots
 1    Bok Choy
 1    Peaches
```

```
1    Melons
1    Ice Cream
1    Mackerel
2    Tea
2    Mustard
2    Pickles
2    Carrots
2    Bok Choy
2    Peaches
2    Melons
2    Ice Cream
2    Mackerel
3    Tea
3    Mustard
3    Pickles
3    Carrots
3    Bok Choy
3    Peaches
3    Melons
3    Ice Cream
3    Mackerel
Non-equijoin query:
1    Tea
2    Mustard
2    Pickles
3    Carrots
3    Bok Choy
Press any key to exit.
     */
```

Example

In the following example, the query must join two sequences based on matching keys that, in the case of the inner (right side) sequence, cannot be obtained prior to the join clause itself. If this join were performed with a `join` clause, then the `Split` method would have to be called for each element. The use of multiple `from` clauses enables the query to avoid the overhead of the repeated method call. However, since `join` is optimized, in this particular case it might still be faster than using multiple `from` clauses. The results will vary depending primarily on how expensive the method call is.

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
```

```
                            select Convert.ToInt32(scoreAsText)).
                            ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
                {student.ExamScores.Average()}.");
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

See also

- Language Integrated Query (LINQ)
- join clause
- Order the results of a join clause

Handle null values in query expressions

1/24/2019 • 2 minutes to read • Edit Online

This example shows how to handle possible null values in source collections. An object collection such as an IEnumerable<T> can contain elements whose value is null. If a source collection is null or contains an element whose value is null, and your query does not handle null values, a NullReferenceException will be thrown when you execute the query.

Example

You can code defensively to avoid a null reference exception as shown in the following example:

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

In the previous example, the `where` clause filters out all null elements in the categories sequence. This technique is independent of the null check in the join clause. The conditional expression with null in this example works because `Products.CategoryID` is of type `int?` which is shorthand for `Nullable<int>`.

Example

In a join clause, if only one of the comparison keys is a nullable value type, you can cast the other to a nullable type in the query expression. In the following example, assume that `EmployeeID` is a column that contains values of type `int?`:

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

See also

- Nullable<T>
- Language Integrated Query (LINQ)
- Nullable types

Handle exceptions in query expressions

1/24/2019 • 2 minutes to read • Edit Online

It's possible to call any method in the context of a query expression. However, we recommend that you avoid calling any method in a query expression that can create a side effect such as modifying the contents of the data source or throwing an exception. This example shows how to avoid raising exceptions when you call methods in a query expression without violating the general .NET guidelines on exception handling. Those guidelines state that it's acceptable to catch a specific exception when you understand why it's thrown in a given context. For more information, see Best Practices for Exceptions.

The final example shows how to handle those cases when you must throw an exception during execution of a query.

Example

The following example shows how to move exception handling code outside a query expression. This is only possible when the method does not depend on any variables local to the query.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

Example

In some cases, the best response to an exception that is thrown from within a query might be to stop the query execution immediately. The following example shows how to handle exceptions that might be thrown from inside a query body. Assume that `SomeMethodThatMightThrow` can potentially cause an exception that requires the query execution to stop.

Note that the `try` block encloses the `foreach` loop, and not the query itself. This is because the `foreach` loop is the point at which the query is actually executed. For more information, see Introduction to LINQ queries.

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

See also

- Language Integrated Query (LINQ)

Contents

Namespaces (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Namespaces are heavily used in C# programming in two ways. First, the .NET Framework uses namespaces to organize its many classes, as follows:

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

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name is not required, as in the following example:

```
using System;
```

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

For more information, see the using Directive.

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# identifier name.

Namespaces Overview

Namespaces have the following properties:

- They organize large code projects.
- They are delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET System namespace.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Using Namespaces
- How to: Use the Global Namespace Alias
- How to: Use the My Namespace
- C# Programming Guide
- Identifier names
- Namespace Keywords
- using Directive
- :: Operator
- . Operator

Using Namespaces (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

Namespaces are heavily used within C# programs in two ways. Firstly, the .NET Framework classes use namespaces to organize its many classes. Secondly, declaring your own namespaces can help control the scope of class and method names in larger programming projects.

Accessing Namespaces

Most C# applications begin with a section of `using` directives. This section lists the namespaces that the application will be using frequently, and saves the programmer from specifying a fully qualified name every time that a method that is contained within is used.

For example, by including the line:

```
using System;
```

At the start of a program, the programmer can use the code:

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

Instead of:

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

Namespace Aliases

The using Directive can also be used to create an alias for a namespace. For example, if you are using a previously written namespace that contains nested namespaces, you might want to declare an alias to provide a shorthand way of referencing one in particular, as in the following example:

```
using Co = Company.Proj.Nested;  // define an alias to represent a namespace
```

Using Namespaces to control scope

The `namespace` keyword is used to declare a scope. The ability to create scopes within your project helps organize code and lets you create globally-unique types. In the following example, a class titled `SampleClass` is defined in two namespaces, one nested inside the other. The . Operator is used to differentiate which method gets called.

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                  "SampleMethod inside NestedNamespace");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Displays "SampleMethod inside SampleNamespace."
            SampleClass outer = new SampleClass();
            outer.SampleMethod();

            // Displays "SampleMethod inside SampleNamespace."
            SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
            outer2.SampleMethod();

            // Displays "SampleMethod inside NestedNamespace."
            NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
            inner.SampleMethod();
        }
    }
}
```

Fully Qualified Names

Namespaces and types have unique titles described by fully qualified names that indicate a logical hierarchy. For example, the statement `A.B` implies that `A` is the name of the namespace or type, and `B` is nested inside it.

In the following example, there are nested classes and namespaces. The fully qualified name is indicated as a comment following each entity.

```
namespace N1     // N1
{
    class C1      // N1.C1
    {
        class C2   // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2   // N1.N2.C2
        {
        }
    }
}
```

In the previous code segment:

- The namespace `N1` is a member of the global namespace. Its fully qualified name is `N1`.
- The namespace `N2` is a member of `N1`. Its fully qualified name is `N1.N2`.
- The class `C1` is a member of `N1`. Its fully qualified name is `N1.C1`.
- The class name `C2` is used two times in this code. However, the fully qualified names are unique. The first instance of `C2` is declared inside `C1`; therefore, its fully qualified name is: `N1.C1.C2`. The second instance of `C2` is declared inside a namespace `N2`; therefore, its fully qualified name is `N1.N2.C2`.

Using the previous code segment, you can add a new class member, `C3`, to the namespace `N1.N2` as follows:

```
namespace N1.N2
{
    class C3   // N1.N2.C3
    {
    }
}
```

In general, use `::` to reference a namespace alias or `global::` to reference the global namespace and `.` to qualify types or members.

It is an error to use `::` with an alias that references a type instead of a namespace. For example:

```
using Alias = System.Console;
```

```
class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}
```

Remember that the word `global` is not a predefined alias; therefore, `global.X` does not have any special meaning. It acquires a special meaning only when it is used with `::`.

Compiler warning CS0440 is generated if you define an alias named global because `global::` always references the global namespace and not an alias. For example, the following line generates the warning:

```
using global = System.Collections;   // Warning
```

Using `::` with aliases is a good idea and protects against the unexpected introduction of additional types. For example, consider this example:

```
using Alias = System;
```

```
namespace Library
{
    public class C : Alias.Exception { }
}
```

This works, but if a type named `Alias` were to subsequently be introduced, `Alias.` would bind to that type instead. Using `Alias::Exception` insures that `Alias` is treated as a namespace alias and not mistaken for a type.

See the topic How to: Use the Global Namespace Alias for more information regarding the `global` alias.

See also

- C# Programming Guide
- Namespaces
- Namespace Keywords
- . Operator
- :: Operator
- extern

How to: Use the Global Namespace Alias (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The ability to access a member in the global namespace is useful when the member might be hidden by another entity of the same name.

For example, in the following code, `Console` resolves to `TestApp.Console` instead of to the `Console` type in the System namespace.

```
using System;
```

```
class TestApp
{
    // Define a new class called 'System' to cause problems.
    public class System { }

    // Define a constant called 'Console' to cause more problems.
    const int Console = 7;
    const int number = 66;

    static void Main()
    {
        // The following line causes an error. It accesses TestApp.Console,
        // which is a constant.
        //Console.WriteLine(number);
    }
}
```

Using `System.Console` still results in an error because the `System` namespace is hidden by the class `TestApp.System`:

```
// The following line causes an error. It accesses TestApp.System,
// which does not have a Console.WriteLine method.
System.Console.WriteLine(number);
```

However, you can work around this error by using `global::System.Console`, like this:

```
// OK
global::System.Console.WriteLine(number);
```

When the left identifier is `global`, the search for the right identifier starts at the global namespace. For example, the following declaration is referencing `TestApp` as a member of the global space.

```
class TestClass : global::TestApp
```

Obviously, creating your own namespaces called `System` is not recommended, and it is unlikely you will encounter any code in which this has happened. However, in larger projects, it is a very real possibility that namespace duplication may occur in one form or another. In these situations, the global namespace qualifier is your guarantee that you can specify the root namespace.

Example

In this example, the namespace `System` is used to include the class `TestClass` therefore, `global::System.Console` must be used to reference the `System.Console` class, which is hidden by the `System` namespace. Also, the alias `colAlias` is used to refer to the namespace `System.Collections`; therefore, the instance of a System.Collections.Hashtable was created using this alias instead of the namespace.

```
using colAlias = System.Collections;
namespace System
{
    class TestClass
    {
        static void Main()
        {
            // Searching the alias:
            colAlias::Hashtable test = new colAlias::Hashtable();

            // Add items to the table.
            test.Add("A", "1");
            test.Add("B", "2");
            test.Add("C", "3");

            foreach (string name in test.Keys)
            {
                // Searching the global namespace:
                global::System.Console.WriteLine(name + " " + test[name]);
            }
        }
    }
}
```

A 1 B 2 C 3

See also

- C# Programming Guide
- Namespaces
- . Operator
- :: Operator
- extern

How to: Use the My Namespace (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The Microsoft.VisualBasic.MyServices namespace (`My` in Visual Basic) provides easy and intuitive access to a number of .NET Framework classes, enabling you to write code that interacts with the computer, application, settings, resources, and so on. Although originally designed for use with Visual Basic, the `MyServices` namespace can be used in C# applications.

For more information about using the `MyServices` namespace from Visual Basic, see Development with My.

Adding a Reference

Before you can use the `MyServices` classes in your solution, you must add a reference to the Visual Basic library.

To add a reference to the Visual Basic library

1. In **Solution Explorer**, right-click the **References** node, and select **Add Reference**.
2. When the **References** dialog box appears, scroll down the list, and select Microsoft.VisualBasic.dll.

   You might also want to include the following line in the `using` section at the start of your program.

```
using Microsoft.VisualBasic.Devices;
```

Example

This example calls various static methods contained in the `MyServices` namespace. For this code to compile, a reference to Microsoft.VisualBasic.DLL must be added to the project.

```
using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to network.");
        }
    }
}
```

Not all the classes in the `MyServices` namespace can be called from a C# application: for example, the FileSystemProxy class is not compatible. In this particular case, the static methods that are part of FileSystem, which are also contained in VisualBasic.dll, can be used instead. For example, here is how to use one such method to duplicate a directory:

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

See also

- C# Programming Guide
- Namespaces
- Using Namespaces

Contents

Nullable types (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Nullable types are instances of the System.Nullable<T> struct. Nullable types can represent all the values of an underlying type `T`, and an additional null value. The underlying type `T` can be any non-nullable value type. `T` cannot be a reference type.

For example, you can assign `null` or any integer value from Int32.MinValue to Int32.MaxValue to a `Nullable<int>` and true, false, or `null` to a `Nullable<bool>`.

You use a nullable type when you need to represent the undefined value of an underlying type. A Boolean variable can have only two values: true and false. There is no "undefined" value. In many programming applications, most notably database interactions, a variable value can be undefined or missing. For example, a field in a database may contain the values true or false, or it may contain no value at all. You use a `Nullable<bool>` type in that case.

Nullable types have the following characteristics:

- Nullable types represent value-type variables that can be assigned the `null` value. You cannot create a nullable type based on a reference type. (Reference types already support the `null` value.)
- The syntax `T?` is shorthand for `Nullable<T>`. The two forms are interchangeable.
- Assign a value to a nullable type just as you would for an underlying value type: `int? x = 10;` or `double? d = 4.108;`. You also can assign the `null` value: `int? x = null;`.
- Use the Nullable<T>.HasValue and Nullable<T>.Value readonly properties to test for null and retrieve the value, as shown in the following example: `if (x.HasValue) y = x.Value;`
  - The HasValue property returns `true` if the variable contains a value, or `false` if it's `null`.
  - The Value property returns a value if HasValue returns `true`. Otherwise, an InvalidOperationException is thrown.
- You can also use the `==` and `!=` operators with a nullable type, as shown in the following example: `if (x != null) y = x.Value;`. If `a` and `b` are both null, `a == b` evaluates to `true`.
- Beginning with C# 7.0, you can use pattern matching to both examine and get a value of a nullable type: `if (x is int valueOfX) y = valueOfX;`.
- The default value of `T?` is an instance whose HasValue property returns `false`.
- Use the GetValueOrDefault() method to return either the assigned value, or the default value of the underlying value type if the value of the nullable type is `null`.
- Use the GetValueOrDefault(T) method to return either the assigned value, or the provided default value if the value of the nullable type is `null`.
- Use the null-coalescing operator, `??`, to assign a value to an underlying type based on a value of the nullable type: `int? x = null; int y = x ?? -1;`. In the example, since `x` is null, the result value of `y` is `-1`.
- If a user-defined conversion is defined between two data types, the same conversion can also be used with the nullable versions of these data types.
- Nested nullable types are not allowed. The following line doesn't compile: `Nullable<Nullable<int>> n;`

For more information, see the Using nullable types and How to: Identify a nullable type topics.

See also

- System.Nullable<T>
- System.Nullable
- ?? Operator
- C# Programming Guide
- C# Guide
- C# Reference
- Nullable Value Types (Visual Basic)

Using nullable types (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

Nullable types are types that represent all the values of an underlying value type `T`, and an additional null value. For more information, see the Nullable types topic.

You can refer to a nullable type in any of the following forms: `Nullable<T>` or `T?`. These two forms are interchangeable.

Declaration and assignment

As a value type can be implicitly converted to the corresponding nullable type, you assign a value to a nullable type as you would for its underlying value type. You also can assign the `null` value. For example:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// Array of nullable type:
int?[] arr = new int?[10];
```

Examination of a nullable type value

Use the following readonly properties to examine an instance of a nullable type for null and retrieve a value of an underlying type:

- Nullable<T>.HasValue indicates whether an instance of a nullable type has a value of its underlying type.
- Nullable<T>.Value gets the value of an underlying type if HasValue is `true`. If HasValue is `false`, the Value property throws an InvalidOperationException.

The code in the following example uses the `HasValue` property to test whether the variable contains a value before displaying it:

```
int? x = 10;
if (x.HasValue)
{
    Console.WriteLine($"x is {x.Value}");
}
else
{
    Console.WriteLine("x does not have a value");
}
```

You also can compare a nullable type variable with `null` instead of using the `HasValue` property, as the following example shows:

```
int? y = 7;
if (y != null)
{
    Console.WriteLine($"y is {y.Value}");
}
else
{
    Console.WriteLine("y does not have a value");
}
```

Beginning with C# 7.0, you can use pattern matching to both examine and get a value of a nullable type:

```
int? z = 42;
if (z is int valueOfZ)
{
    Console.WriteLine($"z is {valueOfZ}");
}
else
{
    Console.WriteLine("z does not have a value");
}
```

Conversion from a nullable type to an underlying type

If you need to assign a nullable type value to a non-nullable type, use the null-coalescing operator `??` to specify the value to be assigned if a nullable type value is null (you also can use the Nullable<T>.GetValueOrDefault(T) method to do that):

```
int? c = null;

// d = c, if c is not null, d = -1 if c is null.
int d = c ?? -1;
Console.WriteLine($"d is {d}");
```

Use the Nullable<T>.GetValueOrDefault() method if the value to be used when a nullable type value is null should be the default value of the underlying value type.

You can explicitly cast a nullable type to a non-nullable type. For example:

```
int? n = null;

//int m1 = n;    // Doesn't compile.
int n2 = (int)n; // Compiles, but throws an exception if n is null.
```

At run time, if the value of a nullable type is null, the explicit cast throws an InvalidOperationException.

A non-nullable value type is implicitly converted to the corresponding nullable type.

Operators

The predefined unary and binary operators and any user-defined operators that exist for value types may also be used by nullable types. These operators produce a null value if one or both operands are null; otherwise, the operator uses the contained values to calculate the result. For example:

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11.
a = a * c;  // a is 110.
a = a + b;  // a is null.
```

For the relational operators (`<` , `>` , `<=` , `>=`), if one or both operands are null, the result is `false` . Do not assume that because a particular comparison (for example, `<=`) returns `false` , the opposite comparison (`>`) returns `true` . The following example shows that 10 is

- neither greater than or equal to null,
- nor less than null.

```
int? num1 = 10;
int? num2 = null;
if (num1 >= num2)
{
    Console.WriteLine("num1 is greater than or equal to num2");
}
else
{
    Console.WriteLine("num1 >= num2 is false (but num1 < num2 also is false)");
}

if (num1 < num2)
{
    Console.WriteLine("num1 is less than num2");
}
else
{
    Console.WriteLine("num1 < num2 is false (but num1 >= num2 also is false)");
}

if (num1 != num2)
{
    Console.WriteLine("num1 != num2 is true!");
}

num1 = null;
if (num1 == num2)
{
    Console.WriteLine("num1 == num2 is true if the value of each is null");
}
// Output:
// num1 >= num2 is false (but num1 < num2 also is false)
// num1 < num2 is false (but num1 >= num2 also is false)
// num1 != num2 is true!
// num1 == num2 is true if the value of each is null
```

The above example also shows that an equality comparison of two nullable types that are both null evaluates to `true` .

Boxing and unboxing

A nullable value type is boxed by the following rules:

- If HasValue returns `false` , the null reference is produced.
- If HasValue returns `true` , a value of the underlying value type `T` is boxed, not the instance of Nullable<T>.

You can unbox the boxed value type to the corresponding nullable type, as the following example shows:

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

The bool? type

The `bool?` nullable type can contain three different values: true, false, and null. The `bool?` type is like the Boolean variable type that is used in SQL. To ensure that the results produced by the `&` and `|` operators are consistent with the three-valued Boolean type in SQL, the following predefined operators are provided:

- `bool? operator &(bool? x, bool? y)`
- `bool? operator |(bool? x, bool? y)`

The semantics of these operators is defined by the following table:

X	Y	X&Y	X|Y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Note that these two operators don't follow the rules described in the Operators section: the result of an operator evaluation can be non-null even if one of the operands is null.

See also

- Nullable types
- C# Programming Guide
- What exactly does 'lifted' mean?

How to: Identify a nullable type (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following example shows how to determine whether a System.Type instance represents a closed generic nullable type, that is, the System.Nullable<T> type with a specified type parameter `T`:

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non nullable")} type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable type
// int is non nullable type
```

As the example shows, you use the typeof operator to create a System.Type object.

If you want to determine whether an instance is of a nullable type, don't use the Object.GetType method to get a Type instance to be tested with the preceding code. When you call the Object.GetType method on an instance of a nullable type, the instance is boxed to Object. As boxing of a non-null instance of a nullable type is equivalent to boxing of a value of the underlying type, GetType returns a Type object that represents the underlying type of a nullable type:

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

Don't use the is operator to determine whether an instance is of a nullable type. As the following example shows, you cannot distinguish types of instances of a nullable type and its underlying type with using the `is` operator:

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

You can use the code presented in the following example to determine whether an instance is of a nullable type:

```
int? a = 14;
int b = 17;
if (IsOfNullableType(a) && !IsOfNullableType(b))
{
    Console.WriteLine("int? a is of a nullable type, while int b -- not");
}

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}

// Output:
// int? a is of a nullable type, while int b -- not
```

See also

- Nullable types
- Using nullable types
- GetUnderlyingType

Contents

Unsafe Code and Pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To maintain type safety and security, C# does not support pointer arithmetic, by default. However, by using the unsafe keyword, you can define an unsafe context in which pointers can be used. For more information about pointers, see the topic Pointer types.

> **NOTE**
> In the common language runtime (CLR), unsafe code is referred to as unverifiable code. Unsafe code in C# is not necessarily dangerous; it is just code whose safety cannot be verified by the CLR. The CLR will therefore only execute unsafe code if it is in a fully trusted assembly. If you use unsafe code, it is your responsibility to ensure that your code does not introduce security risks or pointer errors.

Unsafe Code Overview

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code may increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- In order for C# to compile unsafe code, the application must be compiled with /unsafe.

Related Sections

For more information, see:

- Pointer types
- Fixed Size Buffers
- How to: Use Pointers to Copy an Array of Bytes
- unsafe

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Fixed Size Buffers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C#, you can use the fixed statement to create a buffer with a fixed size array in a data structure. Fixed size buffers are useful when you write methods that interop with data sources from other languages or platforms. The fixed array can take any attributes or modifiers that are allowed for regular struct members. The only restriction is that the array type must be `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`.

```
private fixed char name[30];
```

Remarks

In safe code, a C# struct that contains an array does not contain the array elements. Instead, the struct contains a reference to the elements. You can embed an array of fixed size in a struct when it is used in an unsafe code block.

The following `struct` is 8 bytes in size. The `pathName` array is a reference:

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

A `struct` can contain an embedded array in unsafe code. In the following example, the `fixedBuffer` array has a fixed size. You use a `fixed` statement to establish a pointer to the first element. You access the elements of the array through this pointer. The `fixed` statement pins the `fixedBuffer` instance field to a specific location in memory.

```
internal unsafe struct MyBuffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class MyClass
{
    public MyBuffer myBuffer = default;
}

private static void AccessEmbeddedArray()
{
    MyClass myC = new MyClass();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = myC.myBuffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = myC.myBuffer.fixedBuffer[0];
        Console.WriteLine(c);
        // modify through the index:
        myC.myBuffer.fixedBuffer[0] = 'B';
        Console.WriteLine(myC.myBuffer.fixedBuffer[0]);
    }

}
```

The size of the 128 element `char` array is 256 bytes. Fixed size char buffers always take two bytes per character, regardless of the encoding. This is true even when char buffers are marshaled to API methods or structs with `CharSet = CharSet.Auto` or `CharSet = CharSet.Ansi`. For more information, see CharSet.

The preceding example demonstrates accessing `fixed` fields without pinning, which is available starting with C# 7.3.

Another common fixed-size array is the bool array. The elements in a `bool` array are always one byte in size. `bool` arrays are not appropriate for creating bit arrays or buffers.

> **NOTE**
>
> Except for memory created by using stackalloc, the C# compiler and the common language runtime (CLR) do not perform any security buffer overrun checks. As with all unsafe code, use caution.

Unsafe buffers differ from regular arrays in the following ways:

- You can only use unsafe buffers in an unsafe context.
- Unsafe buffers are always vectors, or one-dimensional arrays.
- The declaration of the array should include a count, such as `char id[8]`. You cannot use `char id[]`.
- Unsafe buffers can only be instance fields of structs in an unsafe context.

See also

- C# Programming Guide
- Unsafe Code and Pointers
- fixed Statement
- Interoperability

Pointer types (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

In an unsafe context, a type may be a pointer type, a value type, or a reference type. A pointer type declaration takes one of the following forms:

```
type* identifier;
void* identifier; //allowed but not recommended
```

The type specified before the `*` in a pointer type is called the **referent type**. Any of the following types may be a referent type:

- Any integral type: sbyte, byte, short, ushort, int, uint, long, ulong.
- Any floating-point type: float, double.
- char.
- bool.
- decimal.
- Any enum type.
- Any pointer type. This allows expressions such as `void**`.
- Any user-defined struct type that contains fields of unmanaged types only.

Pointer types do not inherit from object and no conversions exist between pointer types and `object`. Also, boxing and unboxing do not support pointers. However, you can convert between different pointer types and between pointer types and integral types.

When you declare multiple pointers in the same declaration, the asterisk (*) is written together with the underlying type only; it is not used as a prefix to each pointer name. For example:

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

A pointer cannot point to a reference or to a struct that contains references, because an object reference can be garbage collected even if a pointer is pointing to it. The garbage collector does not keep track of whether an object is being pointed to by any pointer types.

The value of the pointer variable of type `myType*` is the address of a variable of type `myType`. The following are examples of pointer type declarations:

EXAMPLE	DESCRIPTION
`int* p`	`p` is a pointer to an integer.
`int** p`	`p` is a pointer to a pointer to an integer.
`int*[] p`	`p` is a single-dimensional array of pointers to integers.
`char* p`	`p` is a pointer to a char.

EXAMPLE	DESCRIPTION
`void* p`	`p` is a pointer to an unknown type.

The pointer indirection operator * can be used to access the contents at the location pointed to by the pointer variable. For example, consider the following declaration:

```
int* myVariable;
```

The expression `*myVariable` denotes the `int` variable found at the address contained in `myVariable`.

There are several examples of pointers in the topics fixed Statement and Pointer Conversions. The following example uses the `unsafe` keyword and the `fixed` statement, and shows how to increment an interior pointer. You can paste this code into the Main function of a console application to run it. These examples must be compiled with the -unsafe compiler option set.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

You cannot apply the indirection operator to a pointer of type `void*`. However, you can use a cast to convert a void pointer to any other pointer type, and vice versa.

A pointer can be `null`. Applying the indirection operator to a null pointer causes an implementation-defined behavior.

Passing pointers between methods can cause undefined behavior. Consider a method that returns a pointer to a local variable through an `in`, `out`, or `ref` parameter or as the function result. If the pointer was set in a fixed block, the variable to which it points may no longer be fixed.

The following table lists the operators and statements that can operate on pointers in an unsafe context:

OPERATOR/STATEMENT	USE
*	Performs pointer indirection.
->	Accesses a member of a struct through a pointer.
[]	Indexes a pointer.
`&`	Obtains the address of a variable.
++ and --	Increments and decrements pointers.
+ and -	Performs pointer arithmetic.
==, !=, <, >, <=, and >=	Compares pointers.
`stackalloc`	Allocates memory on the stack.
`fixed` statement	Temporarily fixes a variable so that its address may be found.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Unsafe Code and Pointers
- Pointer Conversions
- Pointer Expressions
- Types
- unsafe
- fixed Statement
- stackalloc
- Boxing and Unboxing

Pointer Conversions (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following table shows the predefined implicit pointer conversions. Implicit conversions might occur in many situations, including method invoking and assignment statements.

Implicit pointer conversions

FROM	TO
Any pointer type	void*
null	Any pointer type

Explicit pointer conversion is used to perform conversions, for which there is no implicit conversion, by using a cast expression. The following table shows these conversions.

Explicit pointer conversions

FROM	TO
Any pointer type	Any other pointer type
sbyte, byte, short, ushort, int, uint, long, or ulong	Any pointer type
Any pointer type	sbyte, byte, short, ushort, int, uint, long, or ulong

Example

In the following example, a pointer to `int` is converted to a pointer to `byte`. Notice that the pointer points to the lowest addressed byte of the variable. When you successively increment the result, up to the size of `int` (4 bytes), you can display the remaining bytes of the variable.

```
// compile with: -unsafe
```

```
class ClassConvert
{
    static void Main()
    {
        int number = 1024;

        unsafe
        {
            // Convert to byte:
            byte* p = (byte*)&number;

            System.Console.Write("The 4 bytes of the integer:");

            // Display the 4 bytes of the int variable:
            for (int i = 0 ; i < sizeof(int) ; ++i)
            {
                System.Console.Write(" {0:X2}", *p);
                // Increment the pointer:
                p++;
            }
            System.Console.WriteLine();
            System.Console.WriteLine("The value of the integer: {0}", number);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
```

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

Pointer Expressions (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In this section, the following pointer expressions are discussed:

Obtaining the Value of a Variable

Obtaining the Address of a Variable

How to: Access a Member with a Pointer

How to: Access an Array Element with a Pointer

Manipulating Pointers

See also

- C# Programming Guide
- Pointer Conversions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: Obtain the Value of a Pointer Variable (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Use the pointer indirection operator to obtain the variable at the location pointed to by a pointer. The expression takes the following form, where `p` is a pointer type:

```
*p;
```

You cannot use the unary indirection operator on an expression of any type other than the pointer type. Also, you cannot apply it to a void pointer.

When you apply the indirection operator to a null pointer, the result depends on the implementation.

Example

In the following example, a variable of the type `char` is accessed by using pointers of different types. Note that the address of `theChar` will vary from run to run, because the physical address allocated to a variable can change.

```
// compile with: -unsafe
```

```
unsafe class TestClass
{
    static void Main()
    {
        char theChar = 'Z';
        char* pChar = &theChar;
        void* pVoid = pChar;
        int* pInt = (int*)pVoid;

        System.Console.WriteLine("Value of theChar = {0}", theChar);
        System.Console.WriteLine("Address of theChar = {0:X2}",(int)pChar);
        System.Console.WriteLine("Value of pChar = {0}", *pChar);
        System.Console.WriteLine("Value of pInt = {0}", *pInt);
    }
}
```

Value of theChar = Z Address of theChar = 12F718 Value of pChar = Z Value of pInt = 90

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: obtain the address of a variable (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To obtain the address of a unary expression, which evaluates to a fixed variable, use the address-of operator `&`:

```
int number;
int* p = &number; //address-of operator &
```

The address-of operator can only be applied to a variable. If the variable is a moveable variable, you can use the fixed statement to temporarily fix the variable before obtaining its address.

It's your responsibility to ensure that the variable is initialized. The compiler doesn't issue an error message if the variable is not initialized.

You can't get the address of a constant or a value.

Example

In this example, a pointer to `int`, `p`, is declared and assigned the address of an integer variable, `number`. The variable `number` is initialized as a result of the assignment to `*p`. If you comment out this assignment statement, the initialization of the variable `number` is removed, but no compile-time error is issued.

> **NOTE**
> Compile this example with the `-unsafe` compiler option.

```
class AddressOfOperator
{
    static void Main()
    {
        int number;

        unsafe
        {
            // Assign the address of number to a pointer:
            int* p = &number;

            // Commenting the following statement will remove the
            // initialization of number.
            *p = 0xffff;

            // Print the value of *p:
            System.Console.WriteLine("Value at the location pointed to by p: {0:X}", *p);

            // Print the address stored in p:
            System.Console.WriteLine("The address stored in p: {0}", (int)p);
        }

        // Print the value of the variable number:
        System.Console.WriteLine("Value of the variable number: {0:X}", number);

        System.Console.ReadKey();
    }
}
/* Output:
        Value at the location pointed to by p: FFFF
        The address stored in p: 2420904
        Value of the variable number: FFFF
*/
```

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: access a member with a pointer (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To access a member of a struct that is declared in an unsafe context, you can use the member access operator as shown in the following example in which `p` is a pointer to a struct that contains a member `x`.

```
CoOrds* p = &home;
p -> x = 25; //member access operator ->
```

Example

In this example, a struct, `CoOrds`, that contains the two coordinates `x` and `y` is declared and instantiated. By using the member access operator `->` and a pointer to the instance `home`, `x` and `y` are assigned values.

> **NOTE**
> Notice that the expression `p->x` is equivalent to the expression `(*p).x`, and you can obtain the same result by using either of the two expressions.

```
// compile with: -unsafe
```

```
struct CoOrds
{
    public int x;
    public int y;
}

class AccessMembers
{
    static void Main()
    {
        CoOrds home;

        unsafe
        {
            CoOrds* p = &home;
            p->x = 25;
            p->y = 12;

            System.Console.WriteLine("The coordinates are: x={0}, y={1}", p->x, p->y );
        }
    }
}
```

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types

- Types
- unsafe
- fixed Statement
- stackalloc

How to: access an array element with a pointer (C# Programming Guide)

1/15/2019 • 2 minutes to read • Edit Online

In an unsafe context, you can access an element in memory by using pointer element access, as shown in the following example:

```
char* charPointer = stackalloc char[123];
for (int i = 65; i < 123; i++)
{
    charPointer[i] = (char)i; //access array elements
}
```

The expression in square brackets must be implicitly convertible to `int`, `uint`, `long`, or `ulong`. The operation `p[e]` is equivalent to `*(p+e)`. Like C and C++, the pointer element access does not check for out-of-bounds errors.

Example

In this example, 123 memory locations are allocated to a character array, `charPointer`. The array is used to display the lowercase letters and the uppercase letters in two for loops.

Notice that the expression `charPointer[i]` is equivalent to the expression `*(charPointer + i)`, and you can obtain the same result by using either of the two expressions.

```
// compile with: -unsafe
```

```
class Pointers
{
    unsafe static void Main()
    {
        char* charPointer = stackalloc char[123];

        for (int i = 65; i < 123; i++)
        {
            charPointer[i] = (char)i;
        }

        // Print uppercase letters:
        System.Console.WriteLine("Uppercase letters:");
        for (int i = 65; i < 91; i++)
        {
            System.Console.Write(charPointer[i]);
        }
        System.Console.WriteLine();

        // Print lowercase letters:
        System.Console.WriteLine("Lowercase letters:");
        for (int i = 97; i < 123; i++)
        {
            System.Console.Write(charPointer[i]);
        }
    }
}
```

Uppercase letters:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Lowercase letters:
abcdefghijklmnopqrstuvwxyz

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

Manipulating Pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This section includes the following pointer operations:

Increment and Decrement

Arithmetic Operations

Comparison

See also

- C# Programming Guide
- Pointer Expressions
- C# Operators
- Pointer types
- /unsafe (C# Compiler Options)

How to: increment and decrement pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Use the increment and the decrement operators, `++` and `--`, to change the pointer location by `sizeof(pointer-type)` for a pointer of the type `pointer-type*`. The increment and decrement expressions take the following form:

```
++p;
p++;
--p;
p--;
```

The increment and decrement operators can be applied to pointers of any type except the type `void*`.

The effect of applying the increment operator to a pointer of the type `pointer-type*` is to add `sizeof(pointer-type)` to the address that is contained in the pointer variable.

The effect of applying the decrement operator to a pointer of the type `pointer-type*` is to subtract `sizeof(pointer-type)` from the address that is contained in the pointer variable.

No exceptions are generated when the operation overflows the domain of the pointer, and the result depends on the implementation.

Example

In this example, you step through an array by incrementing the pointer by the size of `int`. With each step, you display the address and the content of the array element.

```
// compile with: -unsafe
```

```
class IncrDecr
{
    unsafe static void Main()
    {
        int[] numbers = {0,1,2,3,4};

        // Assign the array address to the pointer:
        fixed (int* p1 = numbers)
        {
            // Step through the array elements:
            for(int* p2=p1; p2<p1+numbers.Length; p2++)
            {
                System.Console.WriteLine("Value:{0} @ Address:{1}", *p2, (long)p2);
            }
        }
    }
}
```

Value:0 @ Address:12860272 Value:1 @ Address:12860276 Value:2 @ Address:12860280 Value:3 @ Address:12860284 Value:4 @ Address:12860288

See also

- C# Programming Guide
- Pointer Expressions
- C# Operators
- Manipulating Pointers
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc
- sizeof

Arithmetic operations on pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This topic discusses using the arithmetic operators `+` and `-` to manipulate pointers.

> **NOTE**
> You cannot perform any arithmetic operations on void pointers.

Adding and subtracting numeric values to or from pointers

You can add a value `n` of type int, uint, long, or ulong to a pointer. If `p` is a pointer of the type `pointer-type*`, the result `p+n` is the pointer resulting from adding `n * sizeof(pointer-type)` to the address of `p`. Similarly, `p-n` is the pointer resulting from subtracting `n * sizeof(pointer-type)` from the address of `p`.

Subtracting pointers

You can also subtract pointers of the same type. The result is always of the type `long`. For example, if `p1` and `p2` are pointers of the type `pointer-type*`, then the expression `p1-p2` results in:

```
((long)p1 - (long)p2)/sizeof(pointer-type)
```

No exceptions are generated when the arithmetic operation overflows the domain of the pointer, and the result depends on the implementation.

Example

```
// compile with: -unsafe
```

```
class PointerArithmetic
{
    unsafe static void Main()
    {
        int* memory = stackalloc int[30];
        long difference;
        int* p1 = &memory[4];
        int* p2 = &memory[10];

        difference = p2 - p1;

        System.Console.WriteLine("The difference is: {0}", difference);
    }
}
// Output:  The difference is: 6
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Unsafe Code and Pointers
- Pointer Expressions
- C# Operators
- Manipulating Pointers
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

Pointer Comparison (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can apply the following operators to compare pointers of any type:

== != < > <= >=

The comparison operators compare the addresses of the two operands as if they are unsigned integers.

Example

```
// compile with: -unsafe
```

```
class CompareOperators
{
    unsafe static void Main()
    {
        int x = 234;
        int y = 236;
        int* p1 = &x;
        int* p2 = &y;

        System.Console.WriteLine(p1 < p2);
        System.Console.WriteLine(p2 < p1);
    }
}
```

Sample Output

```
True
```

```
False
```

See also

- C# Programming Guide
- Pointer Expressions
- C# Operators
- Manipulating Pointers
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: Use Pointers to Copy an Array of Bytes (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The following example uses pointers to copy bytes from one array to another.

This example uses the unsafe keyword, which enables you to use pointers in the `Copy` method. The fixed statement is used to declare pointers to the source and destination arrays. The `fixed` statement *pins* the location of the source and destination arrays in memory so that they will not be moved by garbage collection. The memory blocks for the arrays are unpinned when the `fixed` block is completed. Because the `Copy` method in this example uses the `unsafe` keyword, it must be compiled with the -unsafe compiler option.

This example accesses the elements of both arrays using indices rather than a second unmanaged pointer. The declaration of the `pSource` and `pTarget` pointers pins the arrays. This feature is available starting with C# 7.3.

Example

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
```

```
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

See also

- C# Programming Guide
- Unsafe Code and Pointers
- -unsafe (C# Compiler Options)
- Garbage Collection

Contents

XML Documentation Comments (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

In Visual C# you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

When you compile with the /doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as DocFX or Sandcastle.

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List&lt;T&gt;"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

> **NOTE**
> The XML documentation comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

In This Section

- Recommended Tags for Documentation Comments
- Processing the XML File
- Delimiters for Documentation Tags
- How to: Use the XML Documentation Features

Related Sections

For more information, see:

- /doc (Process Documentation Comments)

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Recommended Tags for Documentation Comments (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

The C# compiler processes documentation comments in your code and formats them as XML in a file whose name you specify in the **/doc** command-line option. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as DocFX or Sandcastle.

Tags are processed on code constructs such as types and type members.

> **NOTE**
> Documentation comments cannot be applied to a namespace.

The compiler will process any tag that is valid XML. The following tags provide generally used functionality in user documentation.

Tags

<c>	<para>	<see>*
<code>	<param>*	<seealso>*
<example>	<paramref>	<summary>
<exception>*	<permission>*	<typeparam>*
<include>*	<remarks>	<typeparamref>
<list>	<returns>	<value>

(* denotes that the compiler verifies syntax.)

If you want angle brackets to appear in the text of a documentation comment, use `<` and `>`, as shown in the following example.

```
/// <summary cref="C < T >">
/// </summary>
```

See also

- C# Programming Guide
- /doc (C# Compiler Options)
- XML Documentation Comments

<c> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<c>text</c>
```

Parameters

`text`

The text you would like to indicate as code.

Remarks

The <c> tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<code> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<code>content</code>
```

Parameters

`content`

The text you want marked as code.

Remarks

The <code> tag gives you a way to indicate multiple lines as code. Use <c> to indicate that text within a description should be marked as code.

Compile with /doc to process documentation comments to a file.

Example

See the <example> topic for an example of how to use the <code> tag.

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

cref Attribute (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

The `cref` attribute in an XML documentation tag means "code reference." It specifies that the inner text of the tag is a code element, such as a type, method, or property. Documentation tools like DocFX and Sandcastle use the `cref` attributes to automatically generate hyperlinks to the page where the type or member is documented.

Example

The following example shows `cref` attributes used in <see> tags.

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>
```

```
        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

When compiled, the program produces the following XML file. Notice that the `cref` attribute for the `GetZero` method, for example, has been transformed by the compiler to `"M:TestNamespace.TestClass.GetZero"`. The "M:" prefix means "method" and is a convention that is recognized by documentation tools such as DocFX and Sandcastle. For a complete list of prefixes, see Processing the XML File.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
            TestClass contains cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
            This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref
attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
            This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/>
constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
            The GetZero method.
            </summary>
            <example>
            This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
            <code>
            class TestClass
            {
                static int Main()
                {
                    return GetZero();
```

```
                }
            }
            </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue``1(``0)">
            <summary>
            The GetGenericValue method.
            </summary>
            <remarks>
            This sample shows how to specify the <see
cref="M:TestNamespace.TestClass.GetGenericValue``1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
            GenericClass.
            </summary>
            <remarks>
            This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref
attribute.
            </remarks>
        </member>
    </members>    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
            TestClass contains two cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
            The GetZero method.
            </summary>
            <example> This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/>
method.
            <code>
            class TestClass
            {
                static int Main()
                {
                    return GetZero();
                }
            }
            </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue``1(``0)">
            <summary>
            The GetGenericValue method.
            </summary>
            <remarks>
            This sample shows how to specify the <see
cref="M:TestNamespace.TestClass.GetGenericValue``1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
            GenericClass.
            </summary>
            <remarks>
            This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref
attribute.
            </remarks>
        </member>
    </members>
</doc>
```

See also

- XML Documentation Comments
- Recommended Tags for Documentation Comments

<example> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<example>description</example>
```

Parameters

`description`

A description of the code sample.

Remarks

The <example> tag lets you specify an example of how to use a method or other library member. This commonly involves using the <code> tag.

Compile with /doc to process documentation comments to a file.

Example

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
```

```
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>

        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<exception> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<exception cref="member">description</exception>
```

Parameters

cref = " `member` "

A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within double quotation marks (" ").

For more information on how to create a cref reference to a generic type, see <see>.

`description`

A description of the exception.

Remarks

The <exception> tag lets you specify which exceptions can be thrown. This tag can be applied to definitions for methods, properties, events, and indexers.

Compile with /doc to process documentation comments to a file.

For more information about exception handling, see Exceptions and Exception Handling.

Example

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<include> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<include file='filename' path='tagpath[@name="id"]' />
```

Parameters

`filename`

The name of the XML file containing the documentation. The file name can be qualified with a path relative to the source code file. Enclose `filename` in single quotation marks (' ').

`tagpath`

The path of the tags in `filename` that leads to the tag `name`. Enclose the path in single quotation marks (' ').

`name`

The name specifier in the tag that precedes the comments; `name` will have an `id`.

`id`

The ID for the tag that precedes the comments. Enclose the ID in double quotation marks (" ").

Remarks

The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file. By putting the documentation in a separate file, you can apply source control to the documentation separately from the source code. One person can have the source code file checked out and someone else can have the documentation file checked out.

The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your <include> use.

Example

This is a multifile example. The first file, which uses <include>, is listed below:

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

The second file, xml_include_tag.doc, contains the following documentation comments:

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Program Output

The following output is generated when you compile the Test and Test2 classes with the following command line: `/doc:DocFileName.xml.` In Visual Studio, you specify the XML doc comments option in the Build pane of the Project Designer. When the C# compiler sees the <include> tag, it will search for documentation comments in xml_include_tag.doc instead of the current source file. The compiler then generates DocFileName.xml, and this is the file that is consumed by documentation tools such as DocFX and Sandcastle to produce the final documentation.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<list> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<list type="bullet" | "number" | "table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>term</term>
        <description>description</description>
    </item>
</list>
```

Parameters

`term`

A term to define, which will be defined in `description`.

`description`

Either an item in a bullet or numbered list or the definition of a `term`.

Remarks

The <listheader> block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading.

Each item in the list is specified with an <item> block. When creating a definition list, you will need to specify both `term` and `description`. However, for a table, bulleted list, or numbered list, you only need to supply an entry for `description`.

A list or table can have as many <item> blocks as needed.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<para> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<para>content</para>
```

Parameters

`content`

The text of the paragraph.

Remarks

The <para> tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <para>.

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<param> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<param name="name">description</param>
```

Parameters

`name`

The name of a method parameter. Enclose the name in double quotation marks (" ").

`description`

A description for the parameter.

Remarks

The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method. To document multiple parameters, use multiple <param> tags.

The text for the <param> tag will be displayed in IntelliSense, the Object Browser, and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide

<paramref> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<paramref name="name"/>
```

Parameters

`name`

The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

Remarks

The <paramref> tag gives you a way to indicate that a word in the code comments, for example in a <summary> or <remarks> block refers to a parameter. The XML file can be processed to format this word in some distinct way, such as with a bold or italic font.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<permission> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<permission cref="member">description</permission>
```

Parameters

cref = " `member` "

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates `member` to the canonical element name in the output XML. *member* must appear within double quotation marks (" ").

For information on how to create a cref reference to a generic type, see <see>.

`description`

A description of the access to the member.

Remarks

The <permission> tag lets you document the access of a member. The PermissionSet class lets you specify access to a member.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<remarks> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<remarks>description</remarks>
```

Parameters

`Description`

A description of the member.

Remarks

The <remarks> tag is used to add information about a type, supplementing the information specified with <summary>. This information is displayed in the Object Browser window.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<returns> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<returns>description</returns>
```

Parameters

`description`

A description of the return value.

Remarks

The <returns> tag should be used in the comment for a method declaration to describe the return value.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<see> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<see cref="member"/>
```

Parameters

cref = " `member` "

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. Place *member* within double quotation marks (" ").

Remarks

The <see> tag lets you specify a link from within text. Use <seealso> to indicate that text should be placed in a See Also section. Use the cref Attribute to create internal hyperlinks to documentation pages for code elements.

Compile with -doc to process documentation comments to a file.

The following example shows a <see> tag within a summary section.

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<seealso> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<seealso cref="member"/>
```

Parameters

cref = " `member` "

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within double quotation marks (" ").

For information on how to create a cref reference to a generic type, see <see>.

Remarks

The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <seealso>.

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<summary> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<summary>description</summary>
```

Parameters

`description`

A summary of the object.

Remarks

The <summary> tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description. Use the cref Attribute to enable documentation tools such as DocFX and Sandcastle to create internal hyperlinks to documentation pages for code elements.

The text for the <summary> tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser Window.

Compile with /doc to process documentation comments to a file. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as DocFX or Sandcastle.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

The previous example produces the following XML file.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>YourNamespace</name>
    </assembly>
    <members>
        <member name="T:DotNetEvents.TestClass">
            text for class TestClass
        </member>
        <member name="M:DotNetEvents.TestClass.DoWork(System.Int32)">
            <summary>DoWork is a method in the TestClass class.
            <para>Here's how you could make a second paragraph in a description. <see
cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
            <seealso cref="M:DotNetEvents.TestClass.Main"/>
            </summary>
        </member>
        <member name="M:DotNetEvents.TestClass.Main">
            text for Main
        </member>
    </members>
</doc>
```

Example

The following example shows how to make a `cref` reference to a generic type.

```
// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }
```

The previous example produces the following XML file.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>YourNamespace</name>
    </assembly>
    <members>
        <member name="T:ExtensionMethodsDemo1.A">
            <summary cref="T:ExtensionMethodsDemo1.C`1">
            </summary>
        </member>
        <member name="T:ExtensionMethodsDemo1.B">
            <summary cref="T:C`1">
            </summary>
        </member>
        <member name="T:ExtensionMethodsDemo1.C`1">
            <summary cref="T:ExtensionMethodsDemo1.A">
            </summary>
            <typeparam name="T"></typeparam>
        </member>
    </members>
</doc>
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<typeparam> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<typeparam name="name">description</typeparam>
```

Parameters

`name`

The name of the type parameter. Enclose the name in double quotation marks (" ").

`description`

A description for the type parameter.

Remarks

The `<typeparam>` tag should be used in the comment for a generic type or method declaration to describe a type parameter. Add a tag for each type parameter of the generic type or method.

For more information, see Generics.

The text for the `<typeparam>` tag will be displayed in IntelliSense, the Object Browser Window code comment web report.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

See also

- C# Reference
- C# Programming Guide
- Recommended Tags for Documentation Comments

<typeparamref> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<typeparamref name="name"/>
```

Parameters

`name`

The name of the type parameter. Enclose the name in double quotation marks (" ").

Remarks

For more information on type parameters in generic types and methods, see Generics.

Use this tag to enable consumers of the documentation file to format the word in some distinct way, for example in italics.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<value> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<value>property-description</value>
```

Parameters

`property-description`

A description for the property.

Remarks

The <value> tag lets you describe the value that a property represents. Note that when you add a property via code wizard in the Visual Studio .NET development environment, it will add a <summary> tag for the new property. You should then manually add a <value> tag to describe the value that the property represents.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

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

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

Processing the XML File (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

The compiler generates an ID string for each construct in your code that is tagged to generate documentation. (For information about how to tag your code, see Recommended Tags for Documentation Comments.) The ID string uniquely identifies the construct. Programs that process the XML file can use the ID string to identify the corresponding .NET Framework metadata/reflection item that the documentation applies to.

The XML file is not a hierarchical representation of your code; it is a flat list that has a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

- No white space is in the string.
- The first part of the ID string identifies the kind of member being identified, by way of a single character followed by a colon. The following member types are used:

CHARACTER	DESCRIPTION
N	namespace You cannot add documentation comments to a namespace, but you can make cref references to them, where supported.
T	type: class, interface, struct, enum, delegate
F	field
P	property (including indexers or other indexed properties)
M	method (including such special methods as constructors, operators, and so forth)
E	event
!	error string The rest of the string provides information about the error. The C# compiler generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the hash-sign ('#'). It is assumed that no item has a hash-sign directly in its name. For example, the fully qualified name of the String constructor would be "System.String.#ctor".
- For properties and methods, if there are arguments to the method, the argument list enclosed in parentheses follows. If there are no arguments, no parentheses are present. The arguments are separated by commas. The encoding of each argument follows directly how it is encoded in a .NET Framework signature:
  - Base types. Regular types (ELEMENT_TYPE_CLASS or ELEMENT_TYPE_VALUETYPE) are

represented as the fully qualified name of the type.

- Intrinsic types (for example, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF. and ELEMENT_TYPE_VOID) are represented as the fully qualified name of the corresponding full type. For example, System.Int32 or System.TypedReference.
- ELEMENT_TYPE_PTR is represented as a '*' following the modified type.
- ELEMENT_TYPE_BYREF is represented as a '@' following the modified type.
- ELEMENT_TYPE_PINNED is represented as a '^' following the modified type. The C# compiler never generates this.
- ELEMENT_TYPE_CMOD_REQ is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this.
- ELEMENT_TYPE_CMOD_OPT is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
- ELEMENT_TYPE_SZARRAY is represented as "[]" following the element type of the array.
- ELEMENT_TYPE_GENERICARRAY is represented as "[?]" following the element type of the array. The C# compiler never generates this.
- ELEMENT_TYPE_ARRAY is represented as [*lowerbound*: `size`,*lowerbound*: `size`] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is simply omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a 2-dimensional array with 1 as the lower bounds and unspecified sizes is [1:,1:].
- ELEMENT_TYPE_FNPTR is represented as "=FUNC:`type`(*signature*)", where `type` is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this.

The following signature components are not represented because they are never used for differentiating overloaded methods:

- calling convention
- return type
- ELEMENT_TYPE_SENTINEL

- For conversion operators only (op_Implicit and op_Explicit), the return value of the method is encoded as a '~' followed by the return type, as encoded above.
- For generic types, the name of the type is followed by a backtick and then a number that indicates the number of generic type parameters. For example:

  `<member name="T:SampleClass`2">` is the tag for a type that is defined as `public class SampleClass<T, U>`.

  For methods taking generic types as parameters, the generic type parameters are specified as numbers prefaced with backticks (for example `0,`1). Each number representing a zero-based array notation for the type's generic parameters.

Examples

The following examples show how the ID strings for a class and its members would be generated:

```
namespace N
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }


        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public X(int i) { }


        /// <summary>
        /// Enter description here for field q.
        /// ID string generated is "F:N.X.q".
        /// </summary>
        public string q;


        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:N.X.PI".
        /// </summary>
        public const double PI = 3.14;


        /// <summary>
        /// Enter description for method f.
        /// ID string generated is "M:N.X.f".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int f() { return 1; }


        /// <summary>
        /// Enter description for method bb.
        /// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <param name="y">Describe parameter.</param>
        /// <param name="z">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int bb(string s, ref int y, void* z) { return 1; }


        /// <summary>
        /// Enter description for method gg.
        /// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int gg(short[] array1, int[,] array) { return 0; }


        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Addition(N.X,N.X)".
```

```
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <param name="xx">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static X operator +(X x, X xx) { return x; }


        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }


        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:N.X.d".
        /// </summary>
        public event D d;


        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.Item(System.String)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }


        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:N.X.Nested".
        /// </summary>
        public class Nested { }


        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:N.X.D".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void D(int i);


        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(X x) { return 1; }

    }
}
```

See also

- C# Programming Guide
- /doc (C# Compiler Options)
- XML Documentation Comments

Delimiters for Documentation Tags (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The use of XML doc comments requires delimiters, which indicate to the compiler where a documentation comment begins and ends. You can use the following kinds of delimiters with the XML documentation tags:

`///`
Single-line delimiter. This is the form that is shown in documentation examples and used by the Visual C# project templates. If there is a white space character following the delimiter, that character is not included in the XML output.

> **NOTE**
> The Visual Studio IDE has a feature called Smart Comment Editing that automatically inserts the <summary> and </summary> tags and moves your cursor within these tags after you type the `///` delimiter in the Code Editor. You can turn this feature on or off in the Options dialog box.

`/** */`
Multiline delimiters.

There are some formatting rules to follow when you use the `/** */` delimiters.

- On the line that contains the `/**` delimiter, if the remainder of the line is white space, the line is not processed for comments. If the first character after the `/**` delimiter is white space, that white space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
- On the line that contains the `*/` delimiter, if there is only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment, subject to the pattern-matching rules described in the following bullet.
- For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line. The pattern can consist of optional white space and an asterisk (`*`), followed by more optional white space. If the compiler finds a common pattern at the beginning of each line that does not begin with the `/**` delimiter or the `*/` delimiter, it ignores that pattern for each line.

The following examples illustrate these rules.

- The only part of the following comment that will be processed is the line that begins with `<summary>`. The three tag formats produce the same comments.

```
/** <summary>text</summary> */

/**
<summary>text</summary>
*/

/**
 * <summary>text</summary>
*/
```

- The compiler identifies a common pattern of " * " at the beginning of the second and third lines. The pattern is not included in the output.

```
/**
 * <summary>
 * text </summary>*/
```

- The compiler finds no common pattern in the following comment because the second character on the third line is not an asterisk. Therefore, all text on the second and third lines is processed as part of the comment.

```
/**
 * <summary>
   text </summary>
*/
```

- The compiler finds no pattern in the following comment for two reasons. First, the number of spaces before the asterisk is not consistent. Second, the fifth line begins with a tab, which does not match spaces. Therefore, all text from lines two through five is processed as part of the comment.

```
/**
  * <summary>
  * text
 *  text2
    *  </summary>
*/
```

See also

- C# Programming Guide
- XML Documentation Comments
- /doc (C# Compiler Options)
- XML Documentation Comments

How to: Use the XML documentation features

1/23/2019 • 4 minutes to read • Edit Online

The following sample provides a basic overview of a type that has been documented.

Example

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
    }
```

```
    /// <summary>
    /// Some other method.
    /// </summary>
    /// <returns>
    /// Return values are described through the returns tag.
    /// </returns>
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific method.
    /// </seealso>
    public int SomeOtherMethod()
    {
        return 0;
    }

    public int InterfaceMethod(int n)
    {
        return n * n;
    }

    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args"> A list of command line arguments.</param>
    static int Main(System.String[] args)
    {
        // TODO: Add code to start application here.
        return 0;
    }
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}
```

The example generates an .xml file with the following contents:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlsample</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
            Class level summary documentation goes here.
            </summary>
            <remarks>
            Longer comments can be associated with a type or member through
            the remarks tag.
            </remarks>
        </member>
```

```
        <member name="F:TestClass._name">
            <summary>
            Store for the Name property.
            </summary>
        </member>
        <member name="M:TestClass.#ctor">
            <summary>
            The class constructor.
            </summary>
        </member>
        <member name="P:TestClass.Name">
            <summary>
            Name property.
            </summary>
            <value>
            A value tag is used to describe the property value.
            </value>
        </member>
        <member name="M:TestClass.SomeMethod(System.String)">
            <summary>
            Description for SomeMethod.
            </summary>
            <param name="s"> Parameter description for s goes here.</param>
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference a type or member
            and the compiler will check that the reference exists.
            </seealso>
        </member>
        <member name="M:TestClass.SomeOtherMethod">
            <summary>
            Some other method.
            </summary>
            <returns>
            Return values are described through the returns tag.
            </returns>
            <seealso cref="M:TestClass.SomeMethod(System.String)">
            Notice the use of the cref attribute to reference a specific method.
            </seealso>
        </member>
        <member name="M:TestClass.Main(System.String[])">
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments.</param>
        </member>
        <member name="T:TestInterface">
            <summary>
            Documentation that describes the interface goes here.
            </summary>
            <remarks>
            Details about the interface go here.
            </remarks>
        </member>
        <member name="M:TestInterface.InterfaceMethod(System.Int32)">
            <summary>
            Documentation that describes the method goes here.
            </summary>
            <param name="n">
            Parameter n requires an integer argument.
            </param>
            <returns>
            The method returns an integer.
            </returns>
        </member>
    </members>
</doc>
```

Compiling the code

To compile the example, type the following command line:

```
csc XMLsample.cs /doc:XMLsample.xml
```

This command creates the XML file *XMLsample.xml*, which you can view in your browser or by using the TYPE command.

Robust programming

XML documentation starts with ///. When you create a new project, the wizards put some starter /// lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment that says that an error was encountered.
- Developers are free to create their own set of tags. There is a recommended set of tags (see Recommended tags for documentation comments). Some of the recommended tags have special meanings:
  - The <param> tag is used to describe parameters. If used, the compiler verifies that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.
  - The `cref` attribute can be attached to any tag to provide a reference to a code element. The compiler verifies that this code element exists. If the verification failed, the compiler issues a warning. The compiler respects any `using` statements when it looks for a type described in the `cref` attribute.
  - The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

    > **NOTE**
    > The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, the documentation file must be used together with reflection on the actual type or member.

See also

- C# Programming Guide
- /doc (C# Compiler Options)
- XML Documentation Comments