### C# 3.0 Language

The C# 3.0 language and compiler introduce several new language features. These new language constructs are useful individually in various contexts, and collectively for doing Language-Integrated Query (LINQ). For more information about LINQ, see [The LINQ Project](http://go.microsoft.com/fwlink/?LinkId=71829).

The following table lists the new C# 3.0 language features:

|  |  |
| --- | --- |
| **Feature** | **Description** |
| [Implicitly Typed Local Variables and Arrays](http://msdn.microsoft.com/en-us/library/bb384061.aspx) | When used with local variables, the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) keyword instructs the compiler to infer the type of the variable or the array elements from the expression on the right side of the initialization statement. |
| [Object Initializers](http://msdn.microsoft.com/en-us/library/bb384062.aspx) | Enables object initialization without explicit calls to a constructor. |
| [Collection Initializers](http://msdn.microsoft.com/en-us/library/bb384062.aspx) | Enables initialization of collections with an initialization list rather than specific calls to **Add** or another method. |
| [Extension Methods](http://msdn.microsoft.com/en-us/library/bb383977.aspx) | Extend existing classes by using static methods that can be invoked by using instance method syntax. |
| [Anonymous Types](http://msdn.microsoft.com/en-us/library/bb397696.aspx) | Enables on-the-fly creation of unnamed structured types that can be added to collections and accessed by using var. |
| [Lambda Expressions](http://msdn.microsoft.com/en-us/library/bb397687.aspx) | Enables inline expressions with input parameters that can be bound to delegates or expression trees. See also [Anonymous Functions (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb882516.aspx). |
| [Query Keywords](http://msdn.microsoft.com/en-us/library/bb310804.aspx) | Keywords that specify clauses in a query expression:   * from clause(s) * where clause (optional) * ordering clauses (optional) * join clause (optional) * select or group clause * into clause (optional) |
| [Auto-Implemented Properties](http://msdn.microsoft.com/en-us/library/bb384054.aspx) | Enables declaration of properties by using simplified syntax. |
| [Partial Method Definitions](http://msdn.microsoft.com/en-us/library/wa80x488.aspx) | Partial types may now contain partial methods. |

**Implicitly Typed Local Variables (C# Programming Guide)**

Updated: July 2008

Local variables can be given an inferred "type" of var instead of 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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384090.aspx).

The following examples show various ways in which local variables can be declared with var:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl02CSharp');)

// 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](http://msdn.microsoft.com/en-us/library/ch45axte.aspx) initialization statement.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl04other');)

for(var x = 1; x < 10; x++)

* In a [foreach](http://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) initialization statement.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl06other');)

foreach(var item in list){...}

* In a [using Statement](http://msdn.microsoft.com/en-us/library/yh598w02.aspx)

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08other');)

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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb310799.aspx).

 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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397696.aspx).

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.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15CSharp');)

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.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl19CSharp');)

// 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.

**How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression (C# Programming Guide)**

You must use implicitly-typed local variables to store anonymous types. You can also use them in any other situation in which you want the compiler to determine the type of a local variable (in other words a variable declared at method scope). The following examples show how to use implicitly typed variables in both scenarios.

Implicitly-typed local variables are declared by using the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) contextual keyword. For more information, see [Implicitly Typed Local Variables (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384061.aspx) and [Implicitly Typed Arrays (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384090.aspx).

The following example shows how to use the var keyword in a common scenario where it is required: when you are creating and executing a query expression that produces a sequence of anonymous types. Note that in this scenario, not only the query variable, but also the iteration variable in the foreach statement must be implicitly typed by using var.

C#

private static void QueryNames(char firstLetter)

{ // Create the query. var is required because

// the query produces a sequence of anonymous types.

var studentQuery =

from student in students

where student.FirstName[0] == firstLetter

select new { student.FirstName, student.LastName };

// Execute the query.

foreach (var student in studentQuery)

{

Console.WriteLine("First = {0}, Last = {1}", student.FirstName, student.LastName);

}

}

The example later in this topic shows how to use the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) keyword as a syntactic convenience even when it is not required. To illustrate this construction, only the query variable is implicitly typed. The iteration variable in the foreach statement is explicitly typed but it can also be declared by using var. Remember, var itself is not a type, but rather an instruction to the compiler to infer and assign the type.

C#

var queryID =

from student in students

where student.ID > 111

select student.LastName;

foreach (string str in queryID)

{

Console.WriteLine(str);

}

**Object and Collection Initializers (C# Programming Guide)**

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to explicitly invoke a constructor. The following example shows how to use an object initializer with a named type. Note the use of auto-implemented properties in the Test class. For more information, see [Auto-Implemented Properties (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384054.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl02CSharp');)

private class Cat

{

// Auto-implemented properties

public int Age { get; set; }

public string Name { get; set; }

}

static void MethodA()

{

// Object initializer

Cat cat = new Cat { Age = 10, Name = "Sylvester" };

}

 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 with an object initializer. In the select clause, a query expression can transform objects of the original sequence into objects whose value and shape may differ from the original. This is very useful if you want to store only a part of the information in 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.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08CSharp');)

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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09other');)

foreach(var p in productInfos){...}

Each object in the new anonymous type has two public properties which 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.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl10other');)

select new {p.ProductName, Price = p.UnitPrice};

 Object initializers with nullable types

It is a compile-time error to use a collection initializer with a nullable struct.

 Collection Initializers

Collection initializers let you specify one or more element intializers when you initialize a collection class that implements [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx). 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 to the **Add** method of the class in your source code; the compiler adds the calls.

The following examples shows two simple collection initializers:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl21other');)

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 an earlier example. Note that the individual object initializers are enclosed in braces and separated by commas.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl22CSharp');)

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](http://msdn.microsoft.com/en-us/library/edakx9da.aspx) as an element in a collection initializer if the collection's **Add** method allows it.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl24CSharp');)

List<Cat> moreCats = new List<Cat>

{

new Cat(){ Name="Furrytail", Age=5 },

new Cat(){ Name="Peaches", Age=4},

null

};

**Extension Methods (C# Programming Guide)**

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# 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](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx) and [System.Collections.Generic..::.IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) types. To use the standard query operators, first bring them into scope with a using System.Linq directive. Then any type that implements [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) appears to have instance methods such as [GroupBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.groupby.aspx), [OrderBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.orderby.aspx), [Average](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.average.aspx), and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) type such as [List<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx) or [Array](http://msdn.microsoft.com/en-us/library/system.array.aspx).

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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397687.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl21CSharp');)

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](http://msdn.microsoft.com/en-us/library/dk1507sz.aspx) 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](http://msdn.microsoft.com/en-us/library/system.string.aspx) class. Note that it is defined inside a non-nested, non-generic static class:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl25CSharp');)

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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl26other');)

using ExtensionMethods;

And it can be called from an application by using this syntax:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl27other');)

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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb311042.aspx).

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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl29other');)

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<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) types.

|  |
| --- |
| **Note:** |
| Although standard query operators do not appear in IntelliSense for [String](http://msdn.microsoft.com/en-us/library/system.string.aspx), they are still available. |

 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 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.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl40CSharp');)

namespace Extensions

{

using System;

using ExtensionMethodsDemo1;

// Define extension methods for any type 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, because the three classes implement MethodB.

public static void MethodB(this IMyInterface myInterface)

{

Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");

}

}

}

namespace ExtensionMethodsDemo1

{

using System;

using Extensions;

public interface IMyInterface

{

void MethodB();

}

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)

{

A a = new A();

B b = new B();

C c = new C();

TestMethodBinding(a,b,c);

}

static void TestMethodBinding(A a, B b, C c)

{

// A has no methods, so each call resolves to

// the extension methods whose signatures match.

a.MethodA(1); // Extension.MethodA(object, int)

a.MethodA("hello"); // Extension.MethodA(object, string)

a.MethodB(); // A.MethodB()

// B itself has a method with this signature.

b.MethodA(1); // B.MethodA(int)

b.MethodB(); // B.MethodB()

// B has no matching method, but Extension does.

b.MethodA("hello"); // Extension.MethodA(object, string)

// In each case C has a matching instance method.

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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173149.aspx).

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 two 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.

**Anonymous Types (C# Programming Guide)**

Updated: July 2008

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to first explicitly define a type. The type name is generated by the compiler and is not available at the source code level. The type of the properties is inferred by the compiler. The following example shows an anonymous type being initialized with two properties called Amount and Message.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl01other');)

var v = new { Amount = 108, Message = "Hello" };

Anonymous types are typically used in the [select](http://msdn.microsoft.com/en-us/library/bb384087.aspx) 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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397676.aspx).

Anonymous types are created by using the [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx) operator with an object initializer. For more information about object initializers, see [Object and Collection Initializers (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384062.aspx).

Anonymous types are [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) types that consist of one or more public read-only properties. No other kinds of class members such as methods or events are allowed. An anonymous type cannot be cast to any interface or type except for [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx).

The most common scenario is to initialize an anonymous type with some properties from another type. In the following example, assume a class that is named Product that includes Color and Price properties together with several other properties that you are not interested in. Products is a collection of Product objects. The anonymous type declaration starts with the new keyword. It 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 to a property that is being initialized with an expression.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08CSharp');)

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);

}

When an anonymous type is assigned to a variable, that variable must be initialized with the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) construct. This is because only the compiler has access to the underlying name of the anonymous type. For more information about var, see [Implicitly Typed Local Variables (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384061.aspx).

 Remarks

Anonymous types are reference types that derive directly from [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx). The compiler gives them a name 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, except that it cannot be cast to any type except for [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx).

If two or more anonymous types have the same number and type of properties in the same order, the compiler treats them as the same type and they share the same compiler-generated type information.

An anonymous type has method scope. To pass an anonymous type, or a collection that contains anonymous types, outside a method boundary, you must first cast the type to [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx). However, this defeats the strong typing of the anonymous type. If you must store your query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Anonymous types cannot contain [unsafe](http://msdn.microsoft.com/en-us/library/chfa2zb8.aspx) types as properties.

Because the [Equals](http://msdn.microsoft.com/en-us/library/system.object.equals.aspx) and [GetHashCode](http://msdn.microsoft.com/en-us/library/system.object.gethashcode.aspx) methods on anonymous types are defined in terms of the Equals and GetHashcode of the properties, two instances of the same anonymous type are equal only if all their properties are equal.

**Lambda Expressions (C# Programming Guide)**

A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.

All lambda expressions use the lambda operator [=>](http://msdn.microsoft.com/en-us/library/bb311046.aspx), which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block. The lambda expression x => x \* x is read "x goes to x times x." This expression can be assigned to a delegate type as follows:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl02other');)

delegate int del(int i);

del myDelegate = x => x \* x;

int j = myDelegate(5); //j = 25

To create an expression tree type:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl03other');)

using System.Linq.Expressions;

// ...

Expression<del> = x => x \* x;

The => operator has the same precedence as assignment (=) and is right-associative.

Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx).

When you use method-based syntax to call the [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) method in the [Enumerable](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.aspx) class (as you do in LINQ to Objects and LINQ to XML) the parameter is a delegate type [System..::.Func<(Of <(T, TResult>)>)](http://msdn.microsoft.com/en-us/library/bb549151.aspx). 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](http://msdn.microsoft.com/en-us/library/system.linq.queryable.aspx) class (as you do in LINQ to SQL) then the parameter type is an [System.Linq.Expressions..::.Expression](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx)<Func> where Func is any Func delegates with up to five 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](http://msdn.microsoft.com/en-us/library/scekt9xw.aspx) or [as](http://msdn.microsoft.com/en-us/library/cscsdfbt.aspx) operator.

All restrictions that apply to anonymous methods also apply to lambda expressions. For more information, see [Anonymous Methods (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx).

 Expression Lambdas

A lambda expression with an expression on the right side is called an expression lambda. Expression lambdas are used extensively in the construction of [Expression Trees](http://msdn.microsoft.com/en-us/library/bb397951.aspx). An expression lambda returns the result of the expression and takes the following basic form:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl26other');)

(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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl27other');)

(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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl28other');)

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

Specify zero input parameters with empty parentheses:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl29other');)

() => 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 will be consumed in another domain, such as 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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl34other');)

(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.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl35other');)

delegate void TestDelegate(string s);

…

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

myDel("Hello");

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

 Lambdas with the Standard Query Operators

Many Standard query operators have an input parameter whose type is one of the [Func<(Of <(T, TResult>)>)](http://msdn.microsoft.com/en-us/library/bb549151.aspx) family of generic delegates. The [Func<(Of <(T, TResult>)>)](http://msdn.microsoft.com/en-us/library/bb549151.aspx) delegates use type parameters to define the number and type 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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl56other');)

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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl57other');)

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](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.count.aspx) method, is shown here:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl59other');)

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 method will produce a sequence that contains all the elements in the numbers array that are to the left of the “9” because that is the first number in the sequence that does not meet the condition:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl60other');)

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 (>=).

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl61other');)

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 underlying delegate type, and other factors as described in the C# 3.0 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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl66other');)

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](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx) type to which the lambda expression is converted.

 Variable Scope in Lambda Expressions

Lambdas can refer to outer variables that are in scope in the enclosing method or type in which the lambda is defined. Variables that are captured in this manner are stored for use in the lambda expression even if 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:

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl71other');)

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 goes out of scope.
* Variables introduced within a lambda expression are not visible in the outer method.
* A lambda expression cannot directly capture a 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 whose target is outside the body or in the body of a contained anonymous function.

 C# Language Specification

For more information, see the following section in the [C# Language Specification](http://msdn.microsoft.com/en-us/library/ms228593.aspx):

* 5.3.3.29 Anonymous functions

**Query Keywords (C# Reference)**

This section contains the contextual keywords used in query expressions.

 In This Section

|  |  |
| --- | --- |
| **Clause** | **Description** |
| [from](http://msdn.microsoft.com/en-us/library/bb383978.aspx) | Specifies a data source and a range variable (similar to an iteration variable). |
| [where](http://msdn.microsoft.com/en-us/library/bb311043.aspx) | Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators ( && or || ). |
| [select](http://msdn.microsoft.com/en-us/library/bb384087.aspx) | Specifies the type and shape that the elements in the returned sequence will have when the query is executed. |
| [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) | Groups query results according to a specified key value. |
| [into](http://msdn.microsoft.com/en-us/library/bb311045.aspx) | Provides an identifier that can serve as a reference to the results of a join, group or select clause. |
| [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) | Sorts query results in ascending or descending order based on the default comparer for the element type. |
| [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) | Joins two data sources based on an equality comparison between two specified matching criteria. |
| [let](http://msdn.microsoft.com/en-us/library/bb383976.aspx) | Introduces a range variable to store sub-expression results in a query expression. |
| [in](http://msdn.microsoft.com/en-us/library/cc713603.aspx) | Contextual keyword in a [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) clause. |
| [on](http://msdn.microsoft.com/en-us/library/cc713588.aspx) | Contextual keyword in a [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) clause. |
| [equals](http://msdn.microsoft.com/en-us/library/cc713644.aspx) | Contextual keyword in a [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) clause. |
| [by](http://msdn.microsoft.com/en-us/library/cc713645.aspx) | Contextual keyword in a [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) clause. |
| [ascending](http://msdn.microsoft.com/en-us/library/cc713606.aspx) | Contextual keyword in an [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) clause. |
| [descending](http://msdn.microsoft.com/en-us/library/cc713622.aspx) | Contextual keyword in an [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) clause. |

**Auto-Implemented Properties (C# Programming Guide)**

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 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:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl06CSharp');)

// This class is mutable. Its data can be modified from

// outside the class.

class Customer

{

// Auto-Impl 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;

}

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, it is recommended to make the objects immutable by declaring the set accessor as [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx). For more information, see [How to: Implement an Immutable Class That has Auto-Implemented Properties (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb383979.aspx).

Attributes are permitted on auto-implemented properties but obviously not on the backing fields since those are not accessible from your source code. If you must use an attribute on the backing field of a property, just create a regular property.

**Partial Classes and Methods (C# Programming Guide)**

It is possible to split the definition of a [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or a [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx), an [interface](http://msdn.microsoft.com/en-us/library/87d83y5b.aspx) 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](http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx) keyword modifier, as shown here:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl25CSharp');)

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.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl27CSharp');)

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:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl28CSharp');)

[SerializableAttribute]

partial class Moon { }

[ObsoleteAttribute]

partial class Moon { }

They are equivalent to the following declarations:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl29CSharp');)

[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:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl30CSharp');)

partial class Earth : Planet, IRotate { }

partial class Earth : IRevolve { }

They are equivalent to the following declarations:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl31CSharp');)

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:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl32CSharp');)

public partial class A { }

//public class tcA { } // 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:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl33CSharp');)

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](http://msdn.microsoft.com/en-us/library/yzh058ae.aspx)
  + [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx)
  + [protected](http://msdn.microsoft.com/en-us/library/bcd5672a.aspx)
  + [internal](http://msdn.microsoft.com/en-us/library/7c5ka91b.aspx)
  + [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5.aspx)
  + [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw.aspx)
  + base class
  + [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx) modifier (nested parts)
  + generic constraints

For more information, see [Constraints on Type Parameters (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/d5x73970.aspx).

 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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl45CSharp');)

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl49CSharp');)

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. 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.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl63other');)

// 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](http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx) and the method must return [void](http://msdn.microsoft.com/en-us/library/yah0tteb.aspx).
* Partial methods can have [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) but not [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameters.
* Partial methods are implicitly [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx), and therefore they cannot be [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4.aspx).
* Partial methods cannot be [extern](http://msdn.microsoft.com/en-us/library/e59b22c5.aspx), because the presence of the body determines whether they are defining or implementing.
* Partial methods can have [static](http://msdn.microsoft.com/en-us/library/98f28cdx.aspx) and [unsafe](http://msdn.microsoft.com/en-us/library/chfa2zb8.aspx) 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 cannot make a [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) to a partial method.

 C# Language Specification

For more information, see the following sections in the [C# Language Specification](http://msdn.microsoft.com/en-us/library/ms228593.aspx):

* 10.2 Partial Types