Overview of C# 3.0

March 2007

Anders Hejlsberg, Mads Torgersen

Applies to:  
   Visual C# 3.0

**Summary:** Technical overview of C# 3.0 ("C# Orcas"), which introduces several language extensions that build on C# 2.0 to support the creation and use of higher order, functional style class libraries. (38 printed pages)

**Contents**

[Introduction](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic1)  
[26.1 Implicitly Typed Local Variables](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic2)  
[26.2 Extension Methods](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic3)  
   [26.2.1 Declaring Extension Methods](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic4)  
   [26.2.2 Available Extension Methods](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic5)  
   [26.2.3 Extension Method Invocations](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic6)  
[26.3 Lambda Expressions](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic7)  
   [26.3.1 Anonymous Method and Lambda Expression Conversions](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic8)  
   [26.3.2 Delegate Creation Expressions](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic9)  
   [26.3.3 Type Inference](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10)  
      [26.3.3.1 The first phase](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10a)  
      [26.3.3.2 The second phase](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10b)  
      [26.3.3.3 Input types](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10c)  
      [26.3.3.4 Output types](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10d)  
      [26.3.3.5 Dependence](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10e)  
      [26.3.3.6 Output type inferences](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10f)  
      [26.3.3.7 Explicit argument type inferences](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10g)  
      [26.3.3.8 Exact inferences](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10h)  
      [26.3.3.9 Lower-bound inferences](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10i)  
      [26.3.3.10 Fixing](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10j)  
      [26.3.3.11 Inferred return type](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10k)  
      [26.3.3.12 Type inference for conversion of method groups](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10l)  
      [26.3.3.13 Finding the best common type of a set of expressions](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10m)  
   [26.3.4 Overload Resolution](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic11)  
[26.4 Object and Collection Initializers](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic12)  
   [26.4.1 Object Initializers](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic13)  
   [26.4.2 Collection Initializers](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic14)  
[26.5 Anonymous Types](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic15)  
[26.6 Implicitly Typed Arrays](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic16)  
[26.7 Query Expressions](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic17)  
   [26.7.1 Query Expression Translation](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18)  
      [26.7.1.1 Select and groupby clauses with continuations](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18a)  
      [26.7.1.2 Explicit range variable types](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18b)  
      [26.7.1.3 Degenerate query expressions](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18c)  
      [26.7.1.4 From, let, where, join and orderby clauses](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18d)  
      [26.7.1.5 Select clauses](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18e)  
      [26.7.1.6 Groupby clauses](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18f)  
      [26.7.1.7 Transparent identifiers](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18g)  
   [26.7.2 The Query Expression Pattern](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic19)  
[26.8 Expression Trees](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic20)  
   [26.8.1 Overload Resolution](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic21)  
[26.9 Automatically Implemented Properties](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic22)

**Introduction**

This article contains some updates that apply to Visual C# 3.0. A comprehensive specification will accompany the release of the language.

C# 3.0 ("C# Orcas") introduces several language extensions that build on C# 2.0 to support the creation and use of higher order, functional style class libraries. The extensions enable construction of compositional APIs that have equal expressive power of query languages in domains such as relational databases and XML. The extensions include:

* Implicitly typed local variables, which permit the type of local variables to be inferred from the expressions used to initialize them.
* Extension methods, which make it possible to extend existing types and constructed types with additional methods.
* Lambda expressions, an evolution of anonymous methods that provides improved type inference and conversions to both delegate types and expression trees.
* Object initializers, which ease construction and initialization of objects.
* Anonymous types, which are tuple types automatically inferred and created from object initializers.
* Implicitly typed arrays, a form of array creation and initialization that infers the element type of the array from an array initializer.
* Query expressions, which provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.
* Expression trees, which permit lambda expressions to be represented as data (expression trees) instead of as code (delegates).

This document is a technical overview of those features. The document makes reference to the *C# Language Specification Version 1.2* (§1 through §18) and the *C# Language Specification Version 2.0* (§19 through §25), both of which are available on the C# Language Home Page (<http://msdn.microsoft.com/vcsharp/aa336809.aspx>).

**26.1 Implicitly Typed Local Variables**

In an ***implicitly typed local variable declaration***, the type of the local variable being declared is inferred from the expression used to initialize the variable. When a local variable declaration specifies **var** as the type and no type named **var** is in scope, the declaration is an implicitly typed local variable declaration. For example:

var i = 5;

var s = "Hello";

var d = 1.0;

var numbers = new int[] {1, 2, 3};

var orders = new Dictionary<int,Order>();

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

int i = 5;

string s = "Hello";

double d = 1.0;

int[] numbers = new int[] {1, 2, 3};

Dictionary<int,Order> orders = new Dictionary<int,Order>();

A local variable declarator in an implicitly typed local variable declaration is subject to the following restrictions:

* The declarator must include an initializer.
* The initializer must be an expression.
* The initializer expression must have a compile-time type which cannot be the null type.
* The local variable declaration cannot include multiple declarators.
* The initializer cannot refer to the declared variable itself

The following are examples of incorrect implicitly typed local variable declarations:

var x; // Error, no initializer to infer type from

var y = {1, 2, 3}; // Error, collection initializer not permitted

var z = null; // Error, null type not permitted

var u = x => x + 1; // Error, lambda expressions do not have a type

var v = v++; // Error, initializer cannot refer to variable itself

For reasons of backward compatibility, when a local variable declaration specifies **var** as the type and a type named **var** is in scope, the declaration refers to that type. Since a type named **var** violates the established convention of starting type names with an upper case letter, this situation is unlikely to occur.

The *for-initializer* of a **for** statement (§8.8.3) and the *resource-acquisition* of a **using** statement (§8.13) can be an implicitly typed local variable declaration. Likewise, the iteration variable of a **foreach** statement (§8.8.4) may be declared as an implicitly typed local variable, in which case the type of the iteration variable is inferred to be the element type of the collection being enumerated. In the example

int[] numbers = { 1, 3, 5, 7, 9 };

foreach (var n in numbers) Console.WriteLine(n);

the type of **n** is inferred to be **int**, the element type of **numbers**.

Only *local-variable-declaration*, *for-initializer*, *resource-acquisition* and *foreach-statement* can contain implicitly typed local variable declarations.

**26.2 Extension Methods**

***Extension methods*** are static methods that can be invoked using instance method syntax. In effect, extension methods make it possible to extend existing types and constructed types with additional methods.

**Note**   Extension methods are less discoverable and more limited in functionality than instance methods. For those reasons, it is recommended that extension methods be used sparingly and only in situations where instance methods are not feasible or possible. Extension members of other kinds, such as properties, events, and operators, are being considered but are currently not supported.

**26.2.1 Declaring Extension Methods**

Extension methods are declared by specifying the keyword **this** as a modifier on the first parameter of the methods. Extension methods can only be declared in non-generic, non-nested static classes. The following is an example of a static class that declares two extension methods:

namespace Acme.Utilities

{

public static class Extensions

{

public static int ToInt32(this string s) {

return Int32.Parse(s);

}

public static T[] Slice<T>(this T[] source, int index, int count) {

if (index < 0 || count < 0 || source.Length – index < count)

throw new ArgumentException();

T[] result = new T[count];

Array.Copy(source, index, result, 0, count);

return result;

}

}

}

The first parameter of an extension method can have no modifiers other than **this**, and the parameter type cannot be a pointer type.

Extension methods have all the capabilities of regular static methods. In addition, once imported, extension methods can be invoked using instance method syntax.

**26.2.2 Available Extension Methods**

Extension methods are ***available*** in a namespace if declared in a static class or imported through *using-namespace-directive*s (§9.3.2) in that namespace. In addition to importing the types contained in an imported namespace, a *using-namespace-directive* thus imports all extension methods in all static classes in the imported namespace.

In effect, available extension methods appear as additional methods on the types that are given by their first parameter and have lower precedence than regular instance methods. For example, when the **Acme.Utilities** namespace from the example above is imported with the *using-namespace-directive*

using Acme.Utilities;

it becomes possible to invoke the extension methods in the static class **Extensions** using instance method syntax:

string s = "1234";

int i = s.ToInt32(); // Same as Extensions.ToInt32(s)

int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int[] a = digits.Slice(4, 3); // Same as Extensions.Slice(digits, 4, 3)

**26.2.3 Extension Method Invocations**

The detailed rules for extension method invocation are described in the following. In a method invocation (§7.5.5.1) of one of the forms

*expr* . *identifier* ( )

*expr* . *identifier* ( *args* )

*expr* . *identifier* < *typeargs* > ( )

*expr* . *identifier* < *typeargs* > ( *args* )

if the normal processing of the invocation finds no applicable instance methods (specifically, if the set of candidate methods for the invocation is empty), an attempt is made to process the construct as an extension method invocation. The method invocation is first rewritten to one of the following, respectively:

*identifier* ( *expr* )

*identifier* ( *expr* , *args* )

*identifier* < *typeargs* > ( *expr* )

*identifier* < *typeargs* > ( *expr* , *args* )

The rewritten form is then processed as a static method invocation, except for the way in which *identifier* is resolved: Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to process the rewritten method invocation with a method group consisting of all available and accessible extension methods in the namespace with the name given by *identifier*. From this set remove all the methods that are not applicable (§7.4.2.1) and the ones where no implicit identity, reference or boxing conversion exists from the first argument to the first parameter. The first method group that yields a non-empty such set of candidate methods is the one chosen for the rewritten method invocation, and normal overload resolution (§7.4.2) is applied to select the best extension method from the set of candidates. If all attempts yield empty sets of candidate methods, a compile-time error occurs.

The preceding rules mean that instance methods take precedence over extension methods, and extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations. For example:

public static class E

{

public static void F(this object obj, int i) { }

public static void F(this object obj, string s) { }

}

class A { }

class B

{

public void F(int i) { }

}

class C

{

public void F(object obj) { }

}

class X

{

static void Test(A a, B b, C c) {

a.F(1); // E.F(object, int)

a.F("hello"); // E.F(object, string)

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

b.F("hello"); // E.F(object, string)

c.F(1); // C.F(object)

c.F("hello"); // C.F(object)

}

}

In the example, the **B** method takes precedence over the first extension method, and the **C** method takes precedence over both extension methods.

**26.3 Lambda Expressions**

C# 2.0 introduces anonymous methods, which allow code blocks to be written "in-line" where delegate values are expected. While anonymous methods provide much of the expressive power of functional programming languages, the anonymous method syntax is rather verbose and imperative in nature. ***Lambda expressions*** provide a more concise, functional syntax for writing anonymous methods.

A lambda expression is written as a parameter list, followed by the **=>** token, followed by an expression or a statement block.

expression:

assignment

non-assignment-expression

non-assignment-expression:

conditional-expression

lambda-expression

query-expression

lambda-expression:

*(* lambda-parameter-listopt *)* *=>* lambda-expression-body

implicitly-typed-lambda-parameter *=>* lambda-expression-body

lambda-parameter-list:

explicitly-typed-lambda-parameter-list

implicitly-typed-lambda-parameter-list

explicitly-typed-lambda-parameter-list

explicitly-typed-lambda-parameter

explicitly-typed-lambda-parameter-list *,* explicitly-typed-lambda-parameter

explicitly-typed-lambda-parameter:

parameter-modifieropt type identifier

implicitly-typed-lambda-parameter-list

implicitly-typed-lambda-parameter

implicitly-typed-lambda-parameter-list *,* implicitly-typed-lambda-parameter

implicitly-typed-lambda-parameter:

identifier

lambda-expression-body:

expression

block

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

The parameters of a lambda expression can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the lambda expression occurs—specifically, when the lambda expression is converted to a compatible delegate type, that delegate type provides the parameter types (§[26.3.1](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic8)).

In a lambda expression with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, a lambda expression of the form

( *param* ) => *expr*

can be abbreviated to

*param* => *expr*

Some examples of lambda expressions follow below:

x => x + 1 // Implicitly typed, expression body

x => { return x + 1; } // Implicitly typed, statement body

(int x) => x + 1 // Explicitly typed, expression body

(int x) => { return x + 1; } // Explicitly typed, statement body

(x, y) => x \* y // Multiple parameters

() => Console.WriteLine() // No parameters

In general, the specification of anonymous methods, provided in §21 of the *C# 2.0 Specification*, also applies to lambda expressions. Lambda expressions are functionally similar to anonymous methods, except for the following points:

* Anonymous methods permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of parameters.
* Lambda expressions permit parameter types to be omitted and inferred whereas anonymous methods require parameter types to be explicitly stated.
* The body of a lambda expression can be an expression or a statement block whereas the body of an anonymous method can only be a statement block.
* Lambda expressions with an expression body can be converted to expression trees (§[26.8](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic20)).

**26.3.1 Anonymous Method and Lambda Expression Conversions**

**Note**   This section replaces §21.3.

An *anonymous-method-expression* and a *lambda-expression* is classified as a value with special conversion rules. The value does not have a type but can be implicitly converted to a compatible delegate type. Specifically, a delegate type **D** is compatible with an anonymous method or lambda-expression **L** provided:

* **D** and **L** have the same number of parameters.
* If **L** is an anonymous method that does not contain an *anonymous-method-signature*, then **D** may have zero or more parameters of any type, as long as no parameter of **D** has the **out** parameter modifier.
* If **L** has an explicitly typed parameter list, each parameter in **D** has the same type and modifiers as the corresponding parameter in **L**.
* If **L** is a lambda expression that has an implicitly typed parameter list, **D** has no **ref** or **out** parameters.
* If **D** has a **void** return type and the body of **L** is an expression, when each parameter of **L** is given the type of the corresponding parameter in **D**, the body of **L** is a valid expression (wrt §7) that would be permitted as a *statement-expression* (§8.6).
* If **D** has a **void** return type and the body of **L** is a statement block, when each parameter of **L** is given the type of the corresponding parameter in **D**, the body of **L** is a valid statement block (wrt §8.2) in which no **return** statement specifies an expression.
* If **D** has a non-void return type and the body of **L** is an expression, when each parameter of **L** is given the type of the corresponding parameter in **D**, the body of **L** is a valid expression (wrt §7) that is implicitly convertible to the return type of **D**.
* If **D** has a non-void return type and the body of **L** is a statement block, when each parameter of **L** is given the type of the corresponding parameter in **D**, the body of **L** is a valid statement block (wrt §8.2) with a non-reachable end point in which each **return** statement specifies an expression that is implicitly convertible to the return type of **D**.

The examples that follow use a generic delegate type **Func<A,R>** which represents a function taking an argument of type **A** and returning a value of type **R**:

delegate R Func<A,R>(A arg);

In the assignments

Func<int,int> f1 = x => x + 1; // Ok

Func<int,double> f2 = x => x + 1; // Ok

Func<double,int> f3 = x => x + 1; // Error

the parameter and return types of each lambda expression are determined from the type of the variable to which the lambda expression is assigned. The first assignment successfully converts the lambda expression to the delegate type **Func<int,int>** because, when **x** is given type **int, x + 1** is a valid expression that is implicitly convertible to type **int**. Likewise, the second assignment successfully converts the lambda expression to the delegate type **Func<int,double>** because the result of **x + 1** (of type **int**) is implicitly convertible to type **double**. However, the third assignment is a compile-time error because, when **x** is given type **double**, the result of **x + 1** (of type **double**) is not implicitly convertible to type **int**.

**26.3.2 Delegate Creation Expressions**

**Note**   This section replaces §21.10.

Delegate creation expressions (§7.5.10.3) are extended to permit the argument to be an expression classified as a method group, an expression classified as an anonymous method or lambda expression, or a value of a delegate type.

The compile-time processing of a *delegate-creation-expression* of the form **new D(E)**, where **D** is a *delegate-type* and **E** is an *expression*, consists of the following steps:

* If **E** is a method group, a method group conversion (§21.9) must exist from **E** to **D**, and the delegate creation expression is processed in the same way as that conversion.
* If **E** is an anonymous method or lambda expression, an anonymous method or lambda expression conversion (§ [26.3.1](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic8)) must exist from **E** to **D**, and the delegate creation expression is processed in the same way as that conversion.
* If **E** is a value of a delegate type, the method signature of **E** must be consistent (§21.9) with **D**, and the result is a reference to a newly created delegate of type **D** that refers to the same invocation list as **E**. If **E** is not consistent with **D**, a compile-time error occurs.

**26.3.3 Type Inference**

**Note**   This section replaces §20.6.4.

When a generic method is called without specifying type arguments, a ***type inference*** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration:

class Chooser

{

static Random rand = new Random();

public static T Choose<T>(T first, T second) {

return (rand.Next(2) == 0)? first: second;

}

}

it is possible to invoke the **Choose** method without explicitly specifying a type argument:

int i = Chooser.Choose(5, 213); // Calls Choose<int>

string s = Chooser.Choose("foo", "bar"); // Calls Choose<string>

Through type inference, the type arguments **int** and **string** are determined from the arguments to the method.

Type inference occurs as part of the compile-time processing of a method invocation (§20.9.7) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a compile-time error. However, it often leads to a compile-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

Tr M<X1...Xn>(T1 x1 ... Tm xm)

With a method call of the form **M(e1...em)** the task of type inference is to find unique type arguments **S1...Sn** for each of the type parameters **X1...Xn** so that the call **M<S1...Sn>(e1...em)** becomes valid.

During the process of inference each type parameter **Xi** is either *fixed* to a particular type **Si** or *unfixed* with an associated set of *bounds.* Each of the bounds is some type **T**. Initially each type variable **Xi** is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

**Note**   When we refer to *delegate types* throughout the following, this should be taken to include also types of the form **Expression<D>** where **D** is a delegate type. The argument and return types of **Expression<D>** are those of **D**.

**Note**   Type inference takes place not only when a generic method is called. Type inference for conversion of method groups is described in §[26.3.3.12](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10l) and finding the best common type of a set of expressions is described in §[26.3.3.13](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10m).

**26.3.3.1 The first phase**

For each of the method arguments **ei**:

* An explicit argument type inference (§[26.3.3.7](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10g)) is made from **ei** with type **Ti** if **ei** is a lambda expression, an anonymous method, or a method group.
* An output type inference (§[26.3.3.6](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10f)) is made from **ei** with type **Ti** if **ei** is not a lambda expression, an anonymous method, or a method group.

**26.3.3.2 The second phase**

All unfixed type variables **Xi** that *depend on* (§[26.3.3.5](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10e)) no **Xj** are fixed (§[26.3.3.10](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10j)).

If no such type variables exist, all unfixed type variables **Xi** are fixed for which all of the following hold:

* There is at least one type variable **Xj** that depends on **Xi**.
* **Xi** has a non-empty set of bounds.

If no such type variables exist and there are still unfixed type variables, type inference fails. If no further unfixed type variables exist, type inference succeeds. Otherwise, for all arguments **ei** with corresponding argument type **Ti** where the output types (§[26.3.3.4](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10d)) contain unfixed type variables **Xj** but the input types (§[26.3.3.3](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10c)) do not, an output type inference (§[26.3.3.6](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10f)) is made for **ei** with type **Ti**. Then the second phase is repeated.

**26.3.3.3 Input types**

If **e** is a method group or implicitly typed lambda expression and **T** is a delegate type then all the argument types of **T** are *input types* *of* **e** *with type* **T**.

**26.3.3.4 Output types**

If **e** is a method group, an anonymous method, a statement lambda or an expression lambda and **T** is a delegate type then the return type of **T** is an *output type* *of* **e** *with type* **T**.

**26.3.3.5 Dependence**

An unfixed type variable **Xi** *depends directly on* an unfixed type variable **Xj** if for some argument **ek** with type **Tk** **Xj** occurs in an input type of **ek** with type **Tk** and **Xi** occurs in an output type of **ek** with type **Tk**.

**Xj** *depends on* **Xi** if **Xj** depends directly on **Xi** or if **Xi** depends directly on **Xk** and **Xk** depends on **Xj**. Thus "depends on" is the transitive but not reflexive closure of "depends directly on".

**26.3.3.6 Output type inferences**

An output type inference is made from an expression **e** with type **T** in the following way:

* If **e** is a lambda or anonymous method with inferred return type **U** (§[26.3.3.11](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10k)) and **T** is a delegate type with return type **Tb**, then a lower-bound inference (§[26.3.3.9](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10i)) is made from **U** for **Tb**.
* Otherwise, if **e** is a method group and **T** is a delegate type with parameter types **T1...Tk** and overload resolution of **e** with the types **T1...Tk** yields a single method with return type **U**, then a lower-bound inference is made from **U** for **Tb**.
* Otherwise, if **e** is an expression with type **U**, then a lower-bound inference is made from **U** for **T**.
* Otherwise, no inferences are made.

**26.3.3.7 Explicit argument type inferences**

An explicit argument type inference is made from an expression **e** with type **T** in the following way:

* If **e** is an explicitly typed lambda expression or anonymous method with argument types **U1...Uk** and **T** is a delegate type with parameter types **V1...Vk** then for each **Ui** an exact inference (§[26.3.3.8](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic10h)) is made from **Ui** for the corresponding **Vi.**

**26.3.3.8 Exact inferences**

An exact inference from a type **U** for a type **V** is made as follows:

* If **V** is one of the unfixed **Xi** then **U** is added to the set of bounds for **Xi**.
* Otherwise, if **U** is an array type **Ue[...]** and **V** is an array type **Ve[...]** of the same rank then an exact inference from **Ue** to **Ve** is made.
* Otherwise, if **V** is a constructed type **C<V1...Vk>** and **U** is a constructed type **C<U1...Uk>** then an exact inference is made from each **Ui** to the corresponding **Vi**.
* Otherwise, no inferences are made.

**26.3.3.9 Lower-bound inferences**

A lower-bound inference from a type **U** for a type **V** is made as follows:

* If **V** is one of the unfixed **Xi** then **U** is added to the set of bounds for **Xi**.
* Otherwise if **U** is an array type **Ue[...]** and **V** is either an array type **Ve[...]** of the same rank, or if **U** is a one-dimensional array type **Ue[]**and **V** is one of **IEnumerable<Ve>**, **ICollection<Ve>** or **IList<Ve>** then:
  + If **Ue** is known to be a reference type then a lower-bound inference from **Ue** to **Ve** is made.
  + Otherwise, an exact inference from **Ue** to **Ve** is made.
* Otherwise if **V** is a constructed type **C<V1...Vk>** and there is a unique set of types **U1...Uk** such that a standard implicit conversion exists from **U** to **C<U1...Uk>** then an exact inference is made from each **Ui** for the corresponding **Vi**.
* Otherwise, no inferences are made.

**26.3.3.10 Fixing**

An unfixed type variable **Xi** with a set of bounds is *fixed* as follows.

* The set of *candidate types* **Uj** starts out as the set of all types in the set of bounds for **Xi**.
* We then examine each bound for **Xi** in turn. For each bound **U** of **X** all types **Uj** to which there is *not* a standard implicit conversion from **U** are removed from the candidate set.
* If among the remaining candidate types **Uj** there is a unique type **V** from which there is a standard implicit conversion to all the other candidate types, then **Xi** is fixed to **V**.
* Otherwise, type inference fails.

**26.3.3.11 Inferred return type**

For purposes of type inference and overload resolution, the ***inferred return type*** of a lambda expression **or** anonymous method **e** is determined as follows:

* If the body of **e** is an expression, the type of that expression is the inferred return type of **e**.
* If the body of **e** is a statement block, if the set of expressions in the block's **return** statements has a best common type, and if that type is not the null type, then that type is the inferred return type of **L**.
* Otherwise, a return type cannot be inferred for **L**.

As an example of type inference involving lambda expressions, consider the **Select** extension method declared in the **System.Linq.Enumerable** class:

namespace System.Linq

{

public static class Enumerable

{

public static IEnumerable<TResult> Select<TSource,TResult>(

this IEnumerable<TSource> source,

Func<TSource,TResult> selector)

{

foreach (TSource element in source) yield return selector(element);

}

}

}

Assuming the **System.Linq** namespace was imported with a **using** clause, and given a class **Customer** with a **Name** property of type **string**, the **Select** method can be used to select the names of a list of customers:

List<Customer> customers = GetCustomerList();

IEnumerable<string> names = customers.Select(c => c.Name);

The extension method invocation (§[26.2.3](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic6)) of **Select** is processed by rewriting the invocation to a static method invocation:

IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the **customers** argument is related to the **source** parameter, inferring **T** to be **Customer**. Then, using the lambda expression type inference process described above, **c** is given type **Customer**, and the expression **c.Name** is related to the return type of the **selector** parameter, inferring **S** to be **string**. Thus, the invocation is equivalent to:

Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)

and the result is of type **IEnumerable<string>**.

The following example demonstrates how lambda expression type inference allows type information to "flow" between arguments in a generic method invocation. Given the method:

static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {

return f2(f1(value));

}

type inference for the invocation

double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);

proceeds as follows: First, the argument **"1:15:30"** is related to the **value** parameter, inferring **X** to be **string**. Then, the parameter of the first lambda expression, **s**, is given the inferred type **string**, and the expression **TimeSpan.Parse(s)** is related to the return type of **f1**, inferring **Y** to be **System.TimeSpan**. Finally, the parameter of the second lambda expression, **t**, is given the inferred type **System.TimeSpan**, and the expression **t.TotalSeconds** is related to the return type of **f2**, inferring **Z** to be **double**. Thus, the result of the invocation is of type **double**.

**26.3.3.12 Type inference for conversion of method groups**

Similar to calls of generic methods, type inference must also be applied when a method group **M** containing a generic method is assigned to a given delegate type **D**. Given a method

Tr M<X1...Xn>(T1 x1 ... Tm xm)

and the method group **M** being assigned to the delegate type **D** the task of type inference is to find type arguments **S1...Sn** so that the expression:

M<S1...Sn>

becomes assignable to **D**.

Unlike the type inference algorithm for generic method calls, in this case there are only argument *types*, no argument *expressions*. In particular, there are no lambda expressions and hence no need for multiple phases of inference.

Instead, all **Xi** are considered unfixed, and a lower-bound inference is made from each argument type **Uj** of **D** to the corresponding parameter type **Tj** of **M**. If for any of the **Xi** no bounds were found, type inference fails. Otherwise, all **Xi** are fixed to corresponding **Si**, which are the result of type inference.

**26.3.3.13 Finding the best common type of a set of expressions**

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous methods and statement lambdas are found in this way.

Intuitively, given a set of expressions **e1...em** this inference should be equivalent to calling a method

Tr M<X>(X x1 ... X xm)

with the **ei** as arguments.

More precisely, the inference starts out with an unfixed type variable **X**. Output type inferences are then made from each **ei** with type **X**. Finally, **X** is fixed and the resulting type **S** is the resulting common type for the expressions.

**26.3.4 Overload Resolution**

Lambda expressions in an argument list affect overload resolution in certain situations. Please refer to §7.4.2.3 for the exact rules.

The following example illustrates the effect of lambdas on overload resolution.

class ItemList<T>: List<T>

{

public int Sum(Func<T,int> selector) {

int sum = 0;

foreach (T item in this) sum += selector(item);

return sum;

}

public double Sum(Func<T,double> selector) {

double sum = 0;

foreach (T item in this) sum += selector(item);

return sum;

}

}

The **ItemList<T>** class has two **Sum** methods. Each takes a **selector** argument, which extracts the value to sum over from a list item. The extracted value can be either an **int** or a **double** and the resulting sum is likewise either an **int** or a **double**.

The **Sum** methods could for example be used to compute sums from a list of detail lines in an order.

class Detail

{

public int UnitCount;

public double UnitPrice;

...

}

void ComputeSums() {

ItemList<Detail> orderDetails = GetOrderDetails(...);

int totalUnits = orderDetails.Sum(d => d.UnitCount);

double orderTotal = orderDetails.Sum(d => d.UnitPrice \* d.UnitCount);

...

}

In the first invocation of **orderDetails.Sum**, both **Sum** methods are applicable because the lambda expression **d => d.UnitCount** is compatible with both **Func<Detail,int>** and **Func<Detail,double>**. However, overload resolution picks the first **Sum** method because the conversion to **Func<Detail,int>** is better than the conversion to **Func<Detail,double>**.

In the second invocation of **orderDetails.Sum**, only the second **Sum** method is applicable because the lambda expression **d => d.UnitPrice \* d.UnitCount** produces a value of type **double**. Thus, overload resolution picks the second **Sum** method for that invocation.

**26.4 Object and Collection Initializers**

An object creation expression (§7.5.10.1) may include an object or collection initializer which initializes the members of the newly created object or the elements of the newly created collection.

object-creation-expression:

*new* type *(* argument-listopt *)* object-or-collection-initializeropt

*new* type object-or-collection-initializer

object-or-collection-initializer:

object-initializer

collection-initializer

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Execution of an object creation expression that includes an object or collection initializer consists of first invoking the instance constructor and then performing the member or element initializations specified by the object or collection initializer.

It is not possible for an object or collection initializer to refer to the object instance being initialized.

In order to correctly parse object and collection initializers with generics, the disambiguating list of tokens in §20.6.5 must be augmented with the **}** token.

**26.4.1 Object Initializers**

An object initializer specifies values for one or more fields or properties of an object.

object-initializer:

*{* member-initializer-listopt *}*

*{* member-initializer-list *,* *}*

member-initializer-list:

member-initializer

member-initializer-list *,* member-initializer

member-initializer:

identifier = initializer-value

initializer-value:

expression

object-or-collection-initializer

An object initializer consists of a sequence of member initializers, enclosed by **{** and **}** tokens and separated by commas. Each member initializer must name an accessible field or property of the object being initialized, followed by an equals sign and an expression or an object or collection initializer. It is an error for an object initializer to include more than one member initializer for the same field or property. It is not possible for the object initializer to refer to the newly created object it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment (§7.13.1) to the field or property.

A member initializer that specifies an object initializer after the equals sign is a *nested object initializer, i.e., a*n initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the nested object initializer are treated as assignments to members of the field or property. Nested object initializers cannot be applied to properties with a value type, or to read-only fields with a value type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the field or property, the elements given in the initializer are added to the collection referenced by the field or property. The field or property must be of a collection type that satisfies the requirements specified in §[26.4.2](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic14).

The following class represents a point with two coordinates:

public class Point

{

int x, y;

public int X { get { return x; } set { x = value; } }

public int Y { get { return y; } set { y = value; } }

}

An instance of **Point** can be created and initialized as follows:

var a = new Point { X = 0, Y = 1 };

which has the same effect as

var \_\_a = new Point();

\_\_a.X = 0;

\_\_a.Y = 1;

var a = \_\_a;

where **\_\_a** is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

public class Rectangle

{

Point p1, p2;

public Point P1 { get { return p1; } set { p1 = value; } }

public Point P2 { get { return p2; } set { p2 = value; } }

}

An instance of **Rectangle** can be created and initialized as follows:

var r = new Rectangle {

P1 = new Point { X = 0, Y = 1 },

P2 = new Point { X = 2, Y = 3 }

};

which has the same effect as

var \_\_r = new Rectangle();

var \_\_p1 = new Point();

\_\_p1.X = 0;

\_\_p1.Y = 1;

\_\_r.P1 = \_\_p1;

var \_\_p2 = new Point();

\_\_p2.X = 2;

\_\_p2.Y = 3;

\_\_r.P2 = \_\_p2;

var r = \_\_r;

where **\_\_r**, **\_\_p1** and **\_\_p2** are temporary variables that are otherwise invisible and inaccessible.

If the **Rectangle** constructor allocates the two embedded **Point** instances

public class Rectangle

{

Point p1 = new Point();

Point p2 = new Point();

public Point P1 { get { return p1; } }

public Point P2 { get { return p2; } }

}

the following construct can be used to initialize the embedded **Point** instances instead of assigning new instances:

var r = new Rectangle {

P1 = { X = 0, Y = 1 },

P2 = { X = 2, Y = 3 }

};

which has the same effect as

var \_\_r = new Rectangle();

\_\_r.P1.X = 0;

\_\_r.P1.Y = 1;

\_\_r.P2.X = 2;

\_\_r.P2.Y = 3;

var r = \_\_r;

**26.4.2 Collection Initializers**

A collection initializer specifies the elements of a collection.

collection-initializer:

*{* element-initializer-list *}*

*{* element-initializer-list *,* *}*

element-initializer-list:

element-initializer

element-initializer-list *,* element-initializer

element-initializer:

non-assignment-expression

*{* expression-list *}*

A collection initializer consists of a sequence of element initializers, enclosed by **{** and **}** tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by **{** and **}** tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment expression, to avoid ambiguity with member initializers. The *non-assignment-expression* production is defined in §[26.3](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic7).

The following is an example of an object creation expression that includes a collection initializer:

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

The collection object to which a collection initializer is applied must be of a type that implements **System.Collections.IEnumerable** or a compile-time error occurs. For each specified element in order, the collection initializer invokes the Add method on the target object with the expression list of the element initializer, applying normal overload resolution for each invocation.

The following class represents a contact with a name and a list of phone numbers:

public class Contact

{

string name;

List<string> phoneNumbers = new List<string>();

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

public List<string> PhoneNumbers { get { return phoneNumbers; } }

}

A **List<Contact>** can be created and initialized as follows:

var contacts = new List<Contact> {

new Contact {

Name = "Chris Smith",

PhoneNumbers = { "206-555-0101", "425-882-8080" }

},

new Contact {

Name = "Bob Harris",

PhoneNumbers = { "650-555-0199" }

}

};

which has the same effect as

var contacts = new List<Contact>();

var \_\_c1 = new Contact();

\_\_c1.Name = "Chris Smith";

\_\_c1.PhoneNumbers.Add("206-555-0101");

\_\_c1.PhoneNumbers.Add("425-882-8080");

contacts.Add(\_\_c1);

var \_\_c2 = new Contact();

\_\_c2.Name = "Bob Harris";

\_\_c2.PhoneNumbers.Add("650-555-0199");

contacts.Add(\_\_c2);

where **\_\_c1** and **\_\_c2** are temporary variables that are otherwise invisible and inaccessible.

**26.5 Anonymous Types**

C# 3.0 permits the **new** operator to be used with an anonymous object initializer to create an object of an anonymous type.

primary-no-array-creation-expression:

...

anonymous-object-creation-expression

anonymous-object-creation-expression:

*new* anonymous-object-initializer

anonymous-object-initializer:

*{* member-declarator-listopt *}*

*{* member-declarator-list *,* *}*

member-declarator-list:

member-declarator

member-declarator-list *,* member-declarator

member-declarator:

simple-name

member-access

identifier = expression

An anonymous object initializer declares an anonymous type and returns an instance of that type. An anonymous type is a nameless class type that inherits directly from **object**. The members of an anonymous type are a sequence of read/write properties inferred from the object initializer(s) used to create instances of the type. Specifically, an anonymous object initializer of the form

new { *p1* = *e1* , *p2* = *e2* , *...* *pn* = *en* }

declares an anonymous type of the form

class \_\_Anonymous1

{

private *T1* *f1* ;

private *T2* *f2* ;

*...*

private *Tn* *fn* ;

public *T1* *p1* { get { return *f1* ; } set { *f1* = value ; } }

public *T2* *p2* { get { return *f2* ; } set { *f2* = value ; } }

*...*

public *T1* *p1* { get { return *f1* ; } set { *f1* = value ; } }

}

where each **Tx** is the type of the corresponding expression **ex**. It is a compile-time error for an expression in an anonymous object initializer to be of the null type or an unsafe type.

The name of an anonymous type is automatically generated by the compiler and cannot be referenced in program text.

Within the same program, two anonymous object initializers that specify a sequence of properties of the same names and compile-time types in the same order will produce instances of the same anonymous type. (This definition includes the order of the properties because it is observable and material in certain circumstances, such as reflection).

In the example

var p1 = new { Name = "Lawnmower", Price = 495.00 };

var p2 = new { Name = "Shovel", Price = 26.95 };

p1 = p2;

the assignment on the last line is permitted because **p1** and **p2** are of the same anonymous type.

The **Equals** and **GetHashcode** methods on anonymous types are defined in terms of the **Equals** and **GetHashcode** of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name (§7.5.2) or a member access (§7.5.4). This is called a ***projection initializer*** and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

*identifier expr* . *identifier*

are precisely equivalent to the following, respectively:

*identifer* = *identifier* *identifier* = *expr* . *identifier*

Thus, in a projection initializer the *identifier* selects both the value and the field or property to which the value is assigned. Intuitively, a projection initializer projects not just a value, but also the name of the value.

**26.6 Implicitly Typed Arrays**

The syntax of array creation expressions (§7.5.10.2) is extended to support implicitly typed array creation expressions:

array-creation-expression:

...

*new* *[* *]* array-initializer

In an implicitly typed array creation expression, the type of the array instance is inferred from the elements specified in the array initializer. Specifically, the set formed by the types of the expressions in the array initializer must contain exactly one type to which each type in the set is implicitly convertible, and if that type is not the null type, an array of that type is created. If exactly one type cannot be inferred, or if the inferred type is the null type, a compile-time error occurs.

The following are examples of implicitly typed array creation expressions:

var a = new[] { 1, 10, 100, 1000 }; // int[]

var b = new[] { 1, 1.5, 2, 2.5 }; // double[]

var c = new[] { "hello", null, "world” }; // string[]

var d = new[] { 1, "one", 2, "two" }; // Error

The last expression causes a compile-time error because neither **int** nor **string** is implicitly convertible to the other. An explicitly typed array creation expression must be used in this case, for example specifying the type to be **object[]**. Alternatively, one of the elements can be cast to a common base type, which would then become the inferred element type.

Implicitly typed array creation expressions can be combined with anonymous object initializers to create anonymously typed data structures. For example:

var contacts = new[] {

new {

Name = "Chris Smith",

PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }

},

new {

Name = "Bob Harris",

PhoneNumbers = new[] { "650-555-0199" }

}

};

**26.7 Query Expressions**

***Query expressions*** provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

query-expression:

from-clause query-body

from-clause:

from typeopt identifier in expression

query-body:

query-body-clausesopt select-or-group-clause query-continuationopt

query-body-clauses:

query-body-clause

query-body-clauses query-body-clause

query-body-clause:

from-clause

let-clause

where-clause

join-clause

join-into-clause

orderby-clause

let-clause:

let identifier = expression

where-clause:

where boolean-expression

join-clause:

join typeopt identifier in expression on expression equals expression

join-into-clause:

join typeopt identifier in expression on expression equals expression into identifier

orderby-clause:

orderby orderings

orderings:

ordering

orderings , ordering

ordering:

expression ordering-directionopt

ordering-direction:

ascending

descending

select-or-group-clause:

select-clause

group-clause

select-clause:

select expression

group-clause:

group expression by expression

query-continuation:

into identifier query-body

A *query-expression* is classified as a *non-assignment-expression*, the definition of which occurs in §[26.3](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic7).

A query expression begins with a **from** clause and ends with either a **select** or **group** clause. The initial **from** clause can be followed by zero or more **from**, **let**, **where** or **join** clauses. Each **from** clause is a generator introducing a range variable ranging over a sequence. Each **let** clause computes a value and introduces an identifier representing that value. Each **where** clause is a filter that excludes items from the result. Each **join** clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each **orderby** clause reorders items according to specified criteria. The final **select** or **group** clause specifies the shape of the result in terms of the range variable(s). Finally, an **into** clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

**Ambiguities in query expressions**

Query expressions contain a number of new contextual keywords, i.e., identifiers that have special meaning in a given context. Specifically these are: **from**, **join**, **on**, **equals**, **into**, **let**, **orderby**, **ascending**, **descending**, **select**, **group** and **by**. In order to avoid ambiguities caused by mixed use of these identifiers as keywords or simple names in query expressions, they are considered keywords anywhere within a query expression.

For this purpose, a query expression is any expression starting with "**from** *identifier*" followed by any token except "**;**", "**=**" or "**,**".

In order to use these words as identifiers within a query expression, they can be prefixed with "**@**" (§2.4.2).

**26.7.1 Query Expression Translation**

The C# 3.0 language does not specify the exact execution semantics of query expressions. Rather, C# 3.0 translates query expressions into invocations of methods that adhere to the *query expression pattern*. Specifically, query expressions are translated into invocations of methods named **Where**, **Select**, **SelectMany**, **Join**, **GroupJoin**, **OrderBy**, **OrderByDescending**, **ThenBy**, **ThenByDescending**, **GroupBy**, and **Cast** that are expected to have particular signatures and result types, as described in §[26.7.2](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic19). These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of precedence: each section assumes that the translations in the preceding sections have been performed exhaustively.

Certain translations inject range variables with *transparent identifiers* denoted by **\***. The special properties of transparent identifiers are discussed further in §[26.7.1.7](http://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic18g).

**26.7.1.1 Select and groupby clauses with continuations**

A query expression with a continuation

from *...* into *x* *...*

is translated into

from *x* in ( from *...* ) *...*

The translations in the following sections assume that queries have no **into** continuations.

The example

from c in customers

group c by c.Country into g

select new { Country = g.Key, CustCount = g.Count() }

is translated into

from g in

from c in customers

group c by c.Country

select new { Country = g.Key, CustCount = g.Count() }

the final translation of which is

customers.

GroupBy(c => c.Country).

Select(g => new { Country = g.Key, CustCount = g.Count() })

**26.7.1.2 Explicit range variable types**

A **from** clause that explicitly specifies a range variable type

from *T* *x* in *e*

is translated into

from *x* in ( *e* ) . Cast < *T* > ( )

A **join** clause that explicitly specifies a range variable type

join *T* *x* in *e* on *k1* equals *k2*

is translated into

join *x* in ( *e* ) . Cast < *T* > ( ) on *k1* equals *k2*

The translations in the following sections assume that queries have no explicit range variable types.

The example

from Customer c in customers

where c.City == "London"

select c

is translated into

from c in customers.Cast<Customer>()

where c.City == "London"

select c

the final translation of which is

customers.

Cast<Customer>().

Where(c => c.City == "London")

Explicit range variable types are useful for querying collections that implement the non-generic **IEnumerable** interface, but not the generic **IEnumerable<T>** interface. In the example above, this would be the case if **customers** were of type **ArrayList**.

**26.7.1.3 Degenerate query expressions**

A query expression of the form

from *x* in *e* select *x*

is translated into

( *e* ) . Select ( *x* => *x* )

The example

from c in customers

select c

Is translated into

customers.Select(c => c)

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. It is important however to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling **Select** on the source. It is then up to the implementers of **Select** and other query operators to ensure that these methods never return the source object itself.

**26.7.1.4 From, let, where, join and orderby clauses**

A query expression with a second **from** clause followed by a **select** clause

from *x1* in *e1*

from *x2* in *e2*

select *v*

is translated into

( *e1* ) . SelectMany( *x1* => *e2* , ( *x1* , *x2* ) => *v* )

A query expression with a second **from** clause followed by something other than a **select** clause:

from *x1* in *e1*

from *x2* in *e2*

...

is translated into

from \* in ( *e1* ) . SelectMany( *x1* => *e2* , ( *x1* , *x2* ) => new { *x1* , *x2* } )

...

A query expression with a **let** clause

from *x* in *e*

let *y* = *f*

*...*

is translated into

from \* in ( *e* ) . Select ( *x* => new { *x* , *y* = *f* } )

...

A query expression with a **where** clause

from *x* in *e*

where *f*

*...*

is translated into

from *x* in ( *e* ) . Where ( *x* => *f* )

...

A query expression with a **join** clause without an **into** followed by a **select** clause

from *x1* in *e1*

join *x2* in *e2* on *k1* equals *k2*

select *v*

is translated into

( *e1* ) . Join( *e2* , *x1* => *k1* , *x2* => *k2* , ( *x1* , *x2* ) => *v* )

A query expression with a **join** clause without an **into** followed by something other than a **select** clause

from *x1* in *e1*

join *x2* in *e2* on *k1* equals *k2*

...

is translated into

from \* in ( *e1* ) . Join(

*e2* , *x1* => *k1* , *x2* => *k2* , ( *x1* , *x2* ) => new { *x1* , *x2* })

...

A query expression with a **join** clause with an **into** followed by a **select** clause

from *x1* in *e1*

join *x2* in *e2* on *k1* equals *k2* into *g*

select *v*

is translated into

( *e1* ) . GroupJoin( *e2* , *x1* => *k1* , *x2* => *k2* , ( *x1* , *g* ) => *v* )

A query expression with a **join** clause with an **into** followed by something other than a **select** clause

from *x1* in *e1*

join *x2* in *e2* on *k1* equals *k2* into *g*

...

is translated into

from \* in ( *e1* ) . GroupJoin(

*e2* , *x1* => *k1* , *x2* => *k2* , ( *x1* , *g* ) => new { *x1* , *g* })

...

A query expression with an **orderby** clause

from *x* in *e*

orderby *k1* , *k2* , ... ,*kn*

*...*

is translated into

from *x* in ( *e* ) .

OrderBy ( *x* => *k1* ) .

ThenBy ( *x* => *k2* ) .

*...* .

ThenBy ( *x* => *kn* )

...

If an ordering clause specifies a **descending** direction indicator, an invocation of **OrderByDescending** or **ThenByDescending** is produced instead.

The following translations assume that there are no **let**, **where**, **join** or **orderby** clauses, and no more than the one initial **from** clause in each query expression.

The example

from c in customers

from o in c.Orders

select new { c.Name, o.OrderID, o.Total }

is translated into

customers.

SelectMany(c => c.Orders,

(c,o) => new { c.Name, o.OrderID, o.Total }

)

The example

from c in customers

from o in c.Orders

orderby o.Total descending

select new { c.Name, o.OrderID, o.Total }

is translated into

from \* in customers.

SelectMany(c => c.Orders, (c,o) => new { c, o })

orderby o.Total descending

select new { c.Name, o.OrderID, o.Total }

the final translation of which is

customers.

SelectMany(c => c.Orders, (c,o) => new { c, o }).

OrderByDescending(x => x.o.Total).

Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })

where **x** is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

from o in orders

let t = o.Details.Sum(d => d.UnitPrice \* d.Quantity)

where t >= 1000

select new { o.OrderID, Total = t }

is translated into

from \* in orders.

Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) })

where t >= 1000

select new { o.OrderID, Total = t }

the final translation of which is

orders.

Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) }).

Where(x => x.t >= 1000).

Select(x => new { x.o.OrderID, Total = x.t })

where **x** is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

from c in customers

join o in orders on c.CustomerID equals o.CustomerID

select new { c.Name, o.OrderDate, o.Total }

is translated into

customers.Join(orders, c => c.CustomerID, o => o.CustomerID,

(c, o) => new { c.Name, o.OrderDate, o.Total })

The example

from c in customers

join o in orders on c.CustomerID equals o.CustomerID into co

let n = co.Count()

where n >= 10

select new { c.Name, OrderCount = n }

is translated into

from \* in customers.

GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,

(c, co) => new { c, co })

let n = co.Count()

where n >= 10

select new { c.Name, OrderCount = n }

the final translation of which is

customers.

GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,

(c, co) => new { c, co }).

Select(x => new { x, n = x.co.Count() }).

Where(y => y.n >= 10).

Select(y => new { y.x.c.Name, OrderCount = y.n)

where **x** and **y** are compiler generated identifiers that are otherwise invisible and inaccessible.

The example

from o in orders

orderby o.Customer.Name, o.Total descending

select o

has the final translation

orders.

OrderBy(o => o.Customer.Name).

ThenByDescending(o => o.Total)

**26.7.1.5 Select clauses**

A query expression of the form

from *x* in *e* select *v*

is translated into

( *e* ) . Select ( *x* => *v* )

except when **v** is the identifier **x**, the translation is simply

( *e* )

For example

from c in customers.Where(c => c.City == "London")

select c

is simply translated into

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

**26.7.1.6 Groupby clauses**

A query expression of the form

from *x* in *e* group *v* by *k*

is translated into

( *e* ) . GroupBy ( *x* => *k* , *x* => *v* )

except when *v* is the identifier **x**, the translation is

( *e* ) . GroupBy ( *x* => *k* )

The example

from c in customers

group c.Name by c.Country

is translated into

customers.

GroupBy(c => c.Country, c => c.Name)

**26.7.1.7 Transparent identifiers**

Certain translations inject range variables with ***transparent identifiers*** denoted by **\***. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into lambda expressions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

* When a transparent identifier occurs as a parameter in a lambda expression, the members of the associated anonymous type are automatically in scope in the body of the lambda expression.
* When a member with a transparent identifier is in scope, its members are in scope as well.
* When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.

The example

from c in customers

from o in c.Orders

orderby o.Total descending

select new { c.Name, o.Total }

is translated into

from \* in

from c in customers

from o in c.Orders

select new { c, o }

orderby o.Total descending

select new { c.Name, o.Total }

which is further translated into

customers.

SelectMany(c => c.Orders.Select(o => new { c, o })).

OrderByDescending(\* => o.Total).

Select(\* => new { c.Name, o.Total })

which, when transparent identifiers are erased, is equivalent to

customers.

SelectMany(c => c.Orders.Select(o => new { c, o })).

OrderByDescending(x => x.o.Total).

Select(x => new { x.c.Name, x.o.Total })

where **x** is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

from c in customers

join o in orders on c.CustomerID equals o.CustomerID

join d in details on o.OrderID equals d.OrderID

join p in products on d.ProductID equals p.ProductID

select new { c.Name, o.OrderDate, p.ProductName }

is translated into

from \* in

from \* in

from \* in

from c in customers

join o in orders o c.CustomerID equals o.CustomerID

select new { c, o }

join d in details on o.OrderID equals d.OrderID

select new { \*, d }

join p in products on d.ProductID equals p.ProductID

select new { \*, p }

select new { c.Name, o.OrderDate, p.ProductName }

which is further reduced to

customers.

Join(orders, c => c.CustomerID, o => o.CustomerID,

(c, o) => new { c, o }).

Join(details, \* => o.OrderID, d => d.OrderID,

(\*, d) => new { \*, d }).

Join(products, \* => d.ProductID, p => p.ProductID,

(\*, p) => new { \*, p }).

Select(\* => new { c.Name, o.OrderDate, p.ProductName })

the final translation of which is

customers.

Join(orders, c => c.CustomerID, o => o.CustomerID,

(c, o) => new { c, o }).

Join(details, x => x.o.OrderID, d => d.OrderID,

(x, d) => new { x, d }).

Join(products, y => y.d.ProductID, p => p.ProductID,

(y, p) => new { y, p }).

Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })

where **x**, **y**, and **z** are compiler generated identifiers that are otherwise invisible and inaccessible.

**26.7.2 The Query Expression Pattern**

The ***Query Expression Pattern*** establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the methods of the pattern can be implemented as instance methods or as extension methods because the two have the same invocation syntax, and the methods can request delegates or expression trees because lambda expressions are convertible to both.

The recommended shape of a generic type **C<T>** that supports the query expression pattern is shown below. A generic type is used in order to illustrate the proper relationships between parameter and result types, but it is possible to implement the pattern for non-generic types as well.

delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C

{

public C<T> Cast<T>();

}

class C<T>

{

public C<T> Where(Func<T,bool> predicate);

public C<U> Select<U>(Func<T,U> selector);

public C<U> SelectMany<U,V>(Func<T,C<U>> selector,

Func<T,U,V> resultSelector);

public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,

Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,

Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

public O<T> OrderBy<K>(Func<T,K> keySelector);

public O<T> OrderByDescending<K>(Func<T,K> keySelector);

public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,

Func<T,E> elementSelector);

}

class O<T> : C<T>

{

public O<T> ThenBy<K>(Func<T,K> keySelector);

public O<T> ThenByDescending<K>(Func<T,K> keySelector);

}

class G<K,T> : C<T>

{

public K Key { get; }

}

The methods above use the generic delegate types **Func<T1, R>** and **Func<T1, T2, R>**, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between **C<T>** and **O<T>** which ensures that the **ThenBy** and **ThenByDescending** methods are available only on the result of an **OrderBy** or **OrderByDescending**. Also notice the recommended shape of the result of **GroupBy**—a sequence of sequences, where each inner sequence has an additional **Key** property.

The Standard Query Operators (described in a separate specification) provide an implementation of the query operator pattern for any type that implements the **System.Collections.Generic.IEnumerable<T>** interface.

**26.8 Expression Trees**

Expression trees permit lambda expressions to be represented as data structures instead of executable code. A lambda expression that is convertible to a delegate type **D** is also convertible to an expression tree of type **System.Query.Expression<D>**. Whereas the conversion of a lambda expression to a delegate type causes executable code to be generated and referenced by a delegate, conversion to an expression tree type causes code that creates an expression tree instance to be emitted. Expression trees are efficient in-memory data representations of lambda expressions and make the structure of the expression transparent and explicit.

The following example represents a lambda expression both as executable code and as an expression tree. Because a conversion exists to **Func<int,int>**, a conversion also exists to **Expression<Func<int,int>>**.

Func<int,int> f = x => x + 1; // Code

Expression<Func<int,int>> e = x => x + 1; // Data

Following these assignments, the delegate **f** references a method that returns **x + 1**, and the expression tree **e** references a data structure that describes the expression **x + 1**.

**26.8.1 Overload Resolution**

For the purpose of overload resolution there are special rules regarding the **Expression<D>** types. Specifically the following rule is added to the definition of betterness:

* **Expression<D1>** is better than **Expression<D2>** if and only if **D1** is better than **D2**

**Note**   that there is no betterness rule between **Expression<D>** and delegate types.

**26.9 Automatically Implemented Properties**

Oftentimes properties are implemented by trivial use of a backing field, as in the following example:

public Class Point {

private int x;

private int y;

public int X { get { return x; } set { x = value; } }

public int Y { get { return y; } set { y = value; } }

}

Automatically implemented (auto-implemented) properties automate this pattern. More specifically, non-abstract property declarations are allowed to have semicolon accessor bodies. Both accessors must be present and both must have semicolon bodies, but they can have different accessibility modifiers. When a property is specified like this, a backing field will automatically be generated for the property, and the accessors will be implemented to read from and write to that backing field. The name of the backing field is compiler generated and inaccessible to the user.

The following declaration is equivalent to the example above:

public Class Point {

public int X { get; set; }

public int Y { get; set; }

}

Because the backing field is inaccessible, it can be read and written only through the property accessors. This means that auto-implemented read-only or write-only properties do not make sense, and are disallowed. It is however possible to set the access level of each accessor differently. Thus, the effect of a read-only property with a private backing field can be mimicked like this:

Public class ReadOnlyPoint {

public int X { get; private set; }

public int Y { get; private set; }

public ReadOnlyPoint(int x, int y) { X = x; Y = y; }

}

This restriction also means that definite assignment of struct types with auto-implemented properties can only be achieved using the standard constructor of the struct, since assigning to the property itself requires the struct to be definitely assigned.