Appendix A

C# Version 3.0 Specification

Appendix A
C# Version 3.0 Specification

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 1.2

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 cannot be an object or collection initializer by itself, but it can be a new expression that includes an object or collection initializer. The compile-time type of the initializer expression cannot be null.

If the local variable declaration includes multiple declarators, the initializers must all have the same compile-time type.
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
```

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; however, a warning is generated to call attention to the ambiguity. 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 and the *resource-acquisition* of a using statement can be an implicitly typed local variable declaration. Likewise, the iteration variable of a foreach statement 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.

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.

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 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;
                }
        }
}
```

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

Importing extension methods

Extension methods are imported through *using-namespace-directive*s. In addition to importing the types contained in a namespace, a *using-namespace-directive* imports all extension methods in all static classes in the namespace. In effect, imported 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)
```

Extension method invocations

The detailed rules for extension method invocation are described in the following. In a method invocation 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 accessible extension methods with the name given by *identifier* imported by the namespace declaration's *using-namespace-directive*s. The first method group that yields a non-empty set of candidate methods is the one chosen for the rewritten method invocation. If all attempts yield empty sets of candidate methods, a compile-time error occurs.

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

```
using N1;

namespace N1
{
        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, B's method takes precedence over the first extension method, and C's method takes precedence over both extension methods.

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-list*$_{opt}$) => *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-modifier$_{opt}$ *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 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.

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 C# 2.0 Specification, also applies to lambda expressions. Lambda expressions are a functional superset of anonymous methods, providing the following additional functionality:

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 passed as arguments participate in type argument inference and in method overload resolution.
Lambda expressions with an expression body can be converted to expression trees .

Note

The PDC 2005 Technology Preview compiler does not support lambda expressions with a statement block body. In cases where a statement block body is needed, the C# 2.0 anonymous method syntax must be used.

Lambda expression conversions

Similar to an *anonymous-method-expression*, 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 a lambda-expression L provided:

D and L have the same number of parameters.

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 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 that would be permitted as a *statement-expression*.

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 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 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 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.

Type inference

When a generic method is called without specifying type arguments, a type inference process attempts to infer type arguments for the call. Lambda expressions passed as arguments to the generic method participate in this type inference process.

As a type inference first occurs independently for each argument. In this initial phase, nothing is inferred from arguments that are lambda expressions. However, following the initial phase, additional inferences are made from lambda expressions using an iterative process. Specifically, inferences are made as long as one or more arguments exist for which all of the following are true:

The argument is a lambda expression, in the following called L, from which no inferences have yet been made.
The corresponding parameter's type, in the following called P, is a delegate type with a return type that involves one or more method type parameters.

P and L have the same number of parameters, and each parameter in P has the same modifiers as the corresponding parameter in L, or no modifiers if L has an implicitly typed parameter list.

P's parameter types involve no method type parameters or involve only method type parameters for which a consistent set of inferences have already been made.

> If L has an explicitly typed parameter list, when inferred types are substituted for method type parameters in P, each parameter in P has the same type as the the corresponding parameter in L.

If L has an implicitly typed parameter list, when inferred types are substituted for method type parameters in P and the resulting parameter types are given to the parameters of L, the body of L is a valid expression or statement block.

A return type can be inferred for L, as described below.

For each such argument, inferences are made from that argument by relating the return type of P with the inferred return type of L and the new inferences are added to the accumulated set of inferences. This process is repeated until no further inferences can be made.

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

If the body of L is an expression, the type of that expression is the inferred return type of L.

If the body of L is a statement block, if the set formed by the types of the expressions in the block's return statements contains exactly one type to which each type in the set is implicitly convertible, 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.Query.Sequence class:

```
namespace System.Query
{
        public static class Sequence
        {
public static IEnumerable<S> Select<T,S>(
                        this IEnumerable<T> source,
                        Func<T,S> selector)
```

```
                {
                        foreach (T element in source) yield return
    selector(element);
                }
        }
    }
```

Assuming the System.Query 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 of Select is processed by rewriting the invocation to a static method invocation:

```
IEnumerable<string> names = Sequence.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.

Overload resolution

Lambda expressions in an argument list affect overload resolution in certain situations.

The following rule augments:

> Given a lambda expression L for which an inferred return type exists, an implicit conversion of L to a delegate type D_1 is a better conversion than an implicit conversion of L to a delegate type D_2 if D_1 and D_2 have identical parameter lists and the implicit conversion from L's inferred return type to D_1's return type is a better conversion than the implicit conversion from L's inferred return type to D_2's return type. If these conditions are not true, neither conversion is better.

The following example illustrates the effect of this rule.

```
class ItemList<T>: List<T>
{
    public int Sum<T>(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
    public double Sum<T>(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.

Object and collection initializers

An object creation expression 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-list*$_{opt}$) *object-or-collection-initializer*$_{opt}$
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.

Object initializers

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

object-initializer:
{ *member-initializer-list*$_{opt}$ }
{ *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.

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 specifes an object initializer after the equals sign is an initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the object initializer are treated as assignments to members of the field or property. A property of a value type cannot be initialized using this construct.

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 §0.

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 an 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;
```

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;
```

where __p1 and __p2 are temporary variables that are otherwise invisible and inaccessible.

If Rectangle's 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;
```

Collection initializers

A collection initializer specifies the elements of a collection.

collection-initializer:
{ *element-initializer-list*$_{opt}$ }
{ *element-initializer-list* , }
element-initializer-list:
element-initializer
element-initializer-list , *element-initializer*
element-initializer:
non-assignment-expression

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. To avoid ambiguity with member initializers, element initializers cannot be assignment expressions. The *non-assignment-expression* production .

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.Generic.ICollection<T> for exactly one T. Furthermore, an implicit conversion (§6.1) must exist from the type of each element initializer to T. A compile-time error occurs if these requirements are not satisfied.

A collection initializer invokes the ICollection<T>.Add(T) method for each specified element in order.

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 = { "555-0101", "882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "555-0199" }
    }
};
```

which has the same effect as

```
var contacts = new List<Contact>();
var __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("555-0101");
__c1.PhoneNumbers.Add("882-8080");
contacts.Add(__c1);
var __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("555-0199");
contacts.Add(__c2);
```

where __c1 and __c2 are temporary variables that are otherwise invisible and inaccessible.

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-list*$_{opt}$ }
{ *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 { $p_1 = e_1$, $p_2 = e_2$, ... $p_n = e_n$ }

declares an anonymous type of the form

```
class __Anonymous1
{
    private T1 f1 ;
    ...
    private Tn fn ;
    public T1 p1 { get { return f1 ; } set { f1 = value ; } }
    ...
    public T1 p1 { get { return f1 ; } set { f1 = value ; } }
}
```

where each T_x is the type of the corresponding expression e_x. It is a compile-time error for an expression in an anonymous object initializer to be of the null 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 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.

A member declarator can be abbreviated to a simple name or a member access. 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.

Implicitly typed arrays

The syntax of array creation expressions 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[] { "555-0101", "882-8080" }
        },
        new {
                Name = "Bob Harris",
                PhoneNumbers = new[] { "555-0199" }
        }
};
```

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 *from-generators*
from-generators:
from-generator
from-generators , *from-generator*
from-generator:
identifier in *expression*
query-body:
from-or-where-clauses$_{opt}$ *orderby-clause*$_{opt}$ *select-or-group-clause* *into-clause*$_{opt}$
from-or-where-clauses:
from-or-where-clause
from-or-where-clauses *from-or-where-clause*
from-or-where-clause:
from-clause
where-clause
where-clause:
where *boolean-expression*

orderby-clause:
orderby *ordering-clauses*
ordering-clauses:
ordering-clause
ordering-clauses , *ordering-clause*
ordering-clause:
expression *ordering-direction*$_{opt}$
ordering-direction:
ascending
descending
select-or-group-clause:
select-clause
group-clause
select-clause:
select *expression*
group-clause:
group *expression* by *expression*
into-clause:
into *identifier* *query-body*

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 or where clauses. Each from clause is a generator that introduces an iteration variable ranging over a sequence, and each where clause is a filter that excludes items from the result. The final select or group clause specfies the shape of the result in terms of the iteration variable(s). The select or group clause may be preceded by an orderby clause that specifies an ordering for the result. Finally, an into clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

In a query expression, a from clause with multiple generators is exactly equivalent to multiple consecutive from clauses with a single generator.

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, OrderBy, OrderByDescending, ThenBy, ThenByDescending, and GroupBy that are expected to have particular signatures and result types. 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.

The translation of query expressions is demonstrated through a series of examples in the following. A formal description of the translation rules is provided in a later section.

where clauses

A where clause in a query expression:

```
from c in customers
where c.City == "London"
select c
```

translates to an invocation of a Where method with a synthesized lambda expression created by combining the iteration variable identifier and the expression of the where clause:

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

select clauses

The example in the previous section demonstrates how a select clause that selects the innermost iteration variable is erased by the translation to method invocations.

A select clause that selects something other than the innermost iteration variable:

```
from c in customers
where c.City == "London"
select c.Name
```

translates to an invocation of a Select method with a synthesized lambda expression:

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

group clauses

A group clause:

```
from c in customers
group c.Name by c.Country
```

translates to an invocation of a GroupBy method:

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

orderby clauses

An orderby clause:

```
from c in customers
orderby c.Name
select new { c.Name, c.Phone }
```

translates to an invocation of an OrderBy method, or an OrderByDescending method if a descending direction was specified:

```
customers.
OrderBy(c => c.Name).
Select(c => new { c.Name, c.Phone })
```

Secondary orderings in an orderby clause:

```
from c in customers
orderby c.Country, c.Balance descending
select new { c.Name, c.Country, c.Balance }
```

translate to invocations of ThenBy and ThenByDescending methods:

```
customers.
OrderBy(c => c.Country).
ThenByDescending(c => c.Balance).
Select(c => new { c.Name, c.Country, c.Balance })
```

Multiple generators

Multiple generators:

```
from c in customers
where c.City == "London"
from o in c.Orders
where o.OrderDate.Year == 2005
select new { c.Name, o.OrderID, o.Total }
```

translate to invocations of SelectMany for all but the innermost generator:

```
customers.
Where(c => c.City == "London").
SelectMany(c =>
        c.Orders.
        Where(o => o.OrderDate.Year == 2005).
        Select(o => new { c.Name, o.OrderID, o.Total })
)
```

When multiple generators are combined with an orderby clause:

```
from c in customers, o in c.Orders
where o.OrderDate.Year == 2005
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

an additional Select is injected to collect the ordering expressions and the final result in a sequence of tuples. This is necessary such that OrderBy can operate on the entire sequence.

Following OrderBy, the final result is extracted from the tuples:

```
customers.
SelectMany(c =>
        c.Orders.
        Where(o => o.OrderDate.Year == 2005).
        Select(o => new { k1 = o.Total, v = new { c.Name,
o.OrderID, o.Total } })
).
OrderByDescending(x => x.k1).
Select(x => x.v)
```

into clauses

An into clause:

```
from c in customers
group c by c.Country into g
select new {
        Country = g.Key, CustCount = g.Group.Count()
        }
```

is simply a more convenient notation for a nested query:

```
from g in
        from c in customers
        group c by c.Country
select new { Country = g.Key, CustCount = g.Group.Count()
}
```

the translation of which is:

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount =
g.Group.Count() })
```

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<A,R>(A arg);
class C<T>
{
        public C<T> Where(Func<T,bool> predicate);
        public C<S> Select<S>(Func<T,S> selector);
        public C<S> SelectMany<S>(Func<T,C<S>> selector);
        public O<T> OrderBy<K>(Func<T,K> keyExpr);
        public O<T> OrderByDescending<K>(Func<T,K>
keyExpr);
        public C<G<K,T>> GroupBy<K>(Func<T,K> keyExpr);
        public C<G<K,E>> GroupBy<K,E>(Func<T,K> keyExpr,
Func<T,E> elemExpr);
}
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>
{
        public K Key { get; }
        public C<T> Group { get; }
}
```

The methods above use a generic delegate type Func<A, 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, which is a sequence of groupings that each have a Key and Group 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.

Formal translation rules
A query expression is processed by repeatedly applying the following translations in order. Each translation is applied until there are no more occurrences of the specified pattern.
Note that in the translations that produce invocations of OrderBy and ThenBy, if the corresponding ordering clause specifies a descending direction indicator, an invocation of OrderByDescending or ThenByDescending is produced instead.
A query that contains an into clause

$$q_1 \text{ into } x \; q_2$$

is translated into

$$\text{from } x \text{ in } (q_1) \; q_2$$

A from clause with multiple generators

$$\text{from } g_1 , g_2 , \ldots g_n$$

is translated into

$$\text{from } g_1 \text{ from } g_2 \ldots \text{ from } g_n$$

A from clause immediately followed by a where clause

$$\text{from } x \text{ in } e \text{ where } f$$

is translated into

$$\text{from } x \text{ in } (e) . \text{Where } (x => f)$$

A query expression with multiple from clauses, an orderby clause, and a select clause

from x_1 in e_1 from x_2 in e_2 ... orderby k_1 , k_2 ... select v

is translated into

(from x_1 in e_1 from x_2 in e_2 ...
select new { k1 = k_1 , k2 = k_2 ... , v = v })
. OrderBy (x => x . k1) . ThenBy (x => x . k2) ...
. Select (x => x . v)

A query expression with multiple from clauses, an orderby clause, and a group clause

from x_1 in e_1 from x_2 in e_2 ... orderby k_1 , k_2 ... group v by g

is translated into

(from x_1 in e_1 from x_2 in e_2 ...
select new { k1 = k_1 , k2 = k_2 ... , v = v , g = g })
. OrderBy (x => x . k1) . ThenBy (x => x . k2) ...
. GroupBy (x => x . g , x => x . v)

A query expression with multiple from clauses and a select clause

from x in e from x_1 in e_1 ... select v

is translated into

(e) . SelectMany (x => from x_1 in e_1 ... select v)

A query expression with multiple from clauses and a group clause

from x in e from x_1 in e_1 ... group v by g

is translated into

(e) . SelectMany (x => from x_1 in e_1 ... group v by g)

A query expression with a single from clause, no orderby clause, and a select clause

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)

A query expression with a single from clause, no orderby clause, and a group clause

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

is translated into

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

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

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

A query expression with a single from clause, an orderby clause, and a select clause

from *x* in *e* orderby k_1 , k_2 ... select *v*

is translated into

(*e*) . OrderBy (*x* => k_1) . ThenBy (*x* => k_2) ... . Select (*x* => *v*)

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

(*e*) . OrderBy (*x* => k_1) . ThenBy (*x* => k_2) ...

A query expression with a single from clause, an orderby clause, and a group clause

from *x* in *e* orderby k_1 , k_2 ... group *v* by *g*

is translated into

(*e*) . OrderBy (*x* => k_1) . ThenBy (*x* => k_2) ...
. GroupBy (*x* => *g* , *x* => *v*)

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

(*e*) . OrderBy (*x* => k_1) . ThenBy (*x* => k_2) ...
. GroupBy (*x* => *g*)

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.