**What's new in C# 3.0?**

Support for LINQ was the driving force behind the main enhancements in C# 3.0. **Query expressions** are the most obvious example, but **lambda expressions**, **extension methods** and **anonymous types** also fall into this category. However most of these enhancements are useful beyond LINQ.

Query expressions allow a SQL-like query syntax to be used in C#, e.g.

var nameList = new List<string> { "jack", "jill", "sue" };

var results = from name in nameList

where name.StartsWith("j")

select name;

Query expressions are just syntactic sugar - they resolve to standard method calls. For example the query expression above can be rewritten as:

var results = nameList.Where(name => name.StartsWith("j"));

The argument to Where() is a lambda expression, which represents an anonymous method. However a lambda expression is not just a more concise way to represent executable code - it can also be interpreted as a data structure (known as an expression tree), which allows the expression to be easily analysed or transformed at runtime. For example, LINQ-to-SQL makes use of this feature to transform C# queries to SQL queries, which gives much better performance than a simple in-memory filtering of results (as offered by DataTable.Select(), for example).

The Where() method shown above is an example of another new C# 3.0 feature: **extension methods**. Extension methods allow extra methods to be 'attached' to an existing type - any type, even sealed types or types you don't have the source code for. For example, the Where() method can be applied to any type that implements IEnumerable<T>.

Another new feature of C# 3.0 is the **var** keyword, which allows the compiler to infer the type of a variable from context, which is required for anonymous types but can be used with standard named types too:

var list = new List<string>() { "jack", "jill", "sue" }; // optional use with named types

var person = new { name = "jack", age = 20 }; // anonymous type

These examples also demonstrate the new object initializer and collection initializer syntax.

Finally, there are implicitly typed arrays and auto-implemented properties:

var names = new [] { "jack", "jill", "sue" }; // implicitly string[]

public string Name { get; set; } // auto-implemented property - private backing field auto-generated

**Type**

**What is an implicitly typed local variable declaration?**

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 (§26.4) 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 the null type.
* 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 (§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**.

**What is an anonymous type? How is it different from any other reference type?**

Anonymous types are [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) types that derive directly from [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx). The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type, except that it cannot be cast to any type except [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx).

**How to create an object of an anonymous type?**

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.

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 (§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.

**Is it possible to have two or more anonymous types in the same assembly that have the same number and type of properties, in the same order?**

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

**Can we declare a field, a property, an event, or the return type of a method as having an anonymous type?**

You cannot declare a field, a property, an event, or the return type of a method as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type object. However, doing this defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

**When can two instances of the same anonymous type be equal?**

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

**What is the use of dynamic type?**

Visual C# 2010 introduces a new type, dynamic. The type is a static type, but an object of type dynamic bypasses static type checking. In most cases, it functions like it has type object. At compile time, an element that is typed as dynamic is assumed to support any operation. Therefore, you do not have to be concerned about whether the object gets its value from

1. COM API,
2. Dynamic language such as IronPython,
3. HTML Document Object Model (DOM),
4. Reflection, or from somewhere else in the program. However, if the code is not valid, errors are caught at run time.

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

static void Main(string[] args)

{

ExampleClass ec = new ExampleClass();

// The following line causes a compiler error if exampleMethod1 has only

// one parameter.

//ec.exampleMethod1(10, 4);

dynamic dynamic\_ec = new ExampleClass();

// The following line is not identified as an error by the

// compiler, but it causes a run-time exception.

dynamic\_ec.exampleMethod1(10, 4);

// The following calls also do not cause compiler errors, whether

// appropriate methods exist or not.

dynamic\_ec.someMethod("some argument", 7, null);

dynamic\_ec.nonexistentMethod();

}

class ExampleClass

{

public ExampleClass() { }

public ExampleClass(int v) { }

public void exampleMethod1(int i) { }

public void exampleMethod2(string str) { }

}

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

**What are the characteristics of dynamic type?**

**Operations**

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

dynamic d = 1;

var testSum = d + 3;

// Rest the mouse pointer over testSum in the following statement.

System.Console.WriteLine(testSum);

Operations in which the result is not dynamic include conversions from dynamic to another type, and constructor calls that include arguments of type dynamic. For example, the type of testInstance in the following declaration is ExampleClass, not dynamic.

var testInstance = new ExampleClass(d);

**Conversions**

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

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

dynamic d1 = 7;

dynamic d2 = "a string";

dynamic d3 = System.DateTime.Today;

dynamic d4 = System.Diagnostics.Process.GetProcesses();

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

int i = d1;

string str = d2;

DateTime dt = d3;

System.Diagnostics.Process[] procs = d4;

[**Overload Resolution with Arguments of Type dynamic**](javascript:void(0))

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

// Valid.

ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not

// dynamic. A run-time exception is raised because the run-time type of d1 is int.

ec.exampleMethod2(d1);

// The following statement does cause a compiler error.

//ec.exampleMethod2(7);

**Why dynamic is introduced in .NET 4.0?**

The biggest reason is *that it allows a C# program to use dynamic dispatch to more naturally create objects coming from a dynamic language.*

For example, suppose you have a Calculator object declared in C#, meaning it is statically typed. You interact with your object like this:

Calculator calc = GetCalculator();

int sum = calc.Add(10, 20);

That’s pretty simple and straight forward. Now suppose the Calculator is not a statically typed .NET class (or it is a .NET class but you don’t know the specific type of class), you must do something like this:

object calc = GetCalculator();

Type calcType = calc.GetType();

object res = calcType.InvokeMember("Add", BindingFlags.InvokeMethod, null, new object[] { 10, 20 });

int sum = Convert.ToInt32(res);

That’s not nearly as simple. In fact, it’s downright ugly. There is a lot of non-type-safe calls and reflection going on here that you really shouldn’t have to see.

To take this a step further, if we knew that Calculator was a JavaScript class, you must use similar (but still significantly different) code:

ScriptObject calc = GetCalculator();

object res = calc.Invoke("Add", 10, 20);

int sum = Convert.ToInt32(res);

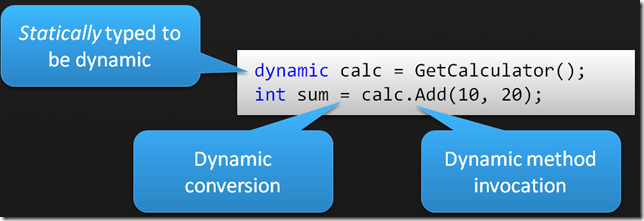
The reason for the differences in syntax is that there is no unification between the two APIs.

In C# 4.0, you can now use the following syntax:

dynamic calc = GetCalculator();

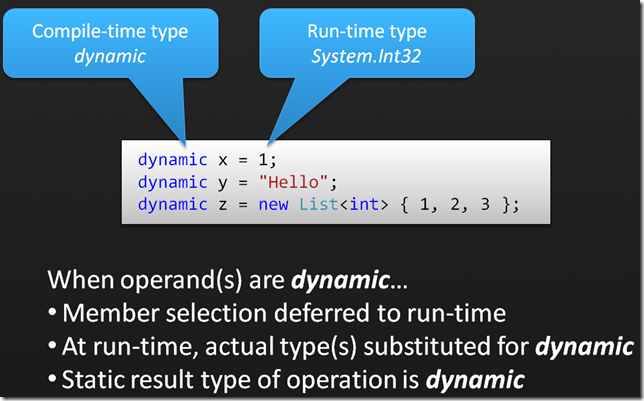
int sum = calc.Add(10, 20);

If you look at this syntax and the earlier statically typed call, you should notice that the only difference is that in C# we are declaring the data type to be dynamic.



Does this mean that C# is loosing it's roots as a statically typed language or that we should all start moving towards dynamic languages? Absolutely not. What is means is that it is now easier for you to write C# code that talks to objects (or APIs) written in dynamically typed languages. It also means that there is a unified API to talk to any dynamic language. You no longer need to worry about what language you are interoperating with to determine which C# code you must write.

So how does the dynamic keyword work? As I mentioned, it's a keyword in a similar fashion to var. You declare at compile-time the type to be dynamic, but at run-time you get a strongly typed object.



The dynamic keyword is great for writing C# code that consumes a dynamic object, but what about going the other direction and writing C# code that can be called from a dynamic language? You do this by implementing the IDynamicObject interface (or more simply, inheriting from the abstract DynamicObject class) and providing your own implementation for the member lookup and invocation.

Using the features and capabilities of the new dynamic keyword, the IDynamicObject interface, and the fact that the dynamic dispatch can dispatch to both dynamic and static types, C# effectively gets support for duck-typing.

**What's the difference between dynamic(C# 4) and var?**

[**http://stackoverflow.com/questions/961581/whats-the-difference-between-dynamicc-4-and-var**](http://stackoverflow.com/questions/961581/whats-the-difference-between-dynamicc-4-and-var)

[**http://msdn.microsoft.com/en-us/magazine/ee336309.aspx**](http://msdn.microsoft.com/en-us/magazine/ee336309.aspx)

[**http://stackoverflow.com/questions/301479/show-me-the-way-to-use-new-dynamic-keyword-in-c-4-0**](http://stackoverflow.com/questions/301479/show-me-the-way-to-use-new-dynamic-keyword-in-c-4-0)

[**Is the C# static constructor thread safe?**](http://stackoverflow.com/questions/7095/is-the-c-static-constructor-thread-safe)

[**http://stackoverflow.com/questions/7095/is-the-c-static-constructor-thread-safe**](http://stackoverflow.com/questions/7095/is-the-c-static-constructor-thread-safe)

[**.Net: Static classes and multithreading?**](http://stackoverflow.com/questions/3270275/net-static-classes-and-multithreading)

[**http://stackoverflow.com/questions/3270275/net-static-classes-and-multithreading**](http://stackoverflow.com/questions/3270275/net-static-classes-and-multithreading)

**Methods**

**What are extension methods? Give an example where do we need to use in real life.**

*Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type.* For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

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

The following example shows how to call the standard query operator **OrderBy** method on an array of integers. The expression in parentheses is a lambda expression. Many standard query operators take lambda expressions as parameters, but this is not a requirement for extension methods.

class ExtensionMethods2

{

static void Main()

{

int[] ints = { 10, 45, 15, 39, 21, 26 };

var result = ints.OrderBy(g => g);

foreach (var i in result)

{

System.Console.Write(i + " ");

}

}

}

//Output: 10 15 21 26 39 45

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on, and the parameter is preceded by the [this](http://msdn.microsoft.com/en-us/library/dk1507sz.aspx) modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

The following example shows an extension method defined for the [System.String](http://msdn.microsoft.com/en-us/library/system.string.aspx) class. Note that it is defined inside a non-nested, non-generic static class:

namespace ExtensionMethods

{

public static class MyExtensions

{

public static int WordCount(this String str)

{

return str.Split(new char[] { ' ', '.', '?' },

StringSplitOptions.RemoveEmptyEntries).Length;

}

}

}

The WordCount extension method can be brought into scope with this using directive:

using ExtensionMethods;

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

string s = "Hello Extension Methods";

int i = s.WordCount();

In your code you invoke the extension method with instance method syntax. However, the intermediate language (IL) generated by the compiler translates your code into a call on the static method. Therefore, the principle of encapsulation is not really being violated. In fact, extension methods cannot access private variables in the type they are extending.

In general, you will probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code. To enable extension methods for a particular type, just add a using directive for the namespace in which the methods are defined. For example, to use the standard query operators, add this using directive to your code:

using System.Linq;

**What is the advantage of extension methods?**

Extension methods enable you to "add" methods to existing types *without creating a new derived type, recompiling, or otherwise modifying the original type*.

**What are the differences between instance methods and extension methods?**

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.

**Which is better: Extension methods versus inheritance?**

<http://richnewman.wordpress.com/2009/06/12/extending-classes-extension-methods-or-inheritance/>

[**Overriding Extension Methods**](http://stackoverflow.com/questions/474074/overriding-extension-methods)

[**http://stackoverflow.com/questions/474074/overriding-extension-methods**](http://stackoverflow.com/questions/474074/overriding-extension-methods)

**What are named and optional arguments?**

Named arguments enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than with the parameter's position in the parameter list. Optional arguments enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters, when used together, enable you to supply arguments for only a few parameters from a list of optional parameters. This capability greatly facilitates calls to COM interfaces such as the Microsoft Office Automation APIs.

**Give example to use named arguments.**

class NamedExample

{

static void Main(string[] args)

{

// The method can be called in the normal way, by using positional arguments.

Console.WriteLine(CalculateBMI(123, 64));

// Named arguments can be supplied for the parameters in either order.

Console.WriteLine(CalculateBMI(weight: 123, height: 64));

Console.WriteLine(CalculateBMI(height: 64, weight: 123));

// Positional arguments cannot follow named arguments.

// The following statement causes a compiler error.

//Console.WriteLine(CalculateBMI(weight: 123, 64));

// Named arguments can follow positional arguments.

Console.WriteLine(CalculateBMI(123, height: 64));

}

static int CalculateBMI(int weight, int height)

{

return (weight \* 703) / (height \* height);

}

}

**Give example to use optional arguments.**

In the following example, the constructor for ExampleClass has one parameter, which is optional. Instance method ExampleMethod has one required parameter, required, and two optional parameters, optionalstr and optionalint. The code in Main shows the different ways in which the constructor and method can be invoked.

namespace OptionalNamespace

{

class OptionalExample

{

static void Main(string[] args)

{

// Instance anExample does not send an argument for the constructor's

// optional parameter.

ExampleClass anExample = new ExampleClass();

anExample.ExampleMethod(1, "One", 1);

anExample.ExampleMethod(2, "Two");

anExample.ExampleMethod(3);

// Instance anotherExample sends an argument for the constructor's

// optional parameter.

ExampleClass anotherExample = new ExampleClass("Provided name");

anotherExample.ExampleMethod(1, "One", 1);

anotherExample.ExampleMethod(2, "Two");

anotherExample.ExampleMethod(3);

// The following statements produce compiler errors.

// An argument must be supplied for the first parameter, and it

// must be an integer.

//anExample.ExampleMethod("One", 1);

//anExample.ExampleMethod();

// You cannot leave a gap in the provided arguments.

//anExample.ExampleMethod(3, ,4);

//anExample.ExampleMethod(3, 4);

// You can use a named parameter to make the previous

// statement work.

anExample.ExampleMethod(3, optionalint: 4);

}

}

class ExampleClass

{

private string \_name;

// Because the parameter for the constructor, name, has a default

// value assigned to it, it is optional.

public ExampleClass(string name = "Default name")

{

\_name = name;

}

// The first parameter, required, has no default value assigned

// to it. Therefore, it is not optional. Both optionalstr and

// optionalint have default values assigned to them. They are optional.

public void ExampleMethod(int required, string optionalstr = "default string",

int optionalint = 10)

{

Console.WriteLine("{0}: {1}, {2}, and {3}.", \_name, required, optionalstr,

optionalint);

}

}

// The output from this example is the following:

// Default name: 1, One, and 1.

// Default name: 2, Two, and 10.

// Default name: 3, default string, and 10.

// Provided name: 1, One, and 1.

// Provided name: 2, Two, and 10.

// Provided name: 3, default string, and 10.

// Default name: 3, default string, and 4.

}

**How the use of named and optional arguments affects overload resolution?**

Use of named and optional arguments affects overload resolution in the following ways:

* A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
* If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
* If two candidates are judged to be equally good, preference goes to a candidate that does not have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

**Object and collection initialzser**

**What are object initializers?**

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to explicitly invoke a constructor. The following example shows how to use an object initializer with a named type, Cat. Note the use of auto-implemented properties in the Cat class.

private class Cat

{

// Auto-implemented properties.

public int Age { get; set; }

public string Name { get; set; }

}

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

**Where can one use object initializers?**

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. Query expressions make frequent use of [anonymous types](http://msdn.microsoft.com/en-us/library/bb397696.aspx), which can only be initialized by using an object initializer, as shown in the following declaration.

var pet = new { Age = 10, Name = "Fluffy" };

Anonymous types enable the select clause in a LINQ query expression to transform objects of the original sequence into objects whose value and shape may differ from the original. This is useful if you want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (p) contains many fields and methods, and that you are only interested in creating a sequence of objects that contain the product name and the unit price.

var productInfos =

from p in products

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

When this query is executed, the productInfos variable will contain a sequence of objects that can be accessed in a foreach statement as shown in this example:

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

Each object in the new anonymous type has two public properties which receive the same names as the properties or fields in the original object. You can also rename a field when you are creating an anonymous type; the following example renames the UnitPrice field to Price.

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

**What are collection initializers? Give an example.**

Collection initializers let you specify one or more element intializers when you initialize a collection class that implements [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx). The element initializers can be a simple value, an expression or an object initializer. By using a collection initializer you do not have to specify multiple calls to the **Add** method of the class in your source code; the compiler adds the calls.

The following examples shows two simple collection initializers:

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

List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };

The following collection initializer uses object initializers to initialize objects of the Cat class defined in a previous example. Note that the individual object initializers are enclosed in braces and separated by commas.

List<Cat> cats = new List<Cat>

{

new Cat(){ Name = "Sylvester", Age=8 },

new Cat(){ Name = "Whiskers", Age=2 },

new Cat(){ Name = "Sasha", Age=14 }

};

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

List<Cat> moreCats = new List<Cat>

{

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

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

null

};

Properties

Arrays

**Delegates**

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

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

delegate int del(int i);

static void Main(string[] args)

{

del myDelegate = x => x \* x;

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

}

To create an expression tree type:

using System.Linq.Expressions;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

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

}

}

}

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

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

When you use method-based syntax to call the [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) method in the [Enumerable](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.aspx) class (as you do in LINQ to Objects and LINQ to XML) the parameter is a delegate type [System.Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151.aspx). *A lambda expression is the most convenient way to create that delegate.* When you call the same method in, for example, the [System.Linq.Queryable](http://msdn.microsoft.com/en-us/library/system.linq.queryable.aspx) class (as you do in LINQ to SQL) then the parameter type is an [System.Linq.Expressions.Expression](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx)<Func> where Func is any Func delegates with up to sixteen input parameters. Again, a lambda expression is just a very concise way to construct that expression tree. The lambdas allow the **Where** calls to look similar although in fact the type of object created from the lambda is different.

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

Lambdas are not allowed on the left side of the [is](http://msdn.microsoft.com/en-us/library/scekt9xw.aspx) or [as](http://msdn.microsoft.com/en-us/library/cscsdfbt.aspx) operator.

All restrictions that apply to anonymous methods also apply to lambda expressions.

**What are the different lambdas?**

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

(input parameters) => expression

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

(x, y) => x == y

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

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

Specify zero input parameters with empty parentheses:

() => SomeMethod()

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

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

(input parameters) => {statement;}

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

delegate void TestDelegate(string s);

…

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

myDel("Hello");

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

**Discuss Type Inference in Lambdas.**

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

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

The general rules for lambdas are as follows:

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

Note that lambda expressions in themselves do not have a type because the common type system has no intrinsic concept of "lambda expression." However, it is sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or [Expression](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx) type to which the lambda expression is converted.

## Defining, Creating, and Using a Delegate

In C#, a delegate is a data structure that refers to either a static method, or an object and an instance method of its class. When you initialize a delegate, you initialize it with either a static method, or a class instance and an instance method.

The following code shows the definition of a delegate and a method that can be used to initialize the delegate:

// Defines a delegate that takes an int and returns an int  
public delegate int ChangeInt(int x);  
  
// Define a method to which the delegate can point  
static public int DoubleIt(int x)  
{  
return x \* 2;  
}

Now, you can create and initialize an instance of the delegate, and then call it:

ChangeInt myDelegate = new ChangeInt(DelegateSample.DoubleIt);  
Console.WriteLine("{0}", myDelegate(5));

This, as you would expect, writes 10 to the console.

## Using an Anonymous Method

With C# 2.0, anonymous methods allow you to write a method and initialize a delegate in place:

ChangeInt myDelegate = new ChangeInt(  
delegate(int x)  
{  
return x \* 2;  
}  
);  
Console.WriteLine("{0}", myDelegate(5));

## Using a Lambda Expression

With Lambda expressions, the syntax gets even terser:

ChangeInt myDelegate = x => x \* 2;  
Console.WriteLine("{0}", myDelegate(5));

This lambda expression is an anonymous method that takes one argument x, and returns x \* 2. In this case, the type of x and the type that the lambda returns are inferred from the type of the delegate to which the lambda is assigned.

If you wanted to, you could have specified the type of the argument, as follows:

ChangeInt myDelegate = (int x) => x \* 2;  
Console.WriteLine("{0}", myDelegate(5));

## Using a Lambda with Two Arguments

When using the Standard Query Operators, on occasion, you need to write a lambda expression that takes two arguments.

If you have a delegate that takes two arguments:

// Defines a delegate that takes two ints and returns an int  
public delegate int MultiplyInts(int arg, int arg2);

You can declare and initialize a delegate:

MultiplyInts myDelegate = (a, b) => a \* b;  
Console.WriteLine("{0}", myDelegate(5, 2));

## Statement Lambda Expressions

You can write a more complicated lambda expression using statements, enclosing the statements in braces. If you use this syntax, you must use the return statement, unless the lambda returns void:

int[] source = new[] { 3, 8, 4, 6, 1, 7, 9, 2, 4, 8 };  
  
foreach (int i in source.Where(  
x =>  
{  
if (x <= 3)  
return true;  
else if (x >= 7)  
return true;  
return false;  
}  
))  
Console.WriteLine(i);

Sometimes developers wonder how to pronounce the => token.

If the lambda expression is a predicate, expressing some condition: c => c.State == "WA" then the => can be spoken as "such that". In this example, you could say "c such that c dot state equals Washington". If the lambda expression is a projection, returning a new type: c => new XElement("CustomerID", c.CustomerID); then the => can be spoken as "becomes". In the above example, you could say "c becomes new XElement with a name of CustomerID and its value is c dot CustomerID". Or "maps to", or "evaluate to", as suggested in the comments below. But most often, I just say "arrow". J

A quick note: predicates are simply boolean expressions that are passed to some method that will use the boolean expression to filter something. A lambda expression used for projection takes one type, and returns a different type. More on both of these concepts later.

## Lambda Expressions that Return Void

A lambda expression that returns void is not very useful in the context of functional programming because the only possible reason for such a function is that it has side-effects, and is not pure (more on this [**later in the tutorial**](http://blogs.msdn.com/ericwhite/pages/Pure-Functions.aspx)), but it is part of C# 3.0 syntax, so I'll cover it here. Sometimes developers will use a void statement lambda expression for writing an event handler. This has the benefit that the syntax is terser, and the program is smaller. In addition, the lambda expression can refer to local variables in the enclosing scope. This is part of C#'s implementation of [**closures**](http://blogs.msdn.com/ericwhite/archive/2008/09/12/closures.aspx). The only way to write a lambda expression that returns void is to write a statement lambda expression. The following example shows defining a void delegate, declaring an instance of it, and calling it.

// Defines a delegate that takes a string and returns void  
public delegate void OutputToConsole(string arg);  
  
static void Main(string[] args)  
{  
OutputToConsole o = a => {  
Console.WriteLine(a);  
};  
o("Hello, World");  
}

If you write a lambda expression for a delegate that returns void and takes no arguments, it results in interesting syntax:

// Defines a delegate that takes no arguments and returns void  
public delegate void OutputHelloToConsole();  
  
static void Main(string[] args)  
{  
OutputHelloToConsole o = () =>  
{  
Console.WriteLine("Hello, World");  
};  
o();  
}

## The Func Delegate Types

The framework defines a number of parameterized delegate types:

public delegate TR Func<TR>();  
public delegate TR Func<T0, TR>(T0 a0);  
public delegate TR Func<T0, T1, TR>(T0 a0, T1 a1);  
public delegate TR Func<T0, T1, T2, TR>(T0 a0, T1 a1, T2 a2);  
public delegate TR Func<T0, T1, T2, T3, TR>(T0 a0, T1 a1, T2 a2, T3 a3);

In the above delegate types, notice that if there is only one type parameter, it is the return type of the delegate. If there are two type parameters, the first type parameter is the type of the one and only argument, and the second type is the return type of the delegate, and so on. Many of the standard query operators (which are just methods that you call) take as an argument a delegate of one of these types. These delegate definitions are useful to you when writing your own methods that take a delegate as an argument.

using System;  
using System.Collections.Generic;  
  
class Program  
{  
static List<T> MyWhereMethod<T>(IEnumerable<T> source,  
Func<T, bool> predicate)  
{  
List<T> l = new List<T>();  
foreach (T item in source)  
if (predicate(item))  
l.Add(item);  
return l;  
}  
  
static void Main(string[] args)  
{  
int[] source = new[] { 3, 8, 4, 6, 1, 7, 9, 2, 4, 8 };  
  
List<int> filteredList = MyWhereMethod(source,  
i => i >= 5);  
foreach (int z in filteredList)  
Console.WriteLine(z);  
}  
}

## The Action Delegate Types

The framework defines a number of parameterized delegate types for delegates that return void:

public delegate void Action();  
public delegate void Action<T0>(T0 a0);  
public delegate void Action<T0, T1>(T0 a0, T1 a1);  
public delegate void Action<T0, T1, T2>(T0 a0, T1 a1, T2 a2);  
public delegate void Action<T0, T1, T2, T3>(T0 a0, T1 a1, T2 a2, T3 a3);

Sometimes API designers will include an event that takes one of these delegate types as an argument, and you can write a lambda expression for the argument. As with the Func delegate types, these delegate definitions are useful to you when writing your own methods that take a delegate as an argument. This uses the interesting syntax of () => { /\* body of void function here \*/ };

using System;  
using System.Collections.Generic;  
using System.Threading;  
  
class Program  
{  
static void SomeAsynchronousMethod(Action complete)  
{  
// just pretending to be asynchronous in this example  
Thread.Sleep(1000);  
complete();  
}  
  
static void Main(string[] args)  
{  
SomeAsynchronousMethod(() => { Console.WriteLine("Done"); });  
}  
}

## Expression Trees

Lambda expressions can also be used as expression trees.

**Generic Delegates**

The .NET framework version 3.5 introduced two new sets of [generic](http://www.blackwasp.co.uk/Generics.aspx), parameterised [delegates](http://www.blackwasp.co.uk/CSharpDelegates.aspx) named Func and Action. The Func delegate can be used to encapsulate a [method](http://www.blackwasp.co.uk/CSharpMethods.aspx) that accepts between zero and four arguments and returns a value. The Action delegate also represents methods with zero to four [parameters](http://www.blackwasp.co.uk/CSharpMethodParameters.aspx) but differs from Func in that the method must return void.

The new delegates can be used to reduce the number of delegates that you define explicitly. In situations where you would need a delegate that matches one of the predefined Func or Action signatures, you may decide to use the built-in version. You should consider the naming of the delegate however, as "Func" or "Action" may not express your intent as clearly as another name.

One of the key reasons for the introduction of Func and Action is their relationship with [lambda expressions](http://www.blackwasp.co.uk/CSharpLambda.aspx). Every lambda expression's underlying type is one of these generic delegates. This means that lambda expressions can be passed to method parameters of the appropriate type without explicitly creating a delegate. Many of the [LINQ](http://www.blackwasp.co.uk/Linq.aspx) standard query operators accept Func arguments to take advantage of this feature.

**Using Generic Delegates as Parameter Types**

The Func delegate is commonly used as the type for parameters of methods that accept lambda expressions. These include LINQ standard query operators and similar methods that you create in your own projects. We will demonstrate this with this article's final example.

The following method defines an [array](http://www.blackwasp.co.uk/CSharpArrays.aspx) containing the names of ten fruits. The method returns a filtered list of these fruits based upon the delegate passed as the only argument. Note that the types assigned to the Func delegate specify that the encapsulated method must receive a single string parameter and return a Boolean value. We could execute the passed method against each fruit string in a [for-each loop](http://www.blackwasp.co.uk/CSharpForEachLoop.aspx). However, for simplicity the example uses the "Where" query operator.

private static string[] Fruit(Func<string, bool> filter)

{

string[] fruit = new string[]

{

"Apple", "Banana", "Cherry", "Damson", "Elderberry",

"Fig", "Grapefruit", "Huckleberry", "Lemon", "Mango"

};

return fruit.Where(filter).ToArray();

}

To test the method, add the following code to the Main method of a console application. This code calls the Fruit method, applying a filter that returns only fruit with a name shorter than six characters. The fruit names are then outputted.

string[] shortFruit = Fruit(f => f.Length < 6);

foreach (string fruit in shortFruit)

Console.WriteLine(fruit);

/\* OUTPUT

Apple

Fig

Lemon

Mango

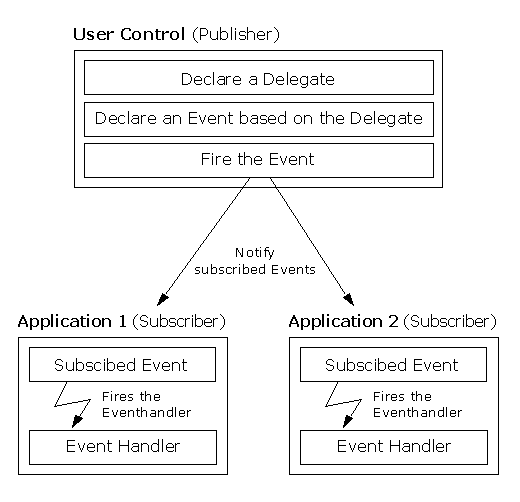
\*/

**Events**

<http://www.akadia.com/services/dotnet_delegates_and_events.html>

The Event model in C# finds its roots in the event programming model that is popular in asynchronous programming. The basic foundation behind this programming model is the idea of "publisher and subscribers." In this model, you have ***publishers*** who will do some logic and publish an "event." Publishers will then send out their event only to ***subscribers*** who have subscribed to receive the specific event.

In C#, any object can *publish* a set of events to which other applications can *subscribe*. When the publishing class raises an event, all the subscribed applications are notified. The following figure shows this mechanism.



The Second Change Event Example

Suppose you want to create a Clock class that uses events to notify potential subscribers whenever the local time changes value by one second. Here is the complete, documented example:

using System;  
using System.Threading;  
  
namespace SecondChangeEvent  
{  
**/\* ======================= Event Publisher =============================== \*/**  
  
// Our subject -- it is this class that other classes  
// will observe. This class publishes one event:  
// SecondChange. The observers subscribe to that event.  
public class Clock  
{  
// Private Fields holding the hour, minute and second  
private int \_hour;  
private int \_minute;  
private int \_second;  
  
// The delegate named SecondChangeHandler, which will encapsulate  
// any method that takes a clock object and a TimeInfoEventArgs  
// object as the parameter and returns no value. It's the  
// delegate the subscribers must implement.  
**public delegate void SecondChangeHandler (  
object clock,  
TimeInfoEventArgs timeInformation  
);**  
  
// The event we publish  
**public event SecondChangeHandler SecondChange;**  
  
// The method which fires the Event  
**protected void OnSecondChange(  
object clock,  
TimeInfoEventArgs timeInformation  
)  
{  
// Check if there are any Subscribers  
if (SecondChange != null)  
{  
// Call the Event  
SecondChange(clock,timeInformation);  
}  
}**  
  
// Set the clock running, it will raise an  
// event for each new second  
public void Run()  
{  
for(;;)  
{  
// Sleep 1 Second  
Thread.Sleep(1000);  
  
// Get the current time  
System.DateTime dt = System.DateTime.Now;  
  
// If the second has changed  
// notify the subscribers  
if (dt.Second != \_second)  
{  
// Create the TimeInfoEventArgs object  
// to pass to the subscribers  
TimeInfoEventArgs timeInformation =  
new TimeInfoEventArgs(  
dt.Hour,dt.Minute,dt.Second);  
  
// If anyone has subscribed, notify them  
**OnSecondChange (this,timeInformation);**  
}  
  
// update the state  
\_second = dt.Second;  
\_minute = dt.Minute;  
\_hour = dt.Hour;  
  
}  
}  
}

// The class to hold the information about the event  
// in this case it will hold only information  
// available in the clock class, but could hold  
// additional state information  
public class TimeInfoEventArgs : EventArgs  
{  
public TimeInfoEventArgs(int hour, int minute, int second)  
{  
this.hour = hour;  
this.minute = minute;  
this.second = second;  
}  
public readonly int hour;  
public readonly int minute;  
public readonly int second;  
}  
  
**/\* ======================= Event Subscribers =============================== \*/**  
// An observer. DisplayClock subscribes to the  
// clock's events. The job of DisplayClock is  
// to display the current time  
public class DisplayClock  
{  
// Given a clock, subscribe to  
// its SecondChangeHandler event  
public void Subscribe(Clock theClock)  
{  
**theClock.SecondChange +=  
new Clock.SecondChangeHandler(TimeHasChanged);**  
}  
  
// The method that implements the  
// delegated functionality  
public void TimeHasChanged(  
object theClock, TimeInfoEventArgs ti)  
{  
Console.WriteLine("Current Time: {0}:{1}:{2}",  
ti.hour.ToString(),  
ti.minute.ToString(),  
ti.second.ToString());  
}  
}  
  
// A second subscriber whose job is to write to a file  
public class LogClock  
{  
public void Subscribe(Clock theClock)  
{  
**theClock.SecondChange +=  
new Clock.SecondChangeHandler(WriteLogEntry);**  
}  
  
// This method should write to a file  
// we write to the console to see the effect  
// this object keeps no state  
public void WriteLogEntry(  
object theClock, TimeInfoEventArgs ti)  
{  
Console.WriteLine("Logging to file: {0}:{1}:{2}",  
ti.hour.ToString(),  
ti.minute.ToString(),  
ti.second.ToString());  
}  
}  
  
**/\* ======================= Test Application =============================== \*/**  
// Test Application which implements the  
// Clock Notifier - Subscriber Sample  
public class Test  
{  
public static void Main()  
{  
// Create a new clock  
Clock theClock = new Clock();  
  
// Create the display and tell it to  
// subscribe to the clock just created  
DisplayClock dc = new DisplayClock();  
dc.Subscribe(theClock);  
  
// Create a Log object and tell it  
// to subscribe to the clock  
LogClock lc = new LogClock();  
lc.Subscribe(theClock);  
  
// Get the clock started  
theClock.Run();  
}  
}  
}

# Conclusion

The Clock class from the last sample could simply print the time rather tahn raising an event, so **why bother with the introduction of using delegates**? The advantage of the publisg / subscribe idiom is that **any number of classes can be notified when an event is raised**. The subscribing classes do not need to know how the Clock works, and the Clock does not need to know what they are going to do in response to the event. Similarly a button can publish an Onclick event, and any number of unrelated objects can subscribe to that event, receiving notification when the button is clicked.

**The publisher and the subscribers are decoupled by the delegate**. This is highly desirable as it makes for more flexible and robust code. The clock can chnage how it detects time without breaking any of the subscribing classes. The subscribing classes can change how they respond to time changes without breaking the Clock. The two classes spin indepentdently of one another, which makes for code that is easier to maintain.

**Covariance and Contravariance**

What are covariance and contravariance?

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments.Covariance preserves [assignment compatibility](http://blogs.msdn.com/ericlippert/archive/2009/11/30/what-s-the-difference-between-covariance-and-assignment-compatibility.aspx) and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

// Assignment compatibility.   
string str = "test";  
// An object of a more derived type is assigned to an object of a less derived type.   
object obj = str;  
  
// Covariance.   
IEnumerable<string> strings = new List<string>();  
// An object that is instantiated with a more derived type argument   
// is assigned to an object instantiated with a less derived type argument.   
// Assignment compatibility is preserved.   
IEnumerable<object> objects = strings;  
  
// Contravariance.             
// Assume that I have this method:   
// static void SetObject(object o) { }   
Action<object> actObject = SetObject;  
// An object that is instantiated with a less derived type argument   
// is assigned to an object instantiated with a more derived type argument.   
// Assignment compatibility is reversed.   
Action<string> actString = actObject;

In C#, variance is supported in the following scenarios:

1. Covariance in arrays (since C# 1.0)
2. Covariance and contravariance in delegates, also known as “method group variance” (since C# 2.0)
3. Variance for generic type parameters in interfaces and delegates (since C# 4.0)

What is array covariance?

Arrays are covariant since C# 1.0. You can always do the following:

object[] obj = new String[10];

In the above code, I assigned an array of strings to an array of objects. So I used a more derived type than that originally specified, which is covariance.   
Covariance in arrays is considered “not safe,” because you can also do this:

obj[0] = 5;

This code compiles, but it throws an exception at run time because **obj** is in fact an array of strings and cannot contain integers.

What is delegate, or method group, variance?

This feature was added in C# 2.0. When you instantiate a delegate, you can assign it a method that has a more derived return type than that specified in the delegate (covariance). You can also assign a method that has parameter types less derived than those in the delegate (contravariance).

Here’s a quick code example illustrating the feature and some of its limitations.

static object GetObject() { return null; }  
static void SetObject(object obj) { }  
  
static string GetString() { return ""; }  
static void SetString(string str) { }  
  
static void Main()  
{  
    // Covariance. A delegate specifies a return type as object,  
    // but I can assign a method that returns a string.  
    Func<object> del = GetString;  
  
    // Contravariance. A delegate specifies a parameter type as string,  
    // but I can assign a method that takes an object.  
    Action<string> del2 = SetObject;  
  
    // But implicit conversion between generic delegates is not supported until C# 4.0.  
    Func<string> del3 = GetString;  
    Func<object> del4 = del3; // Compiler error here until C# 4.0.  
}

By the way, this feature works for all delegates, both generic and non-generic, not just for **Func** and **Action** delegates.

For more information and examples, see [Covariance and Contravariance in Delegates](http://msdn.microsoft.com/en-us/library/ms173174.aspx) on MSDN and Eric Lippert’s post [Covariance and Contravariance in C#, Part Three: Method Group Conversion Variance](http://blogs.msdn.com/ericlippert/archive/2007/10/19/covariance-and-contravariance-in-c-part-three-member-group-conversion-variance.aspx).

What is variance for generic type parameters?

This is a new feature in C# 4.0. Now, when creating a generic interface, you can specify whether there is an implicit conversion between interface instances that have different type arguments. For example, you can use an interface instance that has methods with more derived return types than originally specified (covariance) or that has methods with less derived parameter types (contravariance). The same rules are applied to generic delegates.

While you can create variant interfaces and delegates yourself, this is not the main purpose for this feature. What is more important is that a set of interfaces and delegates in .NET Framework 4 have been updated to become variant.   
Here’s the list of updated interfaces:

* [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(VS.100).aspx) (*T* is covariant)
* [IEnumerator<T>](http://msdn.microsoft.com/en-us/library/78dfe2yb(VS.100).aspx) (*T* is covariant)
* [IQueryable<T>](http://msdn.microsoft.com/en-us/library/bb351562(VS.100).aspx) (*T* is covariant)
* [IGrouping<TKey, TElement>](http://msdn.microsoft.com/en-us/library/bb344977(VS.100).aspx) (TKey and TElement are covariant)
* [IComparer<T>](http://msdn.microsoft.com/en-us/library/8ehhxeaf(VS.100).aspx) (*T* is contravariant)
* [IEqualityComparer<T>](http://msdn.microsoft.com/en-us/library/ms132151(VS.100).aspx) (*T* is contravariant)
* [IComparable<T>](http://msdn.microsoft.com/en-us/library/4d7sx9hd(VS.100).aspx) (*T* is contravariant)

And the list of updated delegates:

* Action delegates from the [System](http://msdn.microsoft.com/en-us/library/system(VS.100).aspx) namespace, for example, [Action<T>](http://msdn.microsoft.com/en-us/library/018hxwa8(VS.100).aspx) and [Action<T1, T2>](http://msdn.microsoft.com/en-us/library/bb549311(VS.100).aspx) (*T*, *T1*, *T2*, and so on are contravariant)
* Func delegates from the [System](http://msdn.microsoft.com/en-us/library/system(VS.100).aspx) namespace, for example, [Func<TResult>](http://msdn.microsoft.com/en-us/library/bb534960(VS.100).aspx) and [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151(VS.100).aspx) (*TResult* is covariant; *T*, *T1*, *T2*, and so on are contravariant)
* [Predicate<T>](http://msdn.microsoft.com/en-us/library/bfcke1bz(VS.100).aspx) (*T* is contravariant)
* [Comparison<T>](http://msdn.microsoft.com/en-us/library/tfakywbh(VS.100).aspx) (*T* is contravariant)
* [Converter<TInput, TOutput>](http://msdn.microsoft.com/en-us/library/kt456a2y(VS.100).aspx) (*TInput* is contravariant; *TOutput* is covariant.)

The most frequent scenario for most users is expected to be something like this one:

IEnumerable<Object> objects = new List<String>();

While this code doesn’t look that impressive, it allows you to reuse a lot of methods that accept **IEnumerable** objects.

class Program  
{  
    // The method has a parameter of the IEnumerable<Person> type.   
    public static void PrintFullName(IEnumerable<Person> persons)  
    {  
        // The method iterates through a sequence and prints some info.   
    }  
  
    public static void Main()  
    {  
        List<Employee> employees = new List<Employee>();  
  
        // I can pass List<Employee>, which is in fact IEnumerable<Employee>,   
        // although the method expects IEnumerable<Person>.   
        PrintFullName(employees);  
    }  
}

A couple of important rules to remember:

* This feature works only for generic interfaces and delegates. If you implement a variant generic interface, the implementing class is still invariant. Classes and structs do not support variance in C# 4.0.   
  So the following doesn’t compile:

// List<T> implements the covariant interface   
// IEnumerable<out T>. But classes are invariant.   
List<Person> list = new List<Employee>(); // Compiler error here.

* Variance is supported only if a type parameter is a reference type. Variance is not supported for value types.   
  The following doesn’t compile either:

// int is a value type, so the code doesn't compile.  
IEnumerable<Object> objects = new List<int>(); // Compiler error here.

Where can I find more examples of using covariance and contravariance?

I wrote a couple of MSDN topics that show how you can benefit from this new feature. They might help you better understand the principles of covariance and contravariance:

* [Using Variance in Interfaces for Generic Collections](http://msdn.microsoft.com/en-us/library/dd465120(VS.100).aspx)
* [Using Variance for Func and Action Generic Delegates](http://msdn.microsoft.com/en-us/library/dd465122(VS.100).aspx)

Also, take a look at the video [How Do I: Use Covariance and Contravariance in VS 2010 Part I?](http://msdn.microsoft.com/en-us/vcsharp/ee672314.aspx) by Eric Lippert.

How can I create variant generic interfaces and delegates myself?

The [out](http://msdn.microsoft.com/en-us/library/dd469487(VS.100).aspx) keyword marks a type parameter as covariant, and the [in](http://msdn.microsoft.com/en-us/library/dd469484(VS.100).aspx) keyword marks it as contravariant. The two most important rules to remember:

* You can mark a generic type parameter as covariant if it is used only as a method return type and is not used as a type of formal method parameters.
* And vice versa, you can mark a type as contravariant if it is used only as a type of formal method parameters and not used as a method return type.

For more information about variance validation, read [Creating Variant Generic Interfaces](http://msdn.microsoft.com/en-us/library/dd997386(VS.100).aspx) and [Variance in Delegates](http://msdn.microsoft.com/en-us/library/dd233060(VS.100).aspx) on MSDN and Eric Lippert’s post [Exact rules for variance validity](http://blogs.msdn.com/ericlippert/archive/2009/12/03/exact-rules-for-variance-validity.aspx).

This example shows how to create a variant generic interface:

interface IVariant<out R, in A>  
{  
    // These methods satisfy the rules.  
    R GetR();  
    void SetA(A sampleArg);  
    R GetRSetA(A sampleArg);  
  
    // And these don’t.  
    // A GetA();  
    // void SetR(R sampleArg);  
    // A GetASetR(R sampleArg);  
}

If you extend a variant interface, the extending interface is invariant by default. You must explicitly specify whether the type parameters are covariant or contravariant by using the [out](http://msdn.microsoft.com/en-us/library/dd469487(VS.100).aspx) or [in](http://msdn.microsoft.com/en-us/library/dd469484(VS.100).aspx) keyword. Here is a quick example from [MSDN](http://msdn.microsoft.com/en-us/library/dd997386(VS.100).aspx):

interface ICovariant<out T> { }   
  
// This interface is invariant because I didn't use the "out" keyword.   
interface IInvariant<T> : ICovariant<T> { }   
  
// And this one is covariant because I explicitly specified this.   
interface IExtCovariant<out T> : ICovariant<T> { }

And once again, this feature is supported for generic interfaces and delegates only. So the following doesn’t compile:

class Sample<out T> { }  // Compiler error here.

For more examples, take a look at the video [How Do I: Use Covariance and Contravariance in VS 2010 Part II?](http://msdn.microsoft.com/en-us/vcsharp/ee672319.aspx) by Eric Lippert.

Where can I find more in-depth information about covariance and contravariance?

This is the MSDN root topic: [Covariance and Contravariance](http://msdn.microsoft.com/en-us/library/ee207183(VS.100).aspx).

And, of course, read [Eric Lippert’s blog](http://blogs.msdn.com/ericlippert/archive/tags/Covariance+and+Contravariance/default.aspx). He designed this feature for C# 4.0, so who knows more about it?

Ques:

**What is background worker? Can one call background Worker from another thread than UI thread?**

Yes. BackgroundWorker uses the Event-based Asynchronous Pattern (EAP). As such, it requires a thread context in which to live. UI threads satisfy this requirement, but manually-created Thread instances do not (unless you install one or make the instance a secondary UI thread).

**BackgroundWorker vs background Thread**

1. Use [BackgroundWorker](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx) if you have a single task that runs in the background and needs to interact with the UI. The task of marshalling data and method calls to the UI thread are handled automatically through its event-based model. Avoid BackgroundWorker if...
   * your assembly does not already reference the System.Windows.Form assembly,
   * you need the thread to be a foreground thread, or
   * you need to manipulate the thread priority.
2. Use a [ThreadPool](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx) thread when efficiency is desired. The ThreadPool helps avoid the overhead associated with creating, starting, and stopping threads. Avoid using the ThreadPool if...
   * the task runs for the lifetime of your application,
   * you need the thread to be a foreground thread,
   * you need to manipulate the thread priority, or
   * you need the thread to have a fixed identity (aborting, suspending, discovering).
3. Use the [Thread](http://msdn.microsoft.com/en-us/library/system.threading.thread.aspx) class for long-running tasks and when you require features offered by a formal threading model, e.g., choosing between foreground and background threads, tweaking the thread priority, fine-grained control over thread execution, etc.

**Explain how it makes easy UI threading with Background Worker.**

[**http://www.i-programmer.info/programming/silverlight/1388-easy-ui-threading-with-background-worker.html?start=1**](http://www.i-programmer.info/programming/silverlight/1388-easy-ui-threading-with-background-worker.html?start=1)

[**Does the BackgroundWorker provide real multithreading?**](http://stackoverflow.com/questions/2257716/does-the-backgroundworker-provide-real-multithreading)

Each BackgroundWorker runs a on a separate thread. You can create as many background workers as you need to run operations in parallel, so in that sense it is true multithreading.

The benefit of BackgroundWorker is the ease with which you can subscribe events that will fire on your UI thread when the time-consuming task completes.

**What are the different ways of implementing multithreading in .NET?**

[**http://www.albahari.com/threading/**](http://www.albahari.com/threading/)

**How to handle exceptions from a BackgroundWorker thread?**

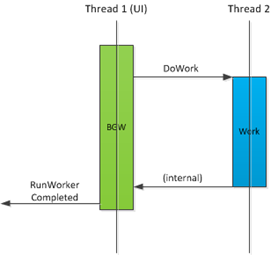
**How to directly access the UI thread from the BackgroundWorker thread in WPF?**

**What kind of background threading should I use in this particular scenario?**

You are in .NET presented with the following options:

1) **BackgroundWorker** (System.ComponentModel.BackgroundWorker)  
2) **ThreadPool** (System.Threading.ThreadPool)  
3) **Thread** (System.Threading.Thread)  
4) **Task/Task<T>** (System.Threading.Task; *.NET 4 only*)

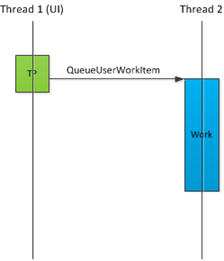
**BackgroundWorker (BGW)**:  
This component can be used if you are doing work which should ultimately interact with the UI. The BackgroundWorker communicates using events raised on the user interface thread (see below), hence it is a good candidate for UI applications.



static void Main(string[] args)  
{  
 var p = new Person();  
   
 //BGW   
 var bgw = new BackgroundWorker();  
 bgw.DoWork += bgw\_DoWork;  
 bgw.RunWorkerCompleted += bgw\_RunWorkerCompleted;  
 bgw.RunWorkerAsync(p); //start working, pass in Person class 'p'  
}  
  
//running on UI-thread  
static void bgw\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)  
{  
 throw new NotImplementedException();  
}  
  
//running on background thread  
static void bgw\_DoWork(object sender, DoWorkEventArgs e)  
{  
 throw new NotImplementedException();  
}

As seen in the above figure – the “RunWorkerCompleted” event is raised on the UI-thread (the backgroundworker handles the marshalling) and you can safely manipulate UI-controls from the eventhandler listening to RunWorkerCompleted event. Technically, the BGW raises the event on the *calling thread*; but as this is the UI-thread in the above scenario, you are effectively seeing an event raised on the UI-thread.

**ThreadPool (TP):**This is an easy way of offloading tasks to the .NET threadpool. You basically compose a “task” (method) and send this off by queuing the task to the threadpool. *It is a fire-and-forget way of programming*. If you intend to interact with the UI afterwards, you should be aware that the operation is running in a non-UI thread!



static void Main(string[] args)  
{  
 //thread pool  
 var p = new Person();  
 ThreadPool.QueueUserWorkItem(ThreadPoolWorker, p);  
}  
  
static void ThreadPoolWorker(object stateObject)  
{  
 //do long running stuff with "stateObject"  
   
}

**Thread**:  
This is the **hardcore** way of doing multithreading. All is left to the developer, but you are in this way given full control of almost any property on the thread.

static void Main(string[] args)  
{  
 //  
 var p = new Person();  
   
 var ts = new ParameterizedThreadStart(WorkMethod);  
 var t = new Thread(ts);  
 t.Name = "my thread";  
 t.Priority = ThreadPriority.Highest; //set high priority  
 t.Start(p); //pass in Person object  
}  
  
   
static void WorkMethod(object state)  
{  
 //do stuff with state-object  
}

**Task/Task<T>**:  
This is presented as the future of multithreading. It is the way multithreading should be made, according to the “dark empire” (MS). The terms and concepts originally comes from TaskParallelLibrary (TPL), which was an incubation project from the PnP group in MS.

static void Main(string[] args)  
{  
 //  
 var p = new Person();  
   
 var t = new Task(WorkMethod, p); //create new task, pass in Person  
 t.Start(); //schedule task (will start afterwards)  
}  
  
   
static void WorkMethod(object p)  
{  
 //do stuff with state-object  
}

There are a large number of tweaks that can be set on the Task/Task<T>. These are not shown, but allows you do control a large number of things.

<http://blog.clauskonrad.net/2011/03/threading-backgroundworker-vs.html>

**Which thread does background worker completed event handler run on?**

<http://stackoverflow.com/questions/4220239/which-thread-does-backgroundworker-completed-event-handler-run-on>

It's going to be run in the same thread that BackgroundWorker is in, i.e., most usually the UI thread.

**I get this error if I click a button that starts the backgroundworker twice. "This BackgroundWorker is currently busy and cannot run multiple tasks concurrently"**

**How can I avoid this?**

Don't start the BackgroundWorker twice. You can check if it is already running by using the IsBusy property, so just change this code:

worker.RunWorkerAsync();

to this:

if( !worker.IsBusy )  
 worker.RunWorkerAsync();  
else  
 MessageBox.Show("Can't run the worker twice!");

You also have the ability to cancel the current job, here is a code sample

worker.WorkerSupportsCancellation = true;  
 if (worker.IsBusy)  
 worker.CancelAsync();  
 else  
 worker.RunWorkerAsync();

// Events  
 void worker\_DoWork(object sender, DoWorkEventArgs e)  
 {  
 for(int i = 0; i < int.MaxValue; i++)  
 {  
 if (worker.CancellationPending)  
 {  
 e.Cancel = true;  
 return;  
 }  
 // Do work here  
 }  
 e.Result = MyResult;  
 }  
  
 void worker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)  
 {  
 // Deal with results  
 }

**When does BackgroundWorker thread get killed?**

|  |
| --- |
| Background worker threads do not get killed: they are returned to the threadpool. |

**I'm creating a C# dll, which is going to be used by others developers in WinForms. For some reasons, I want to detect, if methods from this library, are called from Main (GUI) Thread and warn developer he has done such a thing (ie. in log file). Is there any reasonable way to detect calling method from main thread? Remember I have no access to WinForm application.**

An easy solution in this case is to declare a static control in the library assembly that is created on the Main UI thread. If you want to detect if the library is called from the main thread, then use the following

if (MyLibraryControl.InvokeRequired)  
 //do your thing here

**How does background worker object not able to know running in the UI thread? Explain the implementation of background worker?**

**What are the standard patterns for performing I/O-bound and compute-bound asynchronous operations?**

The .NET Framework provides the following two standard patterns for performing I/O-bound and compute-bound asynchronous operations:

* Asynchronous Programming Model (APM), in which asynchronous operations are represented by a pair of Begin/End methods such as [FileStream.BeginRead](http://msdn.microsoft.com/en-us/library/zxt5ahzw.aspx) and [Stream.EndRead](http://msdn.microsoft.com/en-us/library/system.io.stream.endread.aspx).
* Event-based asynchronous pattern (EAP), in which asynchronous operations are represented by a method/event pair that are named OperationNameAsync and OperationNameCompleted, for example, [WebClient.DownloadStringAsync](http://msdn.microsoft.com/en-us/library/system.net.webclient.downloadstringasync.aspx) and [WebClient.DownloadStringCompleted](http://msdn.microsoft.com/en-us/library/system.net.webclient.downloadstringcompleted.aspx). (EAP was introduced in the .NET Framework version 2.0.)

The Task Parallel Library (TPL) can be used in various ways in conjunction with either of the asynchronous patterns. You can expose both APM and EAP operations as Tasks to library consumers, or you can expose the APM patterns but use Task objects to implement them internally. In both scenarios, by using Task objects, you can simplify the code and take advantage of the following useful functionality:

* Register callbacks, in the form of task continuations, at any time after the task has started.
* Coordinate multiple operations that execute in response to a **Begin\_** method, by using the [ContinueWhenAll](http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskfactory.continuewhenall.aspx) and [ContinueWhenAny](http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskfactory.continuewhenany.aspx) methods, or the [WaitAll](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.waitall.aspx) method or the [WaitAny](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.waitany.aspx) method.
* Encapsulate asynchronous I/O-bound and compute-bound operations in the same Task object.
* Monitor the status of the Task object.
* Marshal the status of an operation to a Task object by using [TaskCompletionSource<TResult>](http://msdn.microsoft.com/en-us/library/dd449174.aspx).

**Can there be two UI threads in application?**

[**http://stackoverflow.com/questions/1652799/multiple-ui-threads-on-the-same-window**](http://stackoverflow.com/questions/1652799/multiple-ui-threads-on-the-same-window)

[**Write a lambda or anonymous function that accepts an out parameter**](http://stackoverflow.com/questions/1990569/write-a-lambda-or-anonymous-function-that-accepts-an-out-parameter)**.**

**I have a delegate defined in my code:**

public bool delegate CutoffDateDelegate( out DateTime cutoffDate );

**I would like to create delegate and initialize with a lambda or anonymous function, but neither of these compiled.**

CutoffDateDelegate del1 = dt => { dt = DateTime.Now; return true; }  
CutoffDateDelegate del2 = delegate( out dt ) { dt = DateTime.Now; return true; }

**Is there way to do this?**

You can use either lambda or anonymous delegate syntax - you just need to specify the type of the argument, and mark it as out:

public delegate bool CutoffDateDelegate( out DateTime cutoffDate );  
  
// using lambda syntax:  
CutoffDateDelegate d1 =   
 (out DateTime dt) => { dt = DateTime.Now; return true; };  
  
// using anonymous delegate syntax:  
CutoffDateDelegate d2 =   
 delegate( out DateTime dt ) { dt = DateTime.Now; return true; }

While explicitly declaring arguments as ref/out is expected, having to declare argument types in lambda expression is less common since the compiler can normally infer them. In this case, however, the compiler does not currently infer the types for out or ref arguments in lambda/anon expressions. I'm not certain if this behavior is a bug/oversight or if there's a language reason why this must be so, but there's an easy enough workaround.

**EDIT:** I did a quick check in VS2010 β2, and it still looks like you have to define the argument types - they are not inferred for C# 4.

**I have method that return some data type**

MyType MyMethod()

**If I am running this method into a separate thread, how can this return type be retrieve in the calling thread (that invokes other thread executing MyMethod)?**

There are many ways to do it, here's one:

Func<MyType> func = MyMethod;  
func.BeginInvoke(ar =>  
{  
 MyType result = (MyType)func.EndInvoke(ar);  
 // Do something useful with result  
 ...  
},  
null);

Here's another, using the Task API:

Task.Factory  
 .StartNew(new Func<MyType>(MyMethod))  
 .ContinueWith(task =>  
 {  
 MyType result = task.Result;  
 // Do something useful with result  
 ...  
 });

And a last one, using the Async CTP (preview of C# 5):

MyType result = await Task.Factory.StartNew<MyType>(MyMethod);  
// Do something useful with result  
...

**What happens if you try and use Windows Forms controls from another thread?**

User controls need to run on the UI thread because that is a restriction in the Windows API. If you try and use Windows Forms controls from another thread you will get an exception.

You can run other code in another thread, but use the UI thread to update the controls. You can use BackgroundWorker for this. Or you can use the InvokeRequired and Invoke or BeginInvoke methods on the control instance to have it execute code on the UI thread.

**How does ref and out differ? For reference type, explain the real use of ref parameter.**

<http://stackoverflow.com/questions/961717/c-what-is-the-use-of-ref-for-reference-type-variables>

**What is the difference between ref and out parameters in .NET? What are the situations where one can be more useful than the other? Can anybody illustrate with a code snippet where one can be used and another can't?**

**I am struggling how to use "ref" (to pass argument by reference) in real app. I would like to have simple and mainly meaningful example. Everything I found so far could be easily redone with adding return type to the method. Any idea someone?**

The best example coming in my mind is a function to Swap two variables values:

static void Swap<T>(ref T el1, ref T el2)  
{  
 var mem = el1;  
 el1 = el2;  
 el2 = mem;  
}

Usage:

static void Main(string[] args)  
{  
 string a = "Hello";  
 string b = "Hi";  
  
 Swap(ref a, ref b);  
 // now a = "Hi" b = "Hello"  
  
 // it works also with array values:  
 int[] arr = new[] { 1, 2, 3 };  
 Swap(ref arr[0], ref arr[2]);  
 // now arr = {3,2,1}  
}

A function like this one cannot be done without the ref keyword.

The TryParse methods built into the framework are typical examples. They use out instead of ref but it is the same semantics, it's just that the caller doesn't need to initialize the value. Example:

int result;  
bool isSuccess = int.TryParse("some string", out result);  
if (isSuccess)  
{  
 // use the result here  
}

As you can see the function returns a boolean indicating whether the operation succeeds but the actual result is returned as out parameter.

**What is finalization queue? What is the algorithm for GC?**