Lambdas in C#

Paul Short

We will cover:

- What are Lambdas?
- Evolution of Delegates in C# w/examples
- Lambdas in C# w/examples
- Lambda Parameters
- Lambda Bodies
- Variable Scope in Lambda Expressions
- .Net 2.0 Generic Delegates
- .Net 3.5 Generic Delegates
- Expression Trees
- How To Effectively Use Delegates and Lambdas

What are Lambdas?

- A Lambda expression in C# is “syntactic sugar” used to simplify the definition of a function body (and the parameters passed to it) where you would normally pass in a delegate (function pointer).
- Lambdas are a terse way of expressing anonymous methods (unnamed **delegates**). When used properly, it can increase compactness and readability of code.

```
var food = new List<string> {"pizza", "jelly donut", "beer"};
Debug.Assert(food.Exists(f => f.Contains("donut"))); // must have
// We'll cover what "f => f" means later
```

- Lambdas in C# are still strongly typed, but they do take part in type inference (compiler is able to reconcile the strong typing), which results in more concise code (parameter type declarations become optional).

- LINQ Extension Methods to the .Net libraries often take in lambda expressions as parameters. This is especially true for .Net generic collections and containers, which heavily use iterator and visitation patterns, with lambdas passed in for double-dispatch.

Why the term "Lambda"?

- The syntax and related terminologies used to describe the Lambda feature in C# are loosely borrowed from the lambda calculus (λ-calculus), a formal system in mathematical logic for expressing variable binding and substitution. Although Turing machines are the more common computational models, the Lambda Calculus plays an important role in the theory of programming languages, compiler theory, and proofs.

- Many "pure" functional languages can be viewed as elaborations on the lambda calculus. A functional language is based on a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data, as opposed to imperative (procedural) languages that define sequences of commands, or declarative languages which do not express control flow.

- C# is borrowing just enough terminology and concepts from functional languages in order to simplify the syntax for delegates. Lambdas in C# aid in type inference, but the compiler still resolves them to strong types.
  - In LINQ, LambdaExpression.Compile blurs this line by running a visitor through an expression tree to generate MSIL at runtime, but this is an interesting advanced feature of LINQ itself ("expression trees," not specific to lambda syntax).

Delegates

- Lambdas may seem like magic if you haven't worked with anonymous methods (unnamed delegates), so before we get into lambdas, let's review how delegates work in C#:

- A Delegate is similar to a function pointer (functor) in C and C++.
- Delegates describe the signature of a method (parameter and return types).
- Delegates can be matched to either static or instance methods.
- **Delegates allow methods to be assigned to variables**.
  - *This means you can change the behavior of a piece of code without having to subclass.*
  - *More granular than traditional interface approach (method rather than class/interface).*

Evolution of Delegates in C#

- **C# 1.0: Named Delegates.** Create an instance of a delegate by explicitly initializing it with a method defined elsewhere in the code.
- **C# 2.0: Anonymous Methods.** Unnamed inline statement blocks that can be executed in a delegate invocation.
- **C# 3.0: Lambda Expression.** Similar to anonymous methods but more expressive and concise (compiles down to the same code in MSIL as anonymous methods).

  We will walk through simple examples in code that illustrate all three.

C# 1.0: Named Delegate

Create an instance of a delegate by explicitly initializing it with a method defined elsewhere in the code. (The oldest, most verbose way.)

```
class Test1
{
    delegate void MyDelegate(string s);
    static void MyMethod(string s)
    {
        Console.WriteLine(s);
    }

    static void Main()
    {
       // Original delegate syntax requires init with named method.
        MyDelegate d = new MyDelegate (MyMethod);
        d("Hello, World!");
    }
}
```

C# 2.0: Anonymous Method

Unnamed inline statement blocks that can be executed in a delegate invocation.

```
class Test2
{
    delegate void MyDelegate(string s);
    static void Main()
    {
        // Delegate can be init with inline code block (unnamed method)
        MyDelegate d = delegate (string s) { Console.WriteLine(s); };
        d("Hello, World!");
    }
}
```

C# 3.0: Lambda

Similar to anonymous methods but more expressive and concise (usually compiles down to the same code in MSIL as anonymous methods).

```
class Test3
{
    delegate void MyDelegate(string s);
    static void Main()
    {
        // Delegate can be Lambda statement. => is pronounced “goes to”
        MyDelegate d = x => { Console.WriteLine(x); };
        d(“Hello, World!”);
    }
}
```

Combined Example: Events (EventHandler)

```
// Type for .Click is EventHandler, the .Net predefined delegate for Events:
//     public delegate void EventHandler(object sender, EventArgs e);
private void Form1_Load(object sender, EventArgs e)
{
    // C# 1.0: Named Delegate
    var namedWay = new Button { Text = "Named" };
    namedWay.Click += namedWay_Click;

    // C# 2.0: Anonymous Method (inline, unnamed delegate)
    var anonWay = new Button { Text = "Anonymous", Location = new Point(0, 30) };
    anonWay.Click += new EventHandler(delegate { MessageBox.Show(anonWay.Text); });

    // C# 3.0: Lambda
    var lambdaWay = new Button { Text = "Lambda", Location = new Point(0, 60) };
    lambdaWay.Click += ((s, e) => MessageBox.Show(lambdaWay.Text));

    this.Controls.AddRange(new Control[] { namedWay, anonWay, lambdaWay} );}
}

private void namedWay_Click(object sender, EventArgs e) // Needed for C# 1.0, Named Delegate
{
    MessageBox.Show(((Control)sender).Text);
}
```

C# Lambda Syntax

- So building on what you know about anonymous delegates:
  - Lambdas are a simpler way to express anonymous delegates.
  - Although parameter *types* can be specified explicitly, they are usually left out (inferred).
  - Parameter *names* are arbitrary. (Since the parameter types are often omitted as well, the names simply just appear.)
- There are two fundamental types of Lambdas:
  - Statement Lambdas: just a shorter version of a C# 2.0 anonymous method (code block), do not return values.
  - Expression Lambdas: return a value. (*Predicate Lambdas* are Expression Lambdas where the return value is simply true or false.)
- The lambda operator "**=>**" is often pronounced "goes to"

C# Lambda Syntax (Continued)

- General syntax is parameters on LHS of the => operator and statement (using parameters) on RHS:
  - **(parameters) => statement;**
  - **(parameters) => { statement1; statement2; *etc.* }**

- Lambdas often return values (a.k.a "Expression Lambdas"):
  - ***SomeVariable* = (parameters) => expression;**
  - ***SomeMethod*((parameters) => expression);** (Most Common Usage)

- Let's see some simple examples.
  - We'll cover the finer points of syntax later.
  - Parameter names are in **gray** since they may be confusing the first time you see the syntax. The parameters aren't declared, so just think of them as named placeholders without the type information. (The names themselves are arbitrary).

Example #1: Lambda Statement with one argument

```
using System;
internal class Example1
{
    public delegate void D(int x);
    private static void Main()
    {
        D d = i => { Console.WriteLine(i); };
        d(888);
    }
}
```

Example #2: Lambda Statement with no arguments

```
using System;
internal class Example2
{
    public delegate void D();
    private static void Main()
    {
        D d = () => { Console.WriteLine("Hey!"); };
        d();
    }
}
```

Example #3: Lambda Expression with 3 arguments and a return type

```
using System;
internal class Example3
{
    public delegate long CalcIntIntLong(int a, int b, long c);
    private static void Main()
    {
        CalcIntIntLong multiply = (x, y, z) => x * y * z;
        Console.WriteLine(multiply(2, 3, 4));   // Output: 24
    }
}
```

Example #4: Same as #3, but using the pre-defined Func delegate shortcut (.Net 3.5)

```
using System;
internal class Example4
{
    private static void Main()
    {
        Func<int, int, int, long> multiply = (x,y,z) => x * y * z;
        Console.WriteLine(multiply(2, 3, 4));   // Output: 24
    }
}
```

Example #5: Lambda with LINQ

```
using System;
using System.Collections.Generic;
internal class Example5
{
    private static void Main()
    {
        var greatApes = new List<string> {
             "Orangutan", "Chimpanzee", "Gorilla", "Bonobos" };

        /* "Find" takes in Predicate: List<String>.Find(Predicate<string> match);
           Predicate<string> is “contravariant” to public delegate bool Predicate<T>(T obj);
           "name" is a string (the “string” part of Predicate<string>).
           String.StartsWith(string value) returns a bool. */
        var b = greatApes.Find(name => name.StartsWith("B"));
        Console.WriteLine(b);
    }
}
```

Lambda Parameters

- **Type is optional when it can be inferred:**
  (string s) => {return s.Length;}
  (s) => {return s.Length;}

- **If no parameters, the parenthesis are empty:**
  () => { return outerValue.Length;}

- **If one parameter, the parenthesis are optional:**
  (s) => {return s.Length;}
  s => {return s.Length;}

- **If more than one parameter, use parenthesis and commas:**
  (x, y) => { return x > y; }

Lambda Parameter Naming

- The standard convention is a short identifier whose meaning can be inferred from context:

```
foreach (var orderType in orderTypes.Where(t => t != null)) {…}
```

- Names are local to the lambda and can be reused if desired to represent an item flowing down a chain (*currying* or "fluent-style"):

```
_dict.Where(i => i.Value.IsExpired) // Where takes in Func<T, bool>
     .Select(i => i.key)            // Select takes in Func<T, TResult>
     .ToList();                     // ToList returns List<T>
```

Lambda Bodies

- **If body contains more than one statement, you must separate with semicolons and enclose in braces:**
  s => { Console.WriteLine(s); Console.Out.Flush(); }

- **If body contains only one statement, semicolon and braces are optional and can/should be omitted:**
  s => { Console.Error.WriteLine(s); }
  s => Console.Error.WriteLine(s);

- **If body has only one expression to evaluate, *return* can/should be omitted (typical with most Expression Lambdas):**
  s => return s.Length;
  s => s.Length;

Variable Scope in Lambda Expressions

- Lambda bodies can refer to outer variables that are in scope in the enclosing method or type in which the lambda is defined. Variables captured in this manner are stored for use in the lambda expression even if they would otherwise go out of scope and be garbage collected.
  - Caveat: an outer variable must be assigned before it can be consumed in a lambda expression.
- This is also known as **Closure**, a concept often encountered in functional languages and scripting languages (with functional features) such as JavaScript.
  - This is how LINQ's so-called "deferred execution" feature is implemented.
  - ReSharper (R#, a Visual Studio plug-in) flags potential closure bugs with the "Access to Modified Closure" warning.
- The next slide attempts to illustrate closure.

Variable Scope in Lambda Expressions: Example of a *Closure*

```
using System;
using System.Threading;
class Program
{
    static void Main()
    {
        DoIt();
        GC.Collect(); // Don't try this in production code; here for dramatic effect.
        Console.WriteLine("Main Exit");
        // Environment.Exit(0); // ...don't do this (still need to wait on other thread)
    }

    static void DoIt()
    {
        int outerValue = 42; // This is the answer, don't forget it!
        (new Thread(() =>    // Use Lamdba for ThreadStart delegate parameter
        {
            Thread.Sleep(1000); // Intentional one-second delay
            Console.WriteLine(outerValue); // Access outer value (b/c of closure)
            Console.WriteLine("Thread Exit");
        }
        )).Start();
    }
} // DISCLAIMER: not an example of how threads should be created or managed in production.
```

Generic Delegates

- Delegates can be useful for specifying pluggable runtime behavior.
- They eliminate the need for inheritance in places to specify different behaviors, especially when combined with Generics syntax.
- .Net 2.0 added some generic delegates**, Predicate** and **Action**
- .Net 3.0 added LINQ
- .Net 3.5 added the more versatile **Func** set of generic delegates.
  - You will find them as arguments to LINQ extension methods on .Net Generic Collections.
- Remember, anywhere where a delegate is specified as a parameter, you can directly specify client visitation code or predicates using the Lambda syntax.

.Net 2.0 Generic Delegates

- **Pre**-LINQ (LINQ introduced in 3.0)

- **Predicate<T>: Returns bool (true/false)**

- **Action<T>: Does not return a value**

```
public delegate void Action();
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
```

... and so on, up to up to 16 parameters

- Example (List<T>.FindAll method takes in parameter Predicate<T>):

```
var myList = new List<string> {"dog", "donut", "beer"};
var newList = myList.FindAll(s => s == "donut");
```

.Net 3.5 Generic Delegates

- **Post**-LINQ (LINQ introduced in 3.0)
- Used by **3.5 LINQ Extension Methods**

- **Func<T>: Can return values of any type (not just bool)**

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, in T2, out
TResult>(T1 arg1, T2 arg2);
```

… and so on, up to up to 16 parameters. Note that:

  - **TResult** parameter uses Generics modifer "out" (out=*covariant*, in=*contravariant*)
  - .Net 3.5's **Func<T, bool>** is equivalent to .Net 2.0's **Predicate<T>**

- Example (functionally equivalent to previous slide, except uses the LINQ extension method *Where*, which takes in Func<T, Boolean>):

```
var myList = new List<string> {"dog", "donut", "beer"};
var newList = myList.Where(s => s == "donut");
```

Expression Trees (Code as Data)

- "Expression trees represent code in a tree-like data structure, where each node is an expression… When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression."
  http://msdn.microsoft.com/en-us/library/bb397951.aspx
  http://msdn.microsoft.com/en-us/library/bb335710.aspx
  http://www.cookcomputing.com/blog/archives/000582.html

- When you write code using frameworks such as LINQ, Entity Framework, or MVC3, you are frequently asked to pass in Lamdba expressions which build up expression trees (Abstract Syntax Trees [AST]).
  - Not always transparent to you as a client, but occurs when you pass in a Lambda expression for methods that take in an Expression<TDelegate> parameter.
  - "Deferred execution" or "lazy evaluation"; delayed compile.
  - Expression<TDelegate> can only be assigned to lambda expressions: you cannot assign it to a lambda statement or even an anonymous delegate.

Expression Trees: MVC3 Razor Example

- Within an MVC3 Razor View:

```
<body>
@using (Html.BeginForm())
{
    @Html.LabelFor(model => model.Name)
    @Html.TextBoxFor(model => model.Name)
}
</body>
```

- Results: Display Label "Name" and TextBox containing actual value.

- HtmlHelper.LabelFor (extension method) function prototype:

```
public static MvcHtmlString LabelFor<TModel, TValue>(
    this HtmlHelper<TModel> html,
    Expression<Func<TModel, TValue>> expression)
```

- You're building up an expression tree. Explanations as to *why* it's done this way:
  http://stackoverflow.com/questions/5848940/mvc-html-helpers-and-lambda-expressions
  http://stackoverflow.com/questions/8738243/asp-net-mvc-strongly-typed-html-helpers

How To Effectively Use Delegates and Lambdas

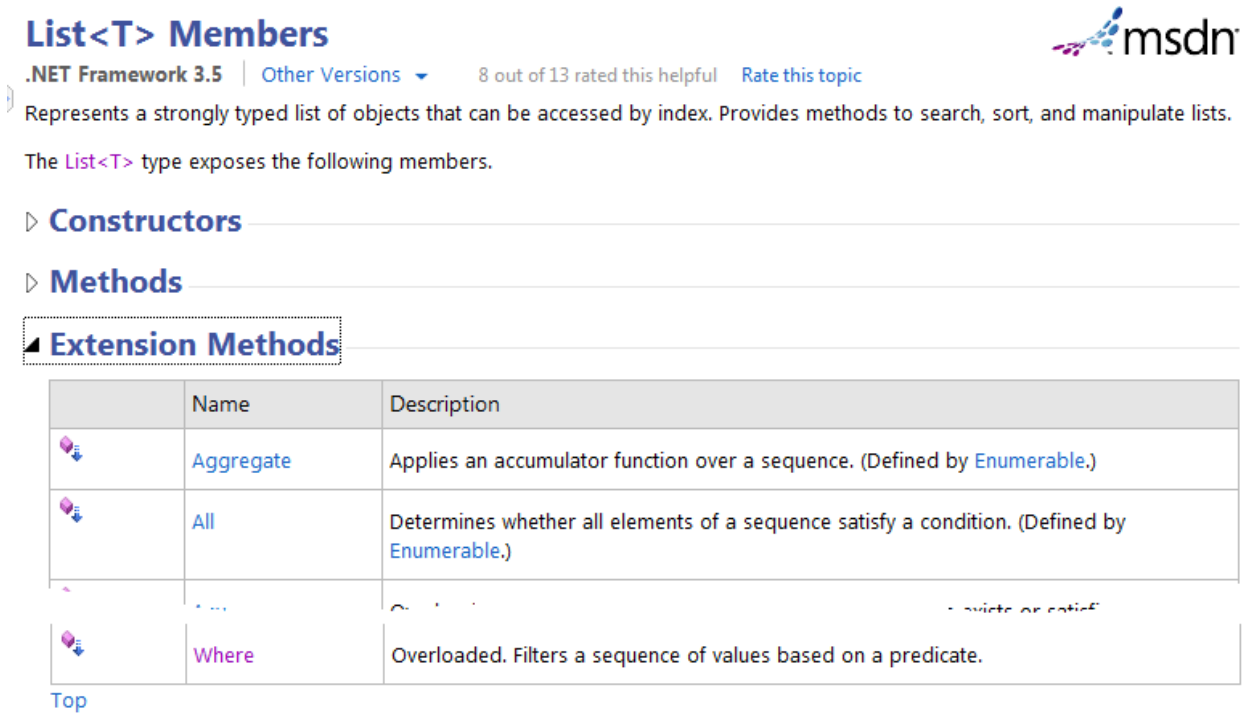
- When you pass in a LINQ expression, take a look at the parameter signatures.
  - *Know your .Net Generic Data Structures:* study the MSDN documentation for **System.Collections.Generic.**
  - *Know your LINQ Extension Methods:* study the MSDN documentation for **System.Collections.Generic** *in detail…* Notice that since .Net 3.5, the documentation for each class contains an "Extension Methods" section. Most extension methods were from **System.Linq**.

- Visual Studio plug-ins such as ReSharper may occasionally suggest and show how to refactor code to simpler syntax using LINQ, Lambdas, and Collections.

- Consider how your delegates, anonymous methods, and lambdas are used in libraries you encounter. Common uses include Sorting, Closures, Currying, Map, Filter, and Fold.

- Stay DRY. Inline is convenient, but isn't an excuse for repetitive copy-and-pasted code. Stick with good design principles. Remember, lambdas are just syntactic tools to help achieve code readability and maintainability.

MSDN DOC Example (1 of 3)



List<T> Members

msdn

.NET Framework 3.5 | Other Versions | 8 out of 13 rated this helpful | Rate this topic

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

The List<T> type exposes the following members.

Constructors

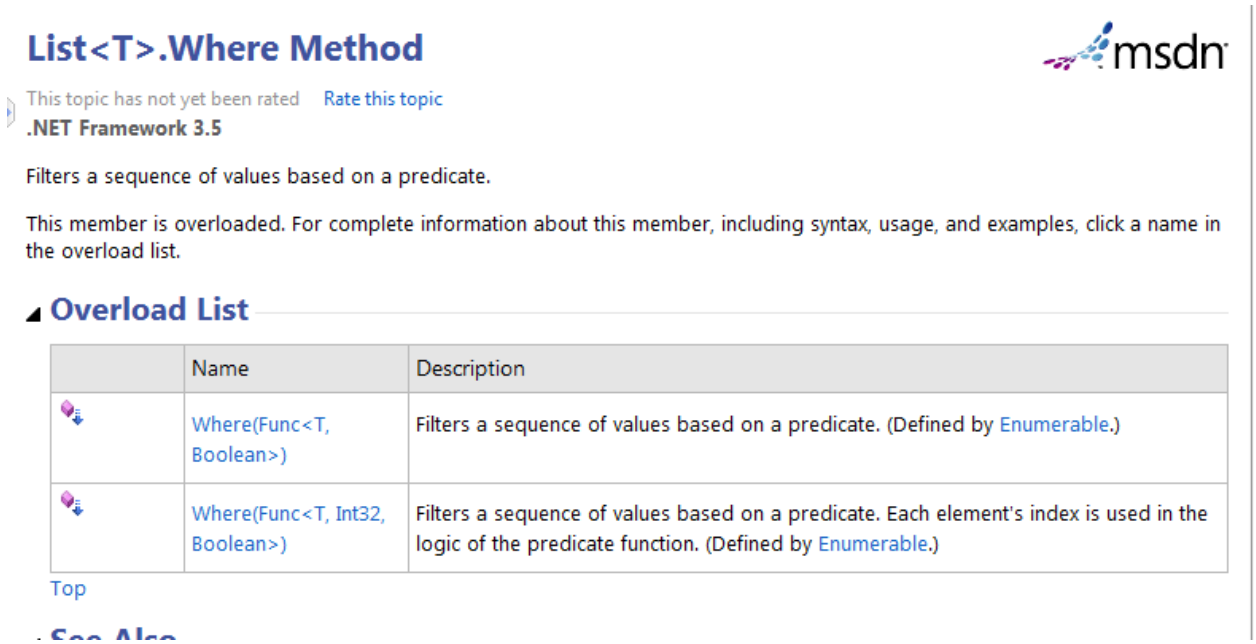
Methods

Extension Methods

	Name	Description
	Aggregate	Applies an accumulator function over a sequence. (Defined by Enumerable.)
	All	Determines whether all elements of a sequence satisfy a condition. (Defined by Enumerable.)
	Where	Overloaded. Filters a sequence of values based on a predicate.

Top

MSDN DOC Example (2 of 3)



List<T>.Where Method

msdn

This topic has not yet been rated Rate this topic

.NET Framework 3.5

Filters a sequence of values based on a predicate.

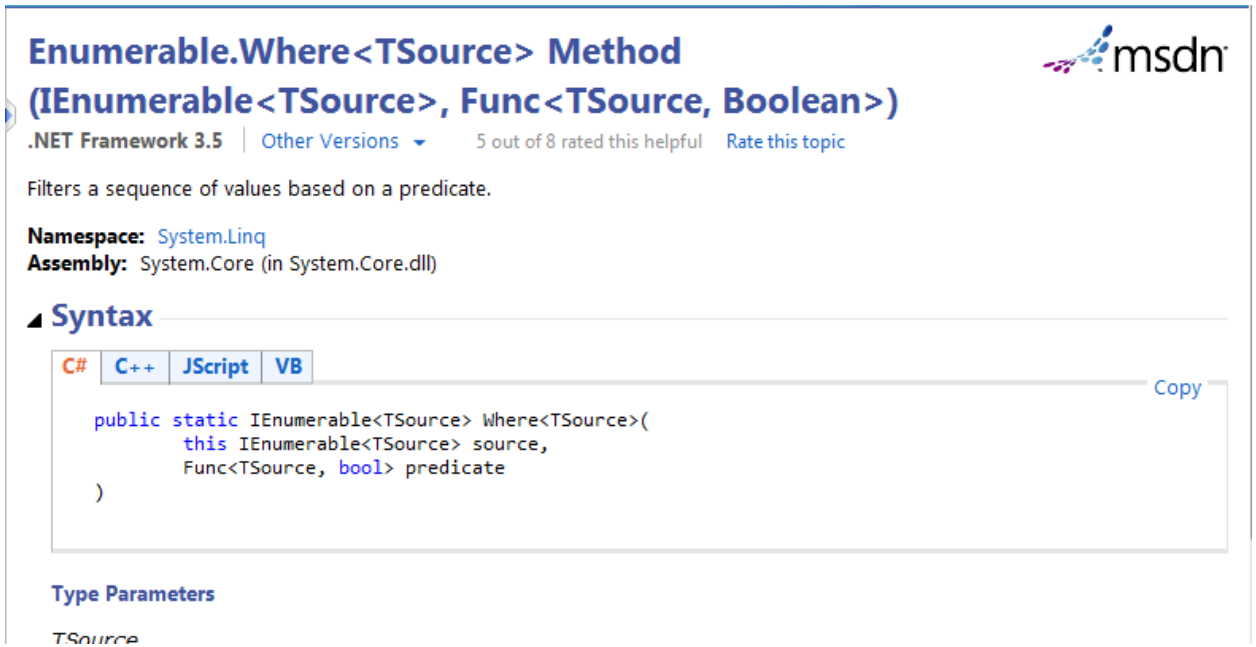
This member is overloaded. For complete information about this member, including syntax, usage, and examples, click a name in the overload list.

Overload List

	Name	Description
	Where(Func<T, Boolean>)	Filters a sequence of values based on a predicate. (Defined by Enumerable.)
	Where(Func<T, Int32, Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function. (Defined by Enumerable.)

Top

MSDN DOC Example (3 of 3)



Enumerable.Where<TSource> Method (IEnumerable<TSource>, Func<TSource, Boolean>)

msdn

.NET Framework 3.5 | Other Versions | 5 out of 8 rated this helpful | Rate this topic

Filters a sequence of values based on a predicate.

Namespace: System.Linq
Assembly: System.Core (in System.Core.dll)

Syntax

C# | C++ | JScript | VB

Copy

```
public static IEnumerable<TSource> Where<TSource>(
        this IEnumerable<TSource> source,
        Func<TSource, bool> predicate
)
```

Type Parameters

TSource

Postscript: History of Delegates

- Delegates were first introduced in Microsoft Visual J++. Sun's (now Oracle's) White Paper "About Microsoft's 'Delegates'" described it as harmful, and recommended interfaces and inner classes instead. http://java.sun.com/docs/white/delegates.html
- That was then, during the Microsoft-Sun J++ war. Hindsight from a decade later, with C# now established, and functional language features such as *delegates*, *lambdas*, and *closures* creeping into most development environments (except for Java, but it is often proposed): http://benhutchison.wordpress.com/2009/02/14/suns-rejection-of-delegates-for-java/ http://weblogs.java.net/blog/2006/08/25/are-closures-just-delegates http://javapapers.com/core-java/java-closures/#&slider1=1
- But it is possible to overuse delegates where interfaces should be used instead. Some simple guidance for C# developers can be found in, "When to Use Delegates Instead of Interfaces": http://msdn.microsoft.com/en-us/library/ms173173(v=vs.100).aspx

Other References

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

Anonymous Functions (C# Programming Guide) http://msdn.microsoft.com/en-us/library/bb882516.aspx

List<T> Members http://msdn.microsoft.com/en-us/library/d9hw1as6(v=vs.90).aspx

Of Lambdas and LINQ. Hare, James Michael. 2012. http://www.slideshare.net/BlackRabbitCoder/of-lambdas-and-linq

You Can't Dance the Lambda. George Mauer, 2009. http://www.slideshare.net/Togakangaroo/cant-dance-the-lambda

"Anonymous Functions," Wikipedia. http://en.wikipedia.org/wiki/Anonymous_function

"Closure (computer science)," Wikipedia. http://en.wikipedia.org/wiki/Closure_(computer_science)

"Lambda Calculus." Wikipedia. http://en.wikipedia.org/wiki/Lambda_calculus

Func And Action Delegates http://www.blackwasp.co.uk/FuncAction.aspx

Functional Programming in C# 3.0 using Lambda Expression (Parts 1 & 2)

http://www.codeproject.com/Articles/33082/Functional-Programming-in-C-3-0-using-Lambda-Expre

http://www.codeproject.com/Articles/33323/Functional-Programming-in-C-3-0-using-Lambda-Expre

Snippet: Simple Lambda example (C#) http://www.codekeep.net/snippets/b12dad17-4d2e-49d1-8975-d0f8ae33fbef.aspx

"Item 29: Support Generic Covariance and Contravariance," Effective C#: 50 Specific Ways to Improve Your C#, 2nd Edition. Wagner, Bill, 2010 http://www.amazon.com/Effective-Covers-4-0-Specific-Development/dp/0321658701

END