

# Learn to Love Lambdas

---

*An overview of Lambda Expressions by JeremyBytes.com*

## Overview

Lambda expressions can be confusing the first time you walk up to them. But once you get to know them, you'll see that they are a great addition to your toolbox. Used properly, they can add elegance and simplicity to your code. And some .NET constructs (such as LINQ) lend themselves to lambda expressions. We'll take a look at how lambda expressions work and see them in action.

Lambda expressions come in two flavors: statement lambdas and expression lambdas. Both types are anonymous delegates (and we'll take a look at what that means). Statement lambdas perform some type of action; expression lambdas return some type of value.

## Anatomy of a Lambda Expression

Lambda expressions consist of 3 parts:

1. A set of parameters
2. The "goes to" operator: =>
3. A set of statements that perform an action or return a value

Let's fire up a Silverlight application and look at lambdas in action.

## The Set Up

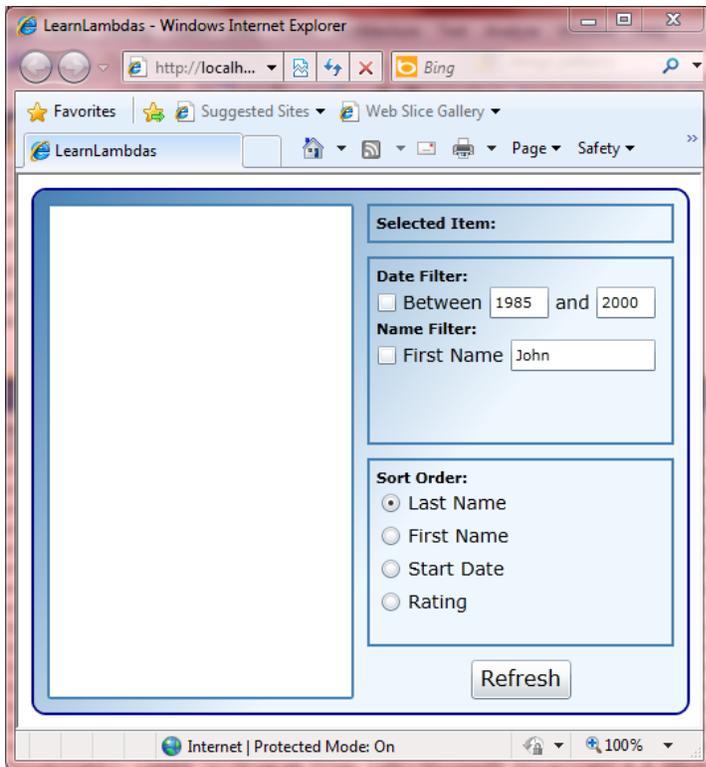
We'll start with a skeleton of a Silverlight project that contains a WCF service. You can download the source code for the application here: <http://www.jeremybytes.com/Downloads.aspx>. The sample code we'll be looking at here is built using Silverlight 4 and Visual Studio 2010. The download includes the starter application and the completed code. The starter application contains the following.

### Projects

The solution contains 2 projects. LearnLambdas is the Silverlight project; LearnLambdas.Web is the asp hosting application. These were both created by choosing new "Silverlight Application" and then opting to "Host the Silverlight application in a new Web site" (the default value).

### Updates

In the Silverlight application, the user interface has been implemented in the MainPage.xaml user control. Here's the running application:



The code-behind page (MainPage.xaml.cs) has a single method added: an empty Click event handler for the Refresh button. We've also added a reference to the LearnLambdas.PersonService namespace; this is the namespace for the service proxy (see PersonService below). Here's the code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using LearnLambdas.PersonService;

namespace LearnLambdas
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void RefreshButton_Click(object sender, RoutedEventArgs e)
        {
        }
    }
}
```

The Silverlight application also contains a Converters.cs file that contains a number of value converters that are used in the display. If you are interested in the XAML and value converters in this project, you

can look up the *Introduction to Data Templates and Value Converters in Silverlight* demo and sample code on the website: <http://www.jeremybytes.com/Downloads.aspx>.

The Web application contains a WCF Service called PersonService. This has the following interface (as defined in IPersonService.cs):

```
[ServiceContract]
public interface IPersonService
{
    [OperationContract]
    List<Person> GetPeople();
}

public class Person
{
    public string FirstName;
    public string LastName;
    public DateTime StartDate;
    public int Rating;
}
```

This is a simple service that defines a `Person` object with 4 properties and a single operation to get a list of `Person` objects. The service implementation is a hard-coded list in `PersonService.cs` (you can check the downloaded code for details).

So with that in place, let's get started!

## Fetching the Data

We want to fetch the data when we click the Refresh button, so we'll add the code we need to the Click event handler. This consists of 4 steps.

1. Create a proxy:

```
var proxy = new PersonServiceClient();
```

2. Hook up the service Completed event handler / callback. We'll let VisualStudio create the reference and stub for us. First, type the "`proxy.GetPeopleCompleted +=`". Visual Studio IntelliSense will offer to create a new event handler. If you press Tab twice, it will add the reference and generate the stub. Here's what we end up with:

```
proxy.GetPeopleCompleted +=
    new EventHandler<GetPeopleCompletedEventArgs>(proxy_GetPeopleCompleted);
```

But we can shorten this a little bit:

```
proxy.GetPeopleCompleted += proxy_GetPeopleCompleted;
```

3. Call the Async method to kick off the method:

```
proxy.GetPeopleAsync();
```

This will start the call to the WCF service asynchronously (as all Silverlight calls are). When the service call returns, the Completed event handler / callback will run.

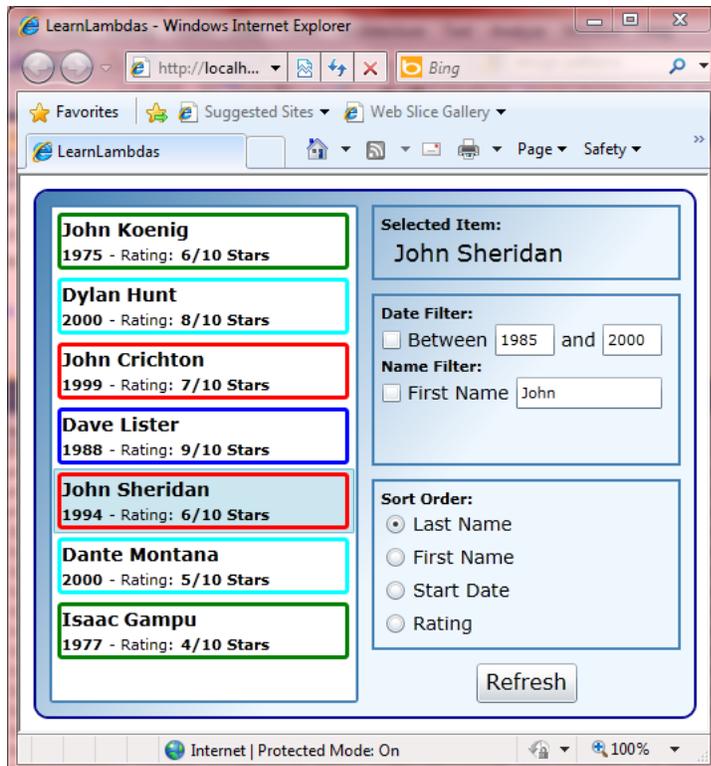
4. Bind the data to the UI. Now, we just need to add code to the callback to put our data into the ListBox on our UI. Here's the code in its entirety:

```
public MainPage()
{
    InitializeComponent();
}

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += proxy_GetPeopleCompleted;
    proxy.GetPeopleAsync();
}

void proxy_GetPeopleCompleted(object sender, GetPeopleCompletedEventArgs e)
{
    PersonListBox.ItemsSource = e.Result;
}
```

Now, if we run the application, click the "Refresh" button, and then select an item in the list, we end up with something that looks like this:



Notice that the “Selected Item” section shows which item we clicked. If you select another item, then this section updates automatically. Here’s something interesting, though: if you click the “Refresh” button again, the selection disappears. This is because we are assigning a new data set to the ListBox each time we click the Refresh button. A little bit later, we’ll take a look at how to keep the selection.

## Upgrading to an Anonymous Delegate

What we have in place right now is a named delegate – or simply a “delegate”. (An event handler is a special kind of delegate.) But instead of a named delegate, we can use an anonymous delegate. This basically in-lines the delegate code.

To do this, we use the `delegate` keyword followed by the parameters, then add the body of the method. Here’s the code:

```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted +=
        delegate(object proxySender, GetPeopleCompletedEventArgs proxyArgs)
        {
            PersonListBox.ItemsSource = proxyArgs.Result;
        };
    proxy.GetPeopleAsync();
}
```

One thing to note: we had to change the names of the parameters since `sender` and `e` are already in scope (from the Button’s Click event). We can now get rid of the named delegate. When we run the application now, we’ll see exactly the same results as above.

## And Now the Lambda Expression

As previously mentioned, a lambda expression is simply an anonymous delegate. This means that we can easily convert our anonymous delegate to a lambda expression. In this case, we will create a statement lambda – a lambda that performs some action (but does not return a value). This conversion is as simple as removing the `delegate` keyword and adding the “=>” operator between the parameters and the statement.

```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted +=
        (object proxySender, GetPeopleCompletedEventArgs proxyArgs) =>
        {
            PersonListBox.ItemsSource = proxyArgs.Result;
        };
    proxy.GetPeopleAsync();
}
```

Lambda expressions have one more property that makes them easy to work with. The parameter types are completely optional if the compiler can figure them out. This is generally true with event handlers since they have a well-defined signature: `object sender` and some kind of `EventArgs`. This leaves us with the following:

```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
    {
        PersonListBox.ItemsSource = proxyArgs.Result;
    };
    proxy.GetPeopleAsync();
}
```

The parameters are still strongly typed. If you hover your cursor over “`proxyArgs`” you will see that it is still of type `GetPeopleCompletedEventArgs`.

A couple more syntactic notes: First, if there is only one statement, then the curly braces are optional. In our case, we’ll be adding some more code, so we’ll leave the braces. Second, if there is only one parameter, then the parentheses are optional. We’ll see this a little later.

If we run the application now, we will see exactly the same behavior as with the named delegate and the anonymous delegate. One of the reasons that I like this syntax is that it keeps the functionality together. Ultimately, when we click the button, we want to populate the data in the list box. Using the lambda expression, we keep the callback logic “inside” the button click method. This works fine for simple callbacks. If our callback starts getting larger, then we would probably want to factor it to a separate method for readability. But as we will see in just a bit, this does not mean that we want to give up the lambda expression.

## Here’s the Cool Part: Captured Variables

We’ve done a lot of work so far, but ultimately, we end up with the same functionality we had when we started. So, what’s the big deal? There’s another cool feature of lambda expressions: captured variables. A lambda expression can actually use a variable after it has gone out of scope. How’s this possible? Let’s take a look at fixing our selection problem.

Problem: when we click the “Refresh” button, the selection goes away. To fix this, we’ll save off the selected item from the `ListBox` before we refresh the data, then after the refresh, we’ll locate that item and re-select it. Here’s the code:

```

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    Person selPerson = PersonListBox.SelectedItem as Person;

    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
    {
        PersonListBox.ItemsSource = proxyArgs.Result;
        if (selPerson != null)
            foreach (Person itm in PersonListBox.Items)
                if (itm.FirstName == selPerson.FirstName &&
                    itm.LastName == selPerson.LastName)
                {
                    PersonListBox.SelectedItem = itm;
                }
    };
    proxy.GetPeopleAsync();
}

```

This is a fairly straight forward process. First, we save off the currently selected item in the `selPerson` variable. Then in our callback, we check to see if the `selPerson` variable is populated. If so, then we loop through the items in the Listbox and try to locate that person (based on the first name and last name properties). If we find a match, then we set the selected item.

But let's think about this for a moment. Here's a timeline of what happens:

1. The Refresh button is clicked in the UI.
2. The Click event fires and the `RefreshButton_Click` method runs.
  - a. The `selPerson` variable is set.
  - b. The service proxy is created.
  - c. The `GetPeopleCompleted` event handler is hooked up.
  - d. The `GetPeopleAsync` method is called.
  - e. The `RefreshButton_Click` method exits.
3. The data is returned from the service and the callback fires.
  - a. The result data is assigned to the Listbox.
  - b. The `SelectedItem` of the Listbox is set using the `selPerson` variable.

Step 3 runs after the `RefreshButton_Click` method exits, which means that the `selPerson` variable is out of scope. You can check this yourself by setting a breakpoint and stepping through the code. So how does this work?

The lambda expression "captures" the variable. This means that the variable stays in scope even though it normally would be released. What makes this good in our case is that we can use a locally scoped variable. If we were to do this same process with the named delegate (where we started), then we would need a class-level variable to hold our selected value since we would need to reference that value in multiple methods.

This is a great feature, but there is one thing that we need to be aware of. When the captured variable is evaluated in the lambda expression, it contains the value at the time of evaluation, not the time of capture. Let's add a line of code to our click event to see what this means:

```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    Person selPerson = PersonListBox.SelectedItem as Person;

    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
    {
        PersonListBox.ItemsSource = proxyArgs.Result;
        if (selPerson != null)
            foreach (Person itm in PersonListBox.Items)
                if (itm.FirstName == selPerson.FirstName &&
                    itm.LastName == selPerson.LastName)
                {
                    PersonListBox.SelectedItem = itm;
                }
    };
    proxy.GetPeopleAsync();
    selPerson = null;
}
```

Now, if we run our application, we will see that the selected person is not saved between refreshes (exactly the same behavior we had initially). This is because the value of `selPerson` is set to null before the lambda expression body is executed. So, even though `selPerson` had a value at the time it was captured (when the `GetPeopleCompleted` handler is assigned), it is null by the time the code is executed.

In this particular instance, we do not need to be concerned about this. But if you are capturing variables that are changing (such as an indexer from a "for" loop), you may find your results are not what you expect.

Note: before continuing, be sure to comment out or remove the line of code that we just added. We want to make sure that we continue to keep our selection as we go forward.

## Lambdas and LINQ

Lambda expressions are used extensively in LINQ (Language Integrated Query). LINQ statements can be built with either query syntax or with extension method syntax. The extension methods included in the `System.Linq` namespace extend the `IEnumerable<T>` and `IQueryable<T>` interfaces. The good news is that most collections implement `IEnumerable<T>`.

Expression lambdas (lambda expressions that return a value) are often used with these LINQ methods. Our UI has a couple of filtering options (by date range and by first name). Let's take a look at implementing these using LINQ. We will be using the `Where` method which has the following signature (from the Help documentation):

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

So what does this mean? First, this is an extension method. We know this from the use of the `this` keyword before the first parameter (`source`). For an overview of extension methods, see *Quick Bytes: Extension Methods* on the website: <http://www.jeremybytes.com/Downloads.aspx>. The second parameter (`predicate`) is listed as a `Func<TSource, bool>`. `Func` is a pre-defined delegate signature. If you want more details on `Func`, see *Quick Bytes: Get Func<>-y* on the website: <http://www.jeremybytes.com/Downloads.aspx>. Here, the “`TSource`” denotes the incoming parameter value, and “`bool`” denotes the return value. This signature lends itself to implementation with a lambda expression.

Since our `Where` statement takes an `IEnumerable<T>` and returns an `IEnumerable<T>`, we can easily set up a method to handle our filters. We’ll start with the easier of the 2 filters (the first name filter):

```
private IEnumerable<Person> AddFilters(IEnumerable<Person> data)
{
    if (NameFilterCheckBox.IsChecked.Value)
        data = data.Where((Person p) => p.FirstName == NameTextBox.Text);

    return data;
}
```

Our method takes an `IEnumerable<Person>` (`data`) and returns an `IEnumerable<Person>`. In going back to our definition of `Where`, the `source` is `data` (the parameter passed to the method) and the `predicate` is our lambda expression. Since `Where` is an extension method on `IEnumerable<T>`, we can treat it as if it were a method of our `data` object. The `predicate` (our lambda) then becomes the only parameter. The parameter for the lambda expression is “`Person p`”; the type is determined by the type of the collection. The parameter name is up to you. By convention, lambda expressions have a single character parameter; I chose “`p`” for `Person` – our underlying data type.

The body of the lambda expression is a comparison of the `FirstName` property and the contents of our text box. As we noted earlier, we can exclude the parameter type from the declaration since the compiler already knows it. Also, since we only have 1 parameter, we can exclude the parentheses as well. Here’s our slightly modified version:

```
private IEnumerable<Person> AddFilters(IEnumerable<Person> data)
{
    if (NameFilterCheckBox.IsChecked.Value)
        data = data.Where(p => p.FirstName == NameTextBox.Text);

    return data;
}
```

A note about the `if` statement: the `NameFilterCheckBox.IsChecked` property is actually a 3-state value (true, false, null/unknown). Because of this, we need to check the `Value` property (which uses false for null/unknown). `if` supports true/false values, but not nulls.

With all of this in mind, let's go ahead and implement the other filter (the date range filter). Our final method looks like this:

```
private IEnumerable<Person> AddFilters(IEnumerable<Person> data)
{
    int startYear = Int32.Parse(StartDateTextBox.Text);
    int endYear = Int32.Parse(EndDateTextBox.Text);

    if (DateFilterCheckBox.IsChecked.Value)
        data = data
            .Where(p => p.StartDate.Year >= startYear)
            .Where(p => p.StartDate.Year <= endYear);

    if (NameFilterCheckBox.IsChecked.Value)
        data = data.Where(p => p.FirstName == NameTextBox.Text);

    return data;
}
```

The first “if” condition checks to see if the `DateFilterCheckBox` is checked and then compares the values for the start and end years. Note the syntax inside the conditional:

```
data = data
    .Where(p => p.StartDate.Year >= startYear)
    .Where(p => p.StartDate.Year <= endYear);
```

The second and third lines are continuations of the first line. This syntax kind of threw me the first time I saw it. But this is the equivalent of this:

```
data = data.Where(...).Where(...);
```

The line breaks get ignored by the compiler. The lambda expressions should look pretty familiar now. Each takes a single parameter (“p”) which is of type `Person`. The expressions compare the `StartDate` of the `Person` to the start and end years from the UI. We could also have included a single “where” statement with an “&&” between the conditions.

This works because most of the LINQ extensions on `IEnumerable<T>` also return an `IEnumerable<T>`. You can keep appending methods together basically piping the result of one method to the parameter of the next. And being able to append multiple `Where` methods is important for the second conditional in our `AddFilters` method (which we saw earlier):

```
data = data.Where(p => p.FirstName == NameTextBox.Text);
```

Note that our `AddFilters` method will work if one, both, or neither of the filters is selected from the UI. If both filters are selected, then the `data` object ends up internally like this:

```
data.Where(p => p.StartDate.Year >= startYear)
    .Where(p => p.StartDate.Year <= endYear);
    .Where(p => p.FirstName == NameTextBox.Text);
```

This is more the magic of LINQ than the magic of lambda expressions, but you can see how being able to append multiple where conditions makes this method fairly readable.

### Calling the AddFilters Method

The last step to getting our filters to work is to call the `AddFilters` method when we Refresh the data. Here's our updated method:

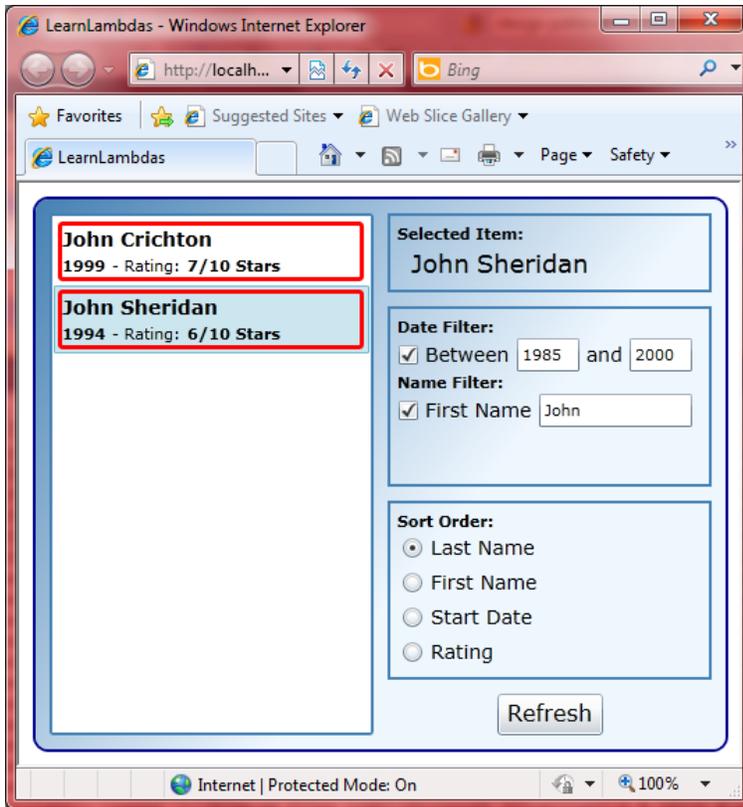
```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    Person selPerson = PersonListBox.SelectedItem as Person;

    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
    {
        PersonListBox.ItemsSource = AddFilters(proxyArgs.Result);
        if (selPerson != null)
            foreach (Person itm in PersonListBox.Items)
                if (itm.FirstName == selPerson.FirstName &&
                    itm.LastName == selPerson.LastName)
                {
                    PersonListBox.SelectedItem = itm;
                }
    };
    proxy.GetPeopleAsync();
}
```

We can use the `AddFilters` method like this because it takes an `IEnumerable<T>` as a parameter and spits out an `IEnumerable<T>` as a result. This makes the types transparent. Our `proxyArgs.Result` implements `IEnumerable<T>` (the actual type in our case is `ObservableCollection<Person>`), so we can use this as our parameter. And the output can be assigned to the `PersonListBox.ItemsSource` property.

In a production application, we probably would not want to implement filters this way. This is just a simplification so that we can see how the filtering with LINQ and lambdas works. One of the problems we would want to avoid in production is the need to refresh the data each time we apply a filter. Instead, we could keep an in-memory copy of the data from our service and then apply the filters as needed, potentially by hooking into the `Click` or `Checked` events of the checkboxes.

Now that everything is hooked up, we can include one or both of our filters. Here's what our application looks like if we check both filters with their default values:



At this point, we can make a selection and experiment with the filters. If you select a person, then include a filter that includes the person, then that person will still be selected after applying the filter. If you select a person that is not included in the filter, then the selection is cleared after applying the filter.

Now that we have filtering, let's look at sorting.

## Implementing the Sort Order

For sorting, we'll use some LINQ methods that have a different signature. The most important of these is `OrderBy`:

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

The first parameter (`source`) is the same as the `Where` method. But the second parameter (`keySelector`) is of type `Func<TSource, TKey>`. To use a lambda expression with this `Func`, we use the same `TSource` as the incoming parameter value, but return a `TKey` – this will be a property of our `Person` object in our case. Also note that the output is an `IOrderedEnumerable` (instead of `IEnumerable`).

Here's the method signature for our `AddSort`:

```
private IOrderedEnumerable<Person> AddSort (IEnumerable<Person> data)
```

This takes the same `IEnumerable<Person>` parameter (`data`) as we had before. But notice that the return type is `IOrderedEnumerable<Person>`.

Here's a simple snippet from the method:

```
if (DateSortButton.IsChecked.Value)
    return data.OrderBy(p => p.StartDate);
```

Our lambda expression is pretty simple. Our one parameter is "p" (of type `Person`), and the body of the lambda is just a property name. This will be the `TKey` and will be sorted using the default comparison method for the data type. For this to be valid, the `TKey` type must implement `IComparable` (and many .NET types do). In our case, `StartDate` is a `DateTime` which does implement `IComparable`. There is also an overload of `OrderBy` that takes a custom `IComparer` if the object does not implement `IComparable` or you want to define a custom method of sorting.

In addition to `OrderBy`, there is `OrderByDescending` (which will order items in descending order – largest to smallest), as well as `ThenBy` (which allows you to include a secondary or tertiary sort). The `ThenBy` method takes an `IOrderedEnumerable<Person>`, so it is generally used in conjunction with an `OrderBy`.

With all of these options in mind, here's our final `AddSort` method:

```
private IOrderedEnumerable<Person> AddSort (IEnumerable<Person> data)
{
    if (LastNameSortButton.IsChecked.Value)
        return data.OrderBy(p => p.LastName);

    if (FirstNameSortButton.IsChecked.Value)
        return data.OrderBy(p => p.FirstName).ThenBy(p => p.LastName);

    if (DateSortButton.IsChecked.Value)
        return data.OrderBy(p => p.StartDate);

    if (RatingSortButton.IsChecked.Value)
        return data.OrderByDescending(p => p.Rating);

    return data.OrderBy(p => p.LastName);
}
```

Note: since our sort selection in the UI is made up of radio buttons, only one sort field is selected at a time. This allows us to simply `return` our value.

### Calling the `AddSort` Method

The last step is to call the `AddSort` method. We're going to nest our method calls. Here's our assignment to the `PersonListBox.ItemsSource` that is in our callback:

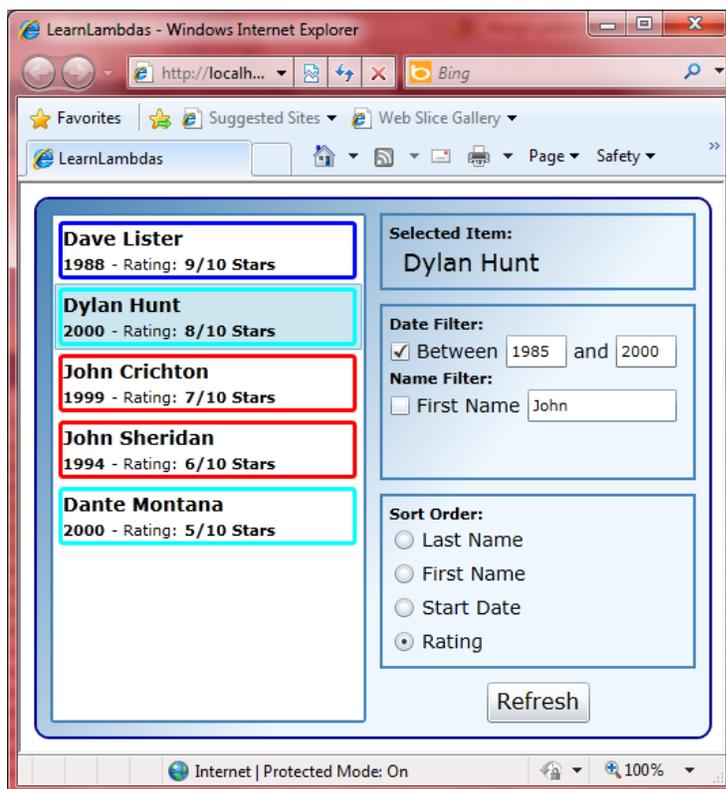
```

proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
{
    PersonListBox.ItemsSource = AddSort(AddFilters(proxyArgs.Result));
    if (selPerson != null)
        foreach (Person itm in PersonListBox.Items)
            if (itm.FirstName == selPerson.FirstName &&
                itm.LastName == selPerson.LastName)
            {
                PersonListBox.SelectedItem = itm;
            }
};

```

We can nest our `AddSort` and `AddFilters` methods because `AddFilters` returns an `IEnumerable<Person>` and `AddSort` takes an `IEnumerable<Person>` as a parameter. Now we can use any combination of filtering and sorting in our application.

Here's our output:



We've got a filter set ("Between 1985 and 2000") and a sort order ("Rating"). Since the Rating implemented with `OrderByDescending`, the highest rating is on top.

## Refactoring a Bit

Now that we're comfortable with lambda expressions and LINQ, let's take a look at refactoring our button click method just a little bit. The problem is with the `foreach` loop that we use to set our selection:

```

foreach (Person itm in PersonListBox.Items)
    if (itm.FirstName == selPerson.FirstName &&
        itm.LastName == selPerson.LastName)
    {
        PersonListBox.SelectedItem = itm;
    }

```

What are we trying to do here? Well, we are trying to set the `SelectedItem` of the List Box to the value of `selPerson` (if we can find it in the list). This `foreach` loop seems a little inside-out to me. Our assignment is nested 2 levels deep. And it's not particularly clear what would happen if the value is not found, or even worse, if there are two matching values found. We can use LINQ and lambdas to make this more readable.

For this, we'll use the `SingleOrDefault` method. Here's the signature (from the Help documentation):

```

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

```

You'll notice that this signature is very similar to the `Where` method that we saw earlier. The parameters (`source` and `predicate`) are the same. The only difference is that it returns a single object (`TSource`) instead of a collection (`IEnumerable<TSource>`).

The `SingleOrDefault` method returns a single value based on a condition (the `predicate`). If there are no values found, then the default value for the type is returned. In our case, we'll be using this with a collection of `Person` objects. If no matching object is found, then a null `Person` object (the "Default") is returned.

Here's our updated implementation:

```

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    Person selPerson = PersonListBox.SelectedItem as Person;

    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
    {
        PersonListBox.ItemsSource = proxyArgs.Result;
        if (selPerson != null)
            PersonListBox.SelectedItem =
                PersonListBox.Items.SingleOrDefault(
                    p => ((Person)p).FirstName == selPerson.FirstName &&
                        ((Person)p).LastName == selPerson.LastName
                );
    };
    proxy.GetPeopleAsync();
    //selPerson = null;
}

```

In going back to the definition of `SingleOrDefault`, the source is `PersonListBox.Items` (which is a collection that implements `IEnumerable<object>`). The `predicate` is our conditional expression. The parameter for our lambda expression is “`object p`”. The type is `object` because this is the type of our collection (the `TSource`).

The body of our lambda expression is a comparison of the `FirstName` and `LastName` properties. Notice that we have to cast our “`p`” to a `Person` since it comes in as type `object`.

Let’s compare our 2 implementations:

```
foreach (Person itm in PersonListBox.Items)
    if (itm.FirstName == selPerson.FirstName &&
        itm.LastName == selPerson.LastName)
    {
        PersonListBox.SelectedItem = itm;
    }

PersonListBox.SelectedItem =
    PersonListBox.Items.SingleOrDefault(
        p => ((Person)p).FirstName == selPerson.FirstName &&
            ((Person)p).LastName == selPerson.LastName
    );
```

These two statements accomplish the same function. But which is more readable? Personally, I like the version with the lambda expression. It’s very clear what we are doing in the first line: we are assigning the `SelectedItem` property of the `ListBox`. In addition, the lambda expression itself makes it clear which element of the “`Items`” collection we’re looking for (assuming that we are comfortable with lambda expression syntax). Finally, since we are using `SingleOrDefault`, we know that we will only get one value back. If there is more than one value matching the `predicate`, then we will get a runtime error. If there are no matching values, then our `SelectedItem` will be set to null.

## Refactoring a Bit More

Earlier I mentioned that we may want to look at factoring out the callback if it gets too complex. Let’s look at how we can do that and still keep our lambda expression and the benefits we’ve seen with the capture variable.

First, we’ll highlight the body of the lambda expression (this is just the part between the curly braces, not the parameters or the “`=>`” operator). Then we right-click, choose “`Refactor`”, then “`Extract Method`”. We can then give it a name like “`UpdateUI`”. Here’s the result we end up with:

```

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    Person selPerson = PersonListBox.SelectedItem as Person;

    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted += (proxySender, proxyArgs) =>
    {
        UpdateUI(selPerson, proxyArgs);
    };
    proxy.GetPeopleAsync();
    //selPerson = null;
}

private void UpdateUI(Person selPerson,
    GetPeopleCompletedEventArgs proxyArgs)
{
    PersonListBox.ItemsSource = AddSort(AddFilters(proxyArgs.Result));
    if (selPerson != null)
        PersonListBox.SelectedItem =
            PersonListBox.Items.SingleOrDefault(
                p => ((Person)p).FirstName == selPerson.FirstName &&
                    ((Person)p).LastName == selPerson.LastName
            );
}

```

Notice that Visual Studio was helpful enough to include the `selPerson` and `proxyArgs` as parameters when it extracted the method. This means that we maintain a reference to our captured variable, and our separate method simply references it through the parameter that is passed in. So, we lose the benefit of keeping everything together in the click method (that was mentioned above), but we still manage to keep the “flow” of the process together. As with all things in programming, most of the time we are finding the balance between readability and maintainability. And it usually makes sense to extract methods when they reach a certain level of complexity.

## Why I Love Lambdas

We’ve taken a look at using lambda expressions in some useful ways. Based on what we’ve done, here’s why I have become a huge fan of lambdas.

### No Explicit Parameter Types Needed

Since the compiler can figure out the parameter types, there is no need to explicitly state them. This is especially useful when you are implementing an unfamiliar event handler. I know that there is a “sender”, but sometimes I don’t know the specific `EventArgs` type that I need to use. This way, I can just name the parameter and let the compiler figure out the specific type.

### Implementation Code Near the Calling Code

In our initial callback implementation with the named delegate, the callback method (`GetPeopleCompleted`) is separated from the initial call (`GetPeopleAsync`). I like how the lambda expression keeps the callback implementation near the initiating call. This is especially useful in the asynchronous world of Silverlight applications where I find myself making multiple calls for fetching data

and saving data. And even if we end up factoring out complex methods, we still keep the “flow” together.

### **Captured Variables**

We saw how useful captured variables are when saving off and restoring a selected item in our list. I like the idea of using a locally-scoped variable as opposed to a class-level variable. This follows the best practice of limiting your variable exposure.

### **Readability**

Once I got used to reading lambda expressions, I found them to be more understandable in most situations. We saw this above when we converted our “SelectedItem” assignment from a foreach loop to a LINQ method with a lambda expression.

### **Easy Implementation of Func<T> and Action<T>**

As we saw when looking at the LINQ methods, Func<T> and Action<T> (and their variants) are used as pre-defined delegates. These were made to be implemented with lambda expressions. It is possible to create a named delegate based on Func<T>, but it is very unusual.

## **Wrap Up**

So, lambda expressions are extremely useful in a variety of situations. We only looked at a couple here (a callback handler plus some LINQ methods), but there are many others. We have seen that lambda expressions are simply anonymous delegates. With this understanding in hand, lambdas become very readable and easy to use – a great addition to your developer toolbox.

Happy coding!