**LINQ Methods With or Without Predicates: What's the Difference?**

I really like lambda expressions, so I talk about them quite a bit in my live presentations, including [Learn to Love Lambdas (and LINQ, Too!)](http://www.jeremybytes.com/Demos.aspx#LLL). I often get questions about different parts of LINQ, so it's time to put some articles together about them.  
  
For the first question:

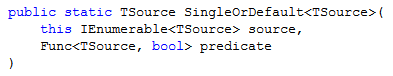
**What's the difference between calling Single with a parameter and calling Where and then Single without a parameter?**

To answer this, let's first look at the code that inspired this.  
  
**The Declarative Programming Approach**  
In my sample code, I show the difference between imperative programming (where we tell the computer *how* to do something) and declarative programming (where we tell the computer *what* we want and let it figure out how to do it). You can see an explanation of this here: [LINQ and the Functional Approach](http://jeremybytes.blogspot.com/2014/02/linq-and-functional-approach.html).  
  
In this block of code, we want to save off the currently selected item of a list box so that we can re-assign the selection after reloading data in the list box. Here are the relevant bits of code:

|  |
| --- |
| <https://3.bp.blogspot.com/-107CAK-iIjA/VOwa32y-TQI/AAAAAAAAEPw/GN2xULWcbEA/s1600/SelectedPerson.png> |
| Saving the Selected Item |

|  |
| --- |
| <https://4.bp.blogspot.com/-dKNpGcK1ybk/VOwa-lmYLOI/AAAAAAAAEP4/Z5zJLWQqDak/s1600/SampleSingleWithPredicate.png> |
| Re-assigning the Selection |

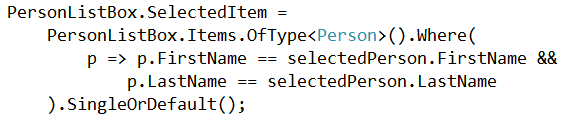
In this code, we use the overload of "SingleOrDefault" that takes a predicate as a parameter. Here's the method signature from the documentation:

[](https://1.bp.blogspot.com/-1u-mnb2-dmo/VOwbqIVBS0I/AAAAAAAAEQA/9k0lHYsxP3w/s1600/SingleOrDefaultWithPredicate.png)

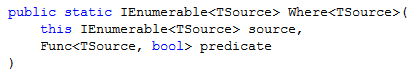
The "predicate" parameter is a "Func<TSource, bool>". This means that we need to provide a delegate that takes a "Person" as a parameter (since our collection has "Person" objects in it), and it needs to return a "bool" value: true or false. In our example, we use a lambda expression (because lambdas are really awesome).  
  
So, this is a filter on our collection. The "SingleOrDefault" method expects to find a single matching item or no matching item. If it finds a single item, it returns that; if it finds no matching items, it returns the default value for the type ("null" for reference types and bitwise 0 for value types). If it happens to find more than one matching item, it throws an exception.  
  
**Another Way to Call SingleOrDefault**  
Now, there is another overload for "SingleOrDefault" -- in this case, it takes no parameters at all. (Well, technically one parameter since it's an extension method; but no additional parameters.) Here's the method signature:

[https://1.bp.blogspot.com/-RYhTpynYj7o/VOwdAGo-ouI/AAAAAAAAEQM/GUtrDXwEgGw/s1600/SingleOrDefault.png](https://1.bp.blogspot.com/-RYhTpynYj7o/VOwdAGo-ouI/AAAAAAAAEQM/GUtrDXwEgGw/s1600/SingleOrDefault.png)

On it's own, this isn't very useful, but it's often used in conjunction with the "Where" method. Here's our example from above re-written in this format:

[](https://3.bp.blogspot.com/-TnuCiFCxKGI/VOwdMV9CaxI/AAAAAAAAEQU/NGExUOZuK74/s1600/SampleSingleWithWhere.png)

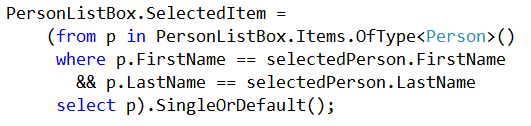
We call "Where" using the same lambda expression as we used above. But notice that after we call "Where", we make a call to "SingleOrDefault" with no parameters.  
  
Here's the method signature for "Where":

[](https://1.bp.blogspot.com/--wIoBB3W7P0/VOwhlUmVPaI/AAAAAAAAEQs/grMl55znhPQ/s1600/Where.png)

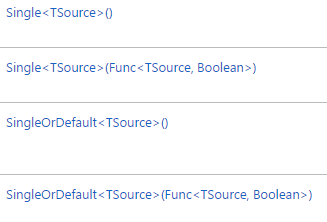
As we see, this uses the same "Func<TSource, bool>" as "SingleOrDefault". That means we can use the same lambda expression with the "Where" method, and we'll end up with the same result.  
  
**What's the Difference?**  
So the question that I get from time to time is "What's the difference?" Does it really matter which of these options we use?

**Behaviorally, there is no difference.**

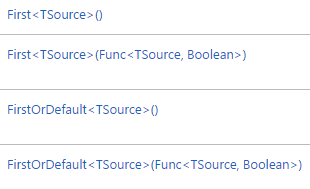
Whether we put the predicate as a parameter in the specific method or use a separate "Where" method, the results will be the same.  
  
**Differences in Readability**  
There are some differences when we talk about readability. I prefer to use the method syntax for LINQ (also often referred to as the "fluent" syntax). This is where we "dot" our methods together.  
  
When using the fluent syntax, I prefer to put the filter conditional into the method itself (and not have the separate "Where"). This results in a shorter syntax since we have one less method in our chain.  
  
But it's also common to use the query syntax. Here's how we would do the same thing using the query syntax in LINQ:

[](https://1.bp.blogspot.com/-DjZ70nqgFXw/VOweiPS9SkI/AAAAAAAAEQg/1jTvawyCUJ4/s1600/SingleWithQuerySyntax.png)

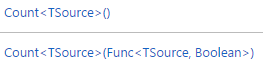
The query syntax lets us write something that looks more like a SQL query. Here we use the keywords "from", "in", "where", and "select". (And there are others as well. We'll be looking at "join" in an upcoming article.)  
  
One of the limitations of using the query syntax is that not all LINQ functionality is available -- "SingleOrDefault" is one of these unavailable items. So if we want to use this, we wrap our entire query in parentheses and then we can use the method syntax to call "SingleOrDefault" (or "First" or "Count" or many others).  
  
When we use the query syntax, it's very common to use the LINQ method that does \***not**\* take a predicate. It's a bit more readable because we can put more things into the "query" part of the statement, and then add on "SingleOrDefault" right at the end.  
  
**Similar LINQ Methods**  
I really love LINQ. There are so many methods that are very easy to use. Be sure to look through the whole list sometime: [IEnumerable<T> Extension Methods](https://msdn.microsoft.com/en-us/librarY/ckzcawb8(v=vs.110).aspx" \t "_blank).  
  
If we go through this list, we find quite a few that have overloads that take no parameter or a "Func<TSource, bool>". Here are some of my favorites:

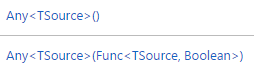
[](https://3.bp.blogspot.com/-W3SLi7jJSrw/VOwiPEWQBJI/AAAAAAAAEQ0/P1_RUknTcHw/s1600/Single.png)

"Single" will return a single item. If it finds no items or more than one item, it throws an exception. And we've already seen how "SingleOrDefault" works.  
  
This also has a useful cousin:

[](https://1.bp.blogspot.com/-TY2dczK6r4Q/VOwilNoO0hI/AAAAAAAAEQ8/JPlE1CAUx2Y/s1600/First.png)

"First" will return a single item. If it finds more than one item, it returns the first one it comes to (but unlike "Single", it does not throw an exception if there is more than one match). If it finds no items, it throws an exception. You can probably figure out what "FirstOrDefault" does based on what we've seen so far.  
  
If you're looking for something a little bit different, be sure to look up "Last" and "LastOrDefault".  
  
In addition, we have "Count" and "Any":

[](https://4.bp.blogspot.com/-YP-VLeorTy4/VOwjE31wvPI/AAAAAAAAERE/Nt-umeKORPk/s1600/Count.png)

[](https://2.bp.blogspot.com/-WMqWZg53g2I/VOwjJUqBTdI/AAAAAAAAERM/TAn4RXxNGJs/s1600/Any.png)

"Count" tells us how many items are in the collection, and "Any" returns "true" if the collection is not empty and "false" if it is. This is very useful if we want to know if there is at least one matching item (but we don't really care what the item is).  
  
None of these methods have matching keywords in the LINQ query syntax. This means that if we want to use them, we need to tack them on to the end of our query like we did in the example above. In those instances, it makes sense to use the versions with no parameters (and put the filter inside the query part).  
  
But if we're using the fluent syntax, it's often easier to read if we use the versions that take the predicate as a parameter.  
  
**Wrap Up**  
So behaviorally, it doesn't really matter which version we choose. That means we need to think about readability. This often comes down to preference. There are many developers who prefer the query syntax, and there are many developers who prefer the fluent syntax.  
  
The best advice I can give is **be consistent**. If we use similar methods in similar ways throughout our code, it will be easier to read and follow. But if we mix things up and use different syntax in different parts of our code, our brains have to do a context switch to understand what's going on. It's best to avoid this if we can.  
  
I have a few more questions to answer about LINQ. Specifically, we'll take a quick look at the "OfType" method, and we'll also take a detailed look at the parameters of the "Join" method. If we use the fluent syntax with "Join", we usually end up with 3 separate lambda expressions as parameters. It's not difficult to understand once we break it down a bit. So, stayed tuned for that.  
  
I really love LINQ. Once you learn to decode the method signatures from the documentation, it's very easy to take advantage of the power and flexibility that these methods provide.  
  
Happy Coding