Understanding Predicate Delegates in C#

[**anshudutta**](http://www.codeproject.com/script/Membership/View.aspx?mid=7117674), 2 Oct 2010[CPOL](http://www.codeproject.com/info/cpol10.aspx)

Introduction

NET Framework 2.0 came with the concept of Predicate Delegates, but compared to the other features of .NET, it never got the attention it deserved. It is really a powerful concept which makes it easy to write searching algorithms on collections. These are also widely used while performing filter operations on WPF object data binding. In this article, we will take a look into the concepts behind predicate delegates and try and understand their importance.

What is a predicate delegate?

A predicate delegate is a delegate with the following signature:

* Return type - bool
* Argument type - generic

So, a predicate delegate is a delegate which points to a boolean function that returns true or false and takes a generic type as an argument. A predicate delegate thus is a delegate which is capable of taking any custom type as an argument. This makes it quite useful because what we get as a result is a generic delegate. The bool function that it points to has logic to evaluate any condition and return true/false accordingly.

Why use a predicate delegate?

The most common use of a predicate delegate is for searching items in a collection. Because a predicate delegate is a delegate of type T or generic, it is most useful when searching items in a generic collection. It really comes handy when you have a collection with tons of items and you need to perform a search operation. Using a predicate delegate not only reduces the lines of code that needs to be written, but also increases performance of the application. These are also used while performing filter operations on IcollectionView objects while performing data binding in WPF.

For example, create a console application and declare an Employee class. Declare these properties for the Employee class: FirstName, LastName, and Designation. See code below:

Hide Shrink http://www.codeproject.com/images/arrow-up-16.pngCopy Code

class Employee

{

private string \_firstName;

private string \_lastName;

*//private int \_empCode;*

private string \_designation;

public Employee()

{ }

public Employee(string firstName, string lastName, string designation)

{

\_firstName = firstName;

\_lastName = lastName;

\_designation = designation;

}

/// *<summary>*

/// *Property First Name*

/// *</summary>*

public string FirstName

{

get { return \_firstName; }

set { \_firstName = value; }

}

/// *<summary>*

/// *Property Last Name*

/// *</summary>*

public string LastName

{

get { return \_lastName; }

set { \_lastName = value; }

}

public string Designation

{

get { return \_designation; }

set { \_designation = value; }

}

In the Main() method of the console application, declare three employees as below:

Hide Copy Code

*// Declare 3 employees*

Employee emp1 = new Employee("Anshu", "Dutta", "SSE");

Employee emp2 = new Employee("John", "Doe", "Manager");

Employee emp3 = new Employee("Jane", "Doe", "Assistant");

Create a generic List and put the above employees in the list:

Hide Copy Code

List<Employee> empList = new List<Employee> { emp1, emp2, emp3 };

We will demonstrate the use of a predicate delegate by performing a search operation on the Employee List.

Method #1 - The traditional way of doing it

Create a bool function that searches an employee first name and returns true if found:

Hide Copy Code

private static bool EmpSearch(Employee emp)

{

if (emp.FirstName == "Anshu")

return true;

else

return false;

}

Declare a predicate delegate pointing to that function:

Hide Copy Code

Predicate<Employee> pred = new Predicate<Employee>(EmpSearch);

Call the Find() method of the List<> and pass the predicate delegate as an argument:

Hide Copy Code

Employee emp = empList.Find(pred);

The Find() method internally iterates through each item in the list and calls the function EmpSearch, passing each item from the list as an argument one by one. When the FirstName of the employee matches the condition, the function returns true. The Find method then returns that employee object to the Employee placeholder emp. That means for each item in the list, the Find() method calls the function EmpSearch passing the items as arguments one by one. Whichever item satisfies the condition in the function is then returned.

Print the result to check:

Hide Copy Code

Console.WriteLine("Employee Found {0}",emp.FirstName);

Console.ReadLine();

This method has a disadvantage. What happens when we need to alter our search? We would need to create another search method and do the same thing as above. Actually, this method was just to explain things in an elaborate way. The better way of doing this is by using anonymous functions of C#.

Method #2 - Using anonymous functions

Consider the following code:

Hide Copy Code

emp = new Employee();

emp = empList.Find(delegate(Employee e)

{

if (e.FirstName == "Anshu")

return true;

else

return false;

});

The functionality here is the same, but instead of declaring a separate predicate delegate and manually pointing it to an external function, we use an anonymous function. The result is much less verbose in terms of lines of code. We directly invoke the anonymous function by the delegate which takes an employee object as argument.

Again, print the result to check:

Hide Copy Code

Console.WriteLine("Employee Found {0}", emp.FirstName);

Method #3 - Using a lambda expression

The lines of code can be further reduced by using => or lambda expressions. Lambda expressions are used to invoke delegates with much lesser lines of code. See the code snippet below:

Hide Copy Code

emp = new Employee();

emp = empList.Find((e)=> {return (e.FirstName == "Anshu");});

The Find() method still takes a predicate delegate. ()=> is used to invoke a delegate with an anonymous function. Here, we need to pass an argument in the anonymous function of type Employee.

(e)=> does just that. Note that intellisense already expects an object of type Employee when typing the Lambda expression. This is because, we are performing a search operation on a generic list of type Employee, and hence the compiler expects a delegate to a bool function which will pass an argument of type Employee used for evaluating the search condition of the anonymous function. Thus we write:

Hide Copy Code

((e)=> {return (e.FirstName == "Anshu");}

Notice that our logic has been reduced to a single line of code.

Finally, print the result again to check:

Hide Copy Code

Console.WriteLine("Employee Found {0}", emp.FirstName);

The Find method here will stop searching when it encounters the first condition. So if we have two employees with the same first name, the employee listed at the top of the list will be returned.

Searching a list of employees

To search more than one result, use the FindAll() method. It is the same as the Find() method, but returns a collection of objects meeting the search condition. The below code searches all employees with J in their first name:

Hide Copy Code

List<Employee> employees = new List<Employee>();

employees = empList.FindAll((e) => { return (e.FirstName.Contains("J")); });

Print the result:

Hide Copy Code

Console.WriteLine("List of employess with J in Firstname");

foreach (Employee e in employees)

{

Console.WriteLine("{0},{1}", e.FirstName, e.LastName);

}

Console.ReadLine();

Index search

You can also narrow down your search radius by specifying the start index and the number of items to search along with the predicate delegate that invokes the search criteria through the anonymous function. See the code below:

Hide Copy Code

int index = empList.FindIndex(0,2,

(e) => { return (e.FirstName.Contains("J")); });

*//print result*

Console.WriteLine("Index search");

Console.WriteLine("Index returned {0}", index);

The code searches any employee with a first name containing J from the zeroth element and spanning two counts (i.e., 0, 1 for our case).

The concepts of the other methods are the same. FindLast() and FindLastIndex do the same things, but return results from the last matches rather than the first as seen above.

Summary

In this article, we took a look into the concepts behind a predicate delegate and how they can be used to implement search operations on any collection.