Delegates and lambdas

Article • 01/05/2022 • 4 minutes to read

A delegate defines a type that represents references to methods that have a particular parameter list and return type. A method (static or instance) whose parameter list and

return type match can be assigned to a variable of that type, then called directly (with the appropriate arguments) or passed as an argument itself to another method and then called. The following example demonstrates delegate use.

C#

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- The `public delegate string Reverse(string s);` line creates a delegate type of a method that takes a string parameter and then returns a string parameter.
- The `static string ReverseString(string s)` method, which has the exact same parameter list and return type as the defined delegate type, implements the delegate.
- The `Reverse rev = ReverseString;` line shows that you can assign a method to a variable of the corresponding delegate type.
- The `Console.WriteLine(rev("a string"));` line demonstrates how to use a variable of a delegate type to invoke the delegate.

In order to streamline the development process, .NET includes a set of delegate types that programmers can reuse and not have to create new types. These types are `Func<>`, `Action<>` and `Predicate<>`, and they can be used without having to define new delegate types. There are some differences between the three types that have to do with the way they were intended to be used:

- `Action<>` is used when there is a need to perform an action using the arguments of the delegate. The method it encapsulates does not return a value.

- `Func<>` is used usually when you have a transformation on hand, that is, you need to transform the arguments of the delegate into a different result. Projections are a good example. The method it encapsulates returns a specified value.
- `Predicate<>` is used when you need to determine if the argument satisfies the condition of the delegate. It can also be written as a `Func<T, bool>`, which means the method returns a boolean value.

We can now take our example above and rewrite it using the `Func<>` delegate instead of a custom type. The program will continue running exactly the same.

C#

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

For this simple example, having a method defined outside of the `Main` method seems a bit superfluous. .NET Framework 2.0 introduced the concept of *anonymous delegates*, which let you create "inline" delegates without having to specify any additional type or method.

In the following example, an anonymous delegate filters a list to just the even numbers and then prints them to the console.

C#

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
```

```
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(
          delegate (int no)
          {
              return (no % 2 == 0);
          }
        );

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

As you can see, the body of the delegate is just a set of expressions, as any other delegate. But instead of it being a separate definition, we've introduced it *ad hoc* in our call to the List<T>.FindAll method.

However, even with this approach, there is still much code that we can throw away. This is where *lambda expressions* come into play. Lambda expressions, or just "lambdas" for short, were introduced in C# 3.0 as one of the core building blocks of Language Integrated Query (LINQ). They are just a more convenient syntax for using delegates. They declare a parameter list and method body, but don't have a formal identity of their own, unless they are assigned to a delegate. Unlike delegates, they can be directly assigned as the right-hand side of event registration or in various LINQ clauses and methods.

Since a lambda expression is just another way of specifying a delegate, we should be able to rewrite the above sample to use a lambda expression instead of an anonymous delegate.

C#

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
```

```
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(i => i % 2 == 0);

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

In the preceding example, the lambda expression used is `i => i % 2 == 0`. Again, it is just a convenient syntax for using delegates. What happens under the covers is similar to what happens with the anonymous delegate.

Again, lambdas are just delegates, which means that they can be used as an event handler without any problems, as the following code snippet illustrates.

C#

```
public MainWindow()
{
    InitializeComponent();

    Loaded += (o, e) =>
    {
        this.Title = "Loaded";
    };
}
```

The `+=` operator in this context is used to subscribe to an event. For more information, see How to subscribe to and unsubscribe from events.

Further reading and resources

- Delegates
- Lambda expressions