WORKING WITH DELEGATES

Gulnaz Zhomartkyzy
D. Serikbayev EKSTU

What You will learn in this lesson

- Understanding delegates and predefined delegate types
- Using anonymous methods
- Using anonymous methods including lambda expressions

Delegates

A ***delegate*** is a type that defines a method signature.

In C++, for example, you would do this with a function pointer.

In C# you can instantiate a ***delegate*** and let it point to another method. You can invoke the method through the *delegate*.

1.1 The following code shows how you can define a delegate type.

[*accessibility*] delegate *returnType DelegateName*([*parameters*]);

Here's a breakdown of that code:

- *accessibility*: An accessibility for the delegate type such as **public** or **private**.
- *delegate*: The delegate **keyword**.
- *returnType*: The **data type** that a method of this delegate type returns such as *void*, *int*, or *string*.
- *delegateName*: The **name** that you want to give the delegate type.
- *parameters*: The parameter list that a method of this delegate type should take.

1.Example 1.

For example, the following code defines a delegate type named **FunctionDelegate**.

```
private delegate float FunctionDelegate(float x);
```

This type represents methods that take a **float as a parameter** and r**eturns an integer**.

After you define a delegate type, you can create a variable of that type.

The following code declares a variable named **TheFunction** that has the **FunctionDelegate** type:

```
private FunctionDelegate TheFunction;
```

Later you can set the **variable equal to a method** that has the appropriate **parameters** and **return type**. The following code defnes a method named **Function1**.

```
  // y = 12 * Sin(3 * x) / (1 + |x|)
private static float Function1(float x)
{
  return (float)(12 * Math.Sin(3 * x) / (1 +
Math.Abs(x)));
}
```

The form's Load event handler then sets the variable **TheFunction** equal to this method.

```
// Initialize TheFunction
private void Form1_Load(object sender, EventArgs e)
{   TheFunction = Function1;
}
```

After the variable **TheFunction** is initialized, the program can use it as if it were the method itself.

For example, the following code snippet sets the variable **y** equal to the value returned by **TheFunction** with parameter.

```
private void button1_Click(object sender, EventArgs e)
{
  float y = TheFunction(1.23f);
  textBox1.Text = y.ToString();
}
```

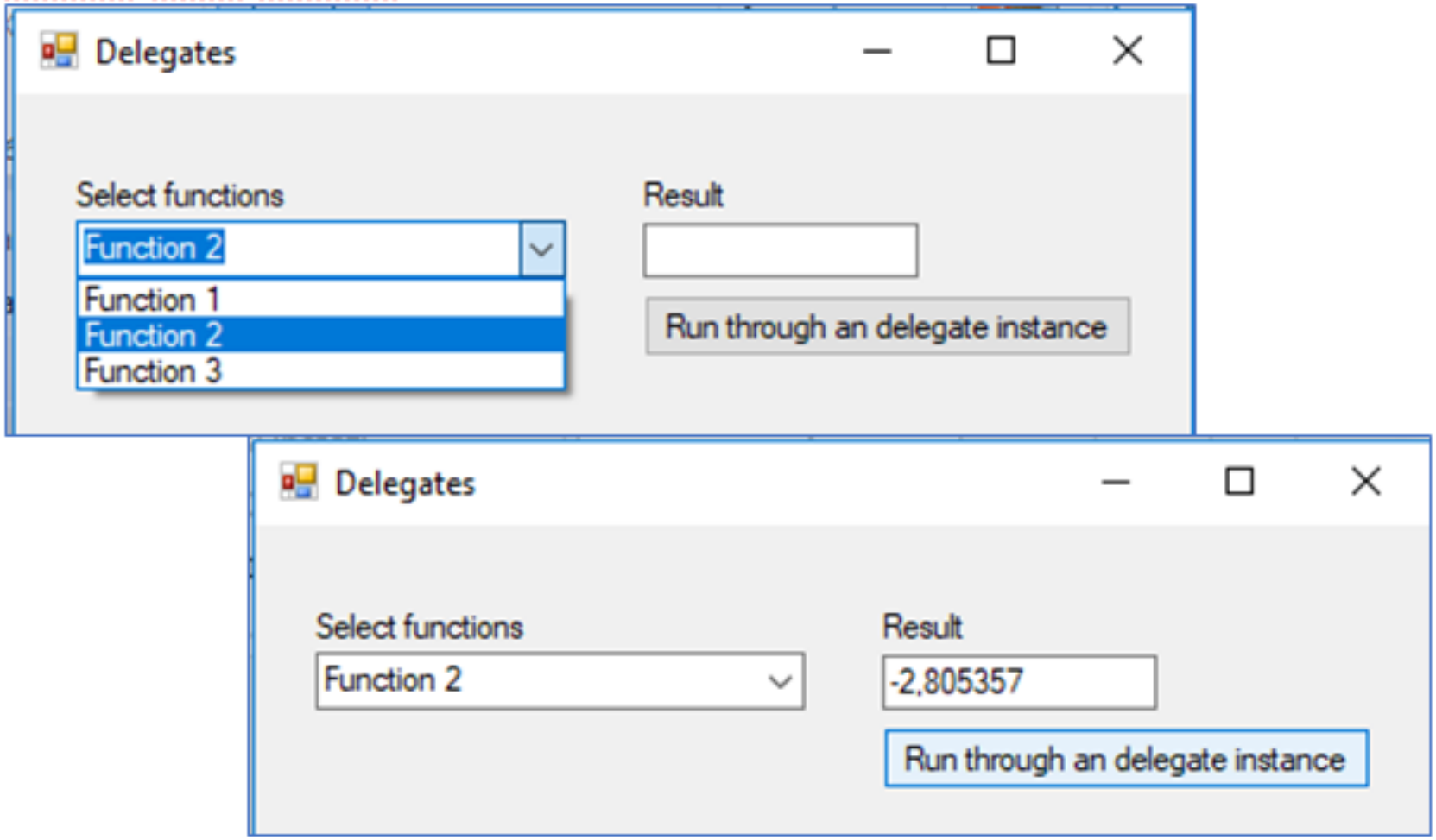
At this point, you don't actually know which method is referred to by **TheFunction**.

The variable could refer to **Function1** or some other method, as long as that method has a signature that matches the **FunctionDelegate** type.

Select a function from the program's ComboBox

```
private void comboBox1_SelectedIndexChanged(object
sender, EventArgs e)
        { switch (comboBox1.SelectedIndex)
            {   case 0:
                    TheFunction = Function1; break;
                case 1:
                    TheFunction = Function2; break;
                case 2:
                    TheFunction = Function3; break;
            }
        }
```

Forms with results.



1.3 Example 2.

Next listing shows an example of declaring a *delegate* and calling a method through it.

Listing. Using a delegate

```
public delegate int Calculate(int x, int y);

public int Add(int x, int y) { return x + y; }

public int Multiply(int x, int y) { return x * y; }

public void UseDelegate()
        {
            Calculate calc = Add;
            Console.WriteLine(calc(3, 4)); // Displays 7
            calc = Multiply;
            Console.WriteLine(calc(3, 4)); // Displays 12
        }
```

1. 4 Anonymous Methods

An ***anonymous method*** is basically a method that **doesn't have a name.**

Instead of creating a method as you **usually do**, you create a delegate that refers to the code that the method should contain.

By using anonymous methods, you reduce the coding overhead in instantiating delegates because you do not have to create a separate method.

You can then use that delegate as if it were a delegate variable holding a reference to the method.

The following shows the syntax for creating an **anonymous method**.

delegate ([*parameters*]) { *code*... }

Here's a breakdown of that code:

- delegate: The delegate keyword.
- parameters: Any parameters that you want the method to take.
- code: Whatever code you want the method to execute. The code can use a return statement if the method should return some value.

Example 1:

```
// Create a delegate.
delegate void Del(int x);
 // Instantiate the delegate using an anonymous method.
Del d = delegate(int k) { /* ... */ };
```

Example 2:

```
// Declare a delegate.
delegate void Printer(string s);
. . .
static void Main()
{
// Instatiate the delegate type using an anonymous method.
  Printer p = delegate(string j)
        { System.Console.WriteLine(j);
        };

// Results from the anonymous delegate call.
p("The delegate using the anonymous method is called.");
}

/* Output:
    The delegate using the anonymous method is called.
```

1.5 Built-in Delegate Types

The .NET Framework defines two generic delegate types that you can use to avoid defining your own delegates in many cases:

- Action and
- Func.

Action Delegates

The generic Action delegate represents a method that returns void. Different versions of Action take between 0 and 18 input parameters.

The following code shows the defnition of the Action delegate that takes two parameters:

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

Func Delegates

The generic Func delegate represents a method that returns a value. As is the case with Action, different versions of Func take between 0 and 18 input parameters.

The following code shows the defnition of the Func delegate that takes two parameters:

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

2 Lambda Expressions

2.1 Lambda Expressions

Definition 1.

Anonymous methods give you a shortcut for creating a short method that will be used in only one place. In case that isn’t short enough, lambda methods provide a shorthand notation for creating those shortcuts.

A ***lambda expression*** uses a concise syntax to create an anonymous method.

Definition 2.

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

2 Lambda Expressions

To create a lambda expression, you specify

1) **input parameters** (if any) on the left side of the lambda operator =>, and

2) you put the **expression** or **statement block** on the other side.

2 Lambda Expressions

For example, the lambda expression **x => x * x** specifies a **parameter** that's named x and **returns** the value of x squared. You can assign this expression to a delegate type, as the following example shows:

```
delegate int del(int i);

static void Main(string[] args)
{    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Compare with the delegate using an anonymous method:

```
Del d = delegate(int k) {/* ... */};
```

Lambda expressions come in a few formats and several variations. To make discussing them a little easier, the following sections group lambda expressions into three categories:

- expression lambdas,
- statement lambdas, and
- async lambdas.

2 Lambda Expressions

2.2 Expression Lambdas

A lambda expression with an expression on the right side is called an ***expression lambda***. An expression lambda returns the result of the expression and takes the following basic form:

```
(input parameters) => expression
```

The parentheses are optional only if the lambda has one input parameter; otherwise they are required. Two or more input parameters are separated by commas enclosed in parentheses:

```
(x, y) => x == y
```

2 Lambda Expressions

Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:

```
(int x, string s) => s.Length > x
```

Specify zero input parameters with empty parentheses:

```
() => SomeMethod()
```

2.3 Statement Lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

```
(input parameters) => {statement;}
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
delegate void TestDelegate(string s);
…
TestDelegate myDel = n => { string s = n + " " +
"World"; Console.WriteLine(s); };
myDel("Hello");
```

3 SUMMARY

This block is designed as a way for you to quickly study the key points of this lesson.

Working with delegates

–A delegate is a type that represents a kind of method. It defnes the method’s parameters and return type.

–Often the name of a delegate type ends with **Delegate** or **Callback**.

–You can use **+** and **–** to combine delegate variables.

For example, if a program executes the statement **del3 = del1 + del2**,

then del3 will execute the methods referred to by del1 and del2.

–If a delegate variable refers to an instance method, it executes with the object on whose instance it was assigned.

3 SUMMARY

–The .NET Framework defnes two built-in delegate types that you can use in many cases: **Action** and **Func**. The following code shows the declarations for Action and Func delegates that take two parameters:

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

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

3 SUMMARY

–An anonymous method is a method with no name. The following code saves a reference to an anonymous method in variable function:

```
Func<float, float> function = delegate(float x)
{ return x * x; };
```

3 SUMMARY

– A lambda expression uses a concise syntax to create an anonymous method. The following code shows examples of lambda expressions:

```
Action note1 = () => MessageBox.Show("Hi");
Action<string> note2 = message => MessageBox.Show(message);
Action<string> note3 = (message) =>
MessageBox.Show(message);
Action<string> note4 = (string message) =>
MessageBox.Show(message);
Func<float, float> square = (float x) => x * x;
```

3 SUMMARY

- An expression lambda evaluates a single expression whose value is returned by the anonymous method.
- A statement lambda executes a series of statements. It must use a return statement to return a value.
- An async lambda is a lambda expression that includes the async keyword.