La Programación Básica en Entorno Visual (II)

UNIDAD N° 3

(Parte 3/3)

2023

Understanding user interface events

Event Data

Data that is associated with an event can be provided through an event data class. The .NET Framework provides many event data classes that you can use in your applications. For example, the SerialDataReceivedEventArgs class is the event data class for the SerialPort.DataReceived event. The .NET Framework follows a naming pattern of ending all event data classes with `EventArgs`.

Event Handler

To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you are handling. In the event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, your event handler method must subscribe to the event. **EventHandler and EventHandler<>**

```
static void c_ThresholdReached(object sender, EventArgs e)
{
    Console.WriteLine("The threshold was reached.");
}
```

The *Func<T, ...>* and *Action<T, ...>* delegate types

The parameter taken by the *Average*, *Max*, *Count*, and other methods of the *List<T>* class are actually generic *Func<T, TResult>* delegates; the type parameters refer to the type of the parameter passed to the delegate and the type of the return value. For the *Average*, *Max*, and *Count* methods of the *List<Person>* class shown in the text, the first type parameter *T* is the type of data in the list (the *Person* struct), whereas the *TResult* type parameter is determined by the context in which the delegate is used. In the following example, the type of *TResult* is *int* because the value returned by the *Count* method should be an integer:

```
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
```

So, in this example, the type of the delegate expected by the *Count* method is *Func<Person, int>*.

Código

Lambda expressions and delegates

The easiest way to specify the predicate is to use a *lambda expression*. A lambda expression is an expression that returns a method. This sounds rather odd because most expressions that you have encountered so far in C# actually return a value. If you are familiar with functional programming languages such as Haskell, you are probably comfortable with this concept. If you are not, fear not: lambda expressions are not particularly complicated, and after you have become accustomed to a new bit of syntax, you will see that they are very useful.

```
// Find the member of the list that has an ID of 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });
```

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
// Create and populate the personnel list
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};
```

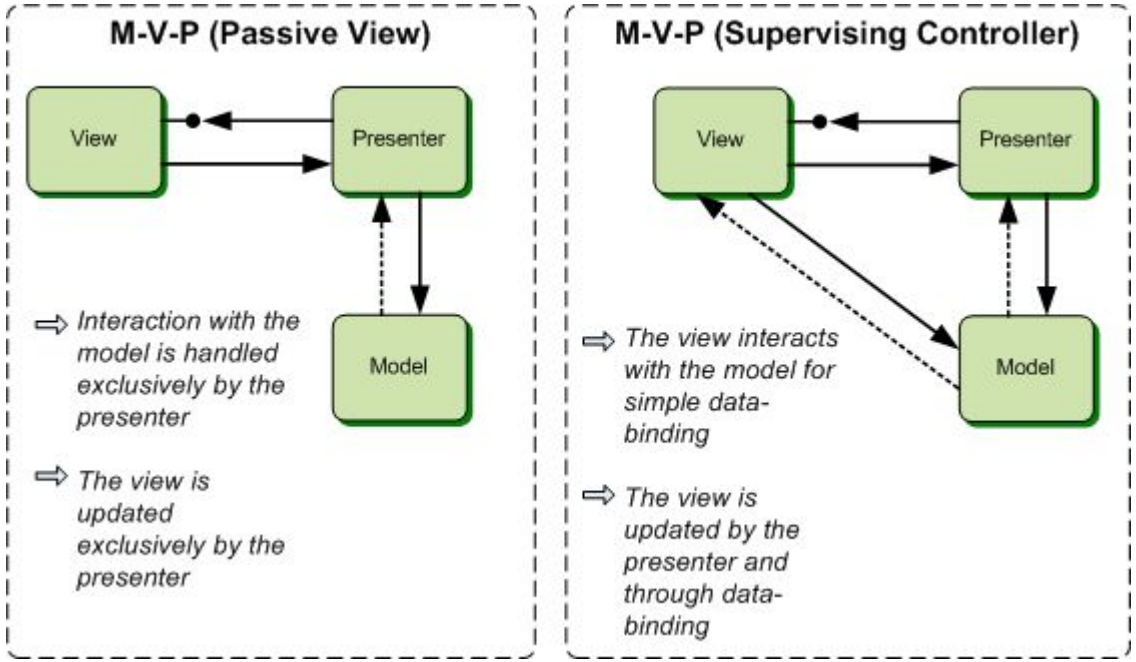
In the call to the *Find* method, the argument *(Person p) => { return p.ID == 3; }* is a lambda expression that actually does the work. It has the following syntactic items:

- A list of parameters enclosed in parentheses. As with a regular method, if the method you are defining (as in the preceding example) takes no parameters, you must still provide the parentheses. In the case of the *Find* method, the predicate is provided with each item from the collection in turn, and this item is passed as the parameter to the lambda expression.
- The => operator, which indicates to the C# compiler that this is a lambda expression.
- The body of the method. The example shown here is very simple, containing a single statement that returns a Boolean value indicating whether the item specified in the parameter matches the search criteria. However, a lambda expression can contain multiple statements, and you can format it in whatever way you feel is most readable. Just remember to add a semicolon after each statement, as you would in an ordinary method.

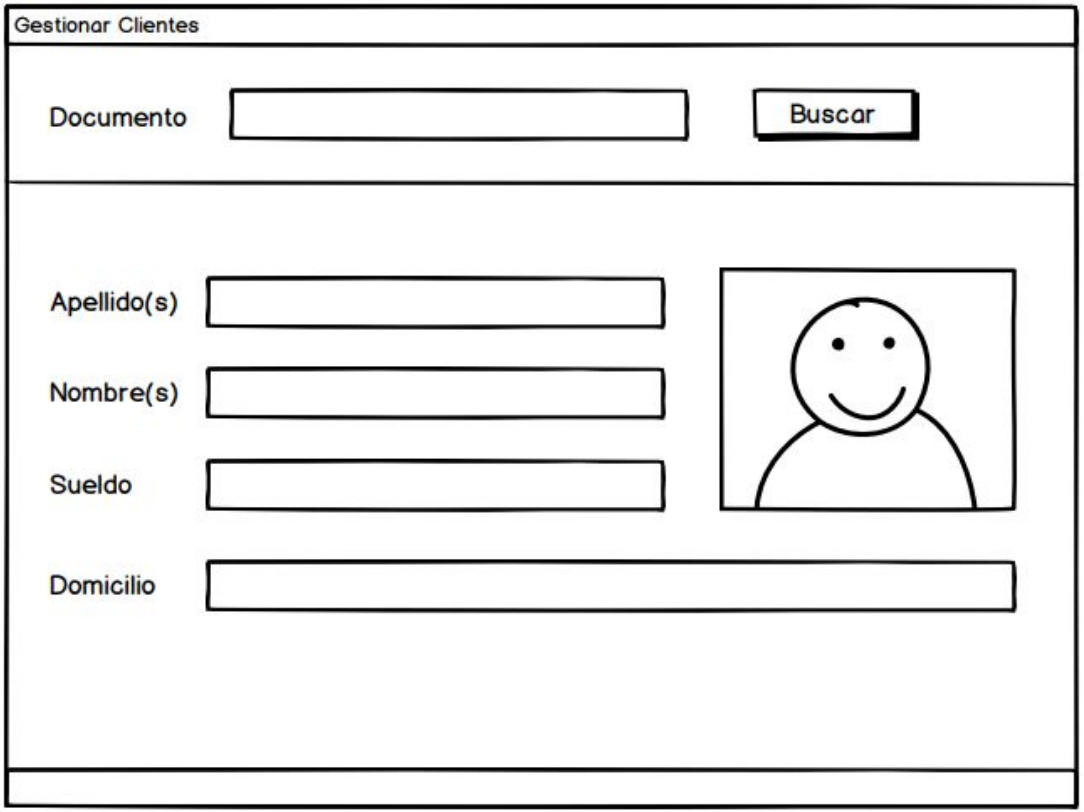
Código

```
x => x * x // A simple expression that returns the square of its parameter
           // The type of parameter x is inferred from the context.

x => { return x * x ; } // Semantically the same as the preceding
                        // expression, but using a C# statement block as
                        // a body rather than a simple expression

(int x) => x / 2 // A simple expression that returns the value of the
                 // parameter divided by 2
                 // The type of parameter x is stated explicitly.

() => folder.StopFolding(0) // Calling a method
                            // The expression takes no parameters.
                            // The expression might or might not
                            // return a value.

(x, y) => { x++; return x / y; } // Multiple parameters; the compiler
                                 // infers the parameter types.
                                 // The parameter x is passed by value, so
                                 // the effect of the ++ operation is
                                 // local to the expression.

(ref int x, int y) => { x++; return x / y; } // Multiple parameters
                                             // with explicit types
                                             // Parameter x is passed by
                                             // reference, so the effect of
                                             // the ++ operation is permanent.
```

MODEL - VIEW - PRESENTER (Modelo Vista Presentador)



Ejemplo


Gestionar Clientes
Documento
Buscar
Apellido(s)
Nombre(s)
Sueldo
Domicilio