The Inside Track
Representative and Reactionary Programming
June 2001

I'm back from my vacation. It was great, I saw many of the great natural wonders the south and west part of this country has to offer. Another great part of this country is the political system. We send people to Washington DC to represent us. The media and other organizations keep an eye on what is happening in Washington. When something interesting happens, we are alerted and can take action. As has been said by others, it's not a perfect system, but it's hard to imagine being happy living under another.

Gee, if it works for people, why not programs!

In C#, we have a couple of concepts that are formalized in the language that I'd like to concentrate on this month: delegates and events. In the December 2000 column, we touched on delegates and events, this month we'll dive in a bit deeper.

Based on the analogy, delegates allow a programmer to get code in one class executed in another class. Yes, you can execute code by using an object.method() call, but with a delegate you can execute a method by using a delegate variable, variable(). The variable *represents* the called object in the context of the calling object. Likewise, events are often used to allow an object to *react* to a situation in another object.

Representative Delegates

A delegate is a method (or function) that we can send off to another object. The other object can then invoke this method. When our delegate is invoked, our code runs, duh. But lots of different classes can send their methods and have them invoked, each with its own particular code/behavior as long as the parameter types and return types match those of the delegate.

Strictly speaking, a delegate is actually a *type*. When you declare a delegate you are defining a new type, for instance from the December 2000 article:

```
public delegate void
             OverCreditLimitHandler(Customer customer,decimal amount);
```

OverCreditLimitHandler is a new type that takes two parameters of types Customer and decimal and returns nothing. It is now possible to declare an object of this type, for instance:

```
public event OverCreditLimitHandler OnOverCreditLimit;
```

This statement declares the event of type OverCreditLimitHandler called OnOverCreditLimit. We could also have declared:

```
public OverCreditLimitHandler OnOverCreditLimit;
```

This statement declares the delegate variable or field OnOverCreditLimit of delegate type OverCreditLimitHandler. We'll explore the difference between a delegate and an event shortly.

One vote per delegate, please!

Let's look at a complete example using a delegate. The example will pass a method as a parameter to another method. The passed method will be used to perform an arithmetic operation on a pair of numbers and return the result. You might think of this as a calculating engine for an expression evaluator or as the basis for calculator program. Depending on which operation key the user presses, you want to perform a different operation on the values in the calculator.

```
using System;

delegate double Operator(double arg1,double arg2);

class Operation
{
	private double arg1, arg2;

	public Operation(double arg1,double arg2)
	{
		this.arg1 = arg1;
		this.arg2 = arg2;
	}
	public double Operate(Operator op)
	{	return op(arg1,arg2); }
	public override string ToString()
	{	return "(" + arg1 + "," + arg2 + ")"; }
}

class Test
{
	static double Addition(double arg1,double arg2)
	{	return arg1 + arg2; }
	static double Subtraction(double arg1,double arg2)
	{	return arg1 - arg2; }
	static double Multiplication(double arg1,double arg2)
	{	return arg1 * arg2; }
	static double Division(double arg1,double arg2)
	{	return arg1 / arg2; }
	public static void Main()
	{
		Operation o1 = new Operation(5,10);
		Operation o2 = new Operation(12,3);
		Operator add = new Operator(Addition);
		Operator sub = new Operator(Subtraction);
		Operator mul = new Operator(Multiplication);
		Operator div = new Operator(Division);

		Console.WriteLine("Add {0}={1}",o1,o1.Operate(add));
		Console.WriteLine("Sub {0}={1}",o1,o1.Operate(sub));
		Console.WriteLine("Mul {0}={1}",o1,o1.Operate(mul));
		Console.WriteLine("Div {0}={1}",o1,o1.Operate(div));
		Console.WriteLine("Add {0}={1}",o2,o2.Operate(add));
		Console.WriteLine("Sub {0}={1}",o2,o2.Operate(sub));
		Console.WriteLine("Mul {0}={1}",o2,o2.Operate(mul));
		Console.WriteLine("Div {0}={1}",o2,o2.Operate(div));
	}
}
```

The output from this code is:

```
Add (5,10)=15
Sub (5,10)=-5
Mul (5,10)=50
Div (5,10)=0.5
Add (12,3)=15
Sub (12,3)=9
Mul (12,3)=36
Div (12,3)=4
```

Let follow the execution of the program. Starting with the first line in Main() where all C# program start, we see:

```
            Operation o1 = new Operation(5,10);
```

The variable o1 is a new object of class Operation with arg1 set to 5 and arg2 set to 10. This happened in the constructor with 2 parameters for the Operation class:

```
      public Operation(double arg1,double arg2)
      {
            this.arg1 = arg1;
            this.arg2 = arg2;
      }
```

Similarly, o2 is created in the next line but with different values. Then we have:

```
            Operator add = new Operator(Addition);
```

Remember that a delegate is a type. So this line creates a variable of type Operator called add. The constructor for a delegate takes one value, the name of a method. In this case, add is a delegate variable with value Test.Addition().

If Addition() were a method in another class called Other, we would have written:

```
            Operator add = new Operator(Other.Addition);
```

Or if Addition() were a method of object obj, we would have written:

```
            Operator add = new Operator(obj.Addition);
```

The next 3 lines create 3 more delegate variables. The next line is:

```
            Console.WriteLine("Add {0}={1}",o1,o1.Operate(add));
```

Console.WriteLine() will output text in a command-line window. The first argument is the format of the text with substitutions indicated by numbers in curly braces. The arguments that follow will be substituted starting with zero. Object o1 is output where you find “{0}” in the format. WriteLine() converts an object to a string by calling the ToString() method. For the Operator class this is:

```
      public override string ToString()
      {     return "(" + arg1 + "," + arg2 + ")"; }
```

Next, WriteLine() will invoke the method o1.Operate(add). Operation.Operate() takes a delegate parameter called op:

```
        public double Operate(Operator op)
        {       return op(arg1,arg2); }
```

The op parameter is a delegate of type Operator which stands for a method that takes two double parameters and returns a double result, just like Addition(), Subtraction(), Multiplication() and Division() in class Test. Any method that follows the pattern specified in the delegate type can be passed as a parameter.

So op is a parameter that refers to a method in another class. Normally, to call Addition() you would code:

```
        {       return Test.Addition(arg1,arg2); }
```

Or with an object called obj you would write:

```
        {       return obj.Addition(arg1,arg2); }
```

By using the delegate parameter everything necessary to represent the method is included in the delegate. You may have heard this referred to as "binding" because when the delegate was created, the method was "bound" to the delegate variable. The expression "op(arg1,arg2)" calls the bound method that returns the result of the arithmetic operation we requested.

As with o1, WriteLine() calls ToString() on the result of the call to o1.Operate() that converts the double result to a string that is then written out.

The next seven lines demonstrate the flexibility of delegates. We invoked Operate() 8 times to get 8 results. This is called "single casting" since only one invocation of a delegate is allowed to call a single method that returns a result.

Proportional Representation?

It is possible to turn this example inside out and have a "multicasting" delegate. For instance:

```
using System;

delegate void Operator(Operation o);

class Operation
{
	public double Arg1,Arg2;
	public Operator Op;

	public void Operate(double arg1,double arg2)
	{	Arg1 = arg1; Arg2 = arg2; Op(this); }
	public override string ToString()
	{	return "(" + Arg1 + "," + Arg2 + ")"; }
}

class Test
{
	static void Addition(Operation o)
	{	Console.WriteLine("Add {0}={1}",o,o.Arg1 + o.Arg2); }
	static void Subtraction(Operation o)
	{	Console.WriteLine("Sub {0}={1}",o,o.Arg1 - o.Arg2); }
	static void Multiplication(Operation o)
	{	Console.WriteLine("Mul {0}={1}",o,o.Arg1 * o.Arg2); }
	static void Division(Operation o)
	{	Console.WriteLine("Div {0}={1}",o,o.Arg1 / o.Arg2); }
	public static void Main()
	{
		Operation o1 = new Operation();

		o1.Op =  new Operator(Addition);
		o1.Op += new Operator(Subtraction);
		o1.Op += new Operator(Multiplication);
		o1.Op += new Operator(Division);

		o1.Operate(5,10);
		o1.Operate(12,3);
	}
}
```

This code produces exactly the same output! How? Each time Operate() is called, it can in turn call multiple methods in the statement “Op(this);”, exactly 4 in this case. The 2 calls to Operate() each caused 4 calls to methods bound to Op and that turned into 8 results total!

Note the statements like “o1.Op += new Operator(Subtraction);”. In these statements we “combined” delegates. Delegates that are combined will be called in the order they were “added”. Try changing the combining order to see the affect it has on the output.

Since multiple methods will be called this way, there are two rules covering multicast delegates. A method used in a multicast must not return a value and may have no parameters marked “out” (meaning they will return a value and may be uninitialized). If the methods did return values, which value would be the appropriate one to return, the first, the last? Since the answer is not clear, the C# designers decided to disallow multicast delegates that return values. One of

the main goals of C# was simplicity and that drove many design decisions like this one.

We've seen that delegates can be passed to a method and assigned and combined via a delegate field; they can be called one-at-a-time or many-at-a-time. Delegates implement that representative programming I mentioned. What about events?

Reactionary Events

Events are a little more than delegates and a little less. If you look back at that December 2000 column, you'll see that I motivated the discussion of properties, methods and events by discussing components.

One of the aspects of components (like ActiveX controls in VB) is that they use a common set of properties, methods and events. Look at virtually any visual ActiveX control and you will find properties like Top, Left, Height and Width, methods like Hide(), SetBounds() and Refresh() and events like Click(), KeyPress() and MouseOver().

The common set of properties, methods and events makes learning a new control quick and easy and is one of the reasons VB is a popular and productive development tool. I'm sure you are not surprised to learn that C# supports this concept of a common "interface" very well and very directly. An interface is a language construct that's be around for a while, C# does interfaces really well.

An interface defines a set of properties, methods and events but no code. A class can inherit from an interface. A class that inherits an interface is required to provide the implementation (that's the code!) for all of the properties, methods and events defined in the interface. Both a button control and a text box control may inherit an interface with a KeyPress() event, the way they are implemented determines how they behave and that will be quite different.

Here is a hypothetical interface for a visual control:

```
public delegate void OnClick();
public delegate void OnKeyPress(int KeyCode);
public delegate void OnMouseOver(int X,int Y);

interface IVisualControl
{
	int Top { get; set; }
	int Left { get; set; }
	int Height { get; set; }
	int Width { get; set; }
	void Hide();
	void SetBound(int Top,int Left,int Height,int Width);
	void ZOrder(int position);
	void Refresh();
	event OnClick Click;
	event OnKeyPress KeyPress;
	event OnMouseOver MouseOver;
}
```

Conventions for Events?

While such an interface is perfectly legal syntax in C#, the .NET Framework Library follows a convention for events; all events have exactly two parameters. The first parameter is always the "sender" of the event and the second is a derived from class EventArgs that contains event data. The event type is suffixed with "Handler" and the event data is suffixed "EventArgs". To follow these rules, the interface above should be declared as:

```
public class KeyPressEventArgs : EventArgs
{
	public int KeyCode;
}

public class MouseOverEventArgs : EventArgs
{
	public int X,Y;
}

public delegate void ClickHandler(object sender, EventArgs e);
public delegate void KeyPressHandler(object sender, KeyPressEventArgs e);
public delegate void MouseOverHandler(object sender, MouseOverEventArgs e);

interface IVisualControl
{
	int Top { get; set; }
	int Left { get; set; }
	int Height { get; set; }
	int Width { get; set; }
	void Hide();
	void SetBound(int Top,int Left,int Height,int Width);
	void ZOrder(int position);
	void Refresh();
	event ClickHandler Click;
	event KeyPressHandler KeyPress;
	event MouseOverHandler MouseOver;
}
```

Here is one of the differences between a delegate and an event: a delegate cannot be declared in an interface, an event can. That's one more for events.

Another difference is that while several operators including "=" (assignment) and "+=" (combining) are possible with delegates, only "+=" (combine delegates) and "-=" (remove from combined delegates) are possible with events. That's one less for events. Why not allow assignment of events?

It is expected that events will be used by unrelated code. Lets say you implemented your event with a delegate. Programmer A uses delegate combining to add a method to the delegate. Programmer B uses delegate assignment so that his method will be invoked when the event occurs. Since Programmer B used assignment there is no combining and Programmer A's method will never be called.

Actually, event syntax with the "+=" and "-+=" operators are much like property get and set methods for assignment, you can define your own. In the case of events, the accessor methods are called add and remove. Why do you want your own accessor methods? An article on MSDN at:

http://msdn.microsoft.com/voices/csharp05172001.asp

The article suggests that one reason is that a class with many events could implement a delegate hash table. Each event is really a wrapper for a delegate as we've seen. If a control has many events but only a few were used, then the extra storage for the delegates might represent a significant overhead. Using the accessor methods, a control (represented by a class) could implement the events as a sparse array by using a hash table! For all the details, read the two part article mentioned at the URL above (part two) and part one at:

http://msdn.microsoft.com/voices/csharp04192001.asp

One last point. Since the class that implements the event is loosely coupled to the code that uses it as in the Programmer A and B example above. The result is that the event may have no methods bound to the multicast delegate. Invoking an unbound delegate produces a runtime error. A common pattern is to have a protected method that invokes the delegate but only if it is not null, i.e. at least one method is bound. This method is named after the event but prefixed with "On". To see the whole thing in action, I've rewritten the Credit Limit example from the December 2000 column following these patterns.

```
using System;

public class OverCreditLimitEventArgs : EventArgs
{
       private decimal amount;
       public OverCreditLimitEventArgs(decimal amount)
       {
```

```
                this.amount = amount;
        }
        public decimal Amount
        {
                get { return amount; }
        }
}

public delegate void OverCreditLimitHandler(
                object sender,OverCreditLimitEventArgs e);

public class Customer
{
        public event OverCreditLimitHandler OverCreditLimit;

        public Customer(int id,string name,decimal creditLimit)
        {
                this.id = id;
                this.name = name;
                this.creditLimit = creditLimit;
                this.currentBalance = 0;
        }
        public bool Purchase(decimal amount)
        {
                if(amount + currentBalance <= creditLimit)
                {
                        currentBalance += amount;
                        return true;
                }
                else
                {
                        OnOverCreditLimit(amount);
                        return false;
                }
        }
        protected void OnOverCreditLimit(decimal amount)
        {
                if(OverCreditLimit != null)
                        OverCreditLimit(this,new OverCreditLimitEventArgs(amount));
        }

        public int id;
        public string name;
        public decimal creditLimit;
        public decimal currentBalance;
}

class Test
{
        static void Over(object o,OverCreditLimitEventArgs e)
        {
                Customer cust = o as Customer;
                Console.WriteLine("Customer {0}:", cust.name);
                Console.WriteLine("\tTransaction Amount = {0}",e.Amount);
                Console.WriteLine("\tCurrent Balance = {0}",cust.currentBalance);
                Console.WriteLine("\tCredit Limit = {0}",cust.currentBalance);
        }

        static public void Main()
        {
                Customer cust = new Customer(100,"Karl Gunderson",100m);

                cust.OverCreditLimit +=
```

```
                            new OverCreditLimitHandler(Over);

            cust.Purchase(50m);
            cust.Purchase(60m);
            cust.Purchase(120m);
            cust.Purchase(50m);
        }
}
```

Following all the rules is a bit more work but the formalism makes events much easier to use.

It was great to have a break, but it's also nice to be back.

C you next month,
Karl Gunderson
Technical Evangelist
solutiondeveloper@greatplains.com