Delegates and Events

Delegates (Harwani, 2015)

Delegate is a class in .NET that encapsulates a method. It is also a reference type data type that holds a reference method. This class can call any method as long as its signature matches.

Declaring and instantiating a delegate

The **`delegate`** keyword is used in declaring a delegate on a `class`. Then, the signature methods specify what kind of delegate to be declared. To declare a delegate, identify what access modifier should be used first. Then, call the `delegate` keyword, the return type (e.g., `void`, `int`, or `string`), the desired name of the delegate, and the parameters inside the open and close parentheses **`()`**. The syntax is

`<access modifier> delegate <return type> <delegate-name> (<parameters>)`.

Example:

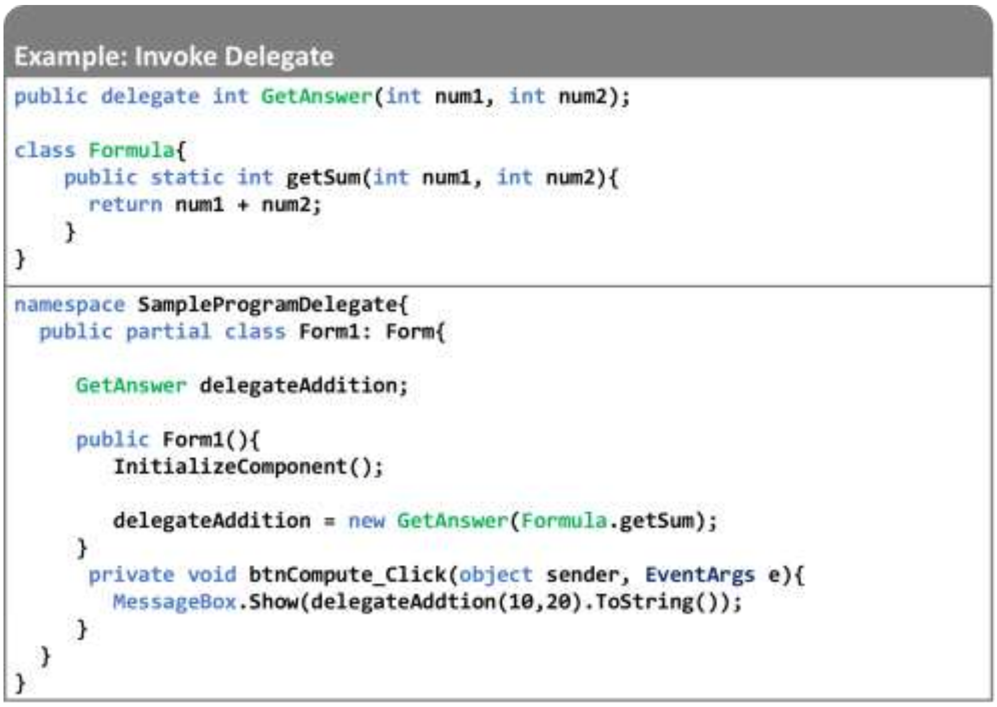
```
public delegate int mathOp(int x, int y);
public delegate void errorMessage();
```

To instantiate the delegate, use the **`new`** keyword that is associated with the method and supposed to refer on the object. For example:

- **`GetAnswer mdAdd = new GetAnswer(Formula.getSum);`**
- **`GetAnswer mdAdd = Formula.Addition;`**

Invoking Delegates

Invoking on referenced methods can also be done in delegates. (See example in *Figure 1*.)


Example: Invoke Delegate

```
public delegate int GetAnswer(int num1, int num2);

class Formula{
    public static int getSum(int num1, int num2){
      return num1 + num2;
    }
}
```

```
namespace SampleProgramDelegate{
  public partial class Form1: Form{

     GetAnswer delegateAddition;

     public Form1(){
        InitializeComponent();

        delegateAddition = new GetAnswer(Formula.getSum);
     }
      private void btnCompute_Click(object sender, EventArgs e){
        MessageBox.Show(delegateAddtion(10,20).ToString());
     }
  }
}
```

Figure 1. Invoking the reference method

In *Figure 1*, there are two (2) classes named **`Formula`** and **`Form1`**. The class named **`Formula`** contains the declaration of delegate and static method. If invoking a referenced method with a return type, declare a delegate that contains parameters. This delegate should have the same count of parameters in the referenced method.

Since the static method named **`getSum()`** returns an integer (**`int`**), a delegate that will reference a method in integer(**`int`**) should be declared.

After declaring the delegate and the method that will be referenced, it needs to call the method in the declared delegate. In the class **`Form1`**, call first the delegate followed by the desired variable name (in *Figure 1*, the variable of the delegate is set to **`delegateAddition`**). Initialize the delegate and call the **`getSum`** method in the **`Formula`**class (take note that the value can be initialized and set inside the constructor).

After initializing the delegate, the same way of calling a return type method, call the delegate with an open and close parentheses. Then, set the arguments and display the following output:

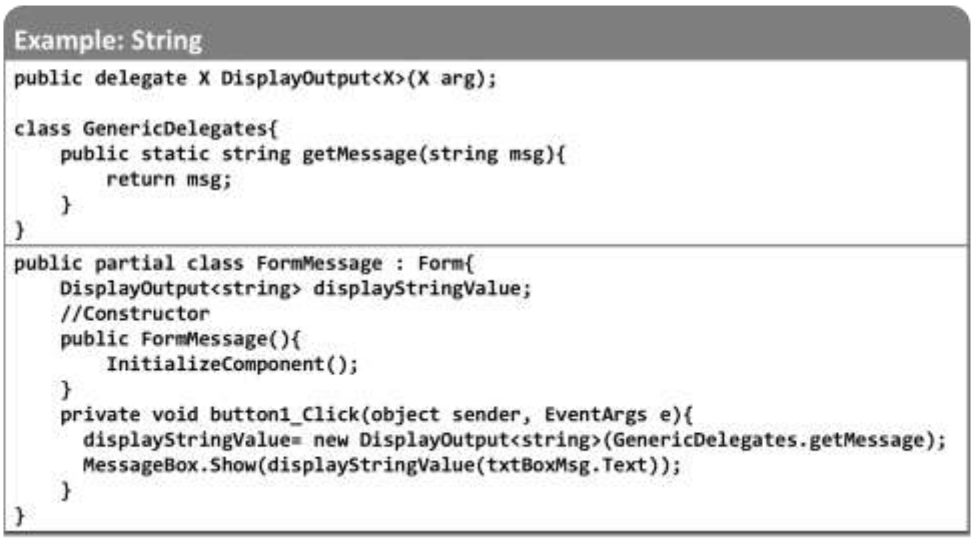
```
MessageBox.Show(delegateAddition(10,20).ToString());
```

Generic Delegate Types

Generic delegates are not bound to any specific type. These can reference a method that returns and takes parameters of different types. Below is an example of declaring a generic delegate:

```
public delegate X DisplayOutput<X>(X arg);
```

In the code above, the declared generic delegate named **DisplayOutput** can point to any method whose return type is **X** and contains a single parameter of **X** type. For example, if X is substituted by **double**, the declared generic delegate can point to any method that returns a **double** data type that has a single parameter. Same with the other data types like string, int, or decimal, the generic delegate can point to these data types as long as the method has a single parameter. *Figures 2* and *3* explain how the generic delegates work.

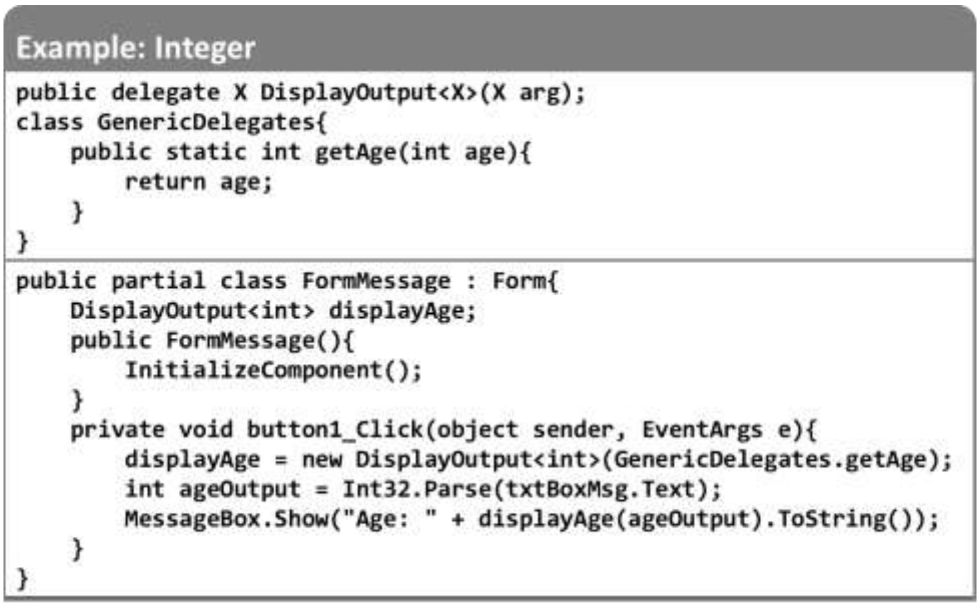


Example: String

```
public delegate X DisplayOutput<X>(X arg);

class GenericDelegates{
    public static string getMessage(string msg){
        return msg;
    }
}
```

```
public partial class FormMessage : Form{
    DisplayOutput<string> displayStringValue;
    //Constructor
    public FormMessage(){
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e){
      displayStringValue= new DisplayOutput<string>(GenericDelegates.getMessage);
      MessageBox.Show(displayStringValue(txtBoxMsg.Text));
    }
}
```

Figure 2. String generic delegate



Example: Integer

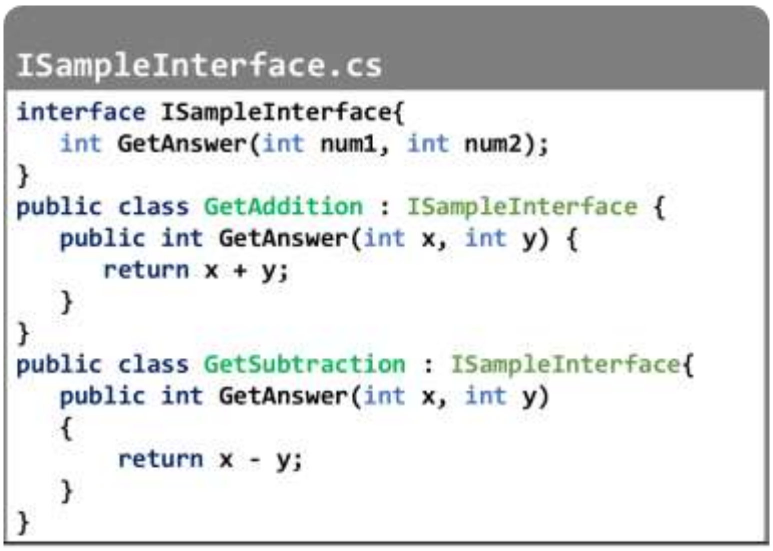
```
public delegate X DisplayOutput<X>(X arg);
class GenericDelegates{
    public static int getAge(int age){
        return age;
    }
}
```

```
public partial class FormMessage : Form{
    DisplayOutput<int> displayAge;
    public FormMessage(){
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e){
        displayAge = new DisplayOutput<int>(GenericDelegates.getAge);
        int ageOutput = Int32.Parse(txtBoxMsg.Text);
        MessageBox.Show("Age: " + displayAge(ageOutput).ToString());
    }
}
```

Figure 3. Integer generic delegate

Difference Between Delegates and Interfaces

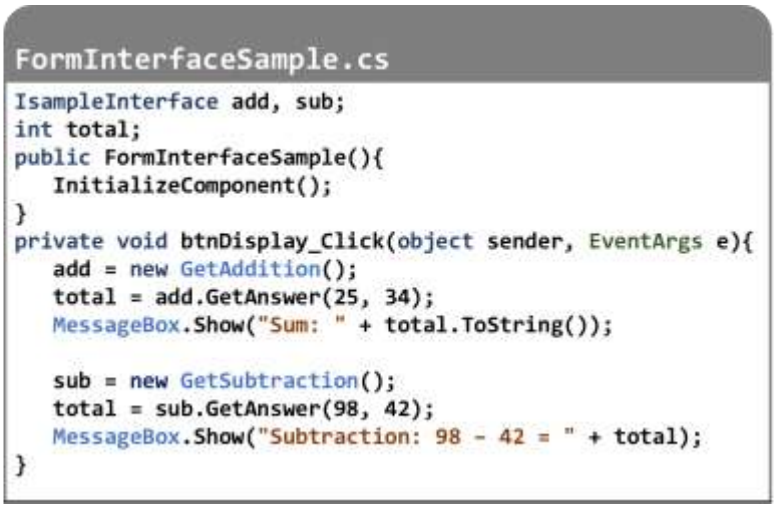
Even if both delegates and interfaces include the declaration, they still have differences. In *Figures 4* and *5*, these two (2) are an example of declaring and calling an interface to display the output. See *Table 1* to check their differences.



ISampleInterface.cs

```
interface ISampleInterface{
   int GetAnswer(int num1, int num2);
}
public class GetAddition : ISampleInterface {
   public int GetAnswer(int x, int y) {
      return x + y;
   }
}
public class GetSubtraction : ISampleInterface{
   public int GetAnswer(int x, int y)
   {
       return x - y;
   }
}
```

Figure 4. Program code using interface



FormInterfaceSample.cs

```
IsampleInterface add, sub;
int total;
public FormInterfaceSample(){
   InitializeComponent();
}
private void btnDisplay_Click(object sender, EventArgs e){
   add = new GetAddition();
   total = add.GetAnswer(25, 34);
   MessageBox.Show("Sum: " + total.ToString());

   sub = new GetSubtraction();
   total = sub.GetAnswer(98, 42);
   MessageBox.Show("Subtraction: 98 - 42 = " + total);
}
```

Figure 5. Calling an interface

DELEGATES	INTERFACES
They have no implementation; they are just safe callbacks and only have a declaration.	A class that implements an interface can implement all the methods associated with it.
These can reference any method of a class that provides arguments and return types.	Using interfaces allows extending some of the objects' functionality.
These are used if a class needs more than one (1) implementation of the method.	These can implement a method only once.
These are faster to execute but slower to get.	These are slower in executing but faster to get.

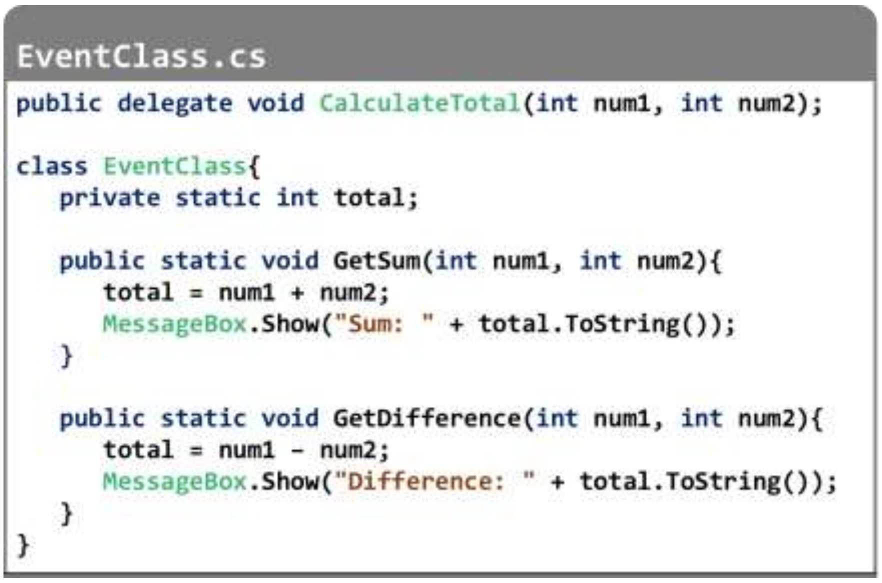
Table 1. Differences between delegates and interfaces (Harwani, 2015)

Event Handling

In **C#.NET**, events are usually task initiators. Clicking a mouse or pressing the Enter key are the common events that one can create while working with any program. An **event** is a member of a class that is fired whenever a specific action takes place. When an event occurs, the method that has been created and registered to this event is automatically invoked. Registration of methods is done through delegates that specify the signature of a method registered for it. The methods that are invoked when an event occurs are known as **event handlers**.

Event Declaration

Before declaring an event, a delegate to which the event is supposed to be associated should be declared first. In *Figure 6*, a class named **`EventClass`** contains reference methods **`GetSum`** and **`GetDifference`**, in which both contain two (2) parameters and a delegate that also contains two (2) parameters. Note that a delegate contains two (2) parameters, which can reference a method that has the same count of parameters.



```
EventClass.cs
public delegate void CalculateTotal(int num1, int num2);

class EventClass{
   private static int total;

   public static void GetSum(int num1, int num2){
      total = num1 + num2;
      MessageBox.Show("Sum: " + total.ToString());
   }

   public static void GetDifference(int num1, int num2){
      total = num1 - num2;
      MessageBox.Show("Difference: " + total.ToString());
   }
}
```

Figure 6. Declaring delegate and reference methods

Once the delegate is declared, the second step is to declare an event. In *Figure 7*, the class named **EventDelegate**—which extends to the **`Form`** class—contains declarations of variables, an event, and a series of codes that will add a delegate. When declaring an event, the following syntax should be

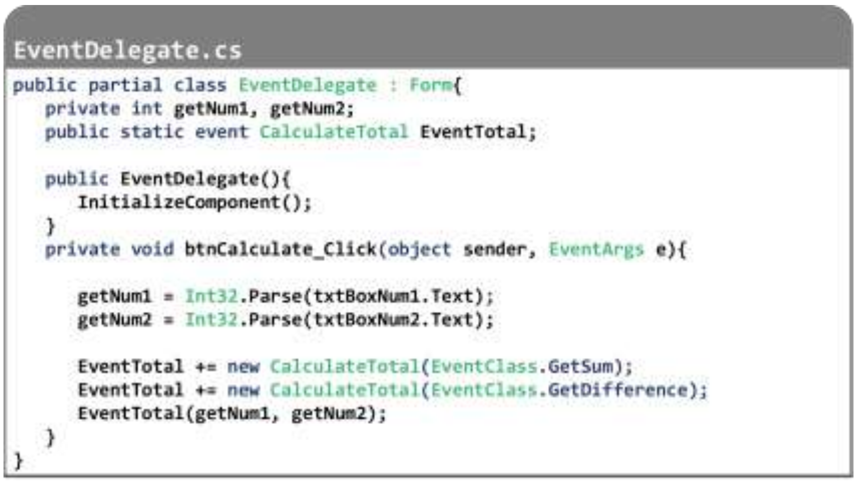
`public event delegate_name event_name`.

The first part of the code is an access modifier, followed by the keyword **`event,`** which is a class, then the name of the delegate, and the preferred event name.

After declaring the event, next is adding a delegate to the event. To add a delegate to an event, use the **+=** operator. The syntax for adding an event is

`event_name += delegate_instance;`.

Figure 7 shows how delegates are added to the event and how this event is executed.



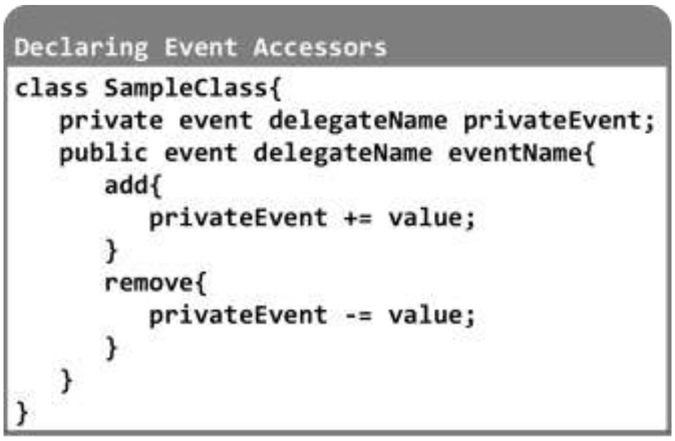
```
EventDelegate.cs
public partial class EventDelegate : Form{
   private int getNum1, getNum2;
   public static event CalculateTotal EventTotal;

   public EventDelegate(){
      InitializeComponent();
   }
   private void btnCalculate_Click(object sender, EventArgs e){

      getNum1 = Int32.Parse(txtBoxNum1.Text);
      getNum2 = Int32.Parse(txtBoxNum2.Text);

      EventTotal += new CalculateTotal(EventClass.GetSum);
      EventTotal += new CalculateTotal(EventClass.GetDifference);
      EventTotal(getNum1, getNum2);
   }
}
```

Figure 7. Declaring an event

Event Accessors

Using the **+=** operator allows adding an event to the class. When this happens, the compiler automatically generates two (2) event accessors. These accessors are the methods **`add`** and **`remove`**, which are similar to the **`get`** and **`set`** accessors used with properties.

To register a new subscription to an event, use the **add** accessor. To unregister the subscription, use the **`remove`** accessor. See *Figure 8* for declaring an event accessor.

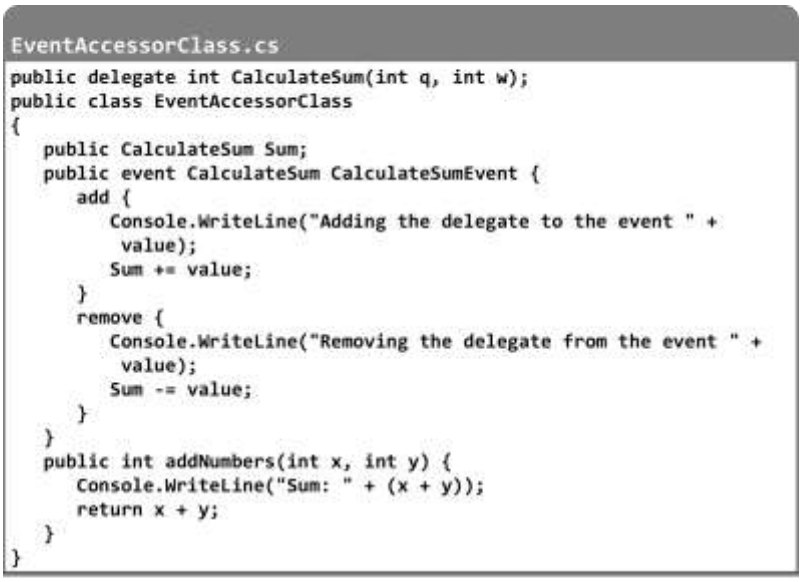


Declaring Event Accessors

```
class SampleClass{
   private event delegateName privateEvent;
   public event delegateName eventName{
      add{
         privateEvent += value;
      }
      remove{
         privateEvent -= value;
      }
   }
}
```

Figure 8. Declaring event accessors

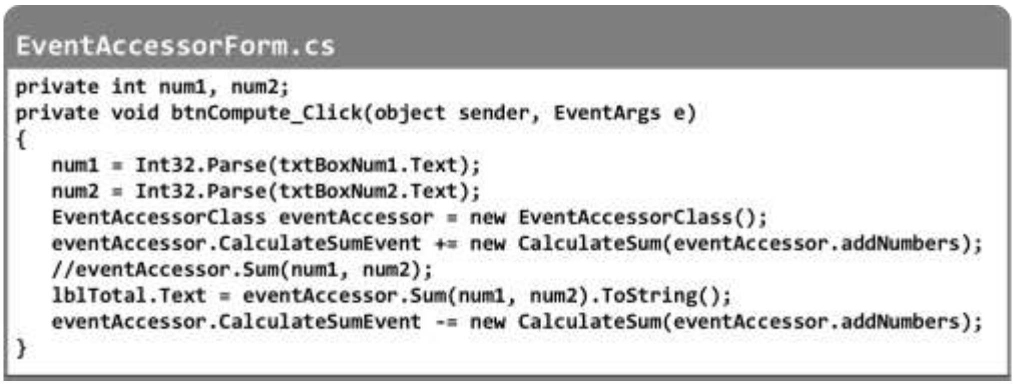
The methods **add** and **remove** allow adding the desired code, most likely checking some conditions before registering or unregistering the subscriber. *Figures 9* and *10* demonstrate the declaration and invoking event accessors.



EventAccessorClass.cs

```
public delegate int CalculateSum(int q, int w);
public class EventAccessorClass
{
   public CalculateSum Sum;
   public event CalculateSum CalculateSumEvent {
      add {
         Console.WriteLine("Adding the delegate to the event " +
          value);
         Sum += value;
      }
      remove {
         Console.WriteLine("Removing the delegate from the event " +
          value);
         Sum -= value;
      }
   }
   public int addNumbers(int x, int y) {
      Console.WriteLine("Sum: " + (x + y));
      return x + y;
   }
}
```

Figure 9. Sample program code for event accessor



EventAccessorForm.cs

```
private int num1, num2;
private void btnCompute_Click(object sender, EventArgs e)
{
   num1 = Int32.Parse(txtBoxNum1.Text);
   num2 = Int32.Parse(txtBoxNum2.Text);
   EventAccessorClass eventAccessor = new EventAccessorClass();
   eventAccessor.CalculateSumEvent += new CalculateSum(eventAccessor.addNumbers);
   //eventAccessor.Sum(num1, num2);
   lblTotal.Text = eventAccessor.Sum(num1, num2).ToString();
   eventAccessor.CalculateSumEvent -= new CalculateSum(eventAccessor.addNumbers);
}
```

Figure 10. Display the output

The declared delegate named **CalculateSum** in the program can reference a method that contains parameters and returns its value. The class named **EventAccessorClass** will compute the sum of two (2) numbers that have been entered by the user. This class contains two (2) variables for data numbers, a constructor that contains parameters, and two (2) methods named **CalculateSumEvent** and **addNumbers**.

REFERENCES:

Deitel, P. and Deitel, H. (2015). *Visual C# 2012 how to program* (5th ed.). USA: Pearson Education, Inc.

Doyle, B. (2015). *C# programming: From problem analysis to program design* (5th ed.). Boston, MA: Cengage Learning.

Gaddis, T. (2016). *Starting out with Visual C#* (4th ed.). USA: Pearson Education, Inc.

Harwani, B. (2015). *Learning object-oriented programming in C# 5.0.* USA: Cengage Learning PTR.

Miles, R. (2016). *Begin to code with C#.* Redmond Washington: Microsoft Press.