Ordering Information:

C# How to Program
The Complete C# Training Course

- View the complete **Table of Contents**
- Read the **Preface**
- Download the **Code Examples**

To view all the Deitel products and services available, visit the Deitel Kiosk on InformIT at **www.informIT.com/deitel**.

To follow the Deitel publishing program, sign-up now for the *DEITEL™ BUZZ ONLINE* e-mail newsletter at **www.deitel.com/newsletter/subscribeinformIT.html**.

To learn more about our **C# and .NET programming courses** or any other Deitel instructor-led corporate training courses that can be delivered at your location, visit **www.deitel.com/training**, contact our Director of Corporate Training Programs at (978) 461-5880 or e-mail: **christi.kelsey@deitel.com.**

Note from the Authors: This article is an excerpt from Chapter 21, Section 21.4 of *C# How to Program* and introduces some basic concepts in the creation and consumption of Web services. In the article we provide an example of creating a Web service in Visual Studio .NET, and an example of creating a client to consume that Web service. Readers should be familiar with C# syntax, and have a basic familiarity of ASP .NET. The code examples included in this article show readers programming examples using the DEITEL™ signature LIVE-CODE™ Approach, which presents all concepts in the context of complete working programs followed by the screen shots of the actual inputs and outputs.

21.4 Publishing and Consuming Web Services

This article presents an example of creating (also known as *publishing*) and using (also known as *consuming*) a Web service. An application that consumes a Web service actually consists of two parts: A *proxy* class that represents the Web service and a client application that accesses the Web service via an instance of the proxy class. The proxy class handles the transferral of the arguments for the Web-service method from the client application to the Web service, as well as the transferral of the result from the Web-service method back to the client application. Visual Studio can generate proxy classes—we demonstrate how to do this momentarily.

Figure 21.6 presents the code-behind file for the **HugeInteger** Web service. The name of the Web service is based on the name of the class that defines it (in this case, **HugeInteger**). This Web service is designed to perform calculations with integers that contain a maximum of 100 digits. As we mentioned earlier, **long** variables cannot handle integers of this size (i.e., an overflow would occur). The Web service provides a client with methods that take two "huge integers" and determine which one is larger or smaller, whether the two numbers are equal, their sum or their difference. The reader can think of these methods as services that one application provides for the programmers of other applications (hence the term, "Web services"). Any programmer can access this Web service, use its methods and thus avoid the writing of over 200 lines of code.

```
1  // Fig. 21.6: HugeInteger.asmx.cs
2  // HugeInteger Web Service.
3
4  using System;
5  using System.Text;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Data;
9  using System.Diagnostics;
10 using System.Web;
11 using System.Web.Services; // contains Web service related classes
12
13 namespace HugeIntegerWebService
14 {
15    /// <summary>
16    /// performs operations on large integers
17    /// </summary>
18    [ WebService(
19       Namespace = "http://www.deitel.com/csphtp1/ch21/",
20       Description = "A Web service which provides methods that" +
21       " can manipulate large integer values." ) ]
22    public class HugeInteger : System.Web.Services.WebService
23    {
24       // default constructor
25       public HugeInteger()
26       {
27          // CODEGEN: This call is required by the ASP .NET Web
28          // Services Designer
```

Fig. 21.6 **HugeInteger** Web service. (Part 1 of 6.)

```
29            InitializeComponent();
30
31            number = new int[ MAXIMUM ];
32         }
33
34         #region Component Designer generated code
35         /// <summary>
36         /// Required method for Designer support - do not modify
37         /// the contents of this method with the code editor.
38         /// </summary>
39         private void InitializeComponent()
40         {
41         }
42         #endregion
43
44         /// <summary>
45         /// Clean up any resources being used.
46         /// </summary>
47         protected override void Dispose( bool disposing )
48         {
49         }
50
51         // WEB SERVICE EXAMPLE
52         // The HelloWorld() example service returns
53         // the string Hello World
54         // To build, uncomment the following lines
55         // then save and build the project
56         // To test this web service, press F5
57
58         //    [WebMethod]
59         //    public string HelloWorld()
60         //    {
61         //       return "Hello World";
62         //    }
63
64         private const int MAXIMUM = 100;
65
66         public int[] number;
67
68         // indexer that accepts an integer parameter
69         public int this[ int index ]
70         {
71            get
72            {
73               return number[ index ];
74            }
75
76            set
77            {
78               number[ index ] = value;
79            }
80
81         } // end indexer
```

Fig. 21.6 **HugeInteger** Web service. (Part 2 of 6.)

```
82
83          // returns string representation of HugeInteger
84          public override string ToString()
85          {
86             StringBuilder returnString = new StringBuilder();
87
88             foreach ( int digit in number )
89                returnString.Insert( 0, digit );
90
91             return returnString.ToString();
92          }
93
94          // creates HugeInteger based on argument
95          public static HugeInteger FromString( string integer )
96          {
97             HugeInteger parsedInteger = new HugeInteger();
98
99             for ( int i = 0; i < integer.Length; i++ )
100               parsedInteger[ i ] = Int32.Parse(
101                  integer[ integer.Length - i - 1 ].ToString() );
102
103            return parsedInteger;
104         }
105
106         // WebMethod that performs integer addition
107         // represented by string arguments
108         [ WebMethod ( Description = "Adds two huge integers." ) ]
109         public string Add( string first, string second )
110         {
111            int carry = 0;
112
113            HugeInteger operand1 = HugeInteger.FromString( first );
114            HugeInteger operand2 =
115               HugeInteger.FromString( second );
116
117            // store result of addition
118            HugeInteger result = new HugeInteger();
119
120            // perform addition algorithm for each digit
121            for ( int i = 0; i < MAXIMUM; i++ )
122            {
123               // add two digits in same column
124               // result is their sum, plus carry from
125               // previous operation modulus 10
126               result[ i ] =
127                  ( operand1[ i ] + operand2[ i ] ) % 10 + carry;
128
129               // store remainder of dividing
130               // sums of two digits by 10
131               carry = ( operand1[ i ] + operand2[ i ] ) / 10;
132            }
133
134            return result.ToString();
```

Fig. 21.6 **HugeInteger** Web service. (Part 3 of 6.)

```
135
136       } // end method Add
137
138       // WebMethod that performs the subtraction of integers
139       // represented by string arguments
140       [ WebMethod (
141          Description = "Subtracts two huge integers." ) ]
142       public string Subtract( string first, string second )
143       {
144          HugeInteger operand1 = HugeInteger.FromString( first );
145          HugeInteger operand2 =
146             HugeInteger.FromString( second );
147          HugeInteger result = new HugeInteger();
148
149          // subtract top digit from bottom digit
150          for ( int i = 0; i < MAXIMUM; i++ )
151          {
152             // if top digit is smaller than bottom
153             // digit we need to borrow
154             if ( operand1[ i ] < operand2[ i ] )
155                Borrow( operand1, i );
156
157             // subtract bottom from top
158             result[ i ] = operand1[ i ] - operand2[ i ];
159          }
160
161          return result.ToString();
162
163       } // end method Subtract
164
165       // borrows 1 from next digit
166       private void Borrow( HugeInteger integer, int place )
167       {
168          // if no place to borrow from, signal problem
169          if ( place >= MAXIMUM - 1 )
170             throw new ArgumentException();
171
172          // otherwise if next digit is zero,
173          // borrow from digit to left
174          else if ( integer[ place + 1 ] == 0 )
175             Borrow( integer, place + 1 );
176
177          // add ten to current place because we borrowed
178          // and subtract one from previous digit -
179          // this is digit borrowed from
180          integer[ place ] += 10;
181          integer[ place + 1 ] -= 1;
182
183       } // end method Borrow
184
185       // WebMethod that returns true if first integer is
186       // bigger than second
```

Fig. 21.6 **HugeInteger** Web service. (Part 4 of 6.)

```
187         [ WebMethod ( Description = "Determines whether first " +
188            "integer is larger than the second integer." ) ]
189         public bool Bigger( string first, string second )
190         {
191            char[] zeroes = { '0' };
192
193            try
194            {
195               // if elimination of all zeroes from result
196               // of subtraction is an empty string,
197               // numbers are equal, so return false,
198               // otherwise return true
199               if ( Subtract( first, second ).Trim( zeroes ) == "" )
200                  return false;
201               else
202                  return true;
203            }
204
205            // if ArgumentException occurs, first number
206            // was smaller, so return false
207            catch ( ArgumentException )
208            {
209               return false;
210            }
211
212         } // end method Bigger
213
214         // WebMethod returns true if first integer is
215         // smaller than second
216         [ WebMethod ( Description = "Determines whether the " +
217            "first integer is smaller than the second integer." ) ]
218         public bool Smaller( string first, string second )
219         {
220            // if second is bigger than first, then first is
221            // smaller than second
222            return Bigger( second, first );
223         }
224
225         // WebMethod that returns true if two integers are equal
226         [ WebMethod ( Description = "Determines whether the " +
227            "first integer is equal to the second integer." ) ]
228         public bool EqualTo( string first, string second )
229         {
230            // if either first is bigger than second, or first is
231            // smaller than second, they are not equal
232            if ( Bigger( first, second ) ||
233               Smaller( first, second ) )
234               return false;
235            else
236               return true;
237         }
238
239      } // end class HugeInteger
```

Fig. 21.6 **HugeInteger** Web service. (Part 5 of 6.)

```
240
241 } // end namespace HugeIntegerWebService
```

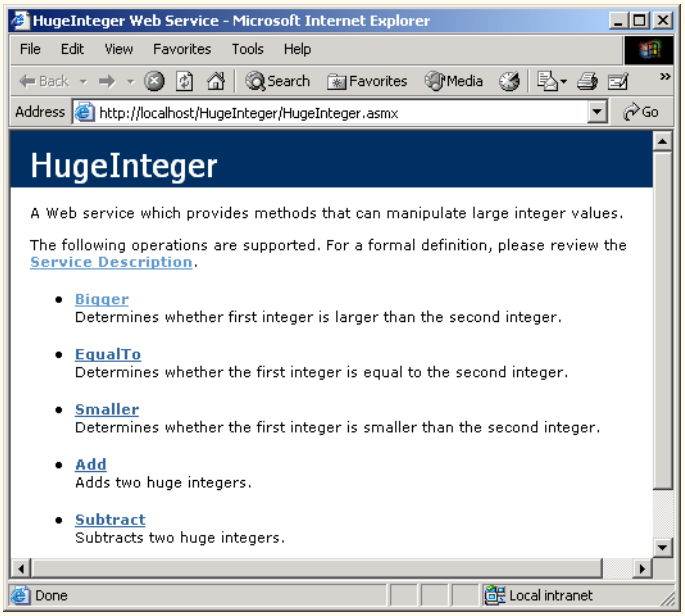


Fig. 21.6 **HugeInteger** Web service. (Part 6 of 6.)

Line 13 places class **HugeInteger** in namespace **HugeIntegerWebService**. Line 19 assigns the Web service namespace to **www.deitel.com/csphtp1/ch21/** to uniquely identify this Web service. The namespace is specified using the ***Namespace*** property of the ***WebService*** attribute. In lines 20–21, we use property ***Description*** to provide information about our Web service. This information will appear in the *ASMX page*, a file that is created as part of the Web service and used to display valuable information about the Web service. The ASMX file is created when the Web service project is created (the ASMX file for the **HugeInteger** Web service is shown in the output of Fig. 21.6). Line 22 specifies that our class derives from ***System.Web.Services.WebService***. By default, Visual Studio defines our Web service so that it inherits from the **WebService** class. Although a Web service class is not required to subclass **WebService**, class **WebService** provides members that are useful in determining information about the client and the Web service itself. Several methods in class **HugeInteger** are tagged with the ***WebMethod*** attribute, which *exposes* the method such that it can be called remotely. When this attribute is absent, the method is not accessible through the Web service. Notice that the **WebMethod** attribute, like the **WebService** attribute, contains a **Description** property, which provides information about the method to the ASMX page.

Good Programming Practice 21.1

Specify a namespace for each Web service so that it can be uniquely identified.

Good Programming Practice 21.2

Specify descriptions for all Web services and Web-service methods so that clients can obtain additional information about the Web service and its contents.

Common Programming Error 21.1

Web-service methods cannot be declared ***static****, or a runtime error will occur when attempting to view the ASMX page. For a client to access a Web-service method, an instance of that Web service must exist.*

Lines 69–81 define an indexer for our class. This enables us to access any digit in **HugeInteger** as if we were accessing it through array **number**. Lines 108–136 and 142–163 define **WebMethod**s **Add** and **Subtract**, which perform addition and subtraction, respectively. Method **Borrow** (lines 166–183) handles the case in which the digit in the left operand is smaller than the corresponding digit in the right operand. For instance, when we subtract 19 from 32, we usually go digit by digit, starting from the right. The number 2 is smaller than 9, so we add 10 to 2 (resulting in 12), which subtracts 9, resulting in 3 for the rightmost digit in the solution. We then subtract 1 from the next digit over (3), making it 2. The corresponding digit in the right operand is now the "1" in 19. The subtraction of 1 from 2 is 1, making the corresponding digit in the result 1. The final result, when both resulting digits are combined, is 13. Method **Borrow** adds 10 to the appropriate digits and subtracts 1 from the digit to the left. Because this is a utility method that is not intended to be called remotely, it is not qualified with attribute **WebMethod**.

The screen capture in Fig. 21.6 lists the available methods of our Web service. A client application can invoke only the five methods listed in the screen shot (i.e., the methods qualified with the **WebMethod** attribute).

Now, let us demonstrate how to create this Web service. To begin, we must create a project of type **ASP.NET Web Service**. Like Web Forms, Web services are by default placed in the Web server's **wwwroot** directory on the server (**localhost**). By default, Visual Studio places the solution file (**.sln**) in the **Visual Studio Projects** folder, in a directory for the solution. (The **Visual Studio Projects** folder is usually located in the **My Documents** folder.)

Notice that, when the project is created, the code-behind file is displayed in design view by default (Fig. 21.7). If this file is not open, it can be opened by clicking **Service1.asmx**. The file that will be opened, however, is **Service1.asmx.cs** (the code-behind file for our Web service). This is because, when creating Web services in Visual Studio, programmers work almost exclusively in the code-behind file. In fact, if a programmer were to open the ASMX file, it would contain only the lines:

```
<%@ WebService Language="c#" Codebehind="Service1.asmx.cs"
    Class="WebService1.Service1" %>
```

indicating the name of the code-behind file, the programming language in which the code-behind file is written and the class that defines our Web service. This is the extent of the information that this file must contain. [*Note*: By default, the code-behind file is not listed in the **Solution Explorer**. The code-behind file is displayed when the ASMX file is double clicked in the **Solution Explorer**. This file can be listed in the **Solution Explorer** by clicking the icon to show all files.]

It might seem strange that there is a design view for Web services, given that Web services do not have graphical user interfaces. A design view is provided because more sophisticated Web services contain methods that manipulate more than just strings or numbers. For example, a Web-service method could manipulate a database. Instead of typing all the code necessary to create a database connection, developers can simply drop the proper ADO .NET components into the design view and manipulate them as we would in a Windows or Web application.

Now that we have defined our Web service, we demonstrate how to use it. First, a client application must be created. In this first example, we create a Windows application as our client. Once this application has been created, the client must add a proxy class for accessing the Web service. A proxy class (or proxy) is a class created from the Web service's WSDL file that enables the client to call Web-service methods over the Internet. The proxy class handles all the "plumbing" required for Web-service method calls. Whenever a call is made in the client application to a Web-service method, the application actually calls a corresponding method in the proxy class. This method takes the name of the method and its arguments, then formats them so that they can be sent as a request in a SOAP message. The Web service receives this request and executes the method call, sending back the result as another SOAP message. When the client application receives the SOAP message containing the response, the proxy class decodes it and formats the results so that they are understandable to the client. This information then is returned to the client. It is important to note that the proxy class essentially is hidden from the programmer. We cannot, in fact, view it in the **Solution Explorer** unless we choose to show all the files. The purpose of the proxy class is to make it seem to clients as though they are calling the Web-service methods directly. It is rarely necessary for the client to view or manipulate the proxy class.

The next example demonstrates how to create a Web service client and its corresponding proxy class. We must begin by creating a project and adding a *Web reference* to that project. When we add a Web reference to a client application, the proxy class is created. The client then creates an instance of the proxy class, which is used to call methods included in the Web service.

To create a proxy in Visual Studio, right click the **References** folder in **Solution Explorer** and select ***Add Web Reference*** (Fig. 21.8). In the **Add Web Reference** dialog that appears (Fig. 21.9), enter the Web address of the Web service and press *Enter*.

Once a Web service is chosen the description of that Web service appears, and the developer can click **Add Reference** (Fig. 21.9). This adds to the **Solution Explorer** (Fig. 21.10) a **Web References** folder with a node named for the domain where the Web service is located. In this case, the name is **`localhost`**, because we are using the local Web server. This means that, when we reference class **`HugeInteger`**, we will be doing so through class **`HugeInteger`** in namespace **`localhost`**, instead of class **`HugeInteger`** in namespace **`HugeIntegerWebService`** [*Note*: The Web service class and the proxy class have the same name. Visual Studio generates a proxy for the Web service and adds it as a reference (Fig. 21.10).]

Good Programming Practice 21.3

When creating a program that will use Web services, add the Web reference first. This will enable Visual Studio to recognize an instance of the Web service class, allowing Intellisense to help the developer use the Web service.

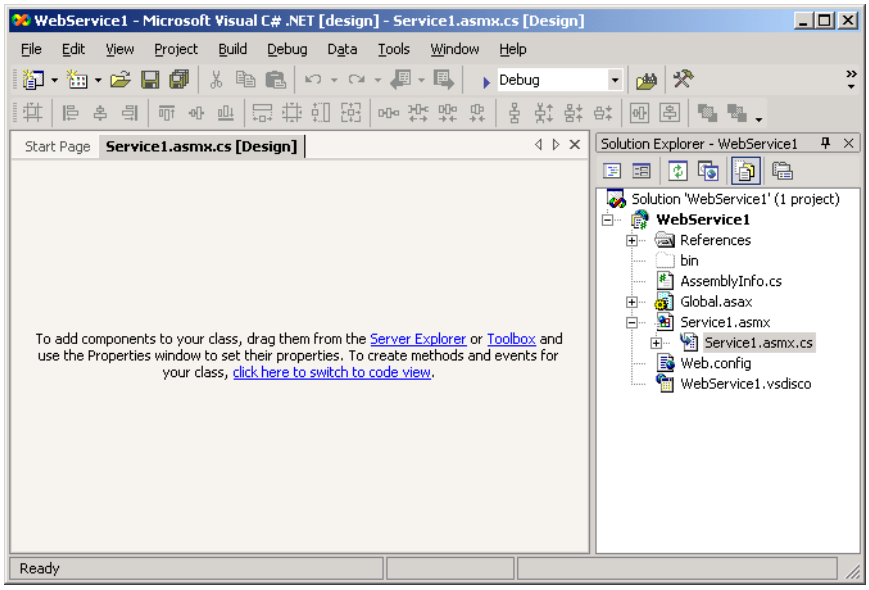


Fig. 21.7 Design view of a Web service.

The steps that we described previously work well if the programmer knows the appropriate Web services reference. However, what if we are trying to locate a new Web service? There are two technologies that facilitate this process: *Universal Description, Discovery and Integration* (*UDDI*) and *Discovery files* (*DISCO*). UDDI is a project for developing a set of specifications that define how Web services should be published so that programmers searching for Web services can find them. Microsoft began this ongoing project to facilitate the locating of Web services that conform to certain specifications, allowing programmers to find different Web services using search engines. UDDI organizes and describes Web services and then places this information in a central location. Although UDDI is beyond the scope of what we are teaching, the reader can learn more about this project and view a demonstration by visiting **`www.uddi.org`** and **`uddi.microsoft.com`**. These sites contain search tools that make finding Web services fast and easy.

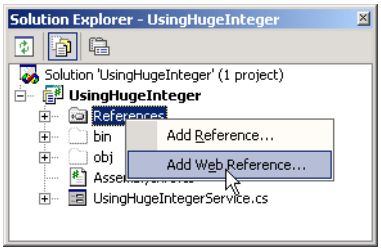


Fig. 21.8 Adding a Web service reference to a project.

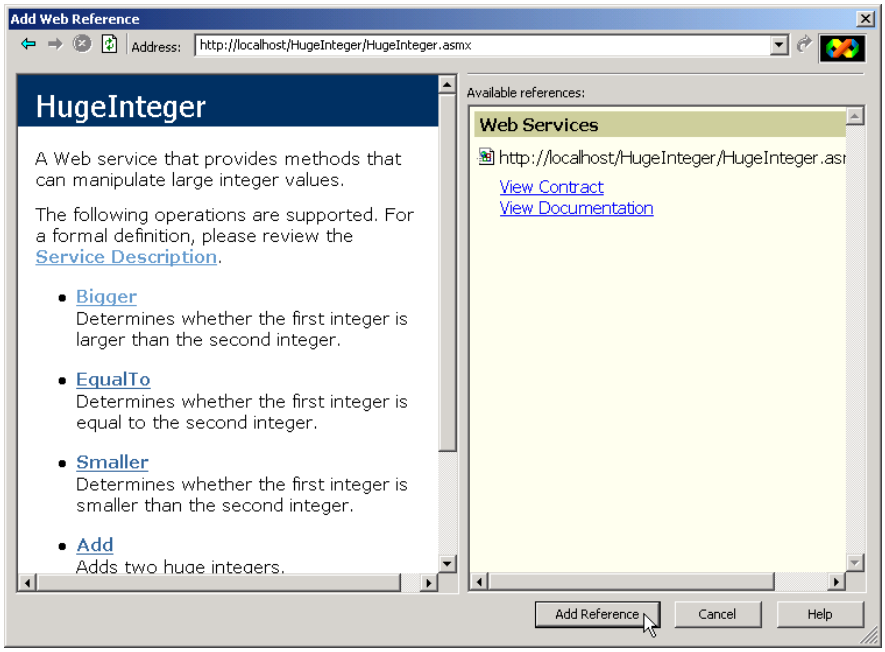


Fig. 21.9 Web reference selection and description.

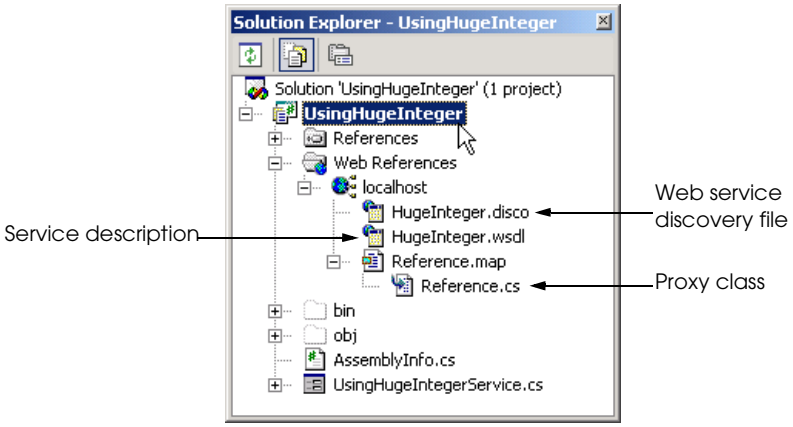


Fig. 21.10 **Solution Explorer** after adding a Web reference to a project.

A DISCO file catalogs Web services in a particular directory. There are two types of discovery files: *Dynamic discovery* files (with a **`.vsdisco`** extension) and *static discovery* files (with a **`.disco`** extension). These files indicate both the location of the ASMX file and the service description (a WSDL file) for each Web service in the current directory, as well as any Web services in the current directory's subdirectories. When a programmer creates a Web service, Visual Studio generates a dynamic discovery file for that

Web service. When a Web reference is added, the client uses the dynamic discovery file to select the desired Web service. Once the Web reference is created, a static discovery file is placed in the client's project. The static discovery file hard codes the location for the ASMX and WSDL files. (By "hard code", we mean that the location is entered directly into the file.) Dynamic discovery files, on the other hand, are created such that a list of Web services is created dynamically on the server when a client is searching for a Web service. The use of dynamic discovery enables certain extra options, such as hiding of certain Web services in subdirectories. Discovery files are a Microsoft-specific technology, whereas UDDI is not. However, the two can work together to enable a client to find a Web service. Using both technologies, the client can use a search engine to find a location with various Web services on a topic, and then use discovery files to view all the Web services in that location.

Once the Web reference is added, the client can access the Web service through our proxy. Because our proxy class is named **`HugeInteger`** and is located in namespace **`localhost`**, we must use **`localhost.HugeInteger`** to reference this class. The Windows Form in Fig. 21.11 uses the **`HugeInteger`** Web service to perform computations with positive integers up to **`100`** digits long. [*Note*: If using the example on this book's CD, the reader might need to regenerate the proxy.]

```
1  // Fig. 21.11: UsingHugeIntegerService.cs
2  // Using the HugeInteger Web Service.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Web.Services.Protocols;
10
11 // allows user to perform operations on large integers
12 public class UsingHugeIntService : System.Windows.Forms.Form
13 {
14    private System.Windows.Forms.Label promptLabel;
15    private System.Windows.Forms.Label resultLabel;
16
17    private System.Windows.Forms.TextBox firstTextBox;
18    private System.Windows.Forms.TextBox secondTextBox;
19
20    private System.Windows.Forms.Button addButton;
21    private System.Windows.Forms.Button subtractButton;
22    private System.Windows.Forms.Button biggerButton;
23    private System.Windows.Forms.Button smallerButton;
24    private System.Windows.Forms.Button equalButton;
25
26    private System.ComponentModel.Container components = null;
27
28    // declare a reference Web service
29    private localhost.HugeInteger remoteInteger;
30
31    private char[] zeroes = { '0' };
```

Fig. 21.11 Using the **`HugeInteger`** Web service. (Part 1 of 5.)

```
32
33     // default constructor
34     public UsingHugeIntService()
35     {
36        InitializeComponent();
37
38        // instantiate remoteInteger
39        remoteInteger = new localhost.HugeInteger();
40     }
41
42     // Visual Studio .NET generated code
43
44     [STAThread]
45     static void Main()
46     {
47        Application.Run( new UsingHugeIntService() );
48
49     } // end Main
50
51     // checks whether two numbers user input are equal
52     protected void equalButton_Click(
53        object sender, System.EventArgs e )
54     {
55        // make sure HugeIntegers do not exceed 100 digits
56        if ( CheckSize( firstTextBox, secondTextBox ) )
57           return;
58
59        // call Web-service method to determine
60        // whether integers are equal
61        if ( remoteInteger.EqualTo(
62           firstTextBox.Text, secondTextBox.Text ) )
63
64           resultLabel.Text =
65              firstTextBox.Text.TrimStart( zeroes ) +
66              " is equal to " +
67              secondTextBox.Text.TrimStart( zeroes );
68        else
69           resultLabel.Text =
70              firstTextBox.Text.TrimStart( zeroes ) +
71              " is NOT equal to " +
72              secondTextBox.Text.TrimStart( zeroes );
73
74     } // end method equalButton_Click
75
76     // checks whether first integer input
77     // by user is smaller than second
78     protected void smallerButton_Click(
79        object sender, System.EventArgs e )
80     {
81        // make sure HugeIntegers do not exceed 100 digits
82        if ( CheckSize( firstTextBox, secondTextBox ) )
83           return;
84
```

Fig. 21.11 Using the **HugeInteger** Web service. (Part 2 of 5.)

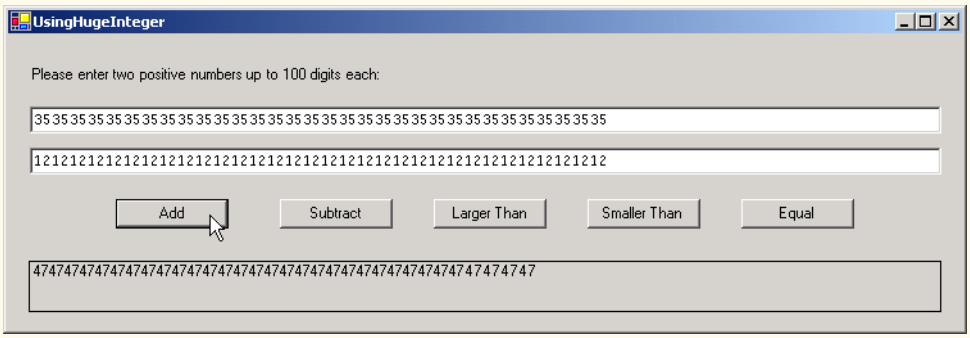
```
85          // call Web-service method to determine whether first
86          // integer is smaller than second
87          if ( remoteInteger.Smaller(
88             firstTextBox.Text, secondTextBox.Text ) )
89
90             resultLabel.Text =
91                firstTextBox.Text.TrimStart( zeroes ) +
92                " is smaller than " +
93                secondTextBox.Text.TrimStart( zeroes );
94          else
95             resultLabel.Text =
96                firstTextBox.Text.TrimStart( zeroes ) +
97                " is NOT smaller than " +
98                secondTextBox.Text.TrimStart( zeroes );
99
100      } // end method smallerButton_Click
101
102      // checks whether first integer input
103      // by user is bigger than second
104      protected void biggerButton_Click(
105         object sender, System.EventArgs e )
106      {
107         // make sure HugeIntegers do not exceed 100 digits
108         if ( CheckSize( firstTextBox, secondTextBox ) )
109            return;
110
111         // call Web-service method to determine whether first
112         // integer is larger than the second
113         if ( remoteInteger.Bigger( firstTextBox.Text,
114            secondTextBox.Text ) )
115
116            resultLabel.Text =
117               firstTextBox.Text.TrimStart( zeroes ) +
118               " is larger than " +
119               secondTextBox.Text.TrimStart( zeroes );
120         else
121            resultLabel.Text =
122               firstTextBox.Text.TrimStart( zeroes ) +
123               " is NOT larger than " +
124               secondTextBox.Text.TrimStart( zeroes );
125
126      } // end method biggerButton_Click
127
128      // subtract second integer from first
129      protected void subtractButton_Click(
130         object sender, System.EventArgs e )
131      {
132         // make sure HugeIntegers do not exceed 100 digits
133         if ( CheckSize( firstTextBox, secondTextBox ) )
134            return;
135
```

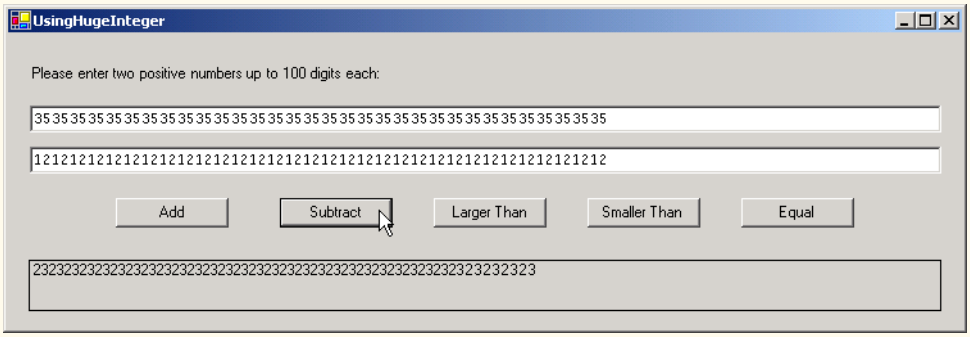
Fig. 21.11 Using the **HugeInteger** Web service. (Part 3 of 5.)

```
136         // perform subtraction
137         try
138         {
139            string result = remoteInteger.Subtract(
140               firstTextBox.Text,
141               secondTextBox.Text ).TrimStart( zeroes );
142
143            resultLabel.Text = ( ( result == "" ) ? "0" : result );
144         }
145
146         // if WebMethod throws an exception, then first
147         // argument was smaller than second
148         catch ( SoapException )
149         {
150            MessageBox.Show(
151               "First argument was smaller than the second" );
152         }
153
154      } // end method subtractButton_Click
155
156      // adds two integers input by user
157      protected void addButton_Click(
158         object sender, System.EventArgs e )
159      {
160         // make sure HugeInteger does not exceed 100 digits
161         // and is not situation where both integers are 100
162         // digits long--result in overflow
163         if ( firstTextBox.Text.Length > 100 ||
164            secondTextBox.Text.Length > 100  ||
165            ( firstTextBox.Text.Length == 100 &&
166            secondTextBox.Text.Length == 100 ) )
167         {
168            MessageBox.Show( "HugeIntegers must not be more "
169               + "than 100 digits\nBoth integers cannot be of"
170               + " length 100: this causes an overflow",
171               "Error", MessageBoxButtons.OK,
172               MessageBoxIcon.Information );
173
174            return;
175         }
176
177         // perform addition
178         resultLabel.Text = remoteInteger.Add( firstTextBox.Text,
179            secondTextBox.Text ).TrimStart( zeroes ).ToString();
180
181      } // end method addButton_Click
182
183      // determines whether size of integers is too big
184      private bool CheckSize( TextBox first, TextBox second )
185      {
186         if ( first.Text.Length > 100 || second.Text.Length > 100 )
187         {
```

Fig. 21.11 Using the **HugeInteger** Web service. (Part 4 of 5.)

```
188            MessageBox.Show( "HugeIntegers must be less than 100"
189               + " digits", "Error", MessageBoxButtons.OK,
190               MessageBoxIcon.Information );
191
192            return true;
193         }
194
195         return false;
196
197      } // end method CheckSize
198
199   } // end class UsingHugeIntegerService
```





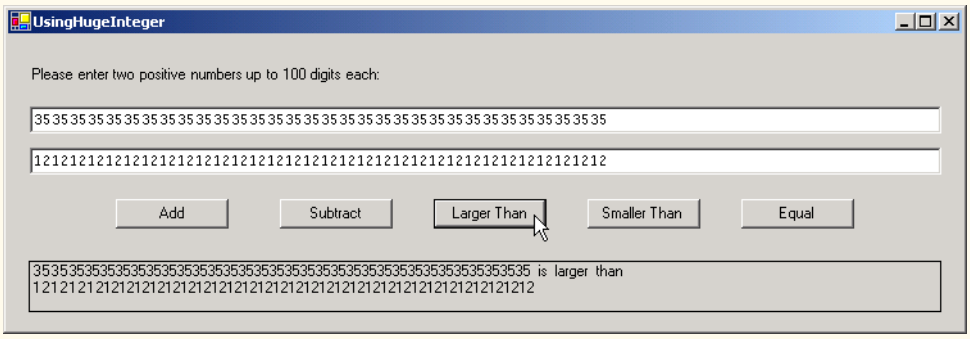


Fig. 21.11 Using the **HugeInteger** Web service. (Part 5 of 5.)

The user inputs two integers, each up to 100 digits long. The clicking of any button invokes a remote method to perform the appropriate calculation and return the result. The return value of each operation is displayed, and all leading zeroes are eliminated using **`string`** method **`TrimStart`**. Note that **`UsingHugeInteger`** does not have the capability to perform operations with 100-digit numbers. Instead, it creates **`string`** representations of these numbers and passes them as arguments to Web-service methods that handle such tasks for us. [*Note*: Fig. 21.11 corresponds to Fig. 21.13 in *C# How to Program*.]