Unit 7

8.0 Error Handling, Logging & Tracing

(Text Book Chapter 07)

8.0 Error Handling, Logging & Tracing

No software can run free from error, and ASP.NET applications are no exception. Sooner or later your code will be interrupted by a programming mistake, invalid data, unexpected circumstances, or even hardware failure. Novice programmers spend sleepless nights worrying about errors. Professional developers recognize that bugs are an inherent part of software applications and code defensively, testing assumptions, logging problems, and writing error-handling code to deal with the unexpected.

In this chapter, you'll learn the error-handling and debugging practices that you can use to defend your ASP.NET applications against common errors, track user problems, and solve mysterious issues. You'll learn how to use structured exception handling, how to use logs to keep a record of unrecoverable errors, and how to set up web pages with custom error messages for common HTTP errors. You'll also learn how to use page tracing to see diagnostic information about ASP.NET pages.

8.1 Common Errors

Errors can occur in a variety of situations. Some of the most common causes of errors include attempts to **divide by zero** (usually caused by invalid input or missing information) and attempts to connect to a limited Resource such as a file or a database (which can fail if the file doesn't exist, the database connection times out, or the code has insufficient security credentials).

One infamous type of error is the **null reference exception**, which usually occurs when a program attempts to use an uninitialized object. As a .NET programmer, you'll quickly learn to recognize and resolve this common but Annoying mistake.

The following code example shows the problem in action, with trying to use a String object before it is instantiated.

➢8.0 Error Handling, Logging and Tracing

```
using System;
.
using System.Xml.Linq;

namespace ExceptionCheck
{
  public partial class _Default : System.Web.UI.Page
  {
    String anInternalString;

    protected void Page_Load(object sender, EventArgs e)
    { }

    protected void showErrorOnClick(object sender, EventArgs e)
    {
      String anotherString = anInternalString.ToString();
    }
  }
}
```

Example 8.1-1 Exception Situation

When an error occurs in your code, .NET checks to see whether any error handlers appear in the current scope. If the error occurs inside a method, .NET searches for local error handlers and then checks for any active error handlers in the calling code. If no error handlers are found, the page processing is aborted and an error page is displayed in the browser.

Depending on whether the request is from the local computer or a remote client, the error page may show a detailed description (as shown in Figure 8.1-1) or a generic message. You'll explore this topic a little later in the "Error Pages" section of this chapter.

➢8.0 Error Handling, Logging and Tracing

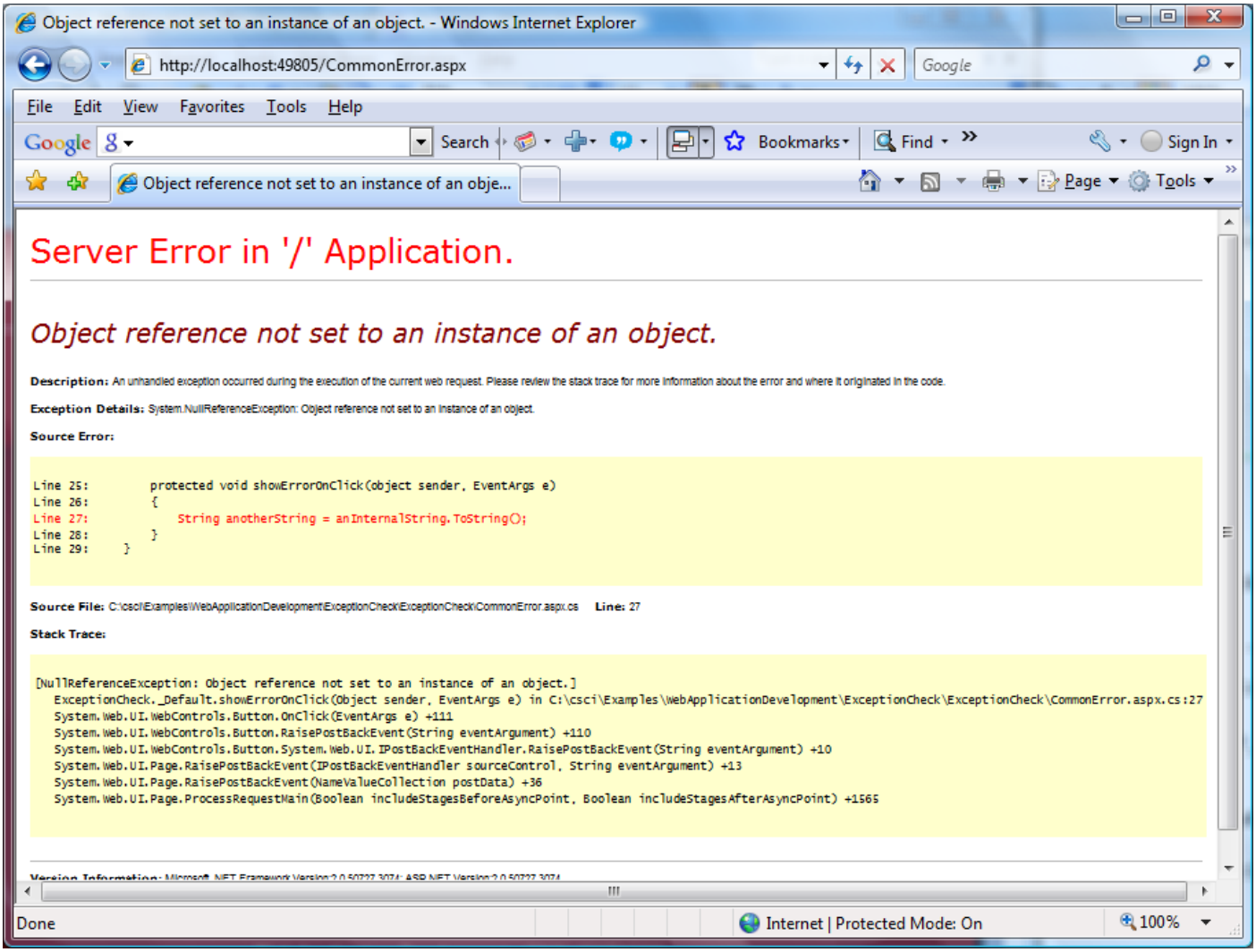


Figure 8.1-1

8.2 Exception Handling in ASP.NET

Exception handling is an in built mechanism in .NET framework to detect and handle run time errors. Exceptions are defined as anomalies that occur during the execution of a program. The .NET framework provides a rich set of standard exceptions that are used during exceptions handling. Exception handling is one of the major feature provide by .NET. There might be various reasons to handle exceptions; **e.g. improper user inputs, improper design logic or system errors etc.** In this scenario if an application does not provide a mechanism to handle these anomalies then the application could crash.

➢8.0 Error Handling, Logging and Tracing

.NET run time environment provide a default structured exception handling mechanism.

All exceptions in C# is class based, so as ASP.NET. This means in the memory there are exception objects created to respond to an unexpected situation. The exception classes are hierarchical. The highest most exception class is called Exception.

Each exception provides a significant amount of diagnostic information wrapped into a neat object, instead of a simple message and error code. These exception objects also support an InnerException property that allows you to wrap a generic error over the more specific error that caused it. You can even create and throw your own exception objects.

Exceptions are caught based on their type: This allows you to streamline error handling code without needing to sift through obscure error codes.

Exception handlers use a modern block structure: This makes it easy to activate and deactivate different error handlers for different sections of code and handle their errors individually.

Exception handlers are multilayered: You can easily layer exception handlers on top of other exception handlers, some of which may check only for a specialized set of errors.

Exceptions are a generic part of the .NET Framework: This means they're completely cross-language compatible. Thus, a .NET component written in C# can throw an exception that you can catch in a web page written in VB.

8.3 The Exception Class

Exception class is the base class for all exceptions and is found in the System namespace. When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error. Once thrown, an exception is handled by the application or by the default exception handler.

The Exception class has the following methods (See Figure 8.3-1)

Method	Description
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetBaseException	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetObjectData	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType	Overloaded.
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
ToString	Creates and returns a string representation of the current exception. (Overrides Object..::.ToString()()().)

Figure 8.3-1

8.4 The .NET Exception Hierarchy

The Figure 8.2 shows the Exception hierarchy in C#. As you see, the highest most class is the exception class. All others inherits the Exception class.

When an exception happens in your code, you cannot expect to see an Exception object is created, Most often, what you see is the exception object relevant to that error created and thrown from that line of code. For example, if a Arithmetic Exception such as divide by zero situation happens, then the **DivideByRezo** exception object is thrown.

ASP.NET Exception hierarchy is shown on next page.

➢8.0 Error Handling, Logging and Tracing

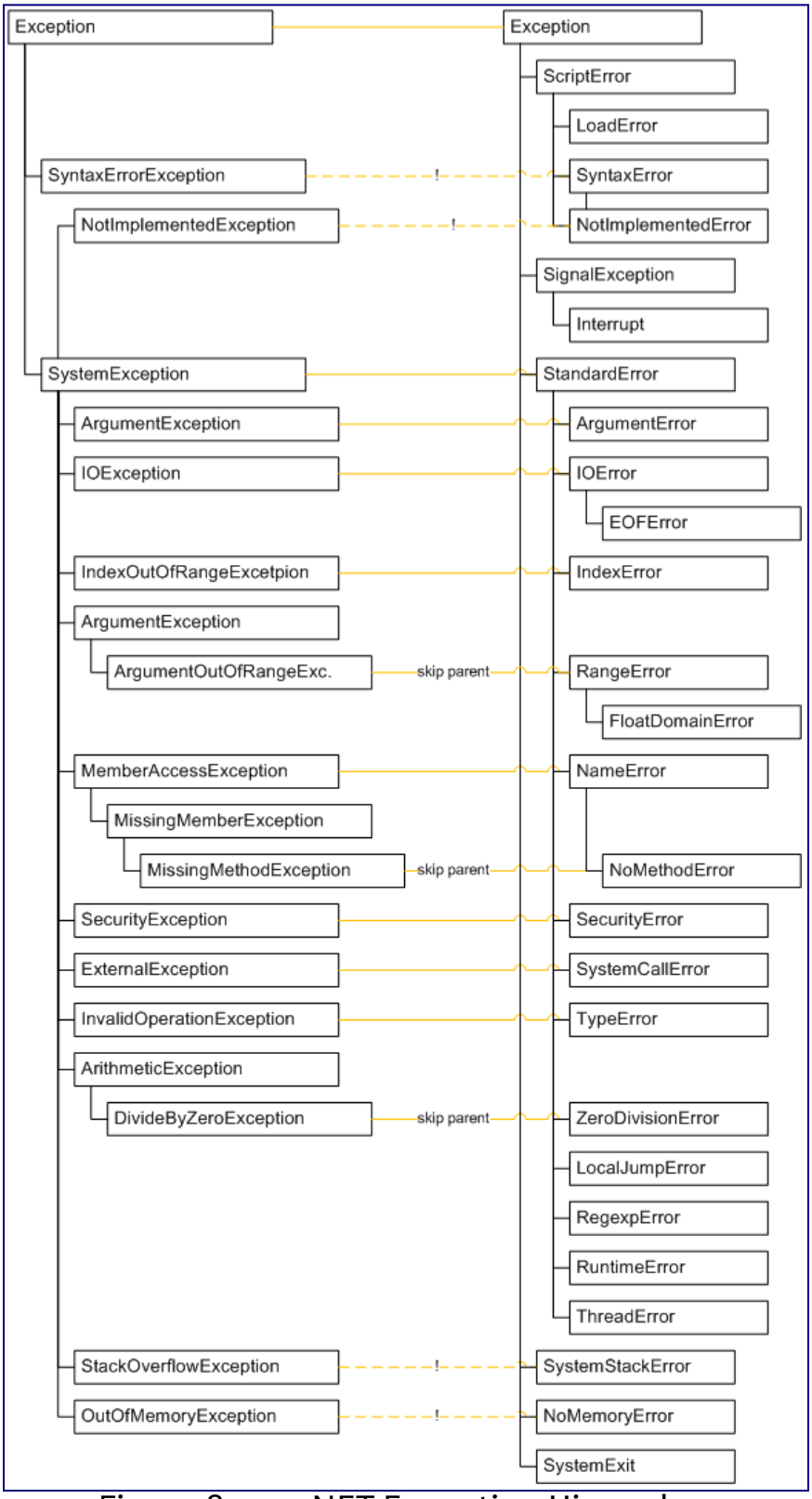


Figure 8.4-1 - .NET Exception Hierarchy

8.5 Visual Studio 's Exceptions Viewer

Visual studio provides a nice tool to view exceptions and their details. Do these steps…

- Click Debug
- Select Exceptions…
- Expand the Common Language Runtime Exceptions

Figure 8.5-1 shows the Exceptions List appearing in Visual Studio .

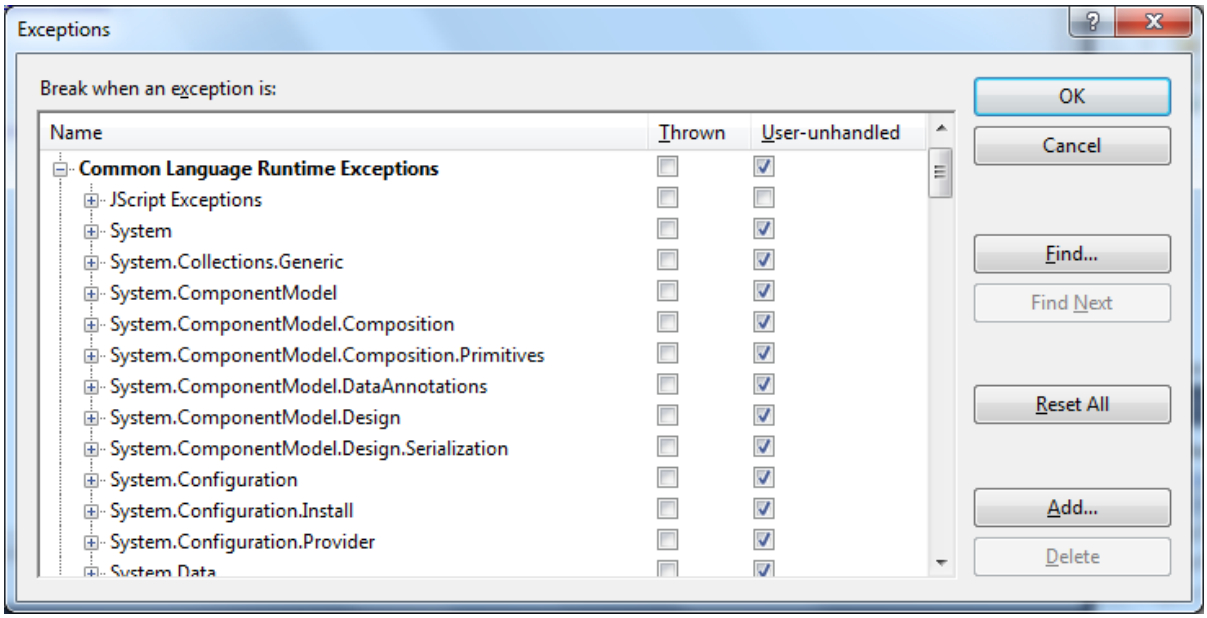


Figure 8.5-1 Visual Studio Exceptions List

8.6 Handling Exceptions

C# provides three keywords **try**, **catch** and **finally** to do exception handling. The try encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for doing any clean up process.

The general form try-catch-finally in C# is shown below:

```
try
{
// Statement which can cause an exception.
}
catch(ClassType exceptionReference)
{
// Statements for handling the exception
}
finally
{
//Any cleanup code
}
```

Example 8.6-1 Common Exception Handling Block

If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.

But in C#, finally blocks is optional. The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks. If there is no exception occurred inside the try block, the control directly transfers to finally block. We can say that the statements inside the finally block is executed always. Note that it is an error to transfer control out of a finally block by using break, continue, return or goto. In C#, exceptions are nothing but objects of the type Exception. The Exception is the ultimate base class for any exceptions in C#. The C# itself provides several of standard exceptions. (See figure 8.4.1)

➢8.0 Error Handling, Logging and Tracing

Developers also can create their own exception classes, provided that this should inherit from either Exception class or one of the standard derived classes of Exception class like **DivideByZeroExcpetion** or **ArgumentException** etc. We will discuss this in few pages down the line.

8.6.1 Uncaught Exceptions

The following program (example 8.6.2) will compile but will show an error during execution. The division by zero is a runtime anomaly and program terminates with an error message. Any uncaught exceptions in the current context propagate to a higher context and looks for an appropriate catch block to handle it. If it can't find any suitable catch blocks, the default mechanism of the .NET runtime will terminate the execution of the entire program.

```
using System;
namespace InnerException
{
 public class ExceptionTest
 {
  public static void Main()
  {
   int x = 0;
   int div = 100 / x;
   Console.WriteLine(div);
  }
 }}
```

Example 8.6-2 Force an Exception

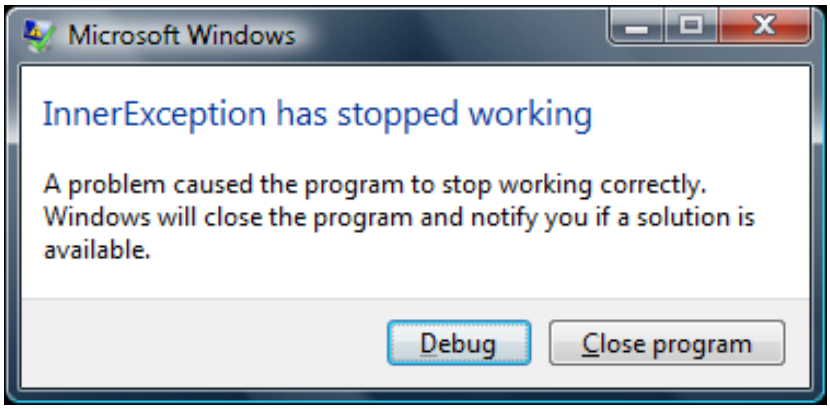


Figure 8.6-3 Uncaught Exception

8.7 Catching Appropriate Exception

You should catch the proper exception for the 'potential' error. For example, the modified form of the above program with exception handling mechanism is as shown in example 8.7-1. Here we are using the object of the standard exception class DivideByZeroException to handle the exception caused by division by zero.

In the above case the program do not terminate unexpectedly. Instead the program control passes from the point where exception occurred inside the try block to the catch blocks. If it finds any suitable catch block, executes the statements inside that catch and continues with the normal execution of the program statements. If a finally block Is present, the code inside the finally block will get also be executed. See example 8.7-2.

```
using System;
class MyClient
{
 public static void Main()
 {
  int x = 0;
  int div = 0;
 try
 {
  div = 100/x;
  Console.WriteLine("This line in not executed");
 }
 catch(DivideByZeroException de)
 {
  Console.WriteLine("Exception occured");
 }
 Console.WriteLine("Result is {0}",div);
 }
}
```

Example 8.7-1

```
using System;
namespace InnerException
{
  class MyClient
  {
    public static void Main()
    {
     int x = 0;
     int div = 0;
     try{
      div = 100 / x;
      Console.WriteLine("Not executed line");
     }
     catch (DivideByZeroException de){
      Console.WriteLine("Exception occured");
     }
     finally{  Console.WriteLine("Finally Block");  }
     Console.WriteLine("Result is {0}", div);
   }
  }
}
```

Example 8.7-2

8.8 Exception Chain(Propagation) & InnerException Property

As said previously, the Base class for all exception is Exception class. When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error. Once thrown, an exception is handled by the application or by the default exception handler. The **Exception** has different kinds of properties. One of these is the **innerException** property.

```
using System;

public class MyAppException:ApplicationException
{
   public MyAppException (String message) : base (message)
   {   }
   public MyAppException (String message, Exception inner) :
                                                  base(message,inner)
   {   }
}
public class ExceptExample {
  public void ThrowInner () {
     throw new MyAppException("ExceptExample inner exception");
   }
public void CatchInner()
{   try {   this.ThrowInner();   }
   catch (Exception e) {
     throw new MyAppException("Error caused by trying ThrowInner.",e);
   }
}}

public class Test {
public static void Main() {
      ExceptExample testInstance = new ExceptExample();
      try   {     testInstance.CatchInner();   }
      catch(Exception e)  {
         Console.WriteLine ("In Main catch block. Caught: {0}", e.Message);

         Console.WriteLine ("Inner Exception is {0}",e.InnerException);
      }
}}
```

Example 8.8-1 InnerException property

Lets take a look at example 8.8-1. Here when you call **the testInstance.CatchInner()** it will call ThrowInner() ethod which will throw an exception "ExceptExample inner exception" . But in the catch block of CatchInner() method it again throws another exception "Error caused by trying ThrowInner" . Now when you print e.Message in class test you will get an exception message as "Error caused by trying ThrowInner". What happened to the first exception thrown by the ThrowInner() method. That will be accessible in InnerException property. In short we can say that InnerException property will help you examine what causes the initial error or you can say that it helps to obtain the set of exceptions that led to the current exception.

8.9 Logging Exceptions

In many cases, it's best not only to detect and catch exceptions but to log them as well. For example, some problems may occur only when your web server is dealing with a particularly large load. Other problems might recur intermittently, with no obvious causes. To diagnose these errors and build a larger picture of site problems, you need to log exceptions so they can be reviewed later.

The .NET Framework provides a wide range of logging tools. When certain errors occur, you can send an e-mail, add a database record, or create and write to a file. We describe many of these techniques in other parts of this book. However, you should keep your logging code as simple as possible. For example, you'll probably run into trouble if you try to log a database exception using another table in the database.

One of the most fail-safe logging tools is the Windows event logging system, which is built into the Windows operating system and available to any application. Using the Windows event logs, your website can write text messages that record errors or unusual events. The Windows event logs store your messages as well as various other details, such as the message type (information, error, and so on) and the time the message was left.

8.9.1 Viewing the Windows Event Logs

To view the Windows event logs, you use the Event Viewer tool that's included with Windows. To launch it, begin by selecting **Start ->Control Panel-> Administrative Tools ->Event Viewer -> Windows Logs,** you'll see the four logs that are described in the Table in figure 8.9-1

Using the Event Viewer(Figure 8.9-2), you can perform a variety of management tasks with the logs. For example, if you right-click one of the logs in the Event Viewer list you'll see options that allow you to clear the events in the log, save the log entries to another file, and import an external log file.

➢8.0 Error Handling, Logging and Tracing

Log Name	Description
Application log	The application log contains events logged by programs. For example, a database program may record a file error in the application log. Events that are written to the application log are determined by the developers of the software program.
Security log	The security log records events such as valid and invalid logon attempts, as well as events related to resource use, such as the creating, opening, or deleting of files. For example, when logon auditing is enabled, an event is recorded in the security log each time a user attempts to log on to the computer. You must be logged on as Administrator or as a member of the Administrators group in order to turn on, use, and specify which events are recorded in the security log.
System log	The system log contains events logged by Windows XP/Vista system components. For example, if a driver fails to load during startup, an event is recorded in the system log. Windows XP/Vista predetermines the events that are logged by system components.
Setup	Used to track issues that occur when installing windows updates or other software. This log only appears in windows vista.

Figure 8.9-1

Each event record in an event log identifies the source (generally, the application or service that created the record), the type of notification (error, information, warning), and the time the log entry was inserted. To see this information, you simply need to select a log entry, and the details will appear in a display area underneath the list of entries.

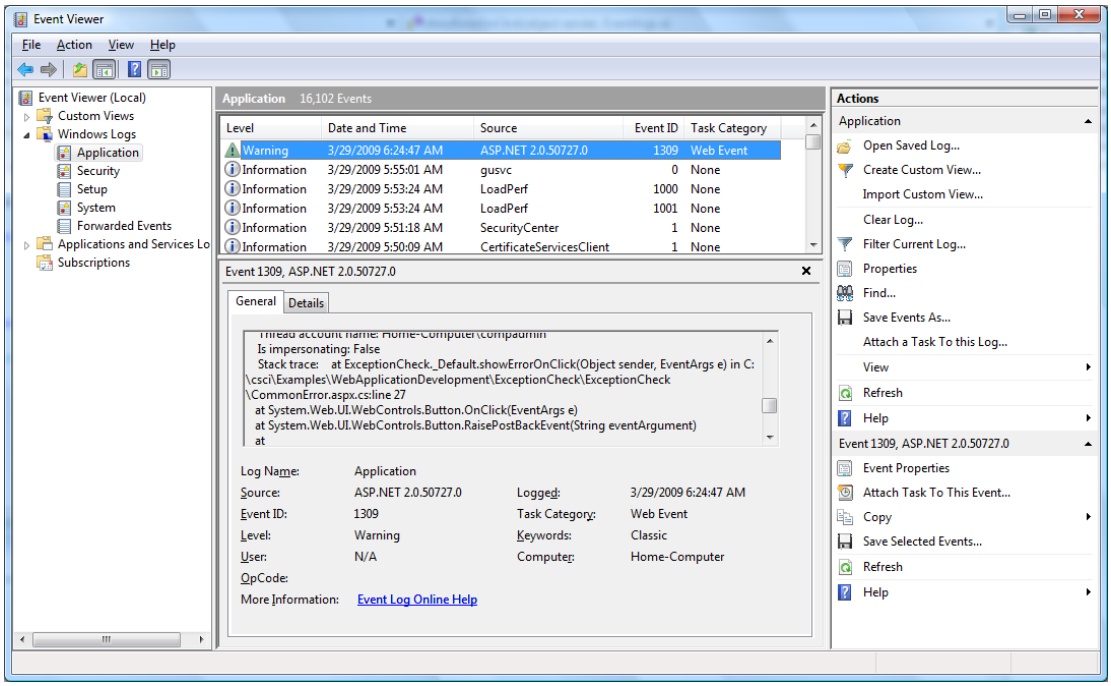


Figure 8.9-2

8.10 Writing to Event Log

The Windows event log is great place to log pertinent information from your application. There is built-in support for accessing the event log in .NET that you can easily implement. It is a simple method to add entries to the Application event log.

When you write an entry to an event log, you specify the message you want to write to the log as a string. A message should contain all information needed to interpret what caused the problem and what to do to correct the problem.

There are two ways you can write an entry to the log; both are equally valid. The most direct way is to register an event source with the log to which you want to write, then instantiate a component and set its Source property to that log, and finally call WriteEntry. If you do this, you do not have to set the Log property for the component instance; the log is automatically determined when you connect to a source that has already been registered. For more information on registering a source, see Adding Your Application as a Source of Event Log Entries.

The other way to approach this process is to instantiate an EventLog component, set its Source, Machine, and Log properties, and call the WriteEntry method. In this case, the WriteEntry method would determine whether the source already existed and register it on the fly if it did not.

The following conditions must be met in order to successfully write a log entry:

1. The source must be registered with the desired log.
   Note You must set the Source property on your EventLog component instance before you can write entries to a log. When your component writes an entry, the system automatically checks to see if the source you specified is registered with the event log to which the component is writing, and calls CreateEventSource if needed.

2. The message you specify cannot be more than 16K in length.

3. Your application must have write access to the log to which it is attempting to write. For more information, see Security Ramifications of Event Logs. You can specify several parameters when you write an entry, including the type of entry you're making, an ID that identifies the event, a category, and any binary data you want to append to the entry. For more information on the properties associated with an entry, see EventLog Members. Take a look at example 8.10-1

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.Diagnostics;

namespace ExceptionCheck
{
 public partial class _Default : System.Web.UI.Page
 {
   String anInternalString;
   protected void Page_Load(object sender, EventArgs e)
   { }

   protected void showErrorOnClick(object sender, EventArgs e)
   {
    try
    {
     String anotherString = anInternalString.ToString();
    }
    catch (Exception exp)
    {
      WriteEvent("Error 1", "This App", EventLogEntryType.Error, "A Log Name");
    }
   }

   public bool WriteEvent(string sEntry, string sAppName, EventLogEntryType eEventType, string sLogName)
   {
     EventLog oEventLog = new EventLog();
     try
     {
      //Register the Application as an Event Source
      if (!EventLog.SourceExists(sAppName))
      {
        EventLog.CreateEventSource(sAppName, sLogName);
      }

       //log the entry
       oEventLog.Source = sAppName;
       oEventLog.WriteEntry(sEntry, eEventType);
       return true;
      }
      catch (Exception Ex)
      {
       return false;
      }
   }
  }
}
```

Example 8.10-1

8.11 Custom Error Log

You can also create your own event log to log errors. For example lets say your company has a specific event log names ABCCompanyLog. You can use this to log your application specific errors.

We use the EventLog.CreateEventSource method to establish an application as a valid event source for writing application specific localized event messages, using the specified configuration properties for the event source and the Corresponding event log.

The CreateEventSource method uses the input sourceData Source, LogName and MachineName properties to create registry values on the target computer for the new source and its associated event log. A new source name cannot match an existing source name or an existing event log name on the target computer. If the LogName property is not set, the source is registered for the Application event log. If the MachineName is not set, the source is registered on the local computer. See example 8.11-1

```
using System;
using System.Diagnostics;
using System.Threading;

class MyLogSample
{
 public static void Main()
 {
  // Create the source, if it does not already exist.
  if(!EventLog.SourceExists("MySource", "MyServer"))
  {
     EventLog.CreateEventSource("MySource",   "ABCCompany", "MyServer");
     Console.WriteLine("CreatingEventSource");
  }

  // Create an EventLog instance and assign its source.
  EventLog myLog = new EventLog();
  myLog.Source = "MySource";

  // Write an informational entry to the event log.
  myLog.WriteEntry("Writing to event log.");
  Console.WriteLine("Message written to event log.");
 }
}
```

Example 8.11-1

8.12Custom Error Pages

ASP.NET provides a simple yet powerful way to deal with errors that occur in your web applications. We will look at several ways to trap errors and display friendly meaningful messages to users. We will then take the discussion a step further and learn how to be instantly notified about problems so you can cope with them right away. As a geek touch we will also track the path 404's travel.

8.12-1 Trapping Errors On Page Level

Every time you create a web form in Visual Studio .NET you see that your page class derives from System.Web.UI.Page. The Page object helps you trap page-level errors. For this to happen you need to override its OnError method as shown in example 8.12-1 For this purpose you can create a common base class named PageBase.

```
using System;
using System.Web;
using System.Web.UI;

namespace AspNetResources.CustomErrors1 {
 public class PageBase : System.Web.UI.Page  {
  protected override void OnError(EventArgs e)
  {
   // At this point we have information about the error
   HttpContext ctx = HttpContext.Current;

   Exception exception = ctx.Server.GetLastError ();

   string errorInfo = "<br>Offending URL: " +  ctx.Request.Url.ToString () +
         "<br>Source: " + exception.Source +  "<br>Message: " + exception.Message +
         "<br>Stack trace: " + exception.StackTrace;

   ctx.Response.Write (errorInfo);

   /* To let the page finish running we clear the error */
   ctx.Server.ClearError ();

   base.OnError (e);
  } } }
```

Example 8.12-1

➢8.o Error Handling, Logging and Tracing

Once this base page (PageBase) is created, you create your code behind of the page inherited from this (PageBase) class (example 8.12-2-a). The page script (the presentation) is also given here (8.12-2-b).

```
using System;
using System.Web;

namespace AspNetResources.CustomErrors1
{
 public class _Default : PageBase
 {
  private void Page_Load (object sender, System.EventArgs e)
  {
   // Just throw an exception //
   throw new Exception ("Silly exception");
  }
 }
}
```

Example 8.12-2-a

```
<%@ Page language="c#" Codebehind="Default.aspx.cs"
AutoEventWireup="false"
Inherits="AspNetResources.CustomErrors1._Default" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN" >
<HTML>
  <HEAD>
    <title>Default page</title>
</HEAD>
<body>
    <form id="Form1" method="post" runat="server">
    </form>
</body>
</HTML>
```

Example 8.12-2-b

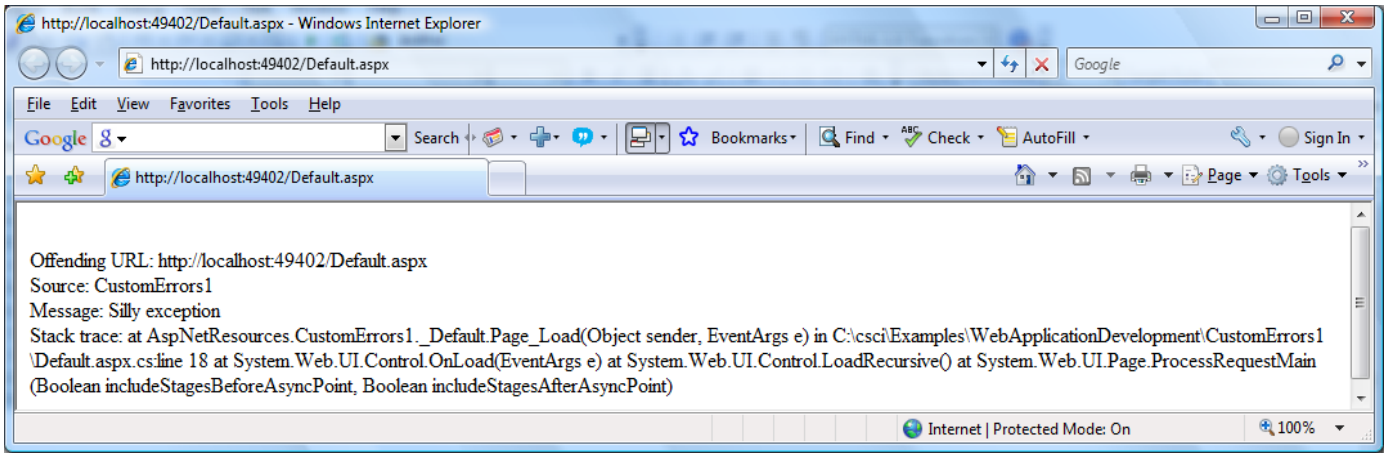


Figure 8.12-1

8.13 Application Level Error Trapping

The idea of capturing errors on the application level is somewhat similar. At this point we need to rehash our understanding of the **Global.asax** file.

From the moment you request a page in your browser to the moment you see a response on your screen a complex process takes place on the server. Your request travels through the ASP.NET pipeline.

In the eyes of IIS each virtual directory is an application. When a request within a certain virtual directory is placed, the pipeline creates an instance of **HttpApplication** to process the request. The runtime maintains a pool of **HttpApplication** objects. The same instance of **HttpApplication** will service a request it is responsible for. This instance can be pooled and reused only after it is done processing a request.

Global.asax is optional which means if you are not interested in any session or application events you can live without it. Otherwise the ASP.NET runtime parses your **Global.asax**, compiles a class derived from **HttpApplication** and hands it a request for your web application.

HttpApplication fires a number of events. One of them is Error. To implement your own handler for application-level errors your **Global.asax** file needs to have code similar to this:

```
protected void Application_Error(object sender, EventArgs e)
{
}
```

When any exception is thrown now—be it a general exception or a 404—it will end up in Application_Error. The following implementation of this handler is similar to the one above – see example 8.13-1.

Be careful when modifying **Global.asax**. The ASP.NET framework detects that you changed it, flushes all session state and closed all browser sessions and—in essence—reboots your application. When a new page request arrives, the framework will parse **Global.asax** and compile a new object derived from HttpApplication again.

➢8.0 Error Handling, Logging and Tracing

```
protected void Application_Error(Object sender, EventArgs e)
{
  // At this point we have information about the error
  HttpContext ctx = HttpContext.Current;

  Exception exception = ctx.Server.GetLastError ();

  string errorInfo =  "<br>Offending URL: " + ctx.Request.Url.ToString () +
    "<br>Source: " + exception.Source +  "<br>Message: " + exception.Message +
    "<br>Stack trace: " + exception.StackTrace;

  ctx.Response.Write (errorInfo);

  // ------------------------------------------------
  // To let the page finish running we clear the error
  // ------------------------------------------------
  ctx.Server.ClearError ();
}
```

Example 8.13-1

8.14 Setting Custom Error Pages in web.config.

If an exception has not been handed by the Page object, or the HttpApplication object and has not been cleared through Server.ClearError() it will be dealt with according to the settings of web.config.

When you first create an ASP.NET web project in Visual Studio .NET you get a web.config for free with a small <customErrors> section:

<customErrors mode="RemoteOnly" />

With this setting your visitors will see a canned error page much like the one from ASP days. To save your face you can have ASP.NET display a nice page with an apology and a suggested plan of action.

The mode attribute can be one of the following:

On	Error details are not shown to anybody, even local users. If you specified a custom error page it will be always used.
Off	Everyone will see error details, both local and remote users. If you specified a custom error page it will NOT be used.
RemoteOnly	Local users will see detailed error pages with a stack trace and compilation details, while remote users with be presented with a concise page notifying them that an error occurred. If a custom error page is available, it will be shown to the remote users only.

Figure 8.14-1

Displaying a concise yet not-so-pretty error page to visitors is still not good enough, so you need to put together a custom error page and specify it this way:

```
<customErrors
    mode="RemoteOnly"
    defaultRedirect="~/errors/GeneralError.aspx”/>
```

Should anything happen now, you will see a detailed stack trace and remote users will be automatically redirected to the custom error page, GeneralError.aspx. How you apologize to users for the inconvenience is up to you.

The <customErrors> tag may also contain several <error> (see MSDN) subtags for more granular error handling. Each <error> tag allows you to set a custom condition based upon an HTTP status code. For example, you may display a custom 404 for missing pages and a general error page for all other exceptions:

```
<customErrors mode="On" defaultRedirect="~/errors/GeneralError.aspx">
    <error statusCode="404" redirect="~/errors/PageNotFound.aspx" />
</customErrors>
```

8.14.2 Clearing Errors

You probably noticed that Server.ClearError() was used in both OnError and Application_Error above. It is to let the page run its course. What happens if you comment it out? The exception will leave Application_Error and continue to crawl up the stack until it's handled and put to rest. If you set custom error pages in web.config the runtime will act accordingly—you get to collect exception information AND see a friendly error page.

8.15 Page Tracing

Page-level tracing enables you to write debugging statements directly to a page's output, and **conditionally** run debugging code when tracing is enabled. To enable tracing for a page, include the following directive at the top of the

```
page code: <%@ Page Trace="true"%>
```

Trace statements can also be organized by category, using the TraceMode attribute of the Page directive. If no **TraceMode** attribute is defined, the default value is **SortByTime.**

```
<%@ Page Trace="true" TraceMode="SortByCategory" %>
```

The following example shows the default output when page-level tracing is enabled. Note that ASP.NET inserts timing information for important places in the page's execution lifecycle:

```
<%@ Page Trace="true" TraceMode="SortByCategory"  Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="WebApplication2._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   </div>
   </form>
</body>
</html>
```

Page Trace is set. Mode is set to sort by category

Example 8.15-1

Figure 8.15-1 shows a dump of Page Tracing.

➢8.0 Error Handling, Logging and Tracing

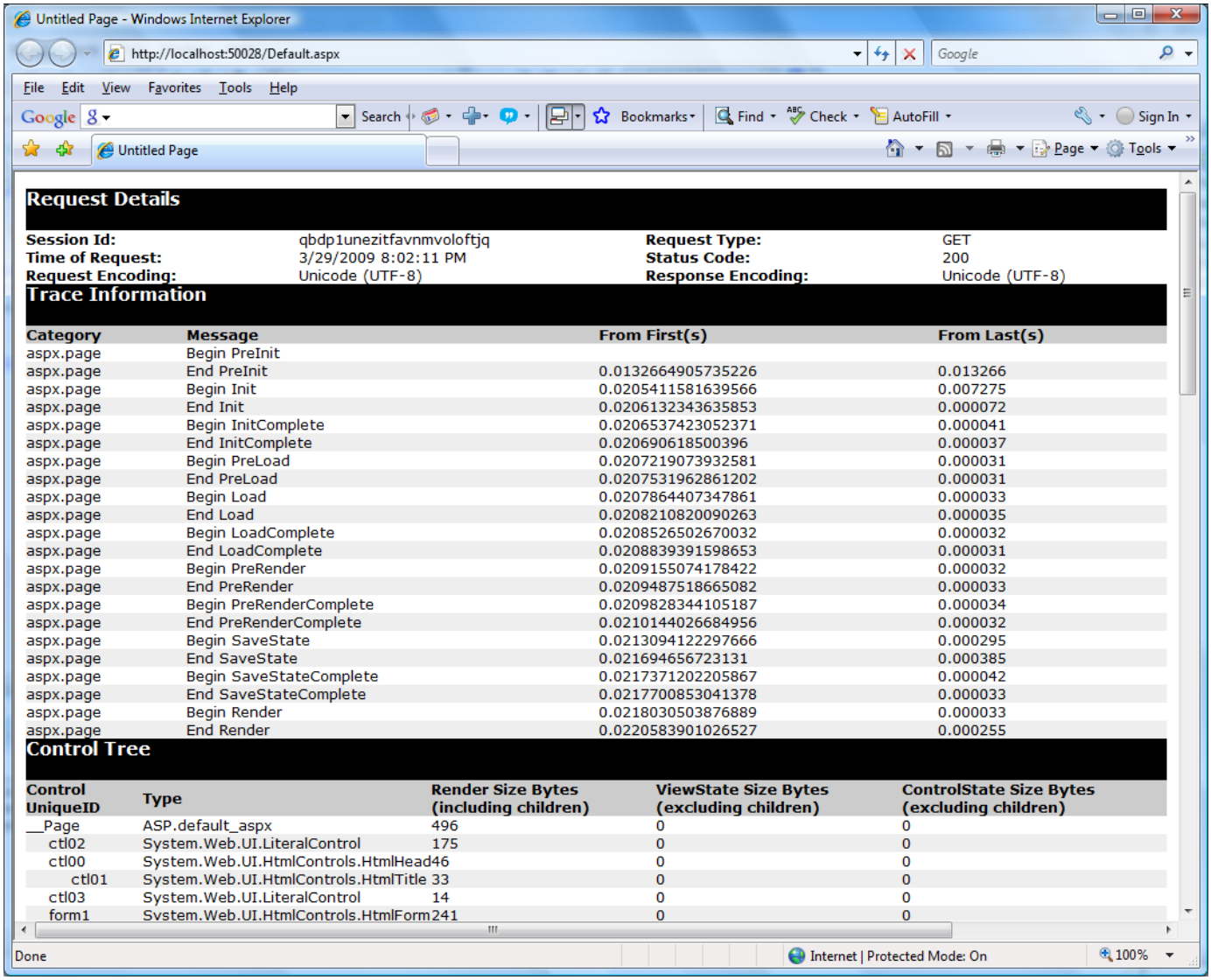


Figure 8.15-1

8.16 Tracing at Application Level

You can enable tracing for the entire application by adding tracing settings in web.config. In below example, pageOutput="false" and requestLimit="20" are used, so trace information is stored for 20 requests, but not displayed on the page because pageOutput attribute is set to false.

```
<configuration>
    <appSettings/>
    <connectionStrings/>
    <system.web>
        <compilation debug="false" />
        <authentication mode="Windows" />
      <trace enabled ="true" pageOutput ="false" requestLimit ="20"
                                          traceMode ="SortByTime " />
    </system.web>
</configuration>
```

The above markup has several attributes and values. This next table (Figure 8.16-1 shows what some of these properties mean, and why they are necessary. The property name is on the left.

Part	What it means
pageOutput	False tells ASP.NET not to embed a large table in each page. You can change this if you want inline trace messages at the bottom of every aspx page.
requestLimit	20 here indicates that ASP.NET should store at most 20 trace messages.
enabled	False here says that tracing should not run at all. You must change this to "true" if you want to enable tracing.
traceMode	Show the list sorted by time.

Figure 8.16-1

9.0 State Management

(Text Book Chapter 08)

9.0 State Management

9.0 Introduction

The most significant difference between programming for the Web and programming for the desktop is state management—how you store information over the lifetime of your application. This information can be as simple as a user's name or as complex as a stuffed -full shopping cart for an e-commerce store.

A traditional Windows program, users interact with a continuously running application. A portion of memory on the desktop computer is allocated to store the current set of working information. In a traditional web application, the story is quite a bit different. A professional ASP.NET site might look like a continuously running application, but that's really just a clever illusion. In a typical web request, the client connects to the web server and requests a page. When the page is delivered, the connection is severed, **and the web server discards all the page objects from memory**. By the time the user receives a page, the web page code has already stopped running, and there's no information left in the web server's memory.

This stateless design has one significant advantage. Because clients need to be connected for only a few seconds at most, a web server can handle a huge number of nearly simultaneous requests without a performance hit. However, if you want to retain information for a longer period of time so it can be used over multiple postbacks or on multiple pages, you need to take additional steps. So in other words, traditional web applications do have a problem.

9.1 Solution

To overcome this inherent limitation of traditional Web programming, ASP.NET includes several options that help you preserve data on both a per-page basis and an application-wide basis. These features are as follows:

View state	Control state	Hidden fields
Cookies	Query strings	Application state
Session state	Profile Properties	

9.0 State Management

View state, **control state**, **hidden fields**, **cookies**, and **query strings** all involve **storing data on the client in various ways**. However, **application state, session state, and profile properties a**ll store data in memory on the server. Each option has distinct advantages and disadvantages, depending on the scenario.

9.2 ViewState

View Sate, is the technique used by an ASP.NET Web page to persist changes to the state of a Web Form across **postbacks** The view state of a page is, by default, placed in a hidden form field named **__VIEWSTATE**. This hidden form field can easily get very large, on the order of tens of kilobytes. Not only does the **__VIEWSTATE** form field cause slower downloads, but, whenever the user posts back the Web page, the contents of this hidden form field must be posted back in the HTTP request, thereby lengthening the request time, as well.

9.2.1 How does ViewState work?

All server controls have a property called ViewState. If this is enabled, the ViewState for the control is also enabled. Where and how is ViewState stored? When the page is first created all controls are serialized to the ViewState, which is rendered as a hidden form field named __ViewState. This hidden field corresponds to the server side object known as the ViewState. ViewState for a page is stored as key-value pairs using the System.Web.UI.StateBag object. When a post back occurs, the page de-serializes the ViewState and recreates all controls. The ViewState for the controls in a page is stored as base 64 encoded strings in name - value pairs. When a page is reloaded two methods pertaining to ViewState are called, namely the LoadViewState method and SaveViewState method. The following is the content of the __ViewState hidden field as generated for a pageIn a particular system.

```
<input type="hidden" name="__VIEWSTATE"
value="dNrATo45Tm5QzQ7Oz8AblWpxPjE9MMl0Aq765QnCmP2TQ==" />
```

9.2.2 The ViewState Collection

The ViewState property of the page provides the current view state information. This property provides an instance of the StateBag collection class. The StateBag is a dictionary collection, which means every item is stored in a separate "slot" using a unique string name. For example, consider this code:

```
// The this keyword refers to the current Page object. It's optional.
this.ViewState["Counter"] = 1;
```

This places the value 1 (or rather, an integer that contains the value 1) into the **ViewState** collection and gives it the descriptive name Counter. If currently no item has the name Counter, a new item will be added automatically. If an item is already stored under the name Counter, it will be replaced. When retrieving a value, you use the key name. You also need to cast the retrieved value to the appropriate data type using the casting. This extra step is required because the **ViewState** collection stores all item s as basic objects, which allows it to handle many different data types. Here's the code that retrieves the counter from view state and converts it to an integer:

```
int counter;
counter = (int)this.ViewState["Counter"];
```

Maintaining the ViewState is the default setting for ASP.NET Web Forms. If you want to NOT maintain the ViewState, include the directive **<%@ Page EnableViewState="false"** %> at the top of an .aspx page or Add the attribute **EnableViewState="false"** to any control.

See example following page.

9.0 State Management

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="ViewStateExample-2.aspx.cs"
Inherits="ViewStateExample_2._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:Button ID="button1" runat="server" Text="Click to
Count" OnClick="Count_Clicks" /> <br/>
   <asp:Label ID="label1" runat="server" Text="" />
   </div>
   </form>
</body>
</html>
```

Example 9.2-1a

```
using System;

using System.Xml.Linq;

namespace ViewStateExample_2
{
   public partial class _Default : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
       label1.Text = "Clicks : " +
          (ViewState["count_clicks"] == null ? 0 :
                (int) ViewState["count_clicks"]);
      }

      protected void Count_Clicks(object sender, EventArgse)
      {
         if (ViewState["count_clicks"] == null)
         {
            ViewState["count_clicks"] = 1;
            label1.Text = "Clicks : " +
                  (int)ViewState["count_clicks"];
         }
         else
         {
            ViewState["count_clicks"] =
               (int)ViewState["count_clicks"] + 1;
            label1.Text = "Clicks : " +
                   (int)ViewState["count_clicks"];
         }
      }
   }}
```

Example 9.2-1a

Start the first browser; initially the ViewState Count is 0

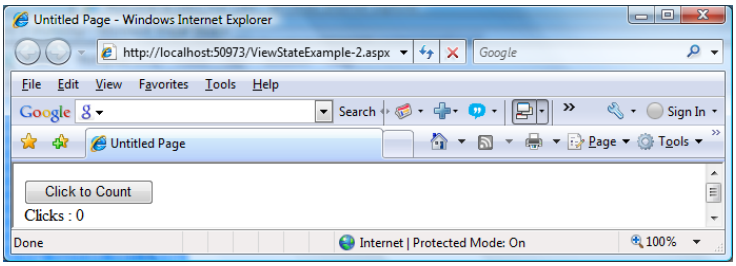


Figure 9.2-1a

Counter updates after Clicking the button several times

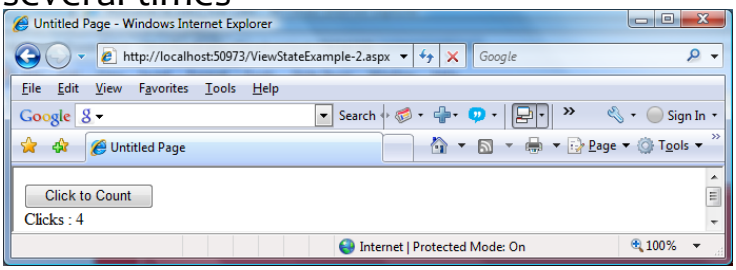


Figure 9.2-1b

Start a second browser while the first is still alive. The Count start at 0 for the new browser

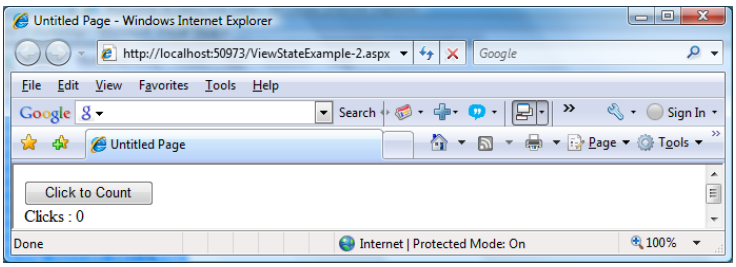


Figure 9.2-2a

Counter updates after Clicking the button several times

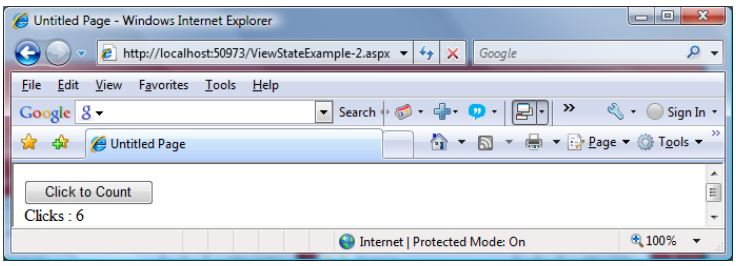


Figure 9.2-2b

9.0 State Management

If a counter need to be shared between the number of clients, then a counter need to be defined as static. Remember static objects are shared by all instances of a class.

```
using System;
.
using System.Xml.Linq;

namespace ViewStateExample_2
{
 public partial class _Default : System.Web.UI.Page
 {
  private static int count = 0;
  protected void Page_Load(object sender, EventArgs e)
  {
   label1.Text = "Clicks : " + count;
  }

  protected void Count_Clicks(object sender, EventArgs e)
  {
   if (ViewState["count_clicks"] == null)
   {
    count = 1;
    ViewState["count_clicks"] = count;
          label1.Text = "Clicks : " +
    (int)ViewState["count_clicks"];
   }
   else
   {
    count = count + 1;
     ViewState["count_clicks"] = count;
        label1.Text = "Clicks : " +
     (int)ViewState["count_clicks"];
   }
 }}}
```

Example 9.2-3

Start a client and click the button several times.

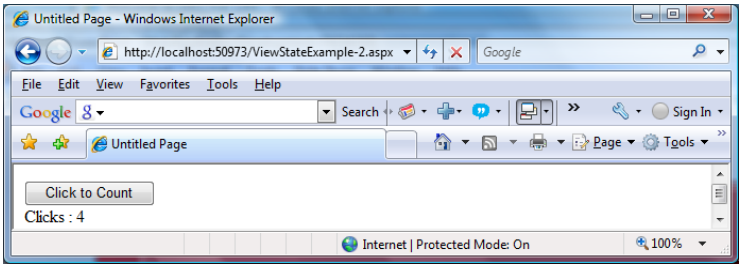


Figure 9.2-2a

Start a second client. The start count is where the last count updated by the first client.

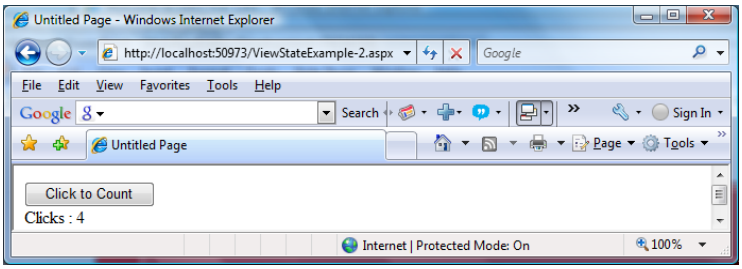


Figure 9.2-2b

9.2.3 Saving and Restoring Values to and from the ViewState

ViewState works with the following types.

Primitive types
Arrays of primitive types
ArrayList and Hashtable
Any other serializable object

To add an ArrayList object to the ViewState use the following statements.

```
ayList obj = new ArrayList(); //Some code ViewState["ViewStateObject"] = obj;
```

To retrieve the ArryList object use the following statement.

```
obj = ViewState["ViewStateObject"];
```

9.2.4 Making View State Secure

Previously we discussed about the format of the view state information string format that usually look like this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="dDw3NDg2NTI5MDg7Oz4=" />
```

As you add more information to view state, this value can become much longer. Because this value isn't formatted as clear text, many ASP.NET programmers assume that their view state data is encrypted. It isn't. Instead, the view state information is simply patched together in memory and converted to a Base64 string (which is a special type of string that's always acceptable in an HTML document because it doesn't include any extended characters). A clever hacker could reverse-engineer this string and examine your view state data in a matter of seconds. Tamper-Proof View State If you want to make view state more secure, you have two choices.

First, you can make sure the view state information is tamperproof by instructing ASP.NET to use a hash code. **A hash code is sometimes described as a cryptographically strong checksum**. The idea is that ASP.NET examines all the data in view state, just before it renders the final page. It runs this data through a hashing algorithm (with the help of a secret key value). The hashing algorithm creates a short segment of data, which is the Hash code. This code is then added at the end of the view state data, in the final HTML that's sent to the browser. When the page is posted back, ASP.NET examines the view state data and recalculates the hash code using the same process. It then checks whether the checksum it calculated matches the hash code that is stored in the view state for the page. If a malicious user changes part of the view state data, ASP.NET will end up with a new hash code that doesn't match. At this point, it will reject the postback completely. (You might think a really clever user could get around this by generating fake view state information and a matching hash code.

However, malicious users can't generate the right hash code, because they don't have the same cryptographic key as ASP.NET. This means the hash

hash codes they create won't match.)Hash codes are actually enabled by default, so if you want this functionality, you don't need to take any extra steps.

9.2.5 Private View State

Even when you use hash codes, the view state data will still be readable by the user. In many cases, this is completely acceptable—after all, the view state tracks information that's often provided directly through other controls. However, if your view state contains some information you want to keep secret, you can enable view state encryption.

You can turn on encryption for an individual page using the **ViewStateEncryptionMode** property of the Page directive:

```
<%@Page ViewStateEncryptionMode="Always" %>
```

Or you can set the same attribute in a configuration file to configure view state encryption for all the pages in your website:

```
<configuration>
 <system.web>
  <pages viewStateEncryptionMode="Always" />
   ...
 </system.web>
</configuration>
```

Either way, this enforces encryption. You have three choices for your view state encryption setting—always encrypt (Always), never encrypt (Never), or encrypt only if a control specifically requests it (Auto). The default is Auto, which means that the page won't encrypt its view state unless a control on that page specifically requests it. (Technically, a control makes this request by calling the **Page.RegisterRequiresViewStateEncryption()** method.) If no control calls this method to indicate it has sensitive information, the view state is not encrypted, thereby saving the encryption overhead. On the other hand, a control doesn't have absolute power—if it calls **Page.RegisterRequiresViewStateEncryption()** and the encryption mode is Never, the view state won't be encrypted.

9.3 Transferring Information Between Pages (Cross-Page Posting)

9.3.1 Using the `PostBackUrl` property

Normally we use the term **postback** when an ASP.NET page submits it's content back to that page itself. But there can be situation when a page needs to submit it's content to a different target page. This is known as cross page **postback**. In this post we will discuss about how to handle and implement cross page **postback** scenario. Now to make a page **postback** to another page we have set the **PostBackUrl** property as shown below:

```
<asp:Button ID="Button2" runat="server"
        Text="CrossPagePostback"
                PostBackUrl="~/TargetForm.aspx" />
```

After the content is submitted to the target url we need to retrieve the control values and properties of the page. to do this we have use the PreviousPage property of the Page class as shown below. The PreviousPage maintains a reference to the instance of the source page.

```
Label1.Text = "Crosspage Postback with value :" +
    (Page.PreviousPage.FindControl("TextBox1") as
    TextBox).Text;
```

Suppose we have a public property defined in the source page as shown below:

```
public string ScreenID { get; set; }
```

If we want to access this property in the target page then we need to specify the type of source page in the @PreviousPageType directive of the target page:

```
<%@ PreviousPageType
    VirtualPath="~/TargetForm.aspx"  %>
```

Then we can access the property in target page as follows:

```
Label1.Text = PreviousPage.ScreenID;
```

9.0 State Management

Using the *PostBackUrl* Property Example: the code for the start page

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CrossPagePosting1.aspx.cs"
Inherits="CrossPagePosting1._Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
    First Name  :
    <asp:TextBox ID="txtFirstName"
          runat="server"></asp:TextBox> <br />
    Last name :
    <asp:TextBox ID="txtLastName“
       runat="server"></asp:TextBox> <br />
    <br />
    <asp:Button ID="btnPostCommand"
        Text="Click to Cross Page“
     PostBackUrl="CrossPagePosting2.aspx" runat="server" />
   </div>
   </form>
</body>
</html>
```

Example 9.3.-1a

```
using System;
.
using System.Xml.Linq;

namespace CrossPagePosting1
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {
  }

  public string FirstName
  {
   get { return txtFirstName.Text; }
  }

  public string LastName
  {
   get { return txtLastName.Text; }
  }
 }
}
```

Example 9.3-1b

Cross page posting can be done in two ways. Both are explained below.

First Method : Here is the code for the Cross-Page using the directive **@PreviousPageType**

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CrossPagePosting2.aspx.cs"
Inherits="CrossPagePosting2._Default" %>
<%@ PreviousPageType VirtualPath="~/CrossPagePosting1.aspx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
 <form id="form1" runat="server">
 <div>
 <asp:Label ID="lblCapture" runat="server"></asp:Label> <br />
 </div>
 </form>
</body>
</html>
```

Example 9.3-1c

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
namespace CrossPagePosting2
{
 public partial class _Default : System.Web.UI.Page
 { protected void Page_Load(object sender, EventArgs e)
  {
   this.lblCapture.Text = "You typed : " +
          PreviousPage.FirstName + " as first name and "+
          PreviousPage.LastName + " as the last name";
    }
  }
}
```

Example 9.3-1d

➢9.0 State Management

Notice that in this method we have set up the `<%@ PreviousPageType VirtualPath= "~/CrossPagePosting1.aspx" %>` Property.

Second Method: is to use a statement like : `CrossPagePosting1._Default previousPage = PreviousPage as CrossPagePosting1._Default;` In the Page_Load() method in code behind of the cross-page.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CrossPagePosting2.aspx.cs"
Inherits="CrossPagePosting2._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
   <asp:Label ID="lblCapture" runat="server"></asp:Label> <br />
  </div>
  </form>
</body>
</html>
```

Example 9.3.-1e

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

namespace CrossPagePosting2
{
  public partial class _Default : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      CrossPagePosting1._Default previousPage = PreviousPage as
                                          CrossPagePosting1._Default;

      this.lblCapture.Text = "You typed : " + previousPage.FirstName +
                                          " as first name and "+
        previousPage.LastName + " as the last name";
    }
  }
}
```

Example 9.3-1f

In both cases you get the same result. You enter a first named and a last name in the first page and then click the "Click to Cross Page" button. The data fields of the first page can be then accessed in the second (Cross Page)

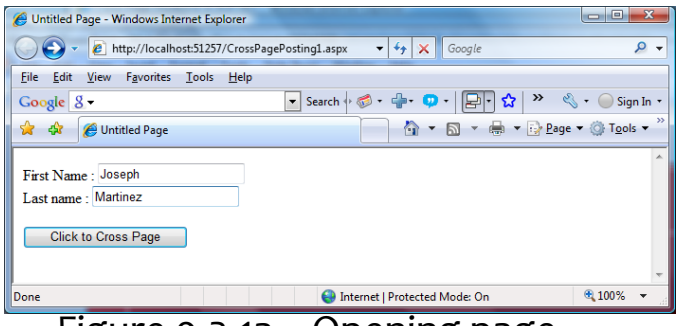


Figure 9.3-1a – Opening page

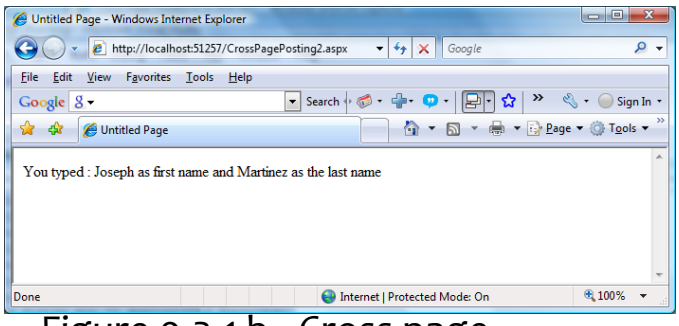


Figure 9.3-1 b– Cross page

9.3.2 Query String Approach (Using QueryString property)

Another common approach is to pass information using a query string in the URL. This approach is commonly found in search engines. For example, if you perform a search on the Google website, you'll be redirected to a new URL that incorporates your search parameters. Here's an example:

http://www.google.ca/search?q=organic+gardening

The query string is the portion of the URL after the question mark. In this case, it defines a single variable named q, which contains the string organic+gardening. The advantage of the query string is that it's lightweight and doesn't exert any kind of burden on the server. However, it also has several limitations:

- Information is limited to simple strings, which must contain URL-legal characters.
- Information is clearly visible to the user and to anyone else who cares to eavesdrop on the Internet.
- The enterprising user might decide to modify the query string and supply new values, which your program won't expect and can't protect against. Many browsers impose a limit on the length of a URL (usually from 1KB to 2KB).

For that reason, you can't place a large amount of information in the query String and still be assured of compatibility with most browsers.

Adding information to the query string is still a useful technique. It's particularly well suited in database applications where you present the user with a list of items that correspond to records in a database, such as products. The user can then select an item and be forwarded to another page with detailed information about the selected item. One easy way to implement this design is to have the first page send the item ID to the second page. The second page then looks that item up in the database and displays the detailed information. You'll notice this technique in e-ommerce sites such as Amazon.

➢9.0 State Management

To store information in the query string, you need to place it there yourself. Unfortunately, you have no collection-based way to do this. Instead, you'll need to insert it into the URL yourself. Here's an example that uses this approach with the Response.Redirect() method:

```
// Go to newpage.aspx. Submit a single query string argument
// named recordID, and set to 10.

Response.Redirect("newpage.aspx?recordID=10");
```

You can send multiple parameters as long as they're separated with an ampersand (&):

```
// Go to newpage.aspx. Submit two query string arguments:
// recordID (10) and mode (full).

Response.Redirect("newpage.aspx?recordID=10&mode=full");
```

The receiving page has an easier time working with the query string. It can receive the values from the QueryString dictionary collection exposed by the built-in Request object:

```
string ID = Request.QueryString["recordID"];
```

Note that information is always retrieved as a string, which can then be converted to another simple data type. Values in the QueryString collection are indexed by the variable name. If you attempt to retrieve a value that isn't present in the query string, you'll get a null reference.

In the example in the next page presents a list of entries. When the user chooses an item by clicking the appropriate item in the list, the user is forwarded to a new page. This page displays the received ID number. This provides a quick and simple query string test with two pages. In a sophisticated application, you would want to combine some of the data control features that are described later.

9.0 State Management

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="QueryStringSender.aspx.cs"
Inherits="QueryStringDemo2.QueryStringSender" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:ListBox ID="aListBox" runat="server" Width="175"
Height="100"></asp:ListBox><br/><br/>
   <asp:CheckBox ID="aCheckBox" runat="server"
Text="Show full details" /> <br/><br/>
   <asp:Button ID="aButton" runat="server" Text="View
Information" OnClick="cmdGo_Click" />
   <asp:Label ID="lblError" runat="server"></asp:Label>
   </div>
   </form>
</body>
</html>
```

Example 9.3.-2a

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="QueryStringRecipient.aspx.cs"
Inherits="QueryStringDemo2.QueryStringRecipient" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:Panel ID="aPanel" runat="server" BorderColor=""
Width="100%" Height="80px" BackColor="Yellow">
     <asp:Label ID="aLabelR" runat="server" Font-Size="XX-
Large" ForeColor="#CC0000"></asp:Label>
   </asp:Panel>
   </div>
   </form>
</body>
</html>
```

Example 9.3.-2c

```
namespace QueryStringDemo2
{
   public partial class QueryStringSender :
System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e) {
         if (!this.IsPostBack)          {
            // Add sample values.
            aListBox.Items.Add("Econo Sofa");
            aListBox.Items.Add("Supreme Leather Drapery");
            aListBox.Items.Add("Threadbare Carpet");
            aListBox.Items.Add("Antique Lamp");
            aListBox.Items.Add("Retro-Finish Jacuzzi");
         }
      }

      protected void cmdGo_Click(object sender, EventArgs e)
      {
         if (aListBox.SelectedIndex == -1)          {
            lblError.Text = "You must select an item.";
         }
         else  {
            // Forward the user to the information page,
            // with the query string data.
            string url = "QueryStringRecipient.aspx?";
            url += "Item=" + aListBox.SelectedItem.Text + "&";
            url += "Mode=" + aCheckBox.Checked.ToString();
            Response.Redirect(url);
         }
      }
   }
}
```

Example 9.3.-2b

```
namespace QueryStringDemo2
{
   public partial class QueryStringRecipient :
System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
         aLabelR.Text = "Item: " +
Request.QueryString["Item"];
         aLabelR.Text += "<br />Show Full Record: ";
         aLabelR.Text += Request.QueryString["Mode"];
      }
   }
}
```

Example 9.3.-2d

9.0 State Management

The following is the output produced by the example in the previous page.

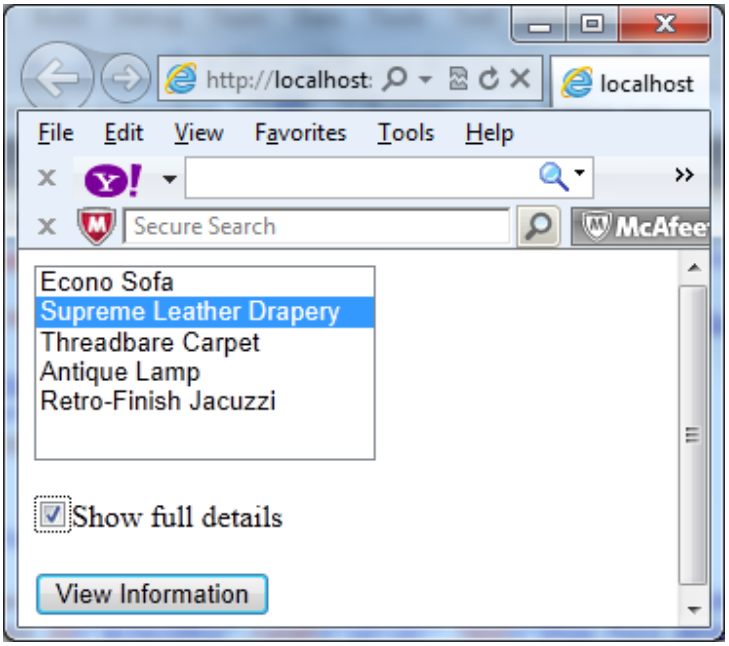


Figure 9.3-2a

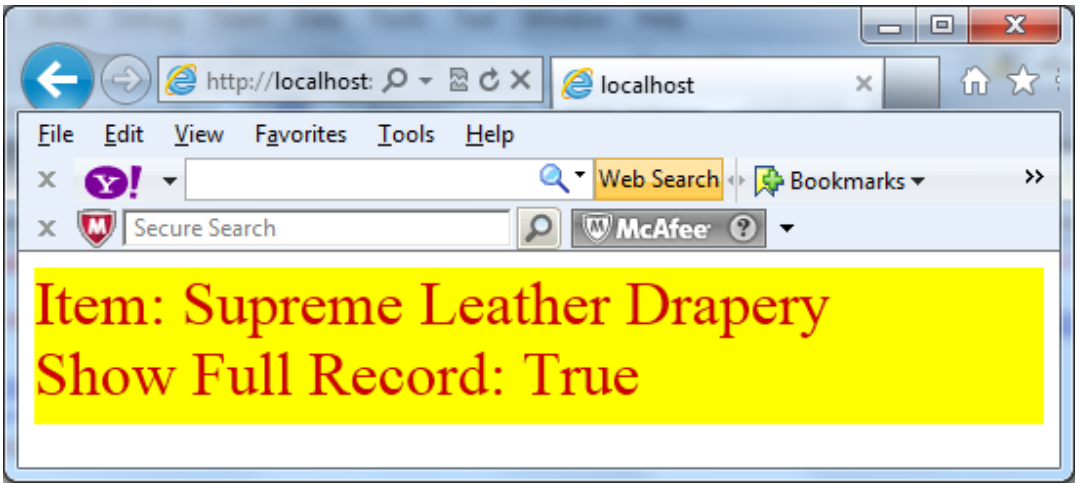


Figure 9.3-2b

One interesting aspect of this example is that it places information in the query string that isn't valid—namely, the space that appears in the item name. When you run the application, you'll notice that ASP.NET encodes the string for you automatically, converting spaces to the valid %20 equivalent escape sequence. The recipient page reads the original values from the QueryString collection without any trouble. This automatic encoding isn't always sufficient. To deal with special characters, you should use the URL encoding technique described in the next section.

9.3.2.1 Potential Problems in Query String Approach

URL Encoding:

One potential problem with the query string is that some characters aren't 1 allowed in a URL. In fact, the list of characters that are allowed in a URL is much shorter than the list of allowed characters in an HTML document. All characters must be alphanumeric or one of a small set of special characters (including $-_.+!*'(),). Some browsers tolerate certain additional special characters (Internet Explorer is notoriously lax), but many do not. Furthermore, some characters have special meaning. For example, the ampersand (&) is used to separate multiple query string parameters, the plus sign (+) is an alternate way to represent a space, and the number sign (#) is used to point to a specific bookmark in a web page. If you try to send query string values that include any of these characters, you'll lose some of your data. You can test this with the previous example by adding items with special characters in the list box.

To avoid potential problems, it's a good idea to perform URL encoding on text values before you place them in the query string. With URL encoding, special characters are replaced by escaped character sequences starting with the percent sign (%), followed by a two-digit hexadecimal representation. For example, the & character becomes %26. The only exception is the space character, which can be represented as the character sequence %20 or the + sign.

To perform URL encoding, you use the **UrlEncode()** and Url**Decode()** methods of the **HttpServerUtility** class. As you learned in Chapter 5, an **HttpServerUtility** object is made available to your code in every web form through the **Page.Server** property. The following code uses the **UrlEncode()** method to rewrite the previous example, so it works with product names that contain special characters:

```
String url = "QueryStringRecipient.aspx?"; url += "Item=“ +
Server.UrlEncode(lstItems.SelectedItem.Text) + "&";
url += "Mode=" + chkDetails.Checked.ToString();
Response.Redirect(url);
```

Notice that it's important not to encode everything. In this example, you can't encode the & character that joins the two query string values, because it truly is a special character.

You can use the **UrlDecode()** method to return a URL-encoded string to its initial value. However, you don't need to take this step with the query string. That's because ASP.NET automatically decodes your values when you access them through the **Request.QueryString** collection. (Many people still make the mistake of decoding the query string values a second time. Usually, decoding already decoded data won't cause a problem. The only exception is if you have a value that includes the + sign. In this case, using **UrlDecode()** will convert the + sign to a space, which isn't what you want.)

9.3.3 Using Cookies

A cookie is often used to identify a user. A cookie is a small file that the server embeds on the user's computer. Each time the same computer requests a page with a browser, it will send the cookie too. With ASP, you can both create and retrieve cookie values. Lets take a look at an example.

```
What is a Cookie?
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Cookie-Page1.aspx.cs" Inherits="Cookie_Page1._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="Form1" runat="server">
<asp:TextBox ID=cookname runat="server" />Cookie Name<br/>
<asp:TextBox ID=cookvalue runat="server" />Cookie value<br/>
<asp:TextBox ID=cookexpiry runat="server" />Days to Expiration<br/>
<asp:Button  ID="submitbutton1" text="Set cookie" OnClick="SubmitButton_Click1" runat="server"/>
<asp:Button  ID="submitbutton2" text="Get cookie" OnClick="SubmitButton_Click2" runat="server"/>
</form>
<asp:Label ID="msg" runat="server" /><br/>
<asp:Label ID="stat" runat="server" />
</body>
</html>
```

Example 9.3-3-a

The Code Behind

```
using System;
using System.Xml.Linq;
namespace Cookie_Page1
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {}
  protected void SubmitButton_Click1(object sender, EventArgs e)
  {  msg.Text = SetCookie(cookname.Text, cookvalue.Text,
    Convert.ToInt32(cookexpiry.Text)).ToString();
    msg.Text += ". Expires: " +
      Response.Cookies[cookname.Text].Expires.ToString();
  }

  protected void SubmitButton_Click2(object sender, EventArgs e)
  { msg.Text = GetCookie(cookname.Text); }

  public bool SetCookie(string cookiename,
          string cookievalue, int iDaysToExpire)
  { try {
        HttpCookie objCookie = new HttpCookie(cookiename);
        Response.Cookies.Clear();
        Response.Cookies.Add(objCookie);
        objCookie.Values.Add(cookiename, cookievalue);
        DateTime dtExpiry = DateTime.Now.AddDays(iDaysToExpire);
        Response.Cookies[cookiename].Expires = dtExpiry;
       }
    catch (Exception e) { return false; }
        return true;
  }}
```

```
public string GetCookie(string cookiename)
{
 string cookyval = "";
 try
 {
  cookyval =
     Request.Cookies[cookiename].Value;
 }
 catch (Exception e)
 {
   cookyval = "";
 }
 return cookyval;
 }
}
```

Example 9.3-3b

9.0 State Management

When you run this example, after you enter the values and click the button, it will take a while to set the cookies. Then close the browser, start the application again, you will see when you start typing on the text boxes, the previous value you entered will pop out in a drop. That's the saved cookie. Here is the result.

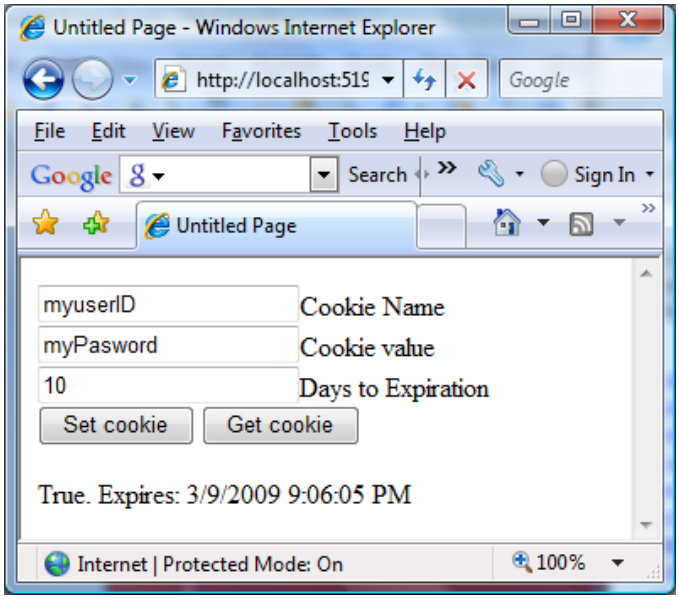


Figure 9.3-3a - First Run. Setting the cookies

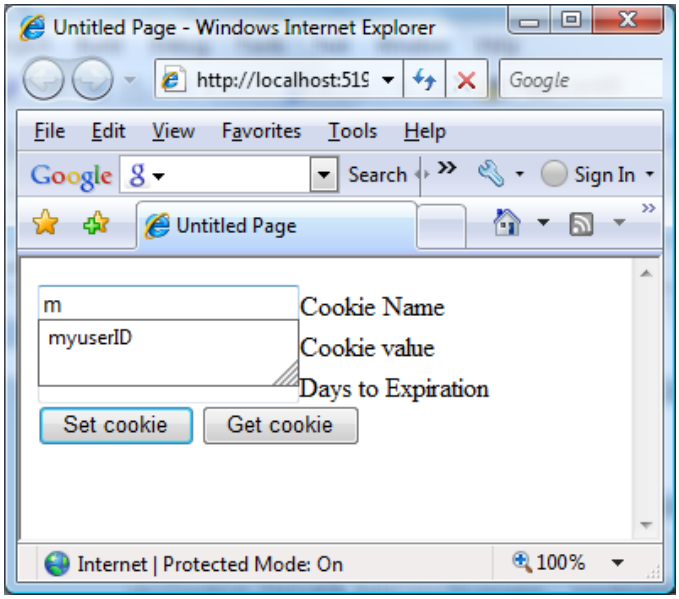


Figure 9.3-3b - Second Run. Notice the cookies

Note : Max number of cookies allowed per website is 20.

9.3.4 Session State

There comes a point in the life of most applications when they begin to have more sophisticated storage requirements. An application might need to store and access complex information such as custom data objects, which can't be easily persisted to a cookie or sent through a query string. Or the application might have stringent security requirements that prevent it from storing information about a client in view state or in a custom cookie. In these situations, **you can use ASP.NET's built-in session state facility**. Session state management is one of ASP.NET's premiere features. It allows you to store any type of data in memory on the server. The information is protected, because it is never transmitted to the client, and it's uniquely bound to a specific session. Every client that accesses the application has a different session and a distinct collection of information. Session state is ideal for storing information such as the items in the current user's shopping basket when the user browses from one page to another.

9.3.4.1 Session Tracking

ASP.NET tracks each session using a unique 120-bit identifier. ASP.NET uses a proprietary algorithm to generate this value, thereby guaranteeing (statistically speaking) that the number is unique and it's random enough that a malicious user can't reverse-engineer or "guess" what session ID a given client will be using. This ID is the only piece of session-related information that is transmitted between the web server and the client. When the client presents the session ID, ASP.NET looks up the corresponding session, retrieves the objects you stored previously, and places them into a special collection so they can be accessed in your code. This process takes place automatically.

For this system to work, the client must present the appropriate session ID with each request. You can accomplish this in two ways:

Using cookies: In this case, the session ID is transmitted in a special cookie (named ASP.NET_SessionId), which ASP.NET creates automatically when the session collection is used. This is the default.

Using modified URLs: In this case, the session ID is transmitted in a specially modified (or munged) URL. This allows you to create applications that use session state with clients that don't support cookies. Session state doesn't come for free. Though it solves many of the problems associated with other forms of state management, it forces the server to store additional information in memory. This extra memory requirement, even if it is small, can quickly grow to performance-destroying levels as hundreds or thousands of clients access the site. In other words, you must think through any use of session state. A careless use of session state is one of the most common reasons that a web application can't scale to serve a large number of clients. Sometimes a better approach is to use caching – which we do not discuss in detail In this course.

9.3.4.2 Using Session State

You can **interact with session state** using the System.Web.SessionState.HttpSessionState class, which is provided in an ASP.NET web page as the built-in **Session** object. The syntax for adding items to the collection and retrieving them is basically the same as for adding items to a page's view state. For example, you might store a DataSet in session memory like this:

```
Session["InfoDataSet"] = dsInfo;
```

You can then retrieve it with an appropriate conversion operation:

```
dsInfo = (DataSet)Session["InfoDataSet"];
```

Of course, before you attempt to use the dsInfo object, you'll need to check that it actually exists—in other words, that it isn't a null reference. If the dsInfo is null, it's up to you to regenerate it. (For example, you might decide to query a database to get the latest data.)

It is important to remember that session variables are now objects. Thus, to avoid a run-time error, you should check whether the variable is set before you try to access it.

9.0 State Management

Session variables are **automatically discarded after they are not used for the time-out setting that is specified in the web.config** file. On each request, the time out is reset. The variables are lost when the session is explicitly abandoned in the code.

When a session is initiated on first request, the server issues a unique session ID to the user. To persist the session ID, store it in an in-memory cookie (which is the default), or embed it within the request URL after the application name. To switch between cookie and **cookieless** session state, set the value of the **cookieless** parameter in the **web.config** file to true or false.

In **cookieless** mode, the server automatically inserts the session ID in the relative URLs only. An absolute URL is not modified, even if it points to the same ASP.NET application, which can cause the loss of session variables.

ASP.NET supports three modes of session state:
InProc: In-Proc mode stores values in the memory of the ASP.NET worker process. Thus, this mode offers the fastest access to these values. However, when the ASP.NET worker process recycles, the state data is lost.

StateServer: Alternately, StateServer mode uses a stand-alone Microsoft Windows service to store session variables. Because this service is independent of Microsoft Internet Information Server (IIS), it can run on a separate server. **You can use this mode for a load-balancing solution** because multiple Web servers can share session variables. Although session variables are not lost if you restart IIS, performance is impacted when you cross process boundaries.

SqlServer: If you are greatly concerned about the persistence of session information, you can use SqlServer mode to leverage Microsoft SQL Server to ensure the highest level of reliability. SqlServer mode is similar to out-of-Process mode, except that **the session data is maintained in a SQL Server**. SqlServer mode also enables you to utilize a state store that is located out of the IIS process and that can be located on the local computer or a remote server.

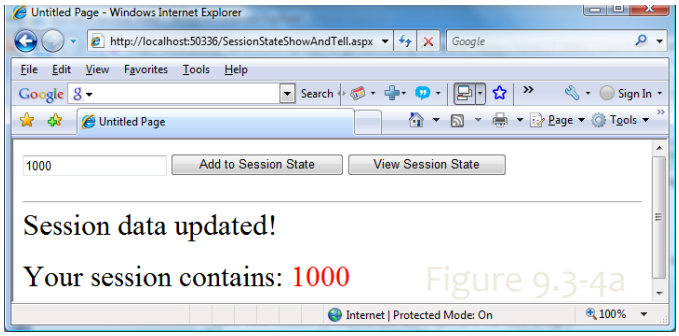
9.0 State Management

In the example below during the session of a client , s session variable is set by the client. Then it is interrogated.

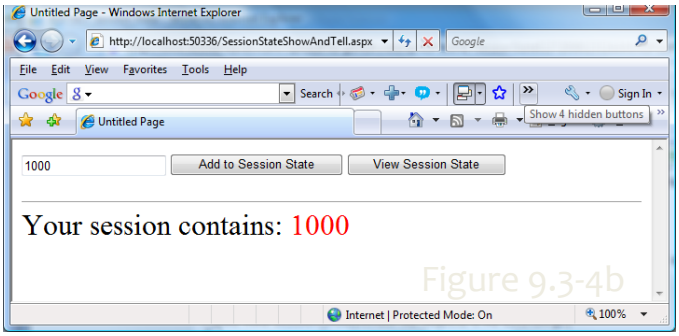
```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="SessionStateShowAndTell.aspx.cs"
Inherits="SessionStateExample._Default" %>=
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:TextBox ID="textBox1Text" Text="just Text“
              runat="server" ></asp:TextBox>
   <asp:Button ID="buttonAddState" runat="server“
    Text="Add to Session State" OnClick="SessionAdd" />
   <asp:Button ID="buttonSeeState" runat="server“
    Text="View Session State" OnClick="CheckSession" />
   <asp:Label ID="label1" runat="server" Text="" />
   </div>
   </form>
   <hr size=1>
   <font size=6><span id=span1 runat=server/></font>
</body>
</html>
```

```
using System;
using System.Xml.Linq;

namespace SessionStateExample
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  { }
  protected void SessionAdd(object sender, EventArgs e)
  {
   Session["MySession"] = textBox1Text.Text;
   span1.InnerHtml = "Session data updated! <P> “ +
         ”Your session contains: <font color=red>" +
          Session["MySession"].ToString() + "</font>";
  }

  protected void CheckSession(object sender, EventArgs e)
  { if (Session["MySession"] == null)
   {span1.InnerHtml = "NOTHING, SESSION DATA LOST!"; }
   else
   {
    span1.InnerHtml = "Your session contains: “ +
    ”<font color=red>" + Session["MySession"].ToString() +
    "</font>";
   } } }}
```

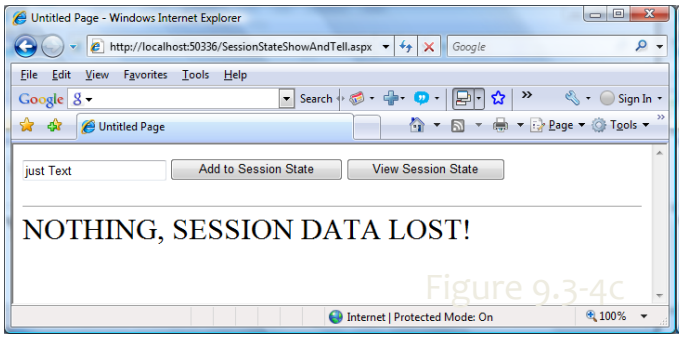
Example 9.3-4a

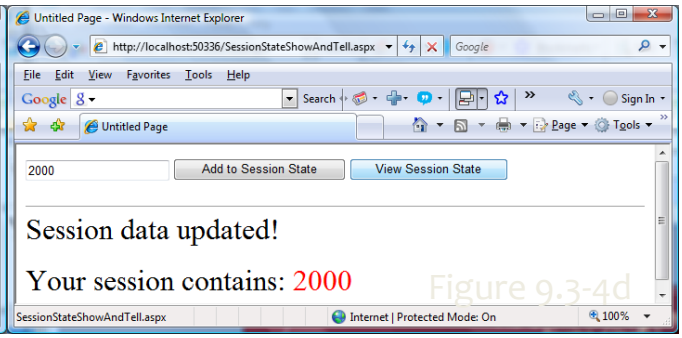


Example 9.3-4b



Then another client is opened. The second client does not see the first clients session data. The second client has it’s own session data.





9.3.4.3 When Does The Session Ends

Session's state is global to the entire application per client. But the client can lose the session data in several ways.

1. If the user closes the browser.
2. If the user accesses the same page through a different browser, although the session will still exist if a web page is accessed through the original browser window. Browsers differ on how they handle this situation.
3. If the session times out due to in activity.
4. If the web page code ends the session by calling the **Session.Abandon()** method.

9.3.4.4 Session Configurations

Sessions can be managed by setting the application specific session configurations. ASP.NET provides three modes of session state storage controlled by mode attribute of **<sessionState>** tag in your web application's **web.config** file. Below is a sample of this tag:

```
<sessionState
   mode="InProc"
   stateConnectionString="tcpip=127.0.0.1:42424"
   sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
   cookieless="false"
   timeout="20"
/>
```

Example 9.3-5

9.3.4.5 Session Timeout

The Session Object : When you are working with an application, you open it, do some changes and then you close it. This is much like a Session. The computer knows who you are. It knows when you start the application and when you end. But on the internet there is one problem: the web server does not know who you are and what you do because the HTTP address doesn't maintain state.

ASP.NET solves this problem by creating a unique cookie for each user. The cookie is sent to the client and it contains information that identifies the user. This interface is called the Session object.

9.0 State Management

The Session object is used to store information about, or change settings for a user session. Variables stored in the Session object hold information about one single user, and are available to all pages in one application. Common information stored in session variables are name, id, and references. The server creates a new Session object for each new user, and destroys the Session object when the session expires.

The Session object's has properties. One is the Timeout property you could use to time out the session when the use do not use the client.

Setting the session time out value:
By default, sessions timeout in 20 minutes. Add the following line to your application's **web.config** to change this value where x is the minutes to wait before a session times out. (Put this line immediately after the <system.web> line.)

```
<sessionState mode="InProc" timeout="x" />
```

Or you can use the **Session.Timeout** in the C# code behind.

In the example below, the **Session.Timeout** is set to one minute. After the application is started, the session will expire in 1 minute. Notice the message "NOTHING, SESSION DATA LOST! when clicked the "View Session State" after one minute.

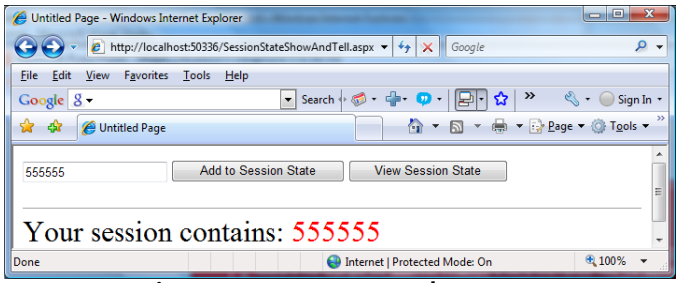


Figure 9.3-5a – At the start

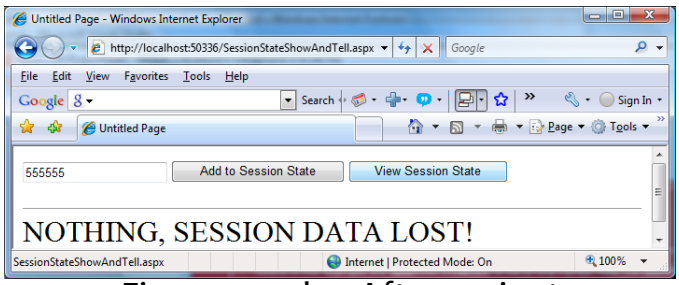


Figure 9.3-5b – After 1 minute

9.3.4.6 What is the difference between ViewState and SessionState?
ViewState persist the values of controls of particular page in the client (browser) when post back operation done. When user requests another page previous page data no longer available.

SessionState persist the data of particular user in the server. This data available till user close the browser or session time completes.

9.3.5 Application State

Application state is something that should be used with care, and in most cases, avoided altogether. Although it is a convenient repository for global data in a Web application, its use can severely limit the scalability of an application, especially if it is used to store shared, updateable state. It is also an unreliable place to store data, because it is replicated with each application instance and is not saved if the application is recycled. With this warning in mind, let's explore how it works.

Application state is accessed through the Application property of the HttpApplication class, which returns an instance of class HttpApplicationState. This class is a named object collection, which means that it can hold data of any type as part of a key/value pair. Example 9.3-6 (a & b) shows a typical use of application state. As soon as the application is started, it loads the data from the database. Subsequent data accesses will not need to go to the database but will instead access the application state object's cached version. Data that is pre-fetched in this way must be static, because it will not be unloaded from the application until the application is recycled or otherwise stopped and restarted.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="ApplicationStateStart.aspx.cs"
Inherits="ApplicationStateExample._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
    <asp:Label ID="lblOne" Text="" runat="server" ></asp:Label>
   </div>
   </form>
</body>
</html>
```

Example 9.3-6a

```
using System;
using System.Xml.Linq;
namespace ApplicationStateExample
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {
   int count = 0;
   if (Application["HitCount"] != null)
   { count = (int)Application["HitCount"]; }

    //Increase the count
    count++;

    //Store it.
    Application["HitCount"] = count;
    lblOne.Text = "Current Hits : " + count;
 }}}
```

Example 9.3-6b

In the example 9.3.6 (a & b), the first client (Figure 9.3-6a starts at hits 1. Then hit the refresh button several times that rings up the hits to 9. Then open up another client (Figure 9.3-6b, you will see the it count at 10.

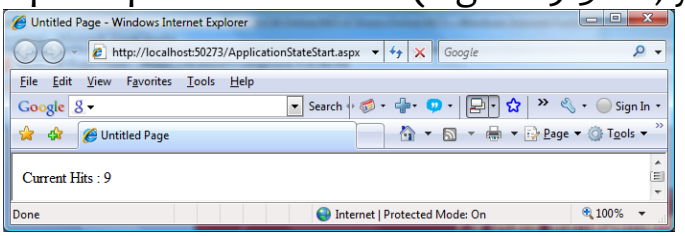


Figure 9.3-6a – At the start

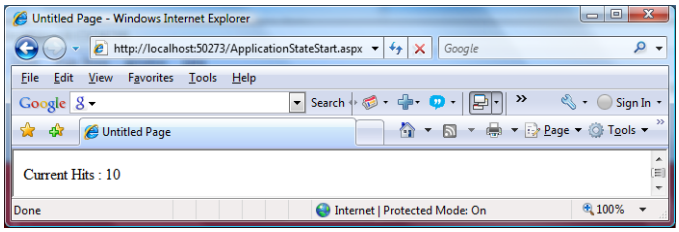


Figure 9.3-6a – At the start

9.3.5.1 Locking Application State

Application State Synchronization:

Multiple threads within an application can simultaneously access values stored in application state. Consequently, when you create something that needs to access application-state values, you must always ensure that the application-state object is free-threaded and performs its own internal synchronization or else performs manual synchronization steps to protect against race conditions, deadlocks, and access violations.

The **HttpApplicationState** class provides two methods, Lock and Unlock, that allow only one thread at a time to access application-state variables.

Calling Lock on the Application object causes ASP.NET to block attempts by code running on other worker threads to access anything in application state. These threads are unblocked only when the thread that called Lock calls the corresponding Unlock method on the Application object.

Application State can be locked for a particular client by calling `Application.Lock()` method. Then when ready it can unlock the state back by using `Application.UnLock();`

In the previous example, we can lock the usage access to count and `Application["HitCount"]` by using the Lock Unlock mechanism as shown below.

The following code Example 9.3-7 demonstrates the use of locking to guard against race conditions.

9.0 State Management

In the example below, when one client is accessing the Application Data, other clients who are trying access the same data are put on hold. So they will feel a delay in processing.

```
using System;
using System.Xml.Linq;

namespace ApplicationStateExample
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {
    Application.Lock();
    int count = 0;
    if (Application["HitCount"] != null)
    { count = (int)Application["HitCount"]; }

     //Increase the count
     count++;

     //Store it.
     Application["HitCount"] = count;
     lblOne.Text = "Current Hits : " + count;
     Application.UnLock();
  }
 }
}
```

Example 9.3-7

10.0 Building Better Forms - Validation (Text Book Chapter 9)

10.0 Understanding Validation

10.1 Common User Mistakes

Common knowledge as developers is that the users, when using any application will make mistakes when entering data. What's particularly daunting is the range of possible mistakes that users can make. Here are some common examples:

- Users might ignore an important field and leave it blank.
- Users might try to type a short string of nonsense to circumvent a required field check, thereby creating endless headaches on your end. For example, you might get stuck with an invalid e-mail address that causes problems for your automatic e-mailing program.
- Users might make an honest mistake, such as entering a typing error, entering a nonnumeric character in a number field, or submitting the wrong type of information. They might even enter several pieces of information that are individually correct but when taken together are inconsistent (for example, entering a MasterCard number after choosing Visa as the payment type).
- Malicious users might try to exploit a weakness in your code by entering carefully structured wrong values. For example, they might attempt to cause a specific error that will reveal sensitive information. A more dramatic example of this technique is the SQL injection attack, where user-supplied values change the operation of a dynamically constructed database command. (Of course, validation is no defense for poor coding. In a later chapter you'll learn how to use parameterized commands, which avoid the danger of SQL injection attacks altogether.) A web application is particularly susceptible to these problems, because it relies on basic HTMLinput controls that don't have all the features of their Windows counterparts.

➢10.0 Building Better Forms - Validation

A web application is particularly susceptible to these problems, because it relies on basic HTML input controls that don't have all the features of their Windows counterparts. For example, a common technique in a Windows application is to handle the KeyPress event of a text box, check to see whether the current character is valid, and prevent it from appearing if it isn't. This technique makes it easy to create a text box that accepts only numeric input.

In web applications, however, you don't have that sort of fine-grained control. To handle a KeyPress event, the page would have to be posted back to the server every time the user types a letter, which would slow down the application hopelessly. Instead, you need to perform all your validation at once when a page (which may contain multiple input controls) is submitted. You then need to create the appropriate user interface to report the mistakes. Some websites report only the first incorrect field, while others use a table, list, or window to describe them all. By the time you've perfected your validation strategy, you'll have spent a considerable amount of effort writing tedious code.

ASP.NET aims to save you this trouble and provide you with a reusable framework of validation controls that manages validation details by checking fields and reporting on errors automatically. These controls can even use client-side JavaScript to provide a more dynamic and responsive interface while still providing ordinary validation for older browsers (often referred to as down-level browsers).

➢10.0 Building Better Forms - Validation

10.1.1 The Validation Controls

ASP.NET provides five validation controls, which are described in Table 10-1. Four are targeted at specific types of validation, while the fifth allows you to apply custom validation routines. You'll also see a **ValidationSummary** control in the Toolbox, which gives you another option for showing a list of Validation error messages in one place. You'll learn about the **ValidationSummary** later in this chapter (see the "Other Display Options" section).

Table Validator Controls	
Control Class	**Description**
RequiredFieldValidator	Validation succeeds as long as the input control doesn't contain an empty string.
RangeValidator	Validation succeeds if the input control contains a value within a specific numeric, alphabetic, or date range.
CompareValidator	Validation succeeds if the input control contains a value that matches the value in another input control, or a fixed value that you specify.
RegularExpressionValidator	Validation succeeds if the value in an input control matches a specified regular expression.
CustomValidator	Validation is performed by a user-defined function.

Figure 10.1-1 – Type of Validation Controls

The above validation controls are found in the System.Web.UI.WebControls namespace and inherit from the **BaseValidator**. Each validation control can be bound to a single input control. In addition, you can apply more than one validation control to the same input control to provide multiple types of validation. If you use the **RangeValidator**, **CompareValidator**, or **RegularExpressionValidator**, validation will automatically succeed if the input control is empty, because there is no value to validate. If this isn't the behavior you want, you should also add a **RequiredFieldValidator** and link it to the same input control. This ensures that two types of validation will be performed, effectively restricting blank values.

10.1.2 Server Side Validation

You can use the validation controls to verify a page automatically when the user submits it or manually in your code. The first approach is the most common.

When using automatic validation, the user receives a normal page and begins to fill in the input controls. When finished, the user clicks a button to submit the page. Every **button** has a **CausesValidation** property, which can be set to true or false. What happens when the user clicks the button depends on the value of the **CausesValidation** property. If **CausesValidation** is false, ASP.NET will ignore the validation controls, the page will be posted back, and your event-handling code will run normally.

If **CausesValidation** is true (the default), ASP.NET will automatically validate the page when the user clicks the button. It does this by performing the validation for each control on the page. If any control fails to validate, ASP.NET will return the page with some error information, depending on your settings. Your click event-handling code may or may not be executed -meaning you'll have to specifically check in the event handler whether the page is valid.

Based on this description, you'll realize that validation happens automatically when certain buttons are clicked. It doesn't happen when the page is posted back because of a change event (such as choosing a new value in an **AutoPostBack** list) or if the user clicks a button that has **CausesValidation** set to false. However, you can still validate one or more controls manually and then make a decision in your code based on the results. You'll learn about this process in more detail a little later (see the "Manual Validation" section).

NOTE: Many other button-like controls that can be used to submit the page also provide the **CausesValidation** property. Examples include the **LinkButton**, **ImageButton**, and **BulletedList**.

In most modern browsers (including Internet Explorer 5 or later and any version of Firefox), ASP.NET automatically adds JavaScript code for client-side validation. In this case, when the user clicks a CausesValidation button

the same error messages will appear without the page needing to be submitted and returned from the server. This increases the responsiveness of your web page.

However, even if the page validates successfully on the client side, ASP.NET still revalidates it when it's received at the server. This is because it's easy for an experienced user to circumvent client-side validation. For example, a malicious user might delete the block of JavaScript validation code and continue working with the page. By performing the validation at both ends, ASP.NET makes sure your application can be as responsive as possible while also remaining secure.

10.1.3 Validation Control Properties

As said, all Validation controls inherit from the **BaseValidator** class which defines the basic functionality for a validation control. The table 10.1-2 describes key properties **BaseValidator** class has. Therefore these properties are inherited to other validation controls.

Table Properties of the BaseValidator Class	
Property	**Description**
ControlToValidate	Identifies the control that this validator will check. Each validator can verify the value in one input control. However, it's perfectly reasonable to "stack" validators—in other words, attach several validators to one input control to perform more than one type of error checking.
ErrorMessage and ForeColor	If validation fails, the validator control can display a text message (set by the ErrorMessage property). By changing the ForeColor, you can make this message stand out in angry red lettering.
Display	Allows you to conFigure whether this error message will be inserted into the page dynamically when it's needed (Dynamic) or whether an appropriate space will be reserved for the message (Static). Dynamic is useful when you're placing several validators next to each other. That way, the space will expand to fit the currently active error indicators, and you won't be left with any unseemly whitespace. Static is useful when the validator is in a table and you don't want the width of the cell to collapse when no message is displayed Finally, you can also choose None to hide the error message altogether.
IsValid	After validation is performed, this returns true or false depending on whether it succeeded or failed. Generally, you'll check the state of the entire page by looking at its IsValid property instead to find out if all the validation controls succeeded.
Enabled	When set to false, automatic validation will not be performed for this control when the page is submitted.
EnableClientScript	If set to true, ASP.NET will add JavaScript and DHTML code to allow client-side validation on browsers that support it.

Figure 10.1-2 – BaseValidator class Properties

➢10.0 Building Better Forms - Validation

When using a validation control, the only properties you need to implement are **ControlToValidate** and **ErrorMessage**. In addition, you may need to implement the properties that are used for your specific validator. Table 10.1-3 outlines these properties.

Table Validator-Specific Properties	
Validator Control	**Added Members**
RequiredFieldValidator	None required
RangeValidator	MaximumValue, MinimumValue, Type
CompareValidator	ControlToCompare, Operator, Type, ValueToCompare
RegularExpressionValidator	ValidationExpression
CustomValidator	ClientValidationFunction, ValidateEmptyText, ServerValidate event

Figure 10.1-3 – BaseValidator's Implementable Properties

RangeValidator Example:

A Simple Validation Example To understand how validation works, you can create a simple web page. This test uses a single Button web control, two TextBox controls, and a RangeValidator control that validates the first text box. If validation fails, the RangeValidator control displays an error message, so you should place this control immediately next to the TextBox it's validating. The second text box does not use any validation. The figure 10.1-4 shows the appearance of the page after a failed validation attempt.

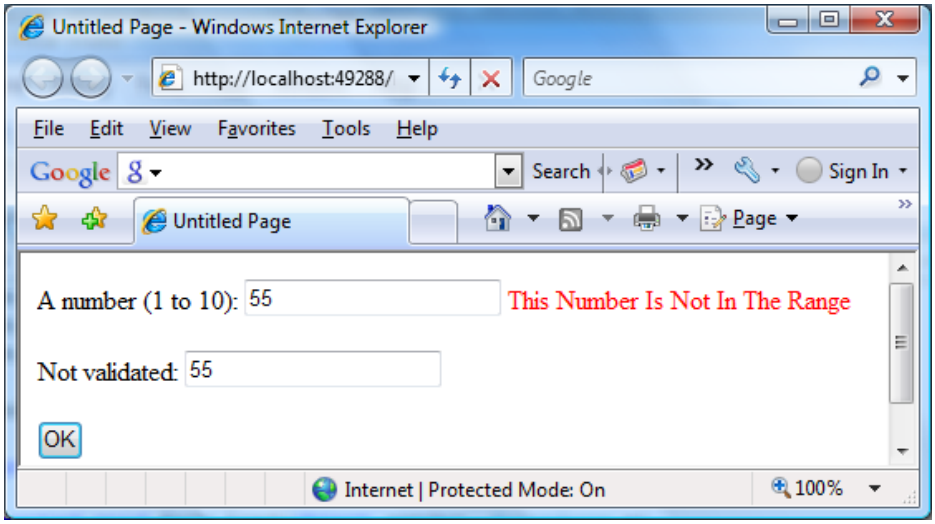


Figure 10.1-4 – Validation Example

In addition, place a Label control at the bottom of the form. This label will report when the page has been posted back and the event-handling code has executed. Disable its **EnableViewState** property to ensure that it will be cleared every time the page is posted back.

The markup for this page defines a **RangeValidator** control, sets the error message, identifies the control that will be validated, and requires an integer from 1 to 10. These properties are set in the .aspx file, but they could also be configured in the event handler for the Page.Load event. The Button automatically has its **CauseValidation** property set to true, because this is the default.

```
A number (1 to 10):
<asp:TextBox id="txtValidated" runat="server" />
<asp:RangeValidator id="RangeValidator" runat="server"
  ErrorMessage="This Number Is Not In The Range“ ControlToValidate="txtValidated"
  MaximumValue="10" MinimumValue="1“ Type="Integer" />
<br /><br />
Not validated:
<asp:TextBox id="txtNotValidated" runat="server" /><br /><br />
<asp:Button id="cmdOK" runat="server" Text="OK" OnClick="cmdOK_Click" />

<br /><br />
<asp:Label id="lblMessage" runat="server"
  EnableViewState="False" />
```

Example 10.1-1a – Using a RangeValidator

Here is the code that responds to the button click:

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
   lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

Example 10.1-1b – Code Behind

If you're testing this web page in a modern browser, you'll notice an interesting trick. When you first open the page, the error message is hidden. But if you type an invalid number (remember, validation will succeed for an empty value) and press the Tab key to move to the second text box, an error message will appear automatically next to the offending control. This is because ASP.NET adds a special JavaScript function that detects when the focus changes.

➢10.0 Building Better Forms - Validation

The actual implementation of this JavaScript code is somewhat complicated, but ASP.NET handles all the details for you automatically. As a result, if you try to click the OK button with an invalid value in **txtValidated**, your actions will be Ignored and the page won't be posted back.

Not all browsers will support client-side validation. To see what will happen on a down-level browser, set the **RangeValidator.EnableClientScript** property to false, and rerun the page. Now error messages won't appear dynamically as you change focus. However, when you click the OK button, the page will be returned from the server with the appropriate error message displayed next to the invalid control.

The potential problem in this scenario is that the click event-handling code will still execute, even though the page is invalid. To correct this problem and ensure that your page behaves the same on modern and older browsers, you must specifically abort the event code if validation hasn't been performed successfully.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
   // Abort the event if the control isn't valid.
   if (!RangeValidator.IsValid) return;
   lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

Example 10.1-1c – Modified Code Behind

The code (next slide) solves the current problem, but it isn't much help if the page contains multiple validation controls. Fortunately, every web form provides its own **IsValid** property. This property will be false if any validation control has failed. It will be true if all the validation controls completed successfully. If validation was not performed (for example, if the Validation controls are disabled or if the button has **CausesValidation** set to false), you'll get an **HttpException** when you attempt to read the **IsValid** property.

➢10.0 Building Better Forms - Validation

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
   // Abort the event if any control on the page is invalid.
   if (!Page.IsValid) return;
   lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

Example 10.1-1d – Code Behind with IsValid property check

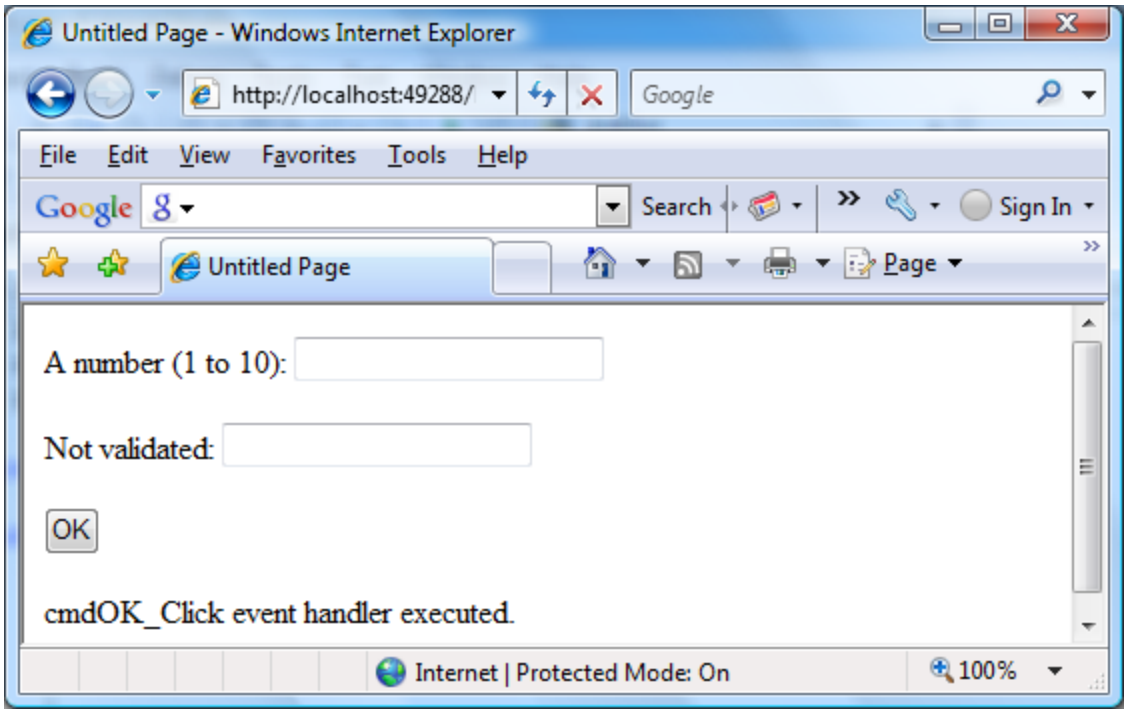


Figure 10.1-5 – Page is invalid

10.1.3 Other Display Options – *ValidationSummary*

In some cases, you might have already created a carefully designed form that combines multiple input fields. Perhaps you want to add validation to this page, but you can't reformat the layout to accommodate all the error messages for all the validation controls. In this case, you can save some work by using the **ValidationSummary** control. To try this, set the Display property of the **RangeValidator** control to None. This ensures the error message will never be displayed. However, validation will still be performed and the user will still be prevented from successfully clicking the OK button if some invalid information exists on the page. Next, add the **ValidationSummary** in a suitable location (such as the bottom of the page). A standard Validation Summary control will look like this:

```
<asp:ValidationSummary id="Errors" runat="server" />
```

```
<body>
   <form id="form1" runat="server">
   <div>
   A number (1 to 10):
   <asp:TextBox id="txtValidated" runat="server" />
   <asp:RangeValidator id="RangeValidator" runat="server"
    ErrorMessage="This Number Is Not In The Range"
    ControlToValidate="txtValidated"
    MaximumValue="10" MinimumValue="1"
    Type="Integer" />
    <br /><br />
  Not validated:
  <asp:TextBox id="txtNotValidated" runat="server" /><br /><br />
  <asp:Button id="cmdOK" runat="server" Text="OK" OnClick="cmdOK_Click" />
  <br /><br />
  <asp:ValidationSummary id="Errors" runat="server" />
 </div>
 </form>
</body>
```

Example 10.1-2a – Adding a ValidationSummary

Code behind is not virtually blank.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
  // Abort the event if any control on the page is invalid.
  //if (!Page.IsValid) return;
  //lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

Example 10.1-2b – Adding a ValidationSummary

But still you see a ValidationSummary

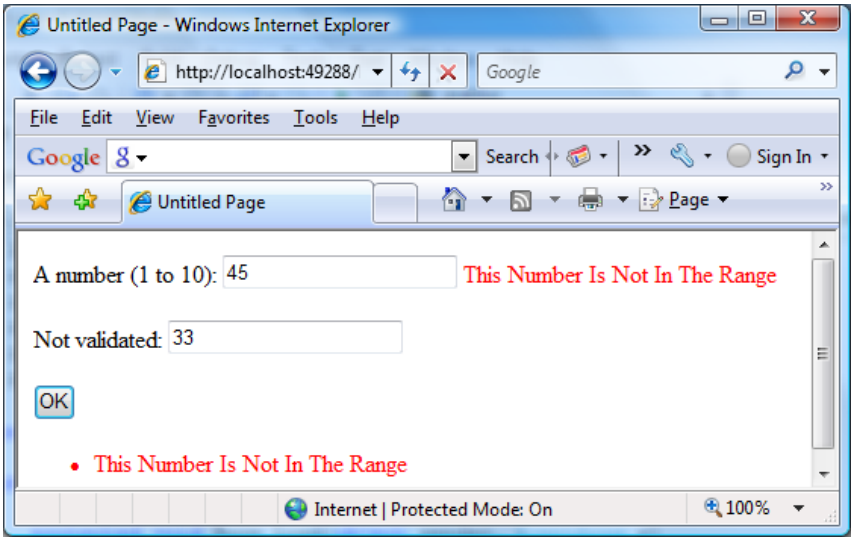


Figure 10.1-6 – Page with a ValidationSummary

10.1.3.1 Validation Summary with Error Indicator

When the **ValidationSummary** displays the list of errors, it automatically retrieves the value of the **ErrorMessage** property from each validator. In some cases, you'll want to display a full message in the summary and some sort of visual indicator next to the offending control. For example, many websites use an error icon or an asterisk to highlight text boxes with invalid input. You can use this technique with the help of the Text property of the validators. Ordinarily, Text is left empty. However, if you set both Text and **ErrorMessage**, the **ErrorMessage** value will be used for the summary while the Text value is displayed in the validator. (Of course, you'll need to make sure you aren't also setting the Display property of your validator to None, which hides it completely.)

➢10.0 Building Better Forms - Validation

Here's an example of a validator that includes a detailed error message (which will appear in the ValidationSummary) and an asterisk indicator (which will appear in the validator, next to the control that has the problem):

```
<asp:RangeValidator id="RangeValidator" runat="server"
  Text="*" ErrorMessage="The First Number Is Not In The Range"
  ControlToValidate="txtValidated"
  MaximumValue="10" MinimumValue="1" Type="Integer" />
```

You can even get a bit fancier by replacing the plain asterisk with a snippet of more interesting HTML. Here's an example that uses the <img> tag to add a small error icon image when validation fails:

```
<asp:RangeValidator id="RangeValidator" runat="server"
  Text="<img src='ErrorIcon.gif' />" alt='Error' ... />
<asp:ValidationSummary id="Errors" runat="server" />
```

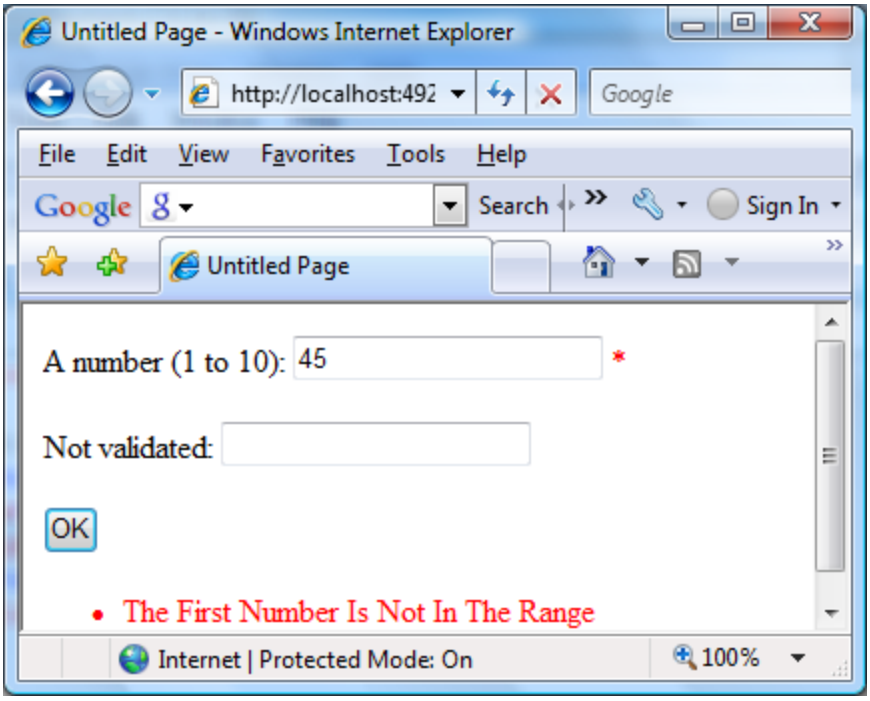


Figure 10.1-7 – A ValidationSummary with Error Indicator

10.1.3.2 Validation Summary with MessageBox

The **ValidationSummary** control provides some useful properties you can use to fine-tune the error display. You can set the **HeaderText** property to display a special title at the top of the list (such as Your page contains the following errors:). You can also change the **ForeColor** and choose a **DisplayMode**. The possible modes are **BulletList** (the default), **List**, and **SingleParagraph**.

Finally, you can choose to have the validation summary displayed in a pop-up dialog box instead of on the page (see Figure 10.1-8). This approach has the advantage of leaving the user interface of the page untouched, but it also forces the user to dismiss the error messages by closing the window before being able to modify the input controls. If users will need to refer to these messages while they fix the page, the inline display is better.

To show the summary in a dialog box, set the **ShowMessageBox** property of the **ValidationSummary** to true. Keep in mind that unless you set the **ShowSummary** property to **false**, you'll see both the message box and the In-page summary (as in Figure 10.1-8).

```
<asp:ValidationSummary id="Errors" runat="server"
                       ShowMessageBox="true" />
```

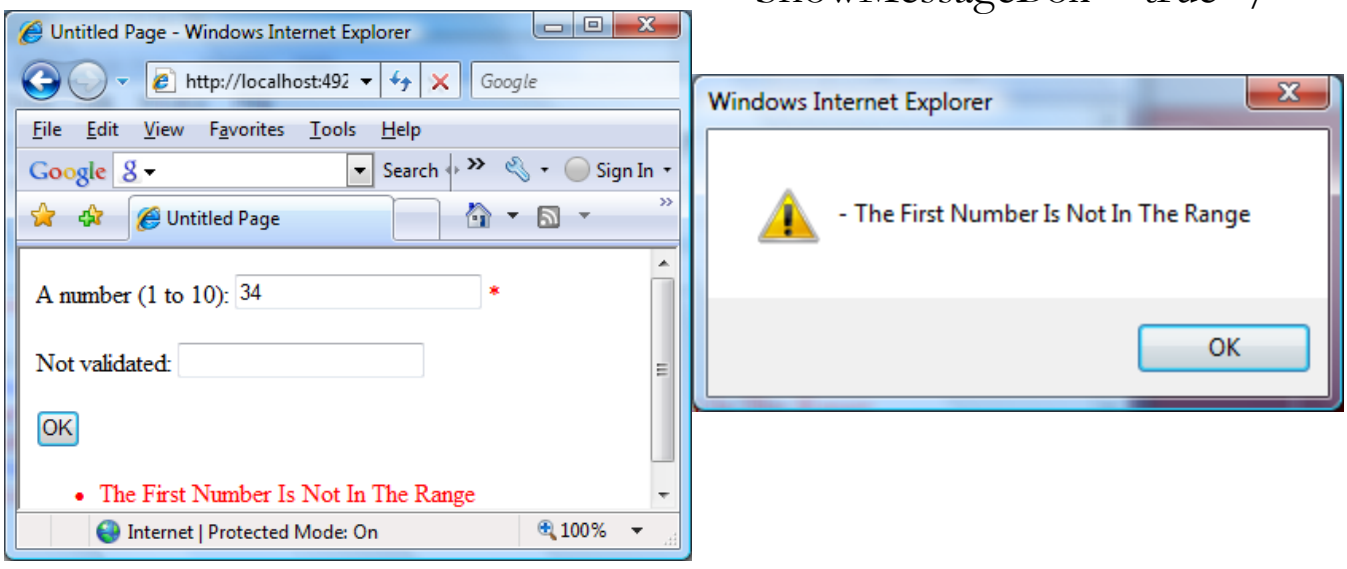


Figure 10.1-8 – A ValidationSummary with MessageBox

10.1.3.3 Manual Validation

Your final option is to disable validation and perform the work on your own, with the help of the validation controls. This allows you to take other information into consideration or create a specialized error message that involves other controls (such as images or buttons).

You can create manual validation in one of three ways:

- Use your own code to verify values. In this case, you won't use any of the ASP.NET validation controls.
- Disable the **EnableClientScript** property for each validation control. This allows an invalid page to be submitted, after which you can decide what to do with it depending on the problems that may exist.
- Add a button with **CausesValidation** set to false. When this button is clicked, manually validate the page by calling the **Page.Validate()** method.

Then examine the **IsValid** property, and decide what to do.
The next example uses the second approach. Once the page is submitted, it examines all the validation controls on the page by looping through the **Page.Validators** collection. Every time it finds a control that hasn't validated successfully, it retrieves the invalid value from the input control and adds it to a string. At the end of this routine, it displays a message that describes which values were incorrect, as shown in Figure 10.19.

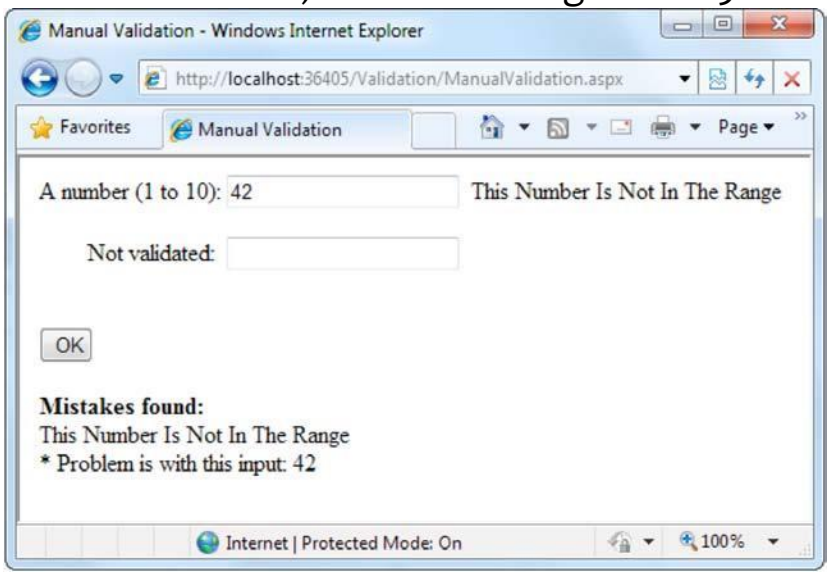


Figure 10.1-9 – A Manual Validation

➢10.0 Building Better Forms - Validation

This technique adds a feature that wouldn't be available with automatic validation, which uses the **ErrorMessage** property. In that case, it isn't possible to include the actual incorrect values in the message.

Here's the event handler that checks for invalid values:

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
   string errorMessage = "<b>Mistakes found:</b><br />";

   // Search through the validation controls.
   foreach (BaseValidator ctrl in this.Validators)
   {
      if (!ctrl.IsValid)
      {
         errorMessage += ctrl.ErrorMessage + "<br />";

         // Find the corresponding input control, and change the
         // generic Control variable into a TextBox variable.
         // This allows access to the Text property.
         TextBox ctrlInput =
          (TextBox)this.FindControl(ctrl.ControlToValidate);
         errorMessage += " * Problem is with this input: ";
         errorMessage += ctrlInput.Text + "<br />";
      }
   }
   lblMessage.Text = errorMessage;
}
```

Example 10.1-3 – Manual Validation Code Behind

This example uses an advanced technique: the Page.FindControl() method. It's required because the ControlToValidate property of each validator simply provides a string with the name of a control, not a reference to the actual control object. To find the control that matches this name (and retrieve its Text property), you need to use the FindControl() method. Once the code has retrieved the matching text box, it can perform other tasks such as clearing the current value, tweaking a property, or even changing the text box color. Note that the FindControl() method returns a generic Control reference, because you might search any type of control. To access all the properties of your control, you need to cast it to the appropriate type (such as TextBox in this example).

10.2 Validation with Regular Expressions

One of ASP.NET's most powerful validation controls is the RegularExpressionValidator, which validates text by determining whether or not it matches a specific pattern. For example, e-mail addresses, phone numbers, and file names are all examples of text that has specific constraints. A phone number must be a set number of digits, an e-mail address must include exactly one @ character (with text on either side), and a file name can't include certain special characters like \ and ?. One way to define patterns like these is with regular expressions.

Regular expressions have appeared in countless other languages and gained popularity as an extremely powerful way to work with strings. In fact, Visual Studio even allows programmers to perform a search-and-replace operation in their code using a regular expression . Regular expressions can almost be considered an entire language of their own. How to master all the ways you can use regular expressions-including pattern matching, back references, and named groups—could occupy an entire book (and several books are dedicated to just that subject). Fortunately, you can understand the basics of regular expressions without nearly that much work.

10.2.1Literals and Metacharacters

All regular expressions consist of two kinds of characters: literals and metacharacters. Literals are not unlike the string literals you type in code. They represent a specific defined character. For example, if you search for the string literal "l", you'll find the character l and nothing else.

Metacharacters provide the true secret to unlocking the full power of regular expressions. You're probably already familiar with two metacharacters from the DOS world (? and *). Consider the command-line expression shown here:

```
Del *.*
```

The expression *.* contains one literal (the period) and two metacharacters (the asterisks). This translates as "delete every file that starts with any number of characters and ends with an extension of any number of characters (or has no extension at all)." Because all files in DOS implicitly have extensions, this has the well-documented effect of “deleting Everything” in the current directory.

Another DOS metacharacter is the question mark, which means "any single character." For example, the following statement deletes any file named hello that has an extension of exactly one character.

```
Del hello.?
```

The regular expression language provides many flexible metacharacters – far more than the DOS command line. For example, \s represents any whitespace character (such as a space or tab). \d represents any digit. Thus, the following expression would match any string that started with the numbers 333, followed by a single whitespace character and any three numbers. Valid matches would include 333 333 and 333 945 but not 334 333 or 3334 945.

```
333\s\d\d\d
```

One aspect that can make regular expressions less readable is that they use special metacharacters that are more than one character long. In the previous example, \s represents a single character, as does \d, even though they both occupy two characters in the expression.

You can use the plus (+) sign to represent a repeated character. For example, 5+7 means "one or more occurrences of the character 5, followed by a single 7." The number 57 would match, as would 555557. You can also use parentheses to group a subexpression. For example, (52)+7 would match any string that started with a sequence of 52. Matches would include 527, 52527, 5252527, and so on.

You can also delimit a range of characters using square brackets. [a-f] would match any single character from a to f (lowercase only). The following expression would match any word that starts with a letter from a to f , contains one or more "word" characters (letters), and ends withing—possible matches. (Examples **acting** and **developing** etc.)

[a-f]\w+ing

The following is a more useful regular expression that can match any e-mail address by verifying that it contains the @ symbol. The dot is a metacharacter used to indicate any character except newline. However, some invalid e-mail addresses would still be allowed, including those that contain spaces and those that don't include a dot (.). You'll see a better example a little later in the customer form example.

.+@.+

10.2.2 Finding a Regular Expression

Clearly, picking the perfect regular expression may require some testing. In fact, numerous reference materials (on the Internet and in paper form) include useful regular expressions for validating common values such as postal codes. To experiment, you can use the simple **RegularExpressionTest** page included with the online samples, which is shown in Figure 10.2-1. It allows you to set a regular expression that will be used to validate a control. Then you can type in some sample values and see whether the regular expression validation succeeds or fails.

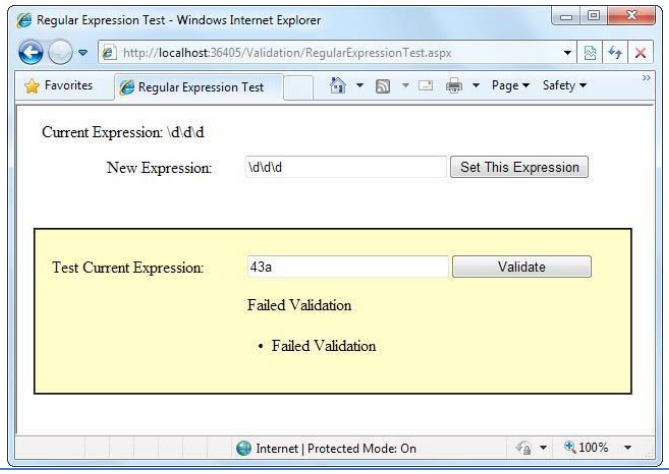


Figure 10.2-1 – A Regular Expression Test

The code is quite simple. The Set This Expression button assigns a new regular expression to the **RegularExpressionValidator** control (using whatever text you have typed). The Validate button simply triggers a **postback**, which causes ASP.NET to perform validation automatically. If an error message appears, validation has failed. Otherwise, it's successful.

```
public partial class RegularExpressionTest : System.Web.UI.Page
{
   protected void cmdSetExpression_Click(Object sender, EventArgs e) {
      TestValidator.ValidationExpression = txtExpression.Text;
      lblExpression.Text = "Current Expression: ";
      lblExpression.Text += txtExpression.Text;
   }
}
```

Example 10.2-1 – Regular Expression Test Code Behind

10.2.3 Regular Expression Building Blocks

Table 11-4 shows some of the fundamental regular expression building blocks. If you need to match a literal character with the same name as a special character, you generally precede it with a \ character. For example, *hello* matches *hello* in a string, because the special asterisk (*) character is preceded by a slash (\).

Table Regular Expression Characters

Character	Description
*	Zero or more occurrences of the previous character or subexpression. For example, 7*8 matches 7778 or just 8.
+	One or more occurrences of the previous character or subexpression. For example, 7+8 matches 7778 but not 8.
( )	Groups a subexpression that will be treated as a single element. For example, (78)+ matches 78 and 787878.
{m,n}	The previous character (or subexpression) can occur from m to n times. For example, A{1,3} matches A, AA, or AAA.
|	Either of two matches. For example, 8|6 matches 8 or 6.
[ ]	Matches one character in a range of valid characters. For example, [A-C] matches A, B, or C.
[^]	Matches a character that isn't in the given range. For example, [^A-B] matches any character except A and B.
.	Any character except newline. For example, .here matches where and there.
\s	Any whitespace character (such as a tab or space).
\S	Any nonwhitespace character.
\d	Any digit character.
\D	Any character that isn't a digit.
\w	Any "word" character (letter, number, or underscore).
\W	Any character that isn't a "word" character (letter, number, or underscore).

Figure 10.2-2 – Regular Expression Characters

10.2.4 Common Regular Expressions

Table in figure 10.2-3 shows a few common (and useful) regular expressions.

Table Commonly Used Regular Expressions		
Content	**Regular Expression**	**Description**
E-mail address	\S+@\S+\.\S+	Check for an at (@) sign and dot (.) and allow nonwhitespace characters only.
Password	\w+	Any sequence of one or more word characters (letter, space, or underscore).
Specific-length password	\w{4,10}	A password that must be at least four characters long but no longer than ten characters.
Advanced password	[a-zA-Z]\w{3,9}	As with the specific-length password, this regular expression will allow four to ten total characters. The twist is that the first character must fall in the range of a-z or A-Z (that is to say. it must start with a nonaccented ordinary letter).
Another advanced password	[a-zA-Z]\w[]\d+\w[]	This password starts with a letter character, followed by zero or more word characters, one or more digits, and then zero or more word characters. In short, it forces a password to contain one or more numbers somewhere inside it. You could use a similar pattern to require two numbers or any other special character.
Limited-length field	\S{4,10}	Like the password example, this allows four to ten characters, but it allows special characters (asterisks, ampersands, and so on).
U.S. Social Security number	\d{3}-\d{2}-\d{4}	A sequence of three, two, then four digits, with each group separated by a dash. You could use a similar pattern when requiring a phone number.

Figure 10.2-3 – Common Regular Expressions

You have many different ways to validate e-mail addresses with regular expressions of varying complexity. See **http://www.4guysfromrolla.com/webtech/validateemail.shtml** for a discussion of the subject and numerous examples.

Some logic is much more difficult to model in a regular expression. An example is the Luhn algorithm, which verifies credit card numbers by first doubling every second digit, then adding these doubled digits together, and finally dividing the sum by ten. The number is valid (although not necessarily connected to a real account) if there is no remainder after dividing the sum. To use the Luhn algorithm, you need a CustomValidator control that runs this logic on the supplied value. (You can find a detailed description of the Luhn algorithm at http://en.wikipedia.org/wiki/Luhn_formula.)

➢10.0 Building Better Forms - Validation

10.2.5 Using Regular Expressions in ASP.NET form.

Regular expressions can be added to most of the ASP components by using a **RegularExpressionValidator**. Following example shows how a regular expression can be added to a web control by using a RegularExpressionValidator. **In the example below, the text field concerned should accept only 1-40 alpha characters.**

```
<%@ language="C#" %>
<form id="form1" runat="server">
   <asp:TextBox ID="txtName" runat="server"/>
   <asp:Button ID="btnSubmit" runat="server" Text="Submit" />
   <asp:RegularExpressionValidator ID="regexpName" runat="server"
                  ErrorMessage="“Allowed only 1-40 alpha characters!"
                  ControlToValidate="txtName"
                  ValidationExpression="^[a-zA-Z'.\s]{1,40}$" />
</form>
```

Example 10.2-2 – Regular Expression usage

The regular expression used in the preceding code example constrains an input name field to alphabetic characters (lowercase and uppercase), space characters, the single quotation mark (or apostrophe) for names such as O'Dell, and the period or dot character. In addition, the field length is constrained to 40 characters.

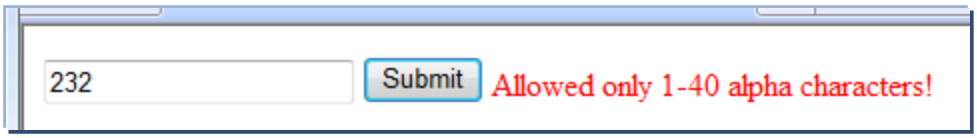


Figure 10.2-4– Regular Expressions in Action

Using ^ and $

Enclosing the expression in the caret (^) and dollar sign ($)markers ensures that the expression consists of the desired content and nothing else. A ^ matches the position at the beginning of the input string and a $ matches the position at the end of the input string. If you omit these markers, an attacker could affix malicious input to the beginning or end of valid content and bypass your filter.

10.2.6 Using Regex Class

If you are not using server controls (which means you cannot use the validation controls) or if you need to validate input from sources other than form fields, such as query string parameters or cookies, or even web controls that you want to check on server side, you can use the **Regex** class within the Sy**stem.Text.RegularExpressions** namespace.

Using the **Regex** class:
Add a `using` statement to reference the `System.Text.RegularExpressions` namespace. Call the `IsMatch` method of the `Regex` class, as shown in the following example.

```
<form id="form1" runat="server">
   <div>
   <asp:TextBox ID="txtName" runat="server"/>
   <asp:Button ID="btnSubmit" runat="server" Text="Submit"
                                                OnClick="CheckValidity"/>
   </div>
</form>
```

Example 10.2-3a – Using Regex class, front end

and on code behind, you validate it like this…

```
protected void CheckValidity(object sender, EventArgs e)
{
 if (!Regex.IsMatch(txtName.Text, @"^[a-zA-Z'.]{1,40}$"))
 {
   Response.Write("Expression Does not match");
 }
}
```

Example 10.2-3b – Using Regex class, code behind

➢10.0 Building Better Forms - Validation

A Validated Customer Form:

To bring together these various topics, you'll now see a full-fledged web form that combines a variety of pieces of information that might be needed to add a user record (for example, an e-commerce site shopper or a content site subscriber). Figure 10.2-5 shows this form.

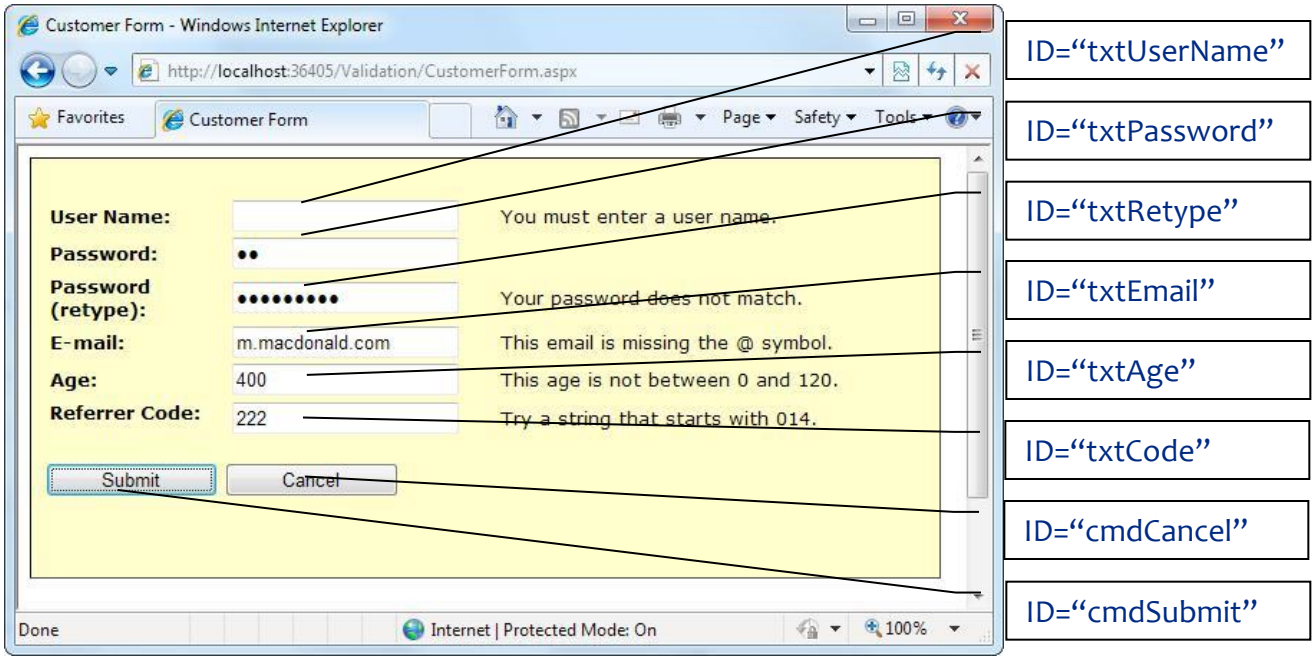


Figure 10.2-5– A form with different Regular Expressions

Several types of validation are taking place on the customer form:

- Three RequiredFieldValidator controls make sure the user enters a user name, a password, and a password confirmation.
- A CompareValidator ensures that the two versions of the masked password match.
- A RegularExpressionValidator checks that the e-mail address contains an at (@) symbol.
- A RangeValidator ensures the age is a number from 0 to 120.
- A CustomValidator performs a special validation on the server of a "referrer code." This code verifies that the first three characters make up a number that is divisible by 7.

➢10.0 Building Better Forms - Validation

The implementation of these Validation Controls are shown below.

```
<asp:RequiredFieldValidator id="vldUserName" runat="server"
  ErrorMessage="You must enter a user name."
  ControlToValidate="txtUserName" />

<asp:RequiredFieldValidator id="vldPassword" runat="server"
  ErrorMessage="You must enter a password."
  ControlToValidate="txtPassword" />

<asp:CompareValidator id="vldRetype" runat="server"
  ErrorMessage="Your password does not match."
  ControlToCompare="txtPassword" ControlToValidate="txtRetype”/>

<asp:RequiredFieldValidator id="vldRetypeRequired"
  runat="server‟ ErrorMessage="You must confirm your password."
  ControlToValidate="txtRetype" />

<asp:RegularExpressionValidator id="vldEmail" runat="server"
  ErrorMessage="This email is missing the @ symbol."
  ValidationExpression=".+@.+" ControlToValidate="txtEmail" />

<asp:RangeValidator id="vldAge" runat="server"
  ErrorMessage="This age is not between 0 and 120."
  Type="Integer‟ MinimumValue="0" MaximumValue="120"
  ControlToValidate="txtAge" />

<asp:CustomValidator id="vldCode" runat="server"
  ErrorMessage="Try a string that starts with 014."
  ValidateEmptyText="False”
  OnServerValidate="vldCode_ServerValidate"
  ControlToValidate="txtCode" />
```

The form provides two validation buttons—one that requires validation and one that allows the user to cancel the task gracefully:

```
<asp:Button id="cmdSubmit" runat="server"
  OnClick="cmdSubmit_Click" Text="Submit"></asp:Button>

<asp:Button id="cmdCancel" runat="server"
  CausesValidation="False" OnClick="cmdCancel_Click”
  Text="Cancel"> </asp:Button>
```

➢10.0 Building Better Forms - Validation

The buttons "Submit" and "Cancel" both handles events in the code behind. Here are the two event handlers.

```
protected void cmdSubmit_Click(Object sender, EventArgs e)
{
   if (Page.IsValid) {
      lblMessage.Text = "This is a valid form.";
   }
}

protected void cmdCancel_Click(Object sender, EventArgs e)
{
   lblMessage.Text = "No attempt was made to validate this form.";
}
```

Example 10.2-4 – Event Handlers for Submit and Cancel buttons

The only form-level code that is required for validation is the custom validation code. The validation takes place in the event handler for the **CustomValidator.ServerValidate** event. This method receives the value it needs to validate (**e.Value**) and sets the result of the validation to true or false (**e.IsValid**).

```
protected void vldCode_ServerValidate(Object source, ServerValidateEventArgs e)
{
   try {
      // Check whether the first three digits are divisible by seven.
      int val = Int32.Parse(e.Value.Substring(0, 3));
      if (val % 7 == 0)  { e.IsValid = true;   }
      else   { e.IsValid = false; }
   }
   catch (Exception exp) {
      // An error occurred in the conversion. The value is not valid.
      e.IsValid = false;
   }
}
```

Example 10.2-5 – ServerValidate event handler

➢10.0 Building Better Forms - Validation

This example also introduces one new detail: error handling. This error-handling code ensures that potential problems are caught and dealt with appropriately. Without error handling, your code may fail, leaving the user with nothing more than a cryptic error page. The reason this example requires error-handling code is because it performs two steps that aren't guaranteed to succeed. First, the Int32.Parse() method attempts to convert the data in the text box to an integer.

An error will occur during this step if the information in the text box is nonnumeric (for example, if the user entered the characters 4G). Similarly, the String.Substring() method, which extracts the first three characters, will fail if fewer than three characters appear in the text box. To guard against these problems, you can specifically check these details before you attempt to use the Parse() and Substring() methods, or you can use error handling to respond to problems after they occur. (Another option is to use the TryParse() method, which returns a Boolean value that tells you whether the conversion succeeded. You saw TryParse() at work in Chapter 5.)

The CustomValidator has another quirk. You'll notice that your custom server-side validation isn't performed until the page is posted back. This means that if you enable the client script code (the default), dynamic messages will appear informing the user when the other values are incorrect, but they will not indicate any problem with the referral code until the page is posted back to the server.

This isn't really a problem, but if it troubles you, you can use the CustomValidator.ClientValidationFunction property. Add a client-side JavaScript or VBScript validation function to the .aspx portion of the web page. (Ideally, it will be JavaScript for compatibility with browsers other than Internet Explorer.) Remember, you can't use client-side ASP.NET code, because C# and VB aren't recognized by the client browser.

Your JavaScript function will accept two parameters (in true .NET style), which identify the source of the event and the additional validation parameters. In fact, the client-side event is modeled on the .NET

ServerValidate event. Just as you did in the **ServerValidate** event handler, in the client validation function, you retrieve the value to validate from the Value property of the event argument object. You then set the **IsValid** property to indicate whether validation succeeds or fails.

The following is the client-side equivalent for the code in the **ServerValidate** event handler. The JavaScript code resembles C# superficially. Once you've added the validation script function, you must set the **ClientValidationFunction** property of the **CustomValidator** control to the name of the function. You can edit the **CustomValidator** tag by hand or use the Properties window in Visual Studio.

ASP.NET will now call this function on your behalf when it's required.

By default, custom validation isn't performed on empty values. However, you can change this behavior by setting the **CustomValidator.ValidateEmptyText** property to true. This is a useful approach if you create a more detailed JavaScript function (for example, one that updates with additional information) and want it to run when the text is cleared.

```
<script type="text/javascript">
<!--
function MyCustomValidation (objSource, objArgs) {
   // Get value.
   var number = objArgs.Value;

   // Check value and return result.
   number = number.substr(0, 3);
   if (number % 7 == 0)  { objArgs.IsValid = true;  }
   else  { objArgs.IsValid = false;  }
}
</script>
```

Example 10.2-6a
Custom Validator, javascript

```
<asp:CustomValidator id="vldCode" runat="server"
    ErrorMessage="Try a string that starts with 014."
    ControlToValidate="txtCode"
    OnServerValidate="vldCode_ServerValidate"
    ClientValidationFunction="MyCustomValidation"
/>
```

Example 10.2-6b
asp:CustomValidator Control

10.3 Validation Groups

In more complex pages, you might have several distinct groups of controls, possibly in separate panels. In these situations, you may want to perform validation separately. For example, you might create a form that includes a box with login controls and a box underneath it with the controls for registering a new user. Each box includes its own submit button, and depending on which button is clicked, you want to perform the validation just for that section of the page.

This scenario is possible thanks to a feature called validation groups. To create a validation group, you need to put the input controls, the validators, and the **CausesValidation** button controls into the same logical group. You do this by setting the **ValidationGroup** property of every control with the same descriptive string (such as "LoginGroup" or "NewUserGroup"). Every control that provides a **CausesValidation** property also includes the **ValidationGroup** property. For example, the following page defines two validation groups, named Group1 and Group2. The controls for each group are placed into separate Panel controls.

```
<form id="form1" runat="server">
 <asp:Panel ID="Panel1" runat="server">
  <asp:TextBox ID="TextBox1" ValidationGroup="Group1" runat="server" />

  <asp:RequiredFieldValidator ID="RequiredFieldValidator1“ ErrorMessage="*Required“
   ValidationGroup="Group1“  runat="server" ControlToValidate="TextBox1" />

  <asp:Button ID="Button1" Text="Validate Group1“  ValidationGroup="Group1" runat="server" />
 </asp:Panel>
 <br />
 <asp:Panel ID="Panel2" runat="server">
  <asp:TextBox ID="TextBox2" ValidationGroup="Group2”  runat="server" />

  <asp:RequiredFieldValidator ID="RequiredFieldValidator2“ ErrorMessage="*Required“
  ValidationGroup="Group2“ ControlToValidate="TextBox2" runat="server" />

  <asp:Button ID="Button2" Text="Validate Group2“  ValidationGroup="Group2" runat="server" />
 </asp:Panel>
</form>
```

If you click the button in the topmost Panel, only the first text box is validated. If you click the button in the second Panel, only the second text box is validated (as shown in Figure 10.3-1).

➢10.0 Building Better Forms - Validation

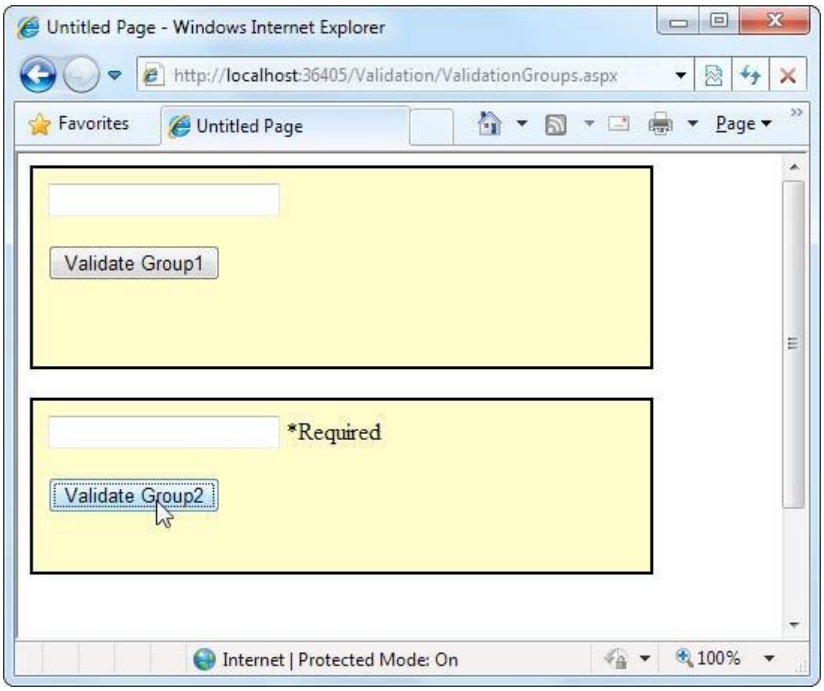


Figure 10.3-1– Validation Groups

What happens if you add a new button that doesn't specify any validation group? In this case, the button validates every control that isn't explicitly assigned to a named validation group. In the current example, no controls fit the requirement, so the page is posted back successfully and deemed to be valid.

If you want to make sure a control is always validated, regardless of the validation group of the button that's clicked, you'll need to create multiple validators for the control, one for each group (and one with no validation group).

11.0 Rich Controls (Text Book Chapter 10)

11.0 Rich Controls

Rich controls are web controls that model complex user interface elements. Although no strict definition exists for what is and what isn't a rich control, the term commonly describes a web control that has an object model that's distinctly separate from the HTML it generates. A typical rich control can be programmed as a single object (and added to a web page with a single control tag) but renders itself using a complex sequence of HTML elements. Rich controls can also react to user actions (like a mouse click on a specific region of the control) and raise more meaningful events that your code can respond to on the web server. In other words, rich controls give you a way to create advanced user interfaces in your web pages without writing lines of convoluted HTML.

In this chapter, you'll take a look at several web controls that have no direct equivalent in the world of ordinary HTML. You'll start with the Calendar, which provides slick date-selection functionality. Next, you'll consider the **AdRotator**, which gives you an easy way to insert a randomly selected image into a web page. Finally, you'll learn how to Create sophisticated pages with multiple views using two advanced container controls: the **MultiView** and the Wizard. These controls allow you to pack a miniature application into a single page. Using them, you can handle a multistep task without redirecting the user from one page to another.

NOTE:
ASP.NET includes numerous rich controls that are discussed elsewhere as needed in these class notes, including rich data controls, security controls, and controls tailored for web portals. In this chapter, you'll focus on a few useful web controls that don't fit neatly into any of these categories. All of these controls appear in the Standard tab of the Visual Studio Toolbox.

➢11.0 Rich Controls

11.1 Calendar Control

11.1.1 Default Calendar

The Calendar control presents a miniature calendar that you can place in any web page. Like most rich controls, the Calendar can be programmed as a single object (and defined in a single simple tag), but it renders itself with dozens of lines of HTML output.

```
<asp:Calendar id="MyCalendar" runat="server" />
```

The Calendar control presents a single-month view, as shown in Figure 12 The user can navigate from month to month using the navigational arrows, at which point the page is posted back and ASP.NET automatically provides a new page with the correct month values. You don't need to write any additional event-handling code to manage this process. When the user clicks a date, the date becomes highlighted in a gray box (by default). You can retrieve the selected day in your code as a **DateTime** object from the **Calendar.SelectedDate** property. This basic set of features may provide everything you need in your application. Alternatively, you can configure different selection modes to allow users to select entire weeks or months or to render the control as a static calendar that doesn't allow selection. The only fact you must remember is that if you allow month selection, the user can also select a single week or a day. Similarly, if you allow week selection, the user can also select a single day.

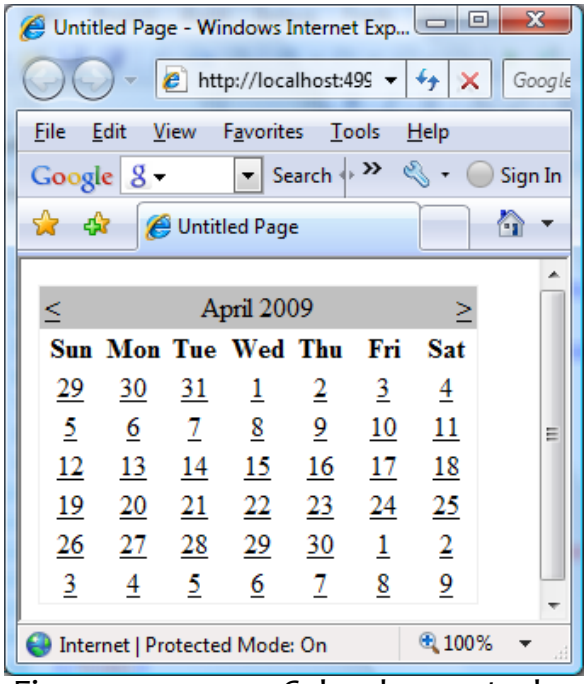


Figure 11.1-1– asp:Calendar control

You set the type of selection through the **Calendar.SelectionMode** property.

You may also need to set the **Calendar.FirstDayOfWeek** property to configure how a week is selected. (For example, set **FirstDayOfWeek** to the enumerated value Sunday, and weeks will be selected from Sunday to Saturday.)

11.0 Rich Controls

When you allow multiple date selection, you need to examine the **SelectedDates** property, which provides a collection of all the selected dates. You can loop through this collection using the `foreach` syntax. The following code demonstrates this technique:

You see a slight modification to the asp:Calendar where the SelectedMode property is now set to "DayWeekMonth". So you can select a day or a week or a whole month at a time.

```
<asp:Calendar id="MyCalendar" SelectionMode="DayWeekMonth" runat="server" />
<asp:Label id="lblDates"  runat="server" />
```

Example 11.1-1a Calendar Control with D/M/W Selectable

```
lblDates.Text = "You selected these dates:<br />";

foreach (DateTime dt in MyCalendar.SelectedDates)
{
  lblDates.Text += dt.ToLongDateString() + "<br />";
}
```

Example 11.1-1b Snippet of code behind

Here are some screen shots

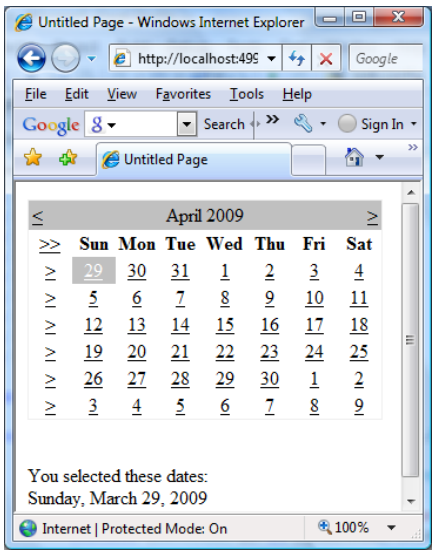


Figure 11.1-2a

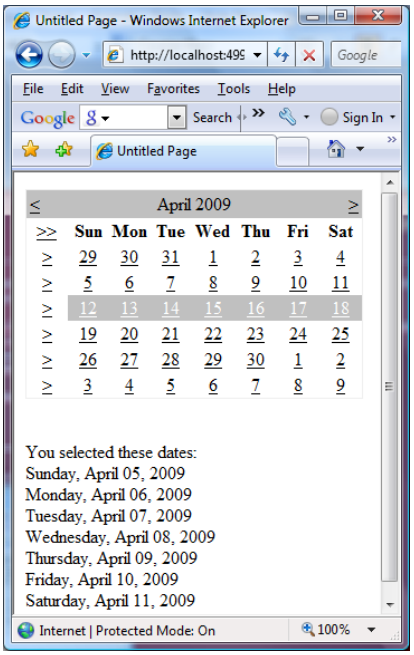


Figure 11.1-2b

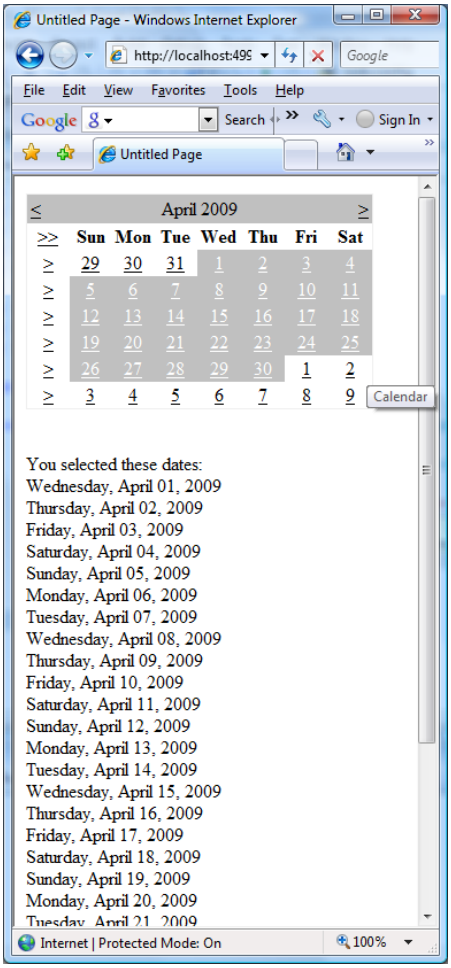


Figure 11.1-2c

11.1.2 Other Common Calendar Properties

The Calendar control provides a whole host of formatting-related properties. You can set various parts of the calendar, like the header, selector, and various day types, by using one of the style properties (for example, **WeekendDayStyle**).

Each of these style properties references a full-featured **TableItemStyle** object that provides properties for coloring, border style, font, and alignment. Taken together, they allow you to modify almost any part of the calendar's appearance. Table in figure 11.1-3 lists the style properties that the Calendar control provides.

Table Properties for Calendar Styles

Member	Description
DayHeaderStyle	The style for the section of the Calendar that displays the days of the week (as column headers).
DayStyle	The default style for the dates in the current month.
NextPrevStyle	The style for the navigation controls in the title section that move from month to month.
OtherMonthDayStyle	The style for the dates that aren't in the currently displayed month. These dates are used to "fill in" the calendar grid. For example, the first few cells in the topmost row may display the last few days from the previous month.
SelectedDayStyle	The style for the selected dates on the calendar.
SelectorStyle	The style for the week and month date-selection controls.
TitleStyle	The style for the title section.
TodayDayStyle	The style for the date designated as today (represented by the TodaysDate property of the Calendar control).
WeekendDayStyle	The style for dates that fall on the weekend.

Figure 11.1-3 – Common style properties in asp:Calendar

You can adjust each style using the Properties window. For a quick shortcut, you can set an entire related color scheme using the Calendar's Auto Format feature. To do so, start by selecting the Calendar on the design surface of a web form. Then, click the arrow icon that appears next to its top-right corner to show the Calendar's smart tag, and click the Auto Format link. You'll be presented with a list of predefined formats that set the style properties, as shown in Figure 11.1-4.

Using BackColor, ForeColor and TodayDayStyle properties:
Here is an example where some of the properties said above are customized in a asp:Calendar control.

```
<asp:Calendar id="MyCalendar" SelectionMode="DayWeekMonth" runat="server" DayHeaderStyle-
    BackColor="#FF3300" TodayDayStyle-ForeColor="#FF9933" WeekendDayStyle-
    ForeColor="#0033CC" /> <br /> <br />
```

Example 11.1-2 Using Common Style properties on a asp:Calendar

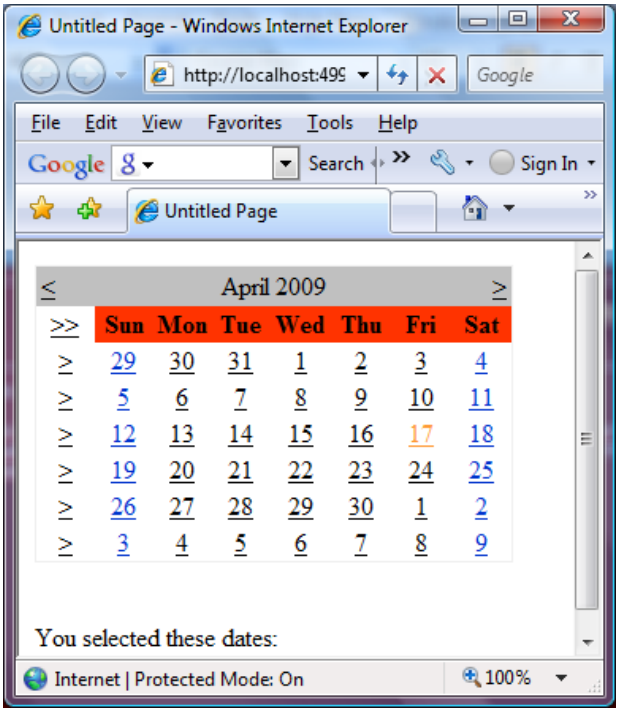


Figure 11.1-4Using Common Style properties on a asp:Calendar

You can also use additional properties to hide some elements or conFigure the text they display. For example, properties that start with "Show" (such as ShowDayHeader, ShowTitle, and ShowGridLines) can be used to hide or show a specific visual element. Properties that end in "Text" (such as PrevMonthText, NextMonthText, and SelectWeekText) allow you to set the text that's shown in part of the calendar.

Restricting Access to Days:

In most situations where you need to use a calendar for selection, you don't want to allow the user to select any date in the calendar. For example, the user might be booking an appointment or choosing a delivery date-two services that are generally provided only on set days. The Calendar control makes it surprisingly easy to implement this logic. In fact, if you've worked with the date and time controls on the Windows platform, you'll quickly recognize that the ASP.NET versions are far superior.

The basic approach to restricting dates is to write an event handler for the **Calendar.DayRender** event. This event occurs when the Calendar control is about to create a month to display to the user. This event gives you the chance to examine the date that is being added to the current month (through the **e.Day** property) and decide whether it should be selectable or restricted.

The following code makes it impossible to select any weekend days or days in years greater than 2013:

```
protected void MyCalendar_DayRender(Object source, DayRenderEventArgs e)
{
   // Restrict dates after the year 2010 and those on the weekend.
   if (e.Day.IsWeekend || e.Day.Date.Year > 2010)
   {
      e.Day.IsSelectable = false;
   }
}
```

Example 11.1-3 Restricting access to days

The **e.Day** object is an instance of the **CalendarDay class**. CalendarDay class provides various properties. Table in figure 11.1-5 describes some of the most useful.

Table CalendarDay Properties	
Property	**Description**
Date	The DateTime object that represents this date.
IsWeekend	True if this date falls on a Saturday or Sunday.
IsToday	True if this value matches the Calendar.TodaysDate property, which is set to the current day by default.
IsOtherMonth	True if this date doesn't belong to the current month but is displayed to fill in the first or last row. For example, this might be the last day of the previous month or the next day of the following month.
IsSelectable	Allows you to conFigure whether the user can select this day.

Figure 11.1-5 CalenderDay class properties

Day**Render Event:**

The **DayRender** event is extremely powerful. Besides allowing you to tailor what dates are selectable, it also allows you to configure the cell where the date is located through the **e.Cell property**. (The calendar is displayed using an HTML table.) For example, you could highlight an important date or even add information. Here's an example that highlights a single day—the fifth of May—by adding a new Label control in the table cell for that day:

```
protected void MyCalendar_DayRender(Object source, DayRenderEventArgs e)
{
   // Check for May 5 in any year, and format it.
   if (e.Day.Date.Day == 5 && e.Day.Date.Month == 5)
   {
      e.Cell.BackColor = System.Drawing.Color.Yellow;

   // Add some static text to the cell.
      Label lbl = new Label();
      lbl.Text = "<br />My Birthday!";
      e.Cell.Controls.Add(lbl);
   }
}
```

Example 11.1-4 Using CalendarDay properties

11.0 Rich Controls

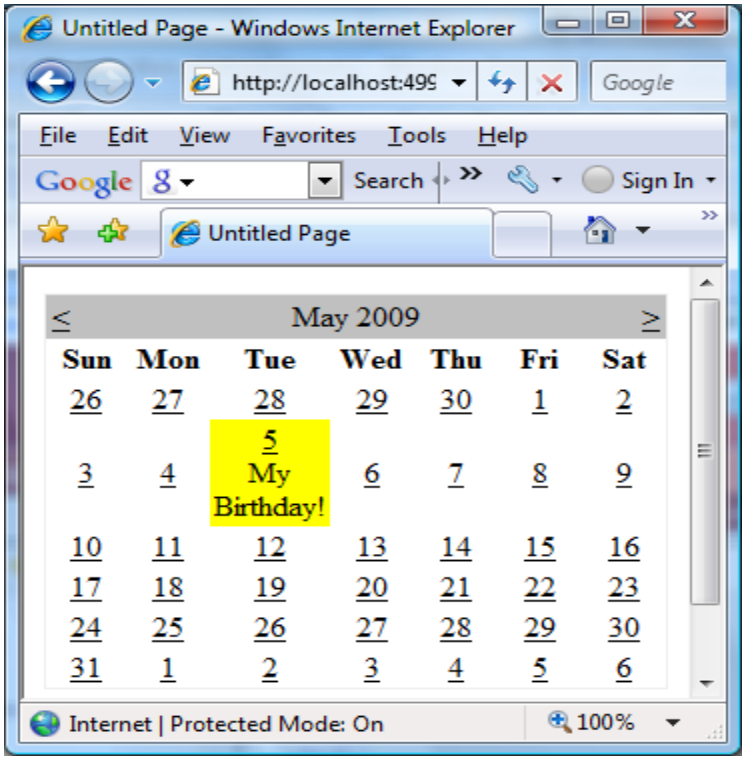


Figure 11.1-6 DayRender

The Calendar control provides two other useful events: SelectionChanged and VisibleMonthChanged. These occur immediately after the user selects a new day or browses to a new month (using the next month and previous month links). You can react to these events and update other portions of the web page to correspond to the current calendar month. For example, you could design a page that lets you schedule a meeting in two steps. First, you choose the appropriate day. Then, you choose one of the available times on that day.

The following code demonstrates an example how SelectionChanged event is trapped and customize the calendar.

```
<form id="form1" runat="server">
 <div>
  <asp:Calendar ID="MyCalendar" runat="server"
      SelectionMode="DayWeekMonth"
     ShowGridLines="True"
      OnSelectionChanged=
                "MyCalendar_SelectionChanged">

       <SelectedDayStyle BackColor="Yellow"
                  ForeColor="Red">
       </SelectedDayStyle>

     </asp:Calendar>

  <asp:Label id="lblDates"  runat="server" />
 </div>
</form>
```

Example 11.1-5a Handling SectionChanged event - Presentation

```
protected void MyCalendar_SelectionChanged
                    (Object source, EventArgs e)
{
 //Clear the current text.
  lblDates.Text = "";

 //Iterate through the SelectedDates collection
 //and display the dates selected in the Calendar
  //control.
 foreach(DateTime day in
     MyCalendar.SelectedDates)
 {
  lblDates.Text +=
    day.Date.ToShortDateString() + "<br />";
 }
}
```

Example 11.1-5b Handling SectionChanged event code behind

Output of example 11.1-5a and 11.1-5b is show in Figure 11.1-7.

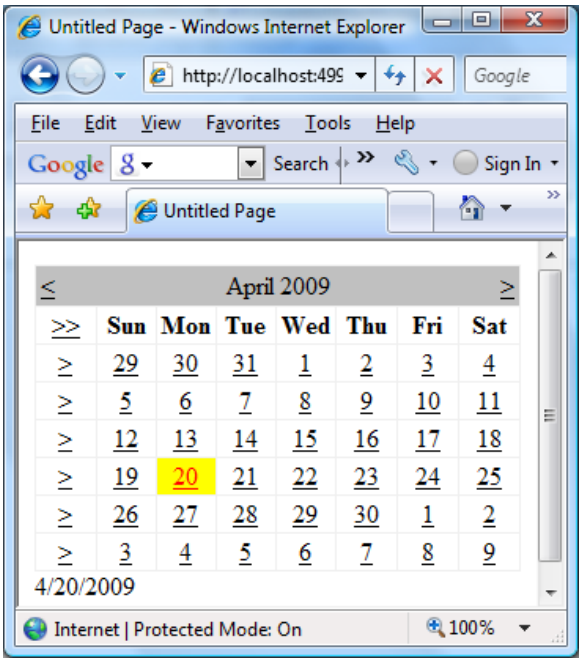


Figure 11.1-7

See other Calendar Class Members

11.2 AdRotator Control

The basic purpose of the AdRotator is to provide a graphic on a page that is chosen randomly from a group of possible images. In other words, every time the page is requested, an image is selected at random and displayed, which is the "rotation" indicated by the name AdRotator. One use of the AdRotator is to show banner-style advertisements on a page, but you can use it any time you want to vary an image randomly.

Using ASP.NET, it wouldn't be too difficult to implement an AdRotator type of design on your own. You could react to the Page.Load event, generate a random number, and then use that number to choose from a list of predetermined image files. You could even store the list in the web.config file so that it can be easily modified separately as part of the application's configuration. Of course, if you wanted to enable several pages with a random image, you would either have to repeat the code or create your own custom control. The AdRotator provides these features for free. The Advertisement File

The AdRotator stores its list of image files in an XML file. This file uses the format shown here:

11.2.1 The Advertisement File

The AdRotator stores its list of image files in an XML file. This file uses the format shown here:

```
<Advertisements>
<Ad>
<ImageUrl>prosetech.jpg</ImageUrl>
<NavigateUrl>http://www.prosetech.com</NavigateUrl>
<AlternateText>ProseTech Site</AlternateText>
<Impressions>1</Impressions>
<Keyword>Computer</Keyword>
</Ad>
</Advertisements>
```

➢11.0 Rich Controls

This example shows a single possible advertisement, which the **AdRotator** control picks at random from the list of advertisements. To add more advertisements, you would create multiple <Ad> elements and place them all inside the root <Advertisements> element:

<Advertisements>
<Ad>
<!-- First ad here. -->
</Ad>
<Ad>
<!-- Second ad here. -->
</Ad>
</Advertisements>

Each <Ad> element has a number of other important properties that configure the link, the image and the frequency, as described in the table in figure 11.2-1

Element	Description
ImageUrl	The image that will be displayed. This can be a relative link (a file in the current directory) or a fully qualified Internet URL.
NavigateUrl	The link that will be followed if the user clicks the banner. This can be a relative or fully qualified URL.
AlternateText	The text that will be displayed instead of the picture if it cannot be displayed. This text will also be used as a tooltip in some newer browsers.
Impressions	A number that sets how often an advertisement will appear. This number isrelative to the numbers specified for other ads. For example, a banner with the value 10 will be shown twice as often (on average) as the banner with the value 5.
Keyword	A keyword that identifies a group of advertisements. You can use this for filtering.For example, you could create ten advertisements and give half of choose to filter the possible advertisements to include only one of these groups.choose to filter the possible advertisements to include only one of these groups.them the keyword Retail and the other half the keyword Computer. The web page can then choose to filter the possible advertisements to include only one of these groups.

Figure 11.2-1 - AdRotator Properties

11.2-2 The AdRotator Class

The actual AdRotator class provides a limited set of properties. You specify both the appropriate advertisement file in the AdvertisementFile property and the type of window that the link should follow (the Target window). The target can name a specific frame, or it can use one of the values defined in table in figure 11.2-2.

Target	Description
_blank	The link opens a new unframed window.
_parent	The link opens in the parent of the current frame.
_self	The link opens in the current frame.
_top	The link opens in the topmost frame of the current window (so the link appears in the full window).

Figure 11.2-1 - AdRotator Target Window Types

Optionally, you can set the KeywordFilter property so that the banner will be chosen from a specific keyword group. This is a fully configured

AdRotator tag:

```
<asp:AdRotator id="Ads" runat="server"
    AdvertisementFile="MainAds.xml"
Target="_blank" KeywordFilter="Computer" />
```

some related content or a link, as shown in Figure 11.2-3

Additionally, you can react to the AdRotator.AdCreated event. This occurs when the page is being created and an image is randomly chosen from the advertisements file. This event provides you with information about the image that you can use to customize the rest of your page. For example, you might display some related content or a link, as shown in Figure 11.2-3

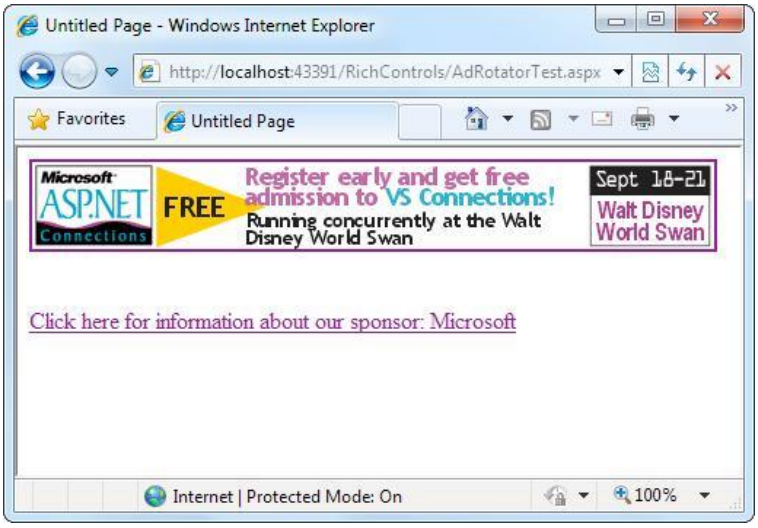


Figure 11.2-3 - AdRotator An AdRotator with synchronized content

The event handling code for this example simply configures a HyperLink control named lnkBanner based on the randomly selected advertisement:

```
protected void Ads_AdCreated(Object sender, AdCreatedEventArgs e)
{
//Synchronize the Hyperlink control.
  lnkBanner.NavigateUrl = e.NavigateUrl;
//Syncrhonize the text of the link.
  lnkBanner.Text = "Click here for information about our sponsor: ";
  lnkBanner.Text += e.AlternateText;
}
```

As you can see, rich controls such as the Calendar and AdRotator don't just add a sophisticated HTML output; they also include an event framework that allows you to take charge of the control's behavior and integrate it into your application.

11.3 Multiview Control

The MultiView control represents a control that acts as a container for groups of View controls. It allows you to define a group of View controls, where each View control contains child controls, for example, in an online survey application.

Your application can then render a specific View control to the client based on criteria such as user identity, user preferences, or information passed in a query string parameter. The following figure illustrates a MultiView control hosting 3 View controls.

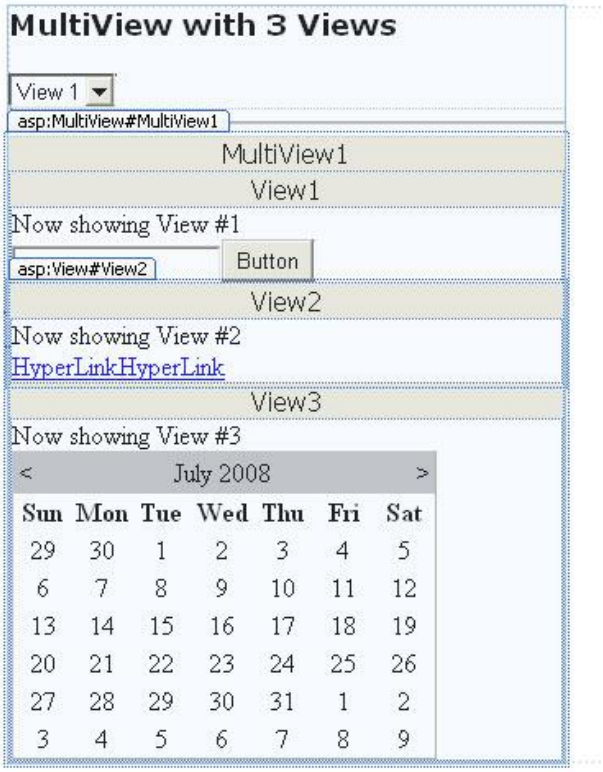


Figure 11.3-1 - 3 View MultiView control

The source code for the above example is shown in next page.

11.0 Rich Controls

```
<%@ Page Language="C#" AutoEventWireup="true"
 CodeFile="how-to-use-MultiView-c.aspx.cs" Inherits="how_to_use_MultiView_c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>How to Use MultiView</title>
</head>
<body>
   <form id="form1" runat="server">
   <div style="width: 339px">
     <h3>
          <font face="Verdana">MultiView with 3 Views</font>
        </h3>
        <asp:DropDownList ID="DropDownList1“ OnSelectedIndexChanged="DropDownList1_SelectedIndexChanged"
          runat="server" AutoPostBack="True">
          <asp:ListItem Value="0">View 1</asp:ListItem>
          <asp:ListItem Value="1">View 2</asp:ListItem>
          <asp:ListItem Value="2">View 3</asp:ListItem>
        </asp:DropDownList><br />
        <hr />
        <asp:MultiView ID="MultiView1" runat="server" ActiveViewIndex="0">
          <asp:View ID="View1" runat="server">
            Now showing View #1<br />
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><strong> </strong>
            <asp:Button ID="Button1" runat="server" Text="Button" /></asp:View>
          <asp:View ID="View2" runat="server">
            Now showing View #2<br />
            <asp:HyperLink ForeColor="#FF9933" ID="HyperLink1" runat="server"
              NavigateUrl="http://www.asp.net">HyperLink</asp:HyperLink>
           <asp:HyperLink ForeColor="#FF9933" ID="HyperLink2" runat="server“
                    NavigateUrl="http://www.asp.net">HyperLink</asp:HyperLink></asp:View>
          <asp:View ID="View3" runat="server">
            Now showing View #3<br />
            <asp:Calendar ID="Calendar1" runat="server"></asp:Calendar>
          </asp:View>
        </asp:MultiView>
   </div>
   </form>
</body>
</html>
```

Fortunately, the MultiView includes some built-in smarts that can save you a lot of trouble. Here's how it works: the MultiView recognizes button controls with specific command names. (Technically, a button control is any control that implements the IButtonControl interface, including the Button, ImageButton, and LinkButton.) If you add a button control to the view that uses one of these recognized command names, the button gets some automatic functionality. Using this technique, you can create navigation buttons without writing any code.

Table in figure 11.3-2 lists all the recognized command names. Each command name also has a corresponding static field in the MultiView class, so you can easily get the right command name if you choose to set it programmatically.

Table Recognized Command Names for the MultiView		
Command Name	MultiView Field	Description
PrevView	PreviousViewCommandName	Moves to the previous view.
NextView	NextViewCommandName	Moves to the next view.
SwitchViewByID	SwitchViewByIDCommandName	Moves to the view with a specific ID (string name). The ID is taken from the CommandArgument property of the button control.
SwitchViewByIndex	SwitchViewByIndexCommandName	Moves to the view with a specific numeric index. The index is taken from the CommandArgument property of the button control.

Figure 11.3-2 - MultiView Command Names

11.4 GridView Control

A recurring task in software development is to display tabular data. ASP.NET provides a number of tools for showing tabular data in a grid, including the GridView control. With the GridView control, you can display, edit, and delete data from many different kinds of data sources, including databases, XML files, and business objects that expose data.

You can use the GridView control to do the following:

- Automatically bind to and display data from a data source control.
- Select, sort, page through, edit, and delete data from a data source control.

Additionally, you can customize the appearance and behavior of the GridView control by doing the following:

- Specifying custom columns and styles.
- Utilizing templates to create custom user interface (UI) elements.
- Adding your own code to the functionality of the GridView control by handling events.

11.4-2 Data Binding with the GridView Control

The GridView control provides you with two options for binding to data:

Data binding using the DataSourceID property, which allows you to bind the ridView control to a data source control. This is the recommended approach because it allows the GridView control to take advantage of the capabilities of the data source control and provide built-in functionality for sorting, paging, and updating.

Data binding using the DataSource property, which allows you to bind to various objects, including ADO.NET datasets and data readers. This approach requires you to write code for any additional functionality such as sorting, paging, and updating.

11.0 Rich Controls

When you bind to a data source using the DataSourceID property, the GridView control supports two-way data binding. In addition to the control displaying returned data, you can enable the control to automatically support update and delete operations on the bound data. The following example demonstrates how a data source could be bounded to a GridvView.

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>GridView example: how to use GridView in asp.net</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:GridView ID="GridView1" runat="server" AllowPaging="True"
      AllowSorting="True" AutoGenerateColumns="False" DataKeyNames="CustomerID"
      DataSourceID="SqlDataSource1">
      <Columns>
        <asp:BoundField DataField="CustomerID" HeaderText="CustomerID" ReadOnly="True"
          SortExpression="CustomerID" />
        <asp:BoundField DataField="CompanyName" HeaderText="CompanyName"
          SortExpression="CompanyName" />
        <asp:BoundField DataField="ContactName" HeaderText="ContactName"
          SortExpression="ContactName" />
        <asp:BoundField DataField="ContactTitle" HeaderText="ContactTitle"
          SortExpression="ContactTitle" />
        <asp:BoundField DataField="Address" HeaderText="Address"
          SortExpression="Address" />
        <asp:BoundField DataField="City" HeaderText="City" SortExpression="City" />
        <asp:BoundField DataField="Region" HeaderText="Region"
          SortExpression="Region" />
        <asp:BoundField DataField="PostalCode" HeaderText="PostalCode"
          SortExpression="PostalCode" />
        <asp:BoundField DataField="Country" HeaderText="Country"
          SortExpression="Country" />
        <asp:BoundField DataField="Phone" HeaderText="Phone" SortExpression="Phone" />
        <asp:BoundField DataField="Fax" HeaderText="Fax" SortExpression="Fax" />
      </Columns>
    </asp:GridView>
    <asp:SqlDataSource ID="SqlDataSource1" runat="server"
      ConnectionString="<%$ ConnectionStrings:AppConnectionString1 %>"
      SelectCommand="SELECT * FROM [Customers]"></asp:SqlDataSource>
  </div>
  </form>
</body>
</html>
```

Example 11.4-1 Binding a Data Source to a GridView Control

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
TORTU	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
TRADH	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
TRAIH	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
VAFFE	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
VICTE	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
VINET	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
WANDK	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
WARTH	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
WELLI	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	
WHITC	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Region	05023	Mexico	(5) 555-3932	

1 2 3 4 5 6 7 8 9 10

Figure 11.4-1 - GridView output from a Database Table

We will discuss more about data binding with GridView in a later chapter.

12.0 User Controls & Graphics (Text Book Chapter 11)

12.0 User Controls & Graphics

12..1 Introduction

In this chapter, you'll consider two ways to extend your web pages another notch. First, you'll tackle user controls, which give you an efficient way to reuse a block of user interface markup—and the code that goes with it. User controls are a key tool for building modular web applications. They can also help you create consistent website designs and reuse your hard work.

Next, you'll explore custom drawing with GDI+. You'll see how you can paint exactly the image you need on request. You'll also learn the best way to incorporate these images into your web pages.

12.2 User Controls

A well-built web application divides its work into discrete, independent blocks. The more modular your web application is, the easier it is to maintain your code, troubleshoot problems, and reuse key bits of functionality.

Although it's easy enough to reuse code (you simply need to pull it out of your pages and put it into separate classes), it's not as straightforward to reuse web page markup. You can cut and paste blocks of HTML and ASP.NET control tags, but this causes endless headaches if you want to change your markup later. Instead, you need a way to wrap up web page markup in a reusable package, just as you can wrap up ordinary C# code. The trick is to create a user control. User controls look pretty much the same as ASP.NET web forms. Like web forms, they are composed of a markup portion with HTML and control tags (the .ascx file) and can optionally use a code-behind file with event-handling logic. They can also include the same range of HTML content and ASP.NET controls, and they experience the same events as the Page object (such as Load and PreRender). The only differences between user controls and web pages are as follows:

12.0 User Controls & Graphics

- User controls use the file extension .ascx instead of .aspx, and their code-behind files inherit from the System.Web.UI.UserControl class. In fact, the UserControl class and the Page class both inherit from the same base classes, which is why they share so many of the same methods and events, as shown in the inheritance diagram in Figure 12.1-1.
- The .ascx file for a user control begins with a **<%@ Control %>** directive instead of a **<%@ Page %>** directive.
- User controls can't be requested directly by a web browser. Instead, they must be embedded inside other web pages

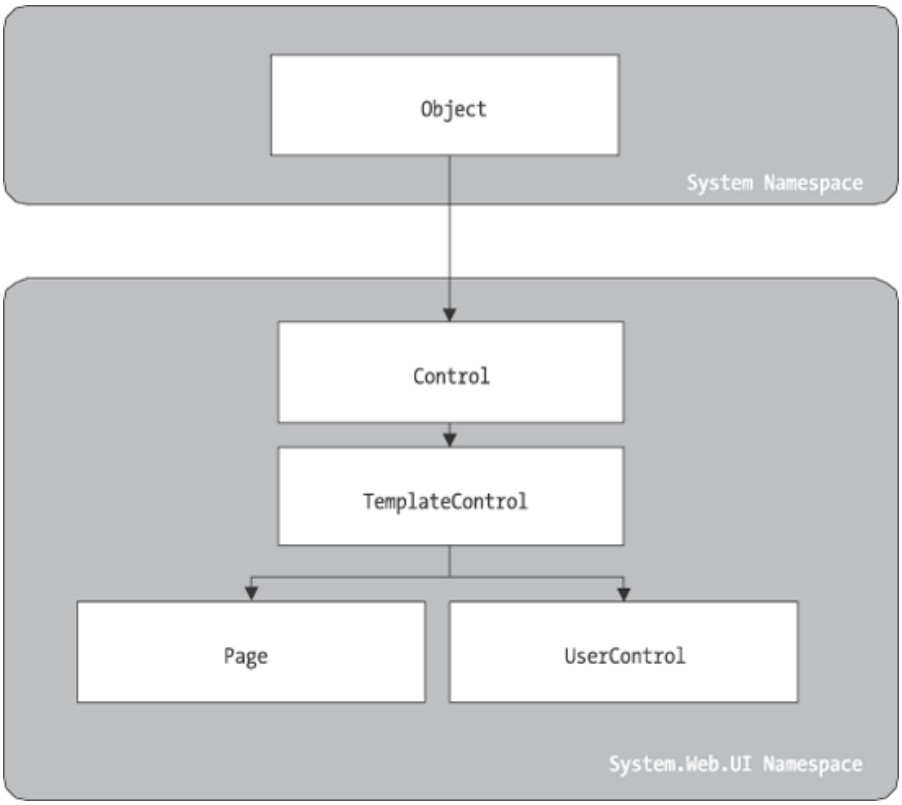


Figure 12.1-1 - User Controls Inheritance Diagram

12..3 Creating a Simple User Control

You can create a user control in Visual Studio in much the same way you add a web page. Just select Website Add New Item, and choose Web User Control from the list.

The following user control contains a single Label control:

```
<%@ Control Language="C#" AutoEventWireup="true"
   CodeFile="Footer.ascx.cs" Inherits="Footer" %>

<asp:Label id="lblFooter" runat="server" />
```

Example 12.3-1a User Control Presentation

```
public partial class Footer : System.Web.UI.UserControl
   {
      protected void Page_Load(object sender, EventArgs e)
      {
         lblFooter.Text = "This page was served at ";
         lblFooter.Text += DateTime.Now.ToString();
      }
   }
```

Example 12.3-1b User Control Code Behind

Note that the Control directive uses the same attributes used in the Page directive for a web page, including Language, AutoEventWireup, and Inherits. The code-behind class for this sample user control is similarly straightforward. It uses the UserControl.Load event to add some text to the label:

To test this user control, you need to insert it into a web page. This is a two-step process. First, you need to add a Register directive to the page that will contain the user control. You place the Register directive immediately after the **Page directive**. The Register directive identifies the control you want to use and associates it with a unique control prefix, as shown here:

<%@ Register TagPrefix="apress" TagName="Footer" Src="Footer.ascx" %>

The Register directive specifies a tag prefix and name. Tag prefixes group sets of related controls (for example, all ASP.NET web controls use the tag prefix asp). Tag prefixes are usually lowercase—technically, they are case-insensitive—and should be unique for your company or organization. The Src directive identifies the location of the user control template file, not the code-behind file.

Second, you can now add the user control whenever you want (and as many times as you want) in the page by inserting its control tag. Consider this page example:

```
<%@ Page Language="C#" AutoEventWireup="true"
   CodeFile="FooterHost.aspx.cs" Inherits="FooterHost"%>
<%@ Register TagPrefix="apress" TagName="Footer" Src="Footer.ascx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Footer Host</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
     <h1>A Page With a Footer</h1><hr />
     Static Page Text<br /><br />
     <apress:Footer id="Footer1" runat="server" />
   </div>
   </form>
</body>
</html>
```

Example 12.3-1c Using the User Control

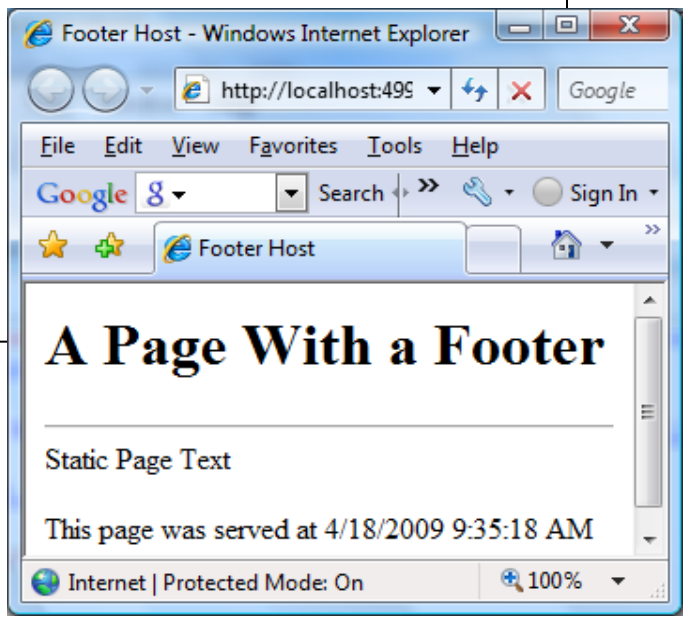


Figure 12.3-1 User Control In Action

This example (shown in Figure 12.3-1) demonstrates a simple way that you can create a header or footer and reuse it in all the pages in your website just by adding a user control. In the case of your simple footer, you won't save much code. However, this approach will become much more useful for a complex control with extensive formatting or several contained controls.

Of course, this only scratches the surface of what you can do with a user control. In the following sections, you'll learn how to enhance a control with properties, methods, and events—transforming it from a simple "include file" into a full-fledged object.

NOTE:
The Page class provides a special LoadControl() method that allows you to create a user control dynamically at runtime from an .ascx file. The user control is returned to you as a control object, which you can then add to the Controls collection of a container control on the web page (such as Place Holder or Panel) to display it on the page. This technique isn't a good substitute for declaratively using a user control, because it's more complex. However, it does have some interesting applications if you want to generate a user interface dynamically.

In Visual Studio, you have a useful shortcut for adding a user control to a page without typing the Register directive by hand. Start by opening the web page you want to use. Then, find the .ascx file for the user control in the Solution Explorer. Drag it from the Solution Explorer and drop it onto the visual design area of your web form (not the source view area). Visual Studio will automatically add the Register directive for the user control, as well as an instance of the user control tag.

12.4 Independent User Controls

Conceptually, two types of user controls exist: independent and integrated. Independent user controls don't interact with the rest of the code on your form. The Footer user control is one such example. Another example might be a LinkMenu control that contains a list of buttons offering links to other pages. This LinkMenu user control can handle the events for all the buttons and then run the appropriate Response.Redirect() code to move to another web page. Or it can just be an ordinary HyperLink control that doesn't have any associated server-side code. Every page in the website can then include the same LinkMenu user control, enabling painless website navigation with no need to worry about frames.

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeBehind="LinkMenu.ascx.cs" Inherits="TestingUserControls.LinkMenu" %>
<div>
 Products:<br />
 <asp:HyperLink id="lnkBooks" runat="server"
  NavigateUrl="TestUserControl.aspx?product=Books">Books
 </asp:HyperLink><br />
 <asp:HyperLink id="lnkToys" runat="server"
  NavigateUrl="TestUserControl.aspx?product=Toys">Toys
 </asp:HyperLink><br />
 <asp:HyperLink id="lnkSports" runat="server"
  NavigateUrl="TestUserControl.aspx?product=Sports">Sports
 </asp:HyperLink><br />
 <asp:HyperLink id="lnkFurniture" runat="server"
  NavigateUrl="TestUserControl.aspx?product=Furniture">Furniture
 </asp:HyperLink>
</div>
```

Example 12.4-1a An independent User Control

NOTE: You can use the more feature-rich navigation controls to provide Website navigation. Creating your own custom controls gives you a simple, more flexible, but less powerful approach to providing navigation. You might use custom controls rather than a whole site map for straightforward navigation between a few pages.

The following sample defines a simple control that presents an attractively formatted list of links. Note that the style attribute of the tag (which defines fonts and formatting) has been omitted for clarity.

The links don't actually trigger any server-side code—instead, they render themselves as ordinary HTML anchor tags with a hard-coded URL.

To test this menu, you can use the following TestingUserControls.aspx web page. It includes two controls: the Menu control and a Label control that displays the product query string parameter. Both are positioned using a table.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="TestUserControl.aspx.cs"
    Inherits="TestingUserControls._Default" %>
<%@ Register TagPrefix="uhclKP" TagName="LinkMenu" Src="LinkMenu.ascx" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
  <title>Menu Host</title>
</head>
<body>
  <form id="form1" runat="server">
<div>
    <table>
      <tr>
        <td><uhclKP:LinkMenu id="Menu1" runat="server" /></td>
        <td><asp:Label id="lblSelection" runat="server" /></td>
      </tr>
    </table>
  </div>
  </form>
</body>
</html>
```

Example 12.4-1b Using independent User Control

When the TestingUserControls.aspx page loads, it adds the appropriate information to the lblSelection control:

```
protected void Page_Load(Object sender, EventArgs e)
{
  if (Request.Params["product"] != null)
  {
    lblSelection.Text = "You chose: ";
    lblSelection.Text += Request.Params["product"];
  }
}
```

Example 12.4-1c Using Code behind for the independent User Control

Following figure 12.4-1 shows the end result. Whenever you click a button, the page is posted back, and the text is updated.

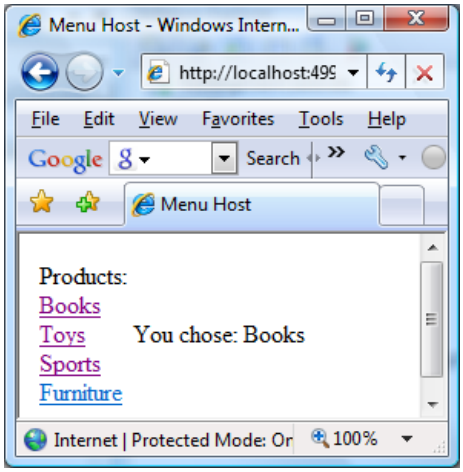


Figure 12.4-1a

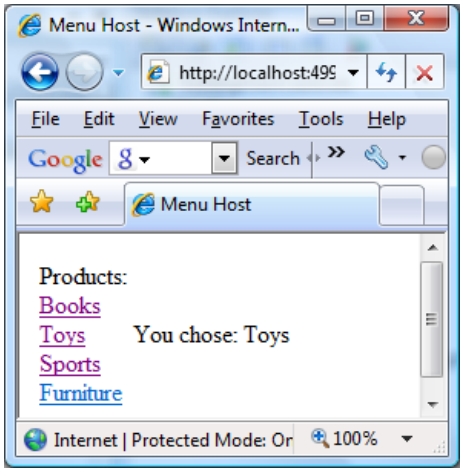


Figure 12.4-1b

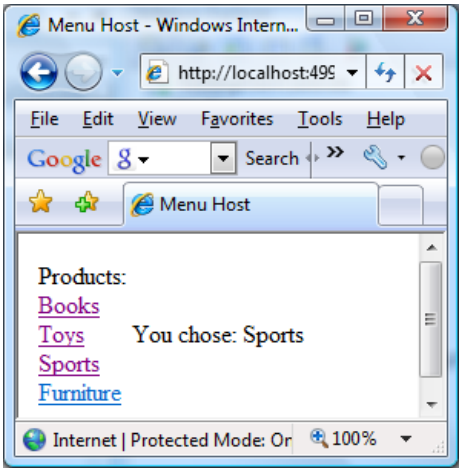


Figure 12.4-1c

You could use the **LinkMenu** control to repeat the same menu on several pages. This is particularly handy in a situation where you can't use master pages to standardize layout (possibly because the pages are too different).

12.5 Integrated User Controls

Integrated user controls interact in one way or another with the web page that hosts them. When you're designing these controls, the class-based design tips you learned in Chapter 4 really become useful.

A typical example is a user control that allows some level of configuration through properties. For instance, you can create a footer that supports two different display formats: long date and short time. To add a further level of refinement, the Footer user control allows the web page to specify the appropriate display format using an enumeration.

The first step is to create an enumeration in the custom Footer class. Remember, an enumeration is simply a type of constant that is internally stored as an integer but is set in code by using one of the allowed names you specify.

Variables that use the FooterFormat enumeration can take the value FooterFormat.LongDate or FooterFormat.ShortTime:

```
public enum FooterFormat
{
  LongDate,
  ShortTime
}
```

The next step is to add a property to the Footer class that allows the web page to retrieve or set the current format applied to the footer. The actual format is stored in a private variable called _format, which is set to the long date format by default when the control is first created. (You can Accomplish the same effect, in a slightly sloppier way, by using a public member variable named Format instead of a full property procedure.) If you're hazy on how property procedures work, feel free to review the explanation in Chapter 3. Finally, the **UserControl.Load** event handler needs to take account of the current footer state and format the output accordingly. The following is the full Footer class code:

```
private FooterFormat format = FooterFormat.LongDate;

public FooterFormat Format
{
   get { return format; }
   set { format = value; }
}
```

```
public partial class Footer : System.Web.UI.UserControl {
  public enum FooterFormat { LongDate, ShortTime }
  private FooterFormat format = FooterFormat.LongDate;

  public FooterFormat Format {
        get { return format; }
        set { format = value; }
  }

  protected void Page_Load(object sender, EventArgs e){
    lblFooter.Text = "This page was served at ";

    if (format == FooterFormat.LongDate)
    {
     lblFooter.Text += DateTime.Now.ToLongDateString();
    }
    else if (format == FooterFormat.ShortTime)
    {
     lblFooter.Text += DateTime.Now.ToShortTimeString();
    }
  }
}
```

Example 12.4-1 Using an Independent User Control

To test this footer, you need to create a page that modifies the Format property of the Footer user control. Figure 12-4.1 shows an example page, which automatically sets the Format property for the user control to match a radio button selection whenever the page is posted back.

To test this footer, you need to create a page that modifies the Format property of the Footer user control. Figure 13-4.2 shows an example page, which automatically sets the Format property for the user control to match a radio button selection whenever the page is posted back.

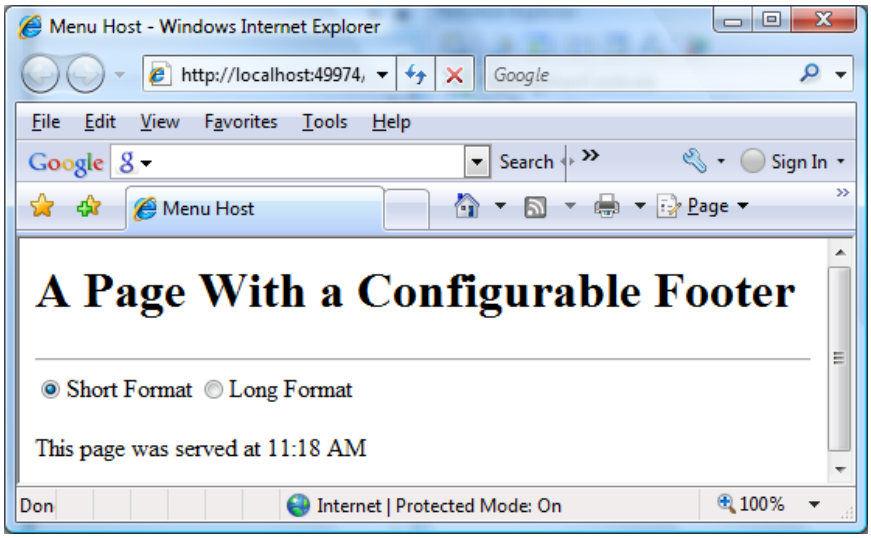


Figure 12.5-1a Format Property Set

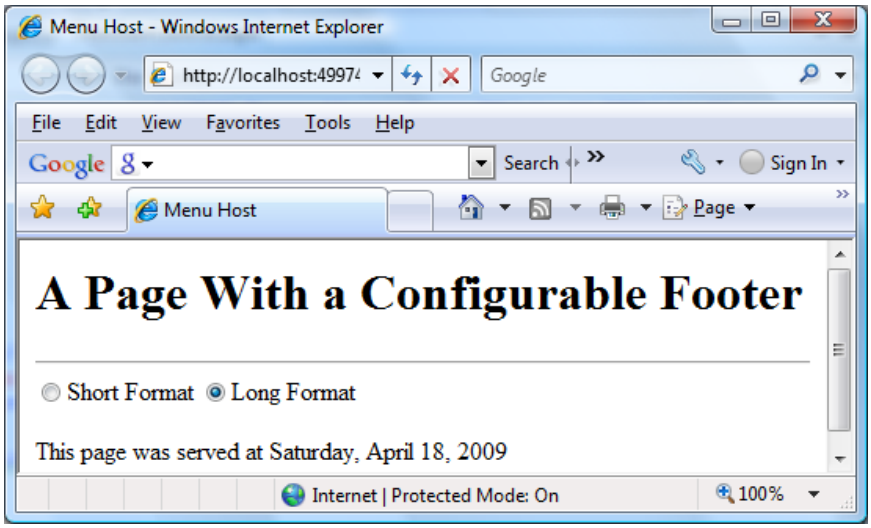


Figure 12.5-1b Format Property Set

12.6 User Controls Events

Another way that communication can occur between a user control and a web page is through events. With methods and properties, the user control reacts to a change made by the web page code. With events, the story is reversed: the user control notifies the web page about an action, and the web page code responds.

Creating a web control that uses events is fairly easy. In the following example, you'll see a version of the LinkMenu control that uses events. Instead of navigating directly to the appropriate page when the user clicks a button, the control raises an event, which the web page can choose to handle.

The first step to create this control is to define the events. Remember, to define an event, you must first choose an event signature. The .NET standard for events specifies that every event should use two parameters. The first one provides a reference to the control that sent the event, while the second incorporates any additional information. This additional information is wrapped into a custom EventArgs object, which inherits from the System.EventArgs class. (If your event doesn't require any additional information, you can just use the predefined EventArgs class, which doesn't contain any additional data. Many events in ASP.NET, such as Page.Load or Button.Click, follow this pattern.) You can refer to Chapter 4 for a quick overview of how to use events in .NET. The LinkMenu2 control uses a single event, which indicates when a link is clicked:

```
public partial class LinkMenu2 : System.Web.UI.UserControl
{
  public event EventHandler LinkClicked;
}
```

Example 12.6-1 Defining an Event

This code defines an event named LinkClicked. The LinkClicked event has the signature specified by the System.EventHandler delegate, which includes two parameters—the event sender and an ordinary EventArgs object. That means that any event handler you create to handle the LinkClicked event must look like this:

```
protected void LinkMenu_LinkClicked(object sender, EventArgs e){ ... }
```

Example 12.6-2 Defining an Event

This takes care of defining the event, but what about raising it? This part is easy. To fire the event, the LinkMenu2 control simply calls the event by name and passes in the two parameters, like this:

```
// Raise the LinkClicked event, passing a reference to
// the current object (the sender) and an empty EventArgs object.
LinkClicked(this, EventArgs.Empty);
```

The LinkMenu2 control actually needs a few more changes. The original version used the HyperLink control. This won't do, because the HyperLink control doesn't fire an event when the link is clicked. Instead, you'll need to use the LinkButton. The LinkButton fires the Click event, which the LinkMenu2 control can intercept, and then raises the LinkClicked event to the web page.

The following is the full user control code:

```
public partial class LinkMenu2 : System.Web.UI.UserControl
{
    public event EventHandler LinkClicked;
    protected void lnk_Click(object sender, EventArgs e)
    {
        // One of the LinkButton controls has been clicked.
        // Raise an event to the page.
        if (LinkClicked != null) { LinkClicked(this, EventArgs.Empty); }
    }
}
```

Example 12.6-3 Using an Event

Notice that before raising the **LinkClicked** event, the LinkMenu2 control needs to test the **LickedClick** event for a null reference. A null reference exists if no event handlers are attached to the event. In this case, you shouldn't try to raise the event, because it would only cause an error.

You can create a page that uses the LinkMenu2 control and add an event handler. Unfortunately, you won't be able to connect these event handlers using the Visual Studio Properties window, because the Properties

window won't show the custom events that the user control provides. Instead, you'll need to modify the **LinkMenu2** tag directly, as shown here:

```
<apress:LinkMenu2 id="Menu1" runat="server"
  OnLinkClicked="LinkClicked" />
```

and here's the event handler that responds in the web page:

Figure 12.6-1 Using LinkMenu2 User Control

Conceptually, this approach should give your web page more power to customize how the user control works. Unfortunately, that's not the case at the moment, because a key piece of information is missing. When the **LinkClicked** event occurs, the web page has no way of knowing what link was clicked, which prevents it from taking any kind of reasonable action. The only way to solve this problem is to create a more intelligent event that can transmit some information through event arguments. You'll see how in the next section.

12.7 Passing Information with Events

In the current LinkMenu2 example no custom information is passed along with the event. In many cases, however, you want to convey additional information that relates to the event. To do so, you need to create a custom class that derives from EventArgs.

The **LinkClickedEventArgs** class that follows allows the LinkMenu2 user control to pass the URL that the user selected through a Url property. It also provides a Cancel property. If set to true, the user control will stop its processing immediately. But if Cancel remains false (the default), the user control will send the user to the new page. This way, the user control still handles the task of redirecting the user, but it allows the web page to plug into this process and change it or stop it (for example, if there's unfinished work left on the current page).

To use this **EventArgs** class, you need to create a new delegate that represents the **LinkClicked** event signature. Here's what it looks like shown In Example 12.7-1

```
public class LinkClickedEventArgs : EventArgs
{
  private string url;
  public string Url
  {
     get { return url; }
     set { url = value; }
  }

  private bool cancel = false;
  public bool Cancel {
      get { return cancel; }
      set { cancel = value; }
   }

   public LinkClickedEventArgs(string url) { Url = url;  }
}
```

Example 12.7-1a Defining an Events with custom class.

```
public delegate void LinkClickedEventHandler(object sender,  LinkClickedEventArgs e);
```

Example 12.7-1b Defining the Event Handler

Both the **LinkClickedEventArgs** class and the **LinkClickedEventHandler** delegate should be placed in the App_Code directory. That way, these classes will be compiled automatically and made available to all web pages. Now you can modify the **LinkClicked** event to use the **LinkClickedEventHandler** delegate:

public event **LinkClickedEventHandler** LinkClicked;

Next, your user control code for raising the event needs to submit the required information when calling the event. But how does the user control determine what link was clicked? The trick is to switch from the **LinkButton.Click** event to the **LinkButton.Command** event. The Command event automatically gets the **CommandArgument** that's defined in the tag. So if you define your **LinkButton** controls like this:

```
<asp:LinkButton ID="lnkBooks" runat="server"
  CommandArgument="Menu2Host.aspx?product=Books" OnCommand="lnk_Command">Books
</asp:LinkButton><br />
<asp:LinkButton ID="lnkToys" runat="server"
  CommandArgument="Menu2Host.aspx?product=Toys" OnCommand="lnk_Command">Toys
</asp:LinkButton><br />
<asp:LinkButton ID="lnkSports" runat="server"
  CommandArgument="Menu2Host.aspx?product=Sports" OnCommand="lnk_Command">Sports
</asp:LinkButton><br />
<asp:LinkButton ID="lnkFurniture" runat="server"
  CommandArgument="Menu2Host.aspx?product=Furniture" OnCommand="lnk_Command">
Furniture</asp:LinkButton>
```

Example 12.7-1c Defining the presentation with LinkButton

you can pass the link along to the web page like this:
Code View: Scroll / Show All

```
LinkClickedEventArgs args = new
  LinkClickedEventArgs((string)e.CommandArgument);
LinkClicked(this, args);
```

Here's the complete user control code(Figure 12.7-1d). It implements one more feature. After the event has been raised and handled by the web page, the LinkMenu2 checks the Cancel property. If it's false, it goes ahead and performs the redirect using Reponse.Redirect().

```
public partial class LinkMenu2 : System.Web.UI.UserControl
{
  public event LinkClickedEventHandler LinkClicked;
  protected void lnk_Command(object sender, CommandEventArgs e)
  {
   // One of the LinkButton controls has been clicked.
   // Raise an event to the page.
   if (LinkClicked != null)
   {
     // Pass along the link information.
     LinkClickedEventArgs args =
              new LinkClickedEventArgs((string)e.CommandArgument);
     LinkClicked(this, args);
     // Perform the redirect.
     if (!args.Cancel){
     // Notice we use the Url from the LinkClickedEventArgs
     // object, not the original link. That means the web page
     // can change the link if desired before the redirect.
      Response.Redirect(args.Url);
    }
   }
  }
}
```

Example 12.7-1d Complete User Control Code

Finally, you need to update the code in the web page (where the user control is placed) so that its event handler uses the new signature. In the following code, the **LinkClicked** event handler checks the URL and allows it in all cases except one:

```
protected void LinkClicked(object sender, LinkClickedEventArgs e)
{
   if (e.Url == "Menu2Host.aspx?product=Furniture")
   {
      lblClick.Text = "This link is not allowed.";
      e.Cancel = true;
   }
   else
   {
      // Allow the redirect, and don't make any changes to the URL.
   }
}
```

Example 12.7-1e Updating the Web Page

If you click the Furniture link, you'll see the message shown in Figure 12.7-1.

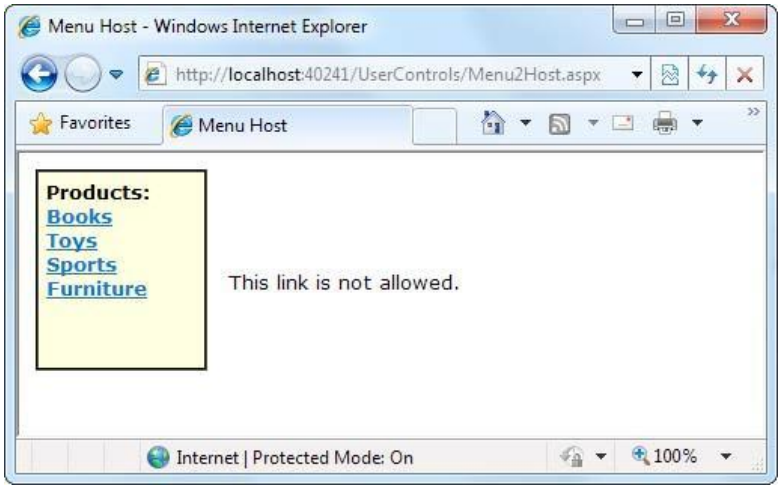


Figure12.7-1 Handling a User Control

The rest of the chapter is outside the scope for this course. But if you plan to develop professional web applications, you might as well read and apply the content.

13.0 Styles, Themes and Master Pages (Text Book Chapter 12)

13.0 Styles, Themes and Master Pages

13.1 Introduction

Using the techniques you've learned so far, you can create polished web pages and let users surf from one page to another. However, to integrate your web pages into a unified, consistent website, you need a few more tools. In this chapter, you'll consider three of the most important tools that you can use: styles, themes, and master pages.

Styles are part of the Cascading Style Sheet (CSS) standard. They aren't directly tied to ASP.NET, but they're still a great help in applying consistent formatting across your entire website. With styles, you can define a set of formatting options once, and reuse it to format different elements on multiple pages. You can even create styles that apply their magic automatically—for example, styles that change the font of all the text in your website without requiring you to modify any of the web page code. Best of all, once you've standardized on a specific set of styles and applied them to multiple pages, you can give your entire website a face-lift just by editing your style sheet.

Styles are genuinely useful, but there are some things they just can't do. Because styles are based on the HTML standard, they have no understanding of ASP.NET concepts like control properties. To fill the gap, ASP.NET includes a themes feature, which plays a similar role to styles but works exclusively with server controls. Much as you use styles to automatically set the formatting characteristics of HTML elements, you use themes to automatically set the properties of ASP.NET controls.

Another feature for standardizing websites is master pages. Essentially, a master page is a blueprint for part of your website. Using a master page, you can define web page layout, complete with all the usual details such as headers, menu bars, and ad banners. Once you've perfected a master page, you can use it to create content pages. Each content page automatically acquires the layout and the content of the linked master page.

➢13.0 Styles, Themes and Master Pages

By using styles, themes, and master pages, you can ensure that all the pages on your website share a standardized look and layout. In many cases, these details are the difference between an average website and one that looks truly professional.

13.2 Styles

In the early days of the Internet, website designers used the formatting features of HTML to decorate these pages. These formatting features were limited, inconsistent, and sometimes poorly supported. Worst of all, HTML formatting led to horribly messy markup, with formatting details littered everywhere.

The solution is the CSS standard, which is supported in all modern browsers. Essentially, CSS gives you a wide range of consistent formatting properties that you can apply to any HTML element. Styles allow you to add borders, set font details, change colors, add margin space and padding, and so on. Many of the examples you've seen so far have in this book have used CSS formatting.

In the following sections, you'll learn the basics of the CSS standard. You'll see how web controls use CSS to apply their formatting, and you'll learn how you can explicitly use styles in your ASP.NET web pages.

13.2.1 Style Types

Web pages can use styles in three different ways:

Style Type	Description
Inline style	An inline style is a style that's placed directly inside an HTML tag. This can get messy, but it's a reasonable approach for one-time formatting. You can remove the style and put it in a style sheet later.
Internal style sheet	An internal style sheet is a collection of styles that are placed in the <head> section of your web page markup. You can then use the styles from this style sheet to format the web controls on that page. By using an internal style sheet, you get a clear separation between formatting (your styles) and your content (the rest of your HTML markup). You can also reuse the same style for multiple elements.
External style sheet	An external style sheet is similar to an internal style sheet, except it's placed in a completely separate file. This is the most powerful approach, because it gives you a way to apply the same style rules to many pages. You can use all types of styles with ASP.NET web pages. You'll see how in the following sections.

Figure13.2-1 Web Page Style Types

➢13.0 Styles, Themes and Master Pages

You can use all types of styles with ASP.NET web pages. You'll see how in the following sections.

To apply a style to an ordinary HTML element, you set the style attribute. Here's an example that gives a blue background to a paragraph:

** This text has a blue background. **

Every style consists of a list of one or more formatting properties. In the preceding example, the style has a single formatting property, named background, which is set to the value Blue. To add multiple style properties, you simply separate them with semicolons, as shown here:

** **
**This text has a blue background. **

This style creates large white text with a blue background and 10 pixels of spacing between the edge of the element (the blue box) and the text content inside.

NOTE: The full list of formatting properties is beyond the scope of this book (although you can get all the details at www.w3schools.com/css). However, you'll soon see that Visual Studio includes tools that can help you build the styles you want, so you don't need to remember style property names or write styles by hand.

You can use the same approach to apply formatting to a web control using a style. However, you don't need to, because web controls provide formatting properties. For example, if you create a Label control like this:

<asp:Label ID="MyLabel" runat="server" ForeColor="White"
BackColor="Blue“ Font-Size="X-Large">Formatted Text</asp:Label>

it's actually rendered into this HTML, which uses an inline style:

<span id="MyLabel“ style="color:White; background-color:Blue;
font-size:X-Large"> Formatted Text

Incidentally, if you specify a theme and set formatting properties that overlap with your style, the properties have the final say.

13.2.2 The Style Builder

Visual Studio provides an indispensable style builder that lets you create styles by picking and choosing your style preferences in a dedicated dialog box. To try it out, begin by creating a new page in Visual Studio. Then drop a few controls onto your page (for example, a label, text box, and button).

Every new page starts with an empty element. This is simply a content container—by default, it doesn't have any appearance. However, by applying style settings to the , you can create a bordered content region, and you can change the font and colors of the content inside. In this example, you'll see how to use Visual Studio to build a style for the element.

```
<div>
   <asp:Label ID="Label1" runat="server">Type something here:
   </asp:Label>
   <asp:TextBox ID="TextBox1" runat="server">
   </asp:TextBox>
   <br /><br />
   <asp:Button ID="Button1" runat="server" Text="Button">
   </asp:Button>
</div>
```

Example 13.2-1 with styles

NOTE: CSS supports a feature it calls inheritance. With inheritance, some formatting properties (such as the font family) are passed down from a parent element to other nested elements. In other words, if you set the font family for a element, all the elements inside will inherit the same font (unless they explicitly specify otherwise). Other properties, like margin and padding settings, don't use inheritance. To learn more about this behavior and the specific properties that use inheritance, you can experiment on your own, consult a dedicated book such as Eric Meyer's CSS: The Definitive Guide (O'Reilly, 2006), or use the tutorials at www.w3schools.com/css.

Before formatting the page, make sure all your controls are nested inside the element. Your markup should look something like this – Example 13.2-2

In the design window, click somewhere inside the (but not on another control). You'll know you're in the right spot when a border appears around your controls, showing you the outline of the , as shown in Figure 13.2-2

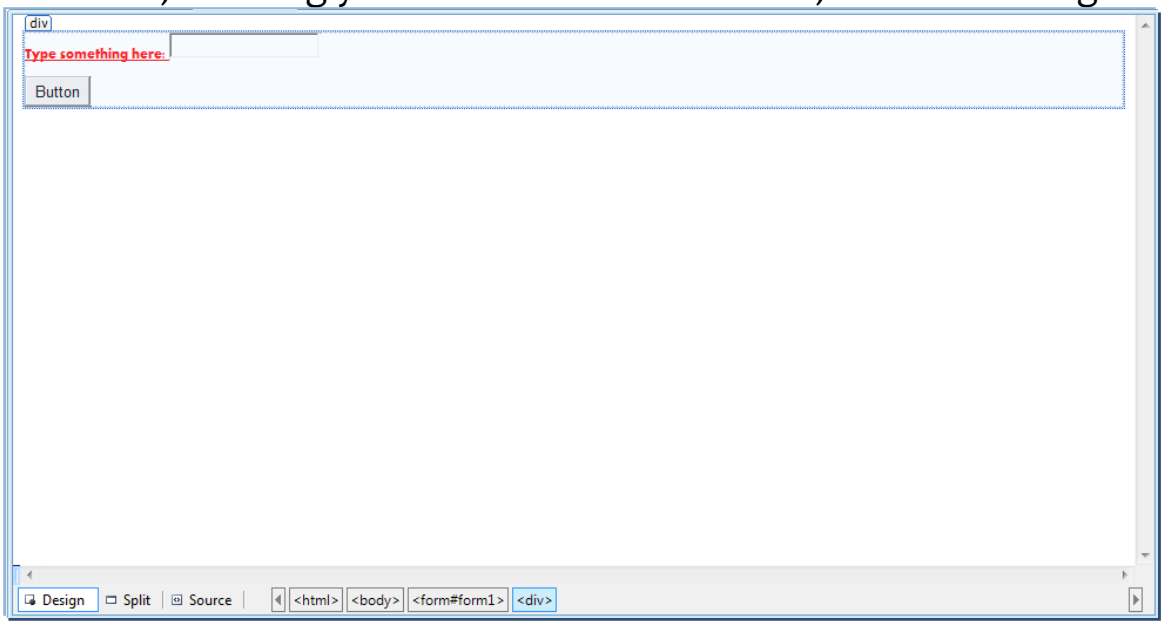


Figure13.2-2 with styles appearance

Next, choose Format New Style from the menu. This opens the New Style dialog box shown in Figure 13.2-2. In the Selector box at the top of the window, choose Inline Style to specify that you're creating your style directly in the HTML markup.

To specify style settings, you first need to choose one of the categories in the Category list. For example, if you choose Font you'll see a list of font-related formatting settings, such as font family, font size, text color, and so on. You can apply settings from as many different categories as you want. Table in figure 13.2-3 provides a brief explanation for each category.

➢13.0 Styles, Themes and Master Pages

Table Style Settings in the New Style Dialog Box	
Category	**Description**
Font	Allows you to choose the font family, font size, and text color, and apply other font characteristics (like italics and bold).
Block	Allows you to fine-tune additional text settings, such as the height of lines in a paragraph, the way text is aligned, the amount of indent in the first list, and the amount of spacing between letters and words.
Background	Allows you to set a background color or image.
Border	Allows you to define borders on one or more edges of the element. You can specify the border style, thickness, and color of each edge.
Box	Allows you to define the margin (the space between the edges of the element and its container) and the padding (the space between the edges of the element and its nested content inside).
Position	Allows you to set a fixed width and height for your element, and use absolute positioning to place your element at a specific position on the page. Use these settings with care. When you make your element a fixed size, there's a danger that the content inside can become too big (in which case it leaks out the bottom or the side). When you position your element using absolute coordinates, there's a chance that it can overlap another element.
Layout	Allows you to control a variety of miscellaneous layout settings. You can specify whether an element is visible or hidden, whether it floats at the side of the page, and what cursor appears when the user moves the mouse overtop, among other settings.
List	If you're configuring a list (a or element), you can set the numbering or bullet style. These settings aren't commonly used in ASP.NET web pages, because you're more likely to use ASP.NET list controls like the BulletedList.
Table	Allows you to set details that only apply to table elements (such as <tr> and <td>). For example, you can control whether borders appear around an empty cell.

Figure13.2 3 Table Style Settings in Style Dialog Box

Creating an Inline Style:

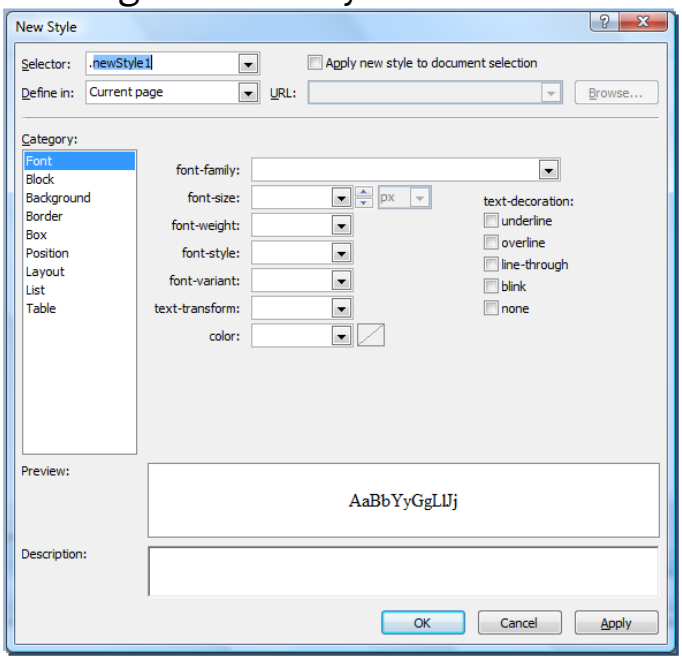


Figure13.2-4 Using The Style Dialog

NOTE : Remember, you can build a style for any HTML element, not just the element. You'll always get exactly the same New Style dialog box with the same formatting options.

➢13.0 Styles, Themes and Master Pages

As you make your selections, Visual Studio shows what your style will look like when applied to some sample text (in the Preview box) and the markup that's needed to define your style (in the Description) box. Figure 13.2-5 shows the New Style dialog box after formatting the into a nicely shaded and bordered box. In the Category list, all the categories with formatting settings are highlighted in bold.

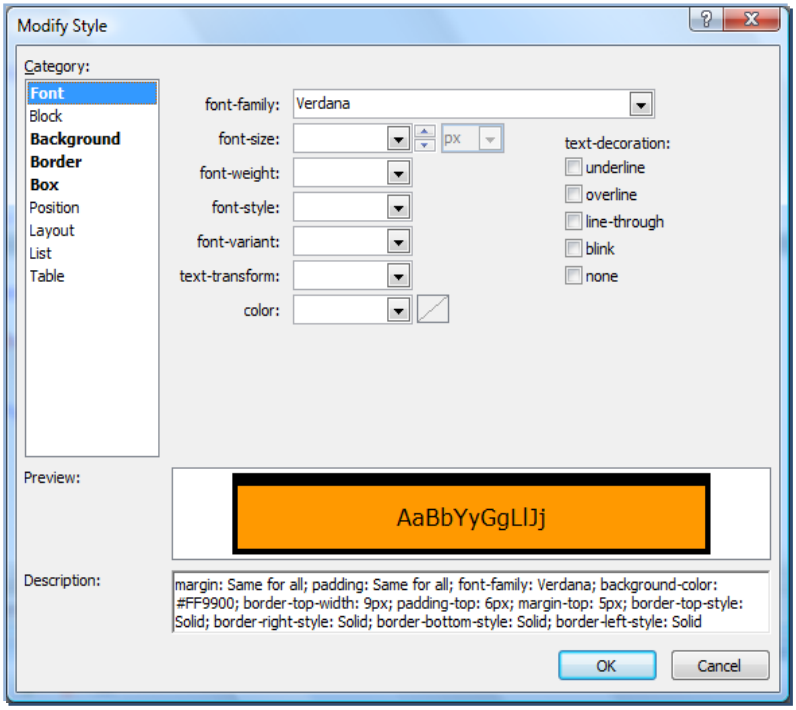


Figure13.2-5 New Style Dialog Box after Formatting

After setting the parameters, this is how the tag will appear

```
<div style="margin: Same for all; padding: Same for all;
    font-family: Verdana; background-color: #FF9900;
    border-top-width: 9px; padding-top: 6px; margin-
    top: 5px; border-top-style: Solid; border-right-style:
    Solid; border-bottom-style: Solid; border-left-style:
    Solid;">
```

Example 13.2-2 with styles

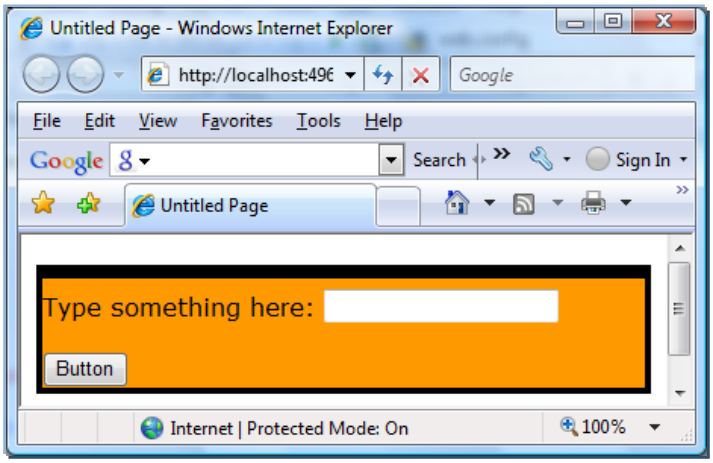


Figure13.2-6 New Style added Dialog Box

13.2.3 CSS Property Window

Once you've created a style, you have two easy options for modifying it in Visual Studio. Both revolve around the CSS Properties window, which allows you to dissect the formatting details of any style.

To show the CSS Properties window, open a web page in Visual Studio and choose View CSS Properties. The CSS Properties window is usually grouped with the Toolbox and Server Explorer windows at the left of the Visual Studio window.

Now that the CSS Properties window is visible, you can use it to view one of your styles. First, find the element or web control that uses the style attribute. Then, click to select it (in design view) or click somewhere in the element's start tag (in source view). Either way, the style information for that element will appear in the CSS Properties window. For example, Figure 13.5 -1shows what you'll see for the element that was formatted in the previous section.

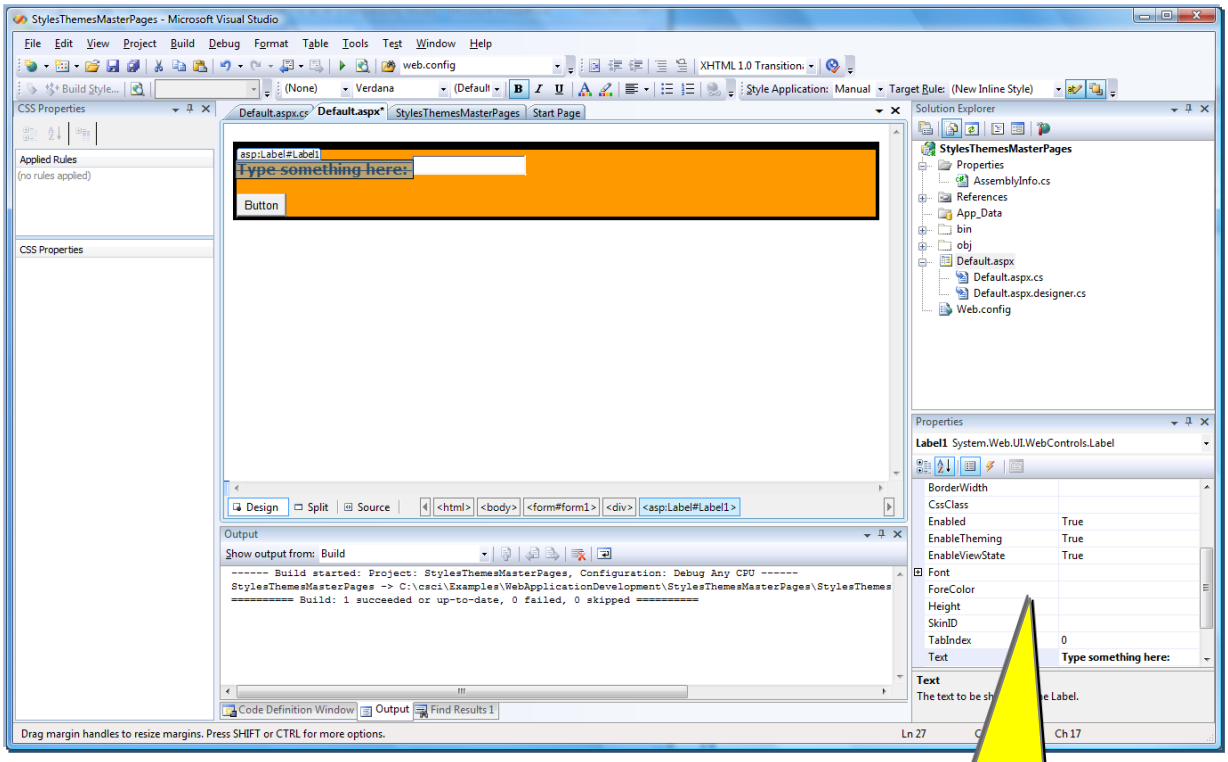


Figure 13.2-7 Properties Window

The CSS Properties window provides an exhaustive list of all the formatting properties you can use in a style. This list is grouped into categories, and the properties in each category are sorted alphabetically. The ones that are currently set are displayed in bold.

You can use the CSS Properties window to modify existing style properties or set new ones, in the same way that you modify the properties for web controls using the Properties window. For example, in Figure 14-5 the font size is being changed.

Depending on your personal taste, you may find that the CSS Properties window is more convenient than the style builder because it gives you access to every style property at once. Or, you may prefer the more organized views in the style builder. (Your preference might also depend on how much screen real estate you have to devote to the CSS Properties window.) If you decide that you want to return to the style builder to change a style, the process is fairly straightforward. First, select the element that has the inline style. Next, look at the Applied Rules list at the top of the CSS Properties window, which should show the text < inline style >. Right-click that text and choose Modify Style to open the Modify Style dialog box, which looks identical to the New Style dialog box you considered earlier.

NOTE : You can't use the CSS Properties window to create a style. If you select an element that doesn't have a style applied, you won't see anything in the CSS Properties window (unless you select an element that's inside another element, and the containing element uses a style).

13.3 Themes

With the convenience of CSS styles, you might wonder why developers need anything more. The problem is that CSS rules are limited to a fixed set of style attributes. They allow you to reuse specific formatting details (fonts, borders, foreground and background colors, and so on), but they obviously can't control other aspects of ASP.NET controls. For example, the CheckBoxList control includes properties that control how it organizes items into rows and columns. Although these properties affect the visual appearance of the control, they're outside the scope of CSS, so you need to set them by hand. Additionally, you might want to define part of the behavior of the control along with the formatting. For example, you might want to standardize the selection mode of a Calendar control or the wrapping in a TextBox. This obviously isn't possible through CSS.

The themes feature fills this gap. Like CSS, themes allow you to define a set of style details that you can apply to controls in multiple pages. However, with CSS, themes aren't implemented by the browser. Instead, ASP.NET processes your themes when it creates the page.

NOTE : Themes don't replace styles. Instead, they complement each other. Styles are particularly useful when you want to apply the same formatting to web controls and ordinary HTML elements. Themes are indispensable when you want to conFigure control properties that can't be tailored with CSS.

All themes are application specific. To use a theme in a web application, you need to create a folder that defines it. This folder needs to be placed in a folder named App_Themes, which must be placed inside the top-level directory for your web application. In other words, a web application named SuperCommerce might have a theme named FunkyTheme in the folder SuperCommerce\App_Themes\FunkyTheme. An application can contain definitions for multiple themes, as long as each theme is in a separate folder. Only one theme can be active on a given page at a time.

➢13.0 Styles, Themes and Master Pages

To actually make your theme accomplish anything, you need to create at least one skin file in the theme folder. A skin file is a text file with the .skin extension. ASP.NET never serves skin files directly—instead, they're used behind the scenes to define a theme.

A skin file is essentially a list of control tags - with a twist. The control tags in a skin file don't need to completely define the control. Instead, they need to set only the properties that you want to standardize. For example, if you're trying to apply a consistent color scheme, you might be interested in setting only properties such as ForeColor and BackColor. When you add a control tag for the ListBox in the skin file, it might look like this:

```
<asp:ListBox runat="server" ForeColor="White" BackColor="Orange"/>
```

The runat="server" portion is always required. Everything else is optional. You should avoid setting the ID attribute in your skin, because the page that contains the ListBox needs to define a unique name for the control in the actual web page.

It's up to you whether you create multiple skin files or place all your control tags in a single skin file. Both approaches are equivalent, because ASP.NET treats all the skin files in a theme directory as part of the same theme definition. Often, it makes sense to put the control tags for complex controls (such as the data controls) in separate skin files. Figure 13.3-1 shows the relationship between themes and skins in more detail.

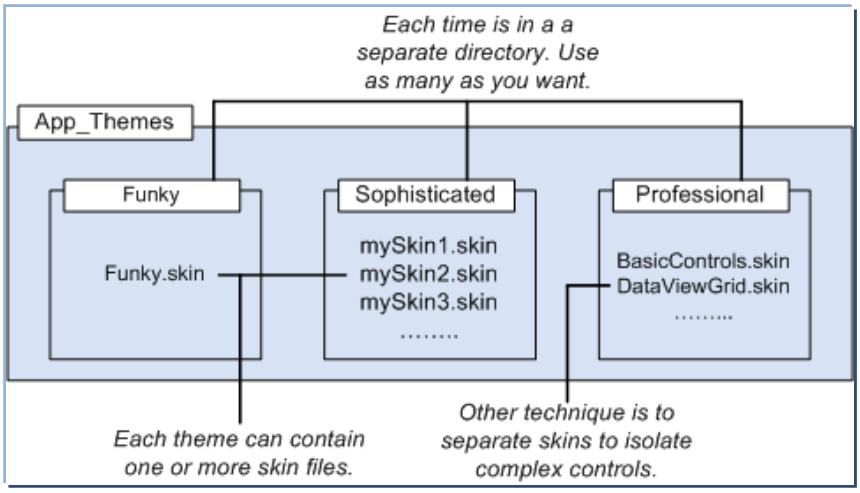


Figure 13.3-1 App Themes

➢13.0 Styles, Themes and Master Pages

ASP.NET also supports global themes. These are themes you place in the :\Inetpub\wwwroot\aspnet_client\system_web\v2.0.50727\Themes folder. However, it's recommended that you use local themes, even if you want to create more than one website that has the same theme. Using local themes makes it easier to deploy your web application, and it gives you the flexibility to introduce site-specific differences in the future. If you have a local theme with the same name as a global theme, the local theme takes precedence, and the global theme is ignored. The themes are not merged together.

To add a theme to your project, select Website Add New Item, and choose Skin File (See Figure 13.3-2). Visual Studio will warn you that skin files need to be placed in a subfolder of the App_Themes folder and ask you whether that's what you intended. If you choose Yes, Visual Studio will create a folder with the same name as your theme file. You can then rename the folder and the file to whatever you'd like to use. Figure 13.3-3 shows an example with a theme that contains a single skin file.

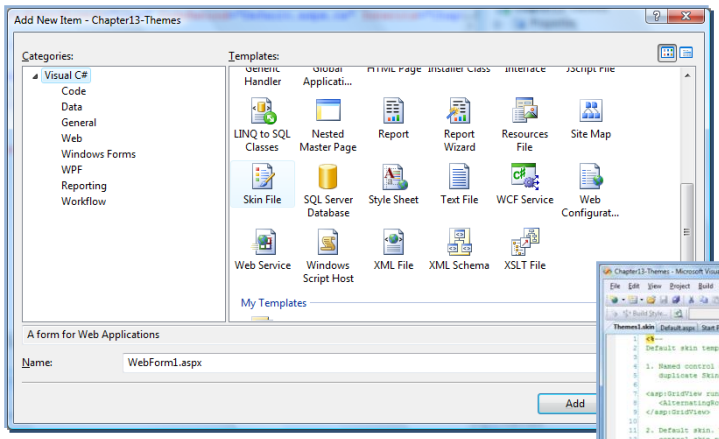


Figure 13.3-2 Adding a Skin

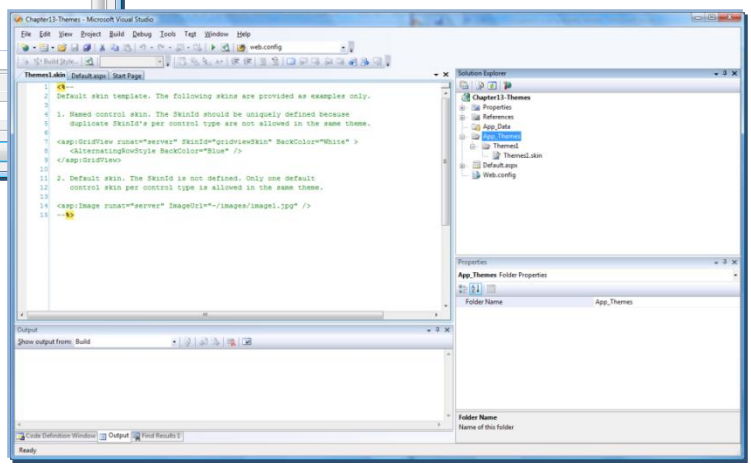


Figure 13.3-3 Theme with a Skin

➢13.0 Styles, Themes and Master Pages

Unfortunately, Visual Studio doesn't include any design-time support for creating themes, so it's up to you to copy and paste control tags from other web pages.

Here's a sample skin that sets background and foreground colors for several common controls:

```
<asp:ListBox runat="server" ForeColor="White" BackColor="Orange"/>
<asp:TextBox runat="server" ForeColor="White" BackColor="Orange"/>
<asp:Button runat="server" ForeColor="White" BackColor="Orange"/>
```

To apply the theme in a web page, you need to set the Theme attribute of the Page directive to the folder name for your theme. (ASP.NET will automatically scan all the skin files in that theme.)

```
<%@ Page Language="C#" AutoEventWireup="true" ... Theme="Themes1" %>
```

You can make this change by hand, or you can select the DOCUMENT object in the Properties window at design time and set the Theme property (which provides a handy drop-down list of all your web application's themes). Visual Studio will modify the Page directive accordingly.

When you apply a theme to a page, ASP.NET considers each control on your web page and checks your skin files to see whether they define any properties for that control. If ASP.NET finds a matching tag in the skin file, the information from the skin file overrides the current properties of the control.

Figure 13.3-4 shows the result of applying the Themes1 to a simple page. You'll notice that conflicting settings (such as the existing background for the list box) are overwritten. However, changes that don't conflict (such as the custom font for the buttons) are left in place.

Here is the example implementation of the **Themes1 .skin** into a set of controls (before and after)

➢13.0 Styles, Themes and Master Pages

```
<%@ Page Language="C#"   AutoEventWireup="true"
      CodeBehind="Default.aspx.cs"
      Inherits="Chapter13_Themes._Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
      1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
      transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:TextBox ID="TextBox1" Text="Testing"
     Width="95" runat="server" /> <br />
   <asp:ListBox ID="ListBox1" Width="100"
     Height="100" runat="server" /> <br />
   <asp:Button ID="Button1" Width="100" Text="Test
     Button" runat="server"/>
   </div>
   </form>
</body>
</html>
```

Example 13.3-1a Before Adding a Theme

```
<%@ Page Language="C#" Theme="Themes1"
      AutoEventWireup="true"
      CodeBehind="Default.aspx.cs"
      Inherits="Chapter13_Themes._Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
      1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
      transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:TextBox ID="TextBox1" Text="Testing"
     Width="95" runat="server" /> <br />
   <asp:ListBox ID="ListBox1" Width="100"
     Height="100" runat="server" /> <br />
   <asp:Button ID="Button1" Width="100" Text="Test
     Button" runat="server"/>
   </div>
   </form>
</body>
</html>
```

Example 13.3-1b After Adding a Theme

Here are the two results – before and after applying g the themes (themes1.skin)

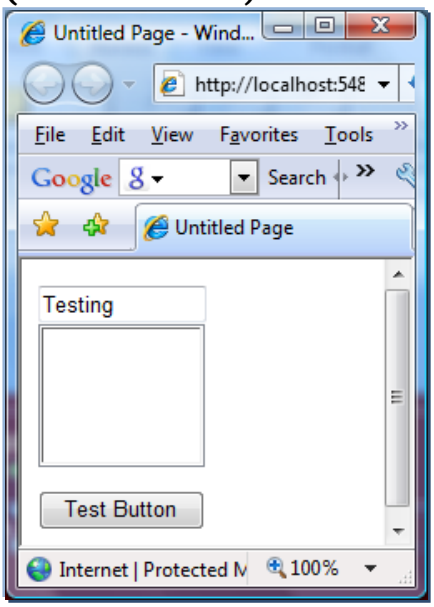


Figure 13.3-4a Before

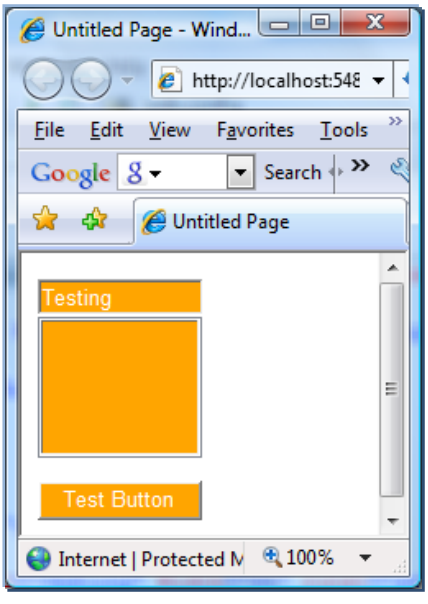


Figure 13.3-4b After

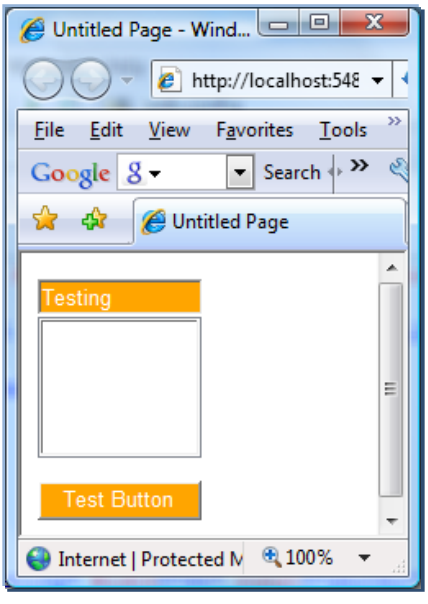


Figure 13.3-4c Controlled

13.3.1 Controlling Themes

You can decide if you wish to avoid adding the theme to a specific control of your page. For example in the example 13.3-1, if you decide to avoid the theme Themes1.skin on the List Box control, all you have to is to set the **EnableTheming** property of that control to **false**.

```
<asp:Button ID="Button1" runat="server" ... EnableTheming="false" />
```

So you get the output as follows (in Figure 13.3-4c)

13.3.2 Applying a Theme to entire Web Site.

Using the Page directive, you can bind a theme to a single page. However, you might decide that your theme is ready to be rolled out for the entire web application. The cleanest way to apply this theme is by configuring the <pages> element in the **web.config** file for your application, as shown here:

```
<configuration>
  <system.web>
    <pages theme="Theme1">
      ...
    </pages>
  </system.web>
</configuration>
```

If you want to use the style sheet behavior so that the theme doesn't overwrite conflicting control properties, use the **StyleSheetTheme** attribute instead of Theme:

```
<configuration>
  <system.web>
    <pages styleSheetTheme="Theme1">
      ...
    </pages>
</system.web>
</configuration>
```

Either way, when you specify a theme in the **web.config** file, the theme will be applied throughout all the pages in your website, provided these pages don't have their own theme settings. If a page specifies the Theme attribute, the page setting will take precedence over the **web.config** setting. If your page specifies the Theme or **StyleSheetTheme** attribute with a blank string (Theme=""), no theme will be applied at all.

Using this technique, it's just as easy to apply a theme to part of a web application. For example, you can create a separate web.config file for each subfolder and use the <pages> setting to configure different themes.

Here is an example of modified Themes1.skin (Example 13.3-2) file along with the modified web.config file.

Web.confic | **Themes1.skin** | Default.aspx | Start Page

```
<%--
Default skin template. The following skins are provided as examples only.

1. Named control skin. The SkinId should be uniquely defined because
   duplicate SkinId's per control type are not allowed in the same theme.

<asp:GridView runat="server" SkinId="gridviewSkin" BackColor="White" >
   <AlternatingRowStyle BackColor="Blue" />
</asp:GridView>

2. Default skin. The SkinId is not defined. Only one default
   control skin per control type is allowed in the same theme.

<asp:Image runat="server" ImageUrl="~/images/image1.jpg" />
--%>

<asp:ListBox runat="server" ForeColor="White" BackColor="Orange"/>
<asp:TextBox runat="server" ForeColor="White" BackColor="Orange"/>
<asp:Button runat="server" ForeColor="White" BackColor="Orange"/>
<asp:TextBox runat="server" ForeColor="White" BackColor="DarkOrange"
 Font-Bold="True" SkinID="Dramatic"/>
<asp:Button runat="server" ForeColor="White" BackColor="DarkOrange"
 Font-Bold="True" SkinID="Dramatic"/>
```

Example 13.3-2 Skin Template

Figure 13.3-5a Before

```
<pages theme="Themes1">
  <controls>
    <add tagPrefix="asp" namespace="System.Web.UI" assembly="System.Web.Extensions, Version=3.5.0.0,
                                                    Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
    <add tagPrefix="asp" namespace="System.Web.UI.WebControls" assembly="System.Web.Extensions, Version=3.5.0.0,
                                                    Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
  </controls>
</pages>
```

Figure 13.3-5b Skin Template added in web.config file

➢13.0 Styles, Themes and Master Pages

Here is the result of the example. In this the Themes-”” tag is removed in the <@page….>

<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="Chapter13_Themes._Default" %>

```
<%@ Page Language="C#"  AutoEventWireup="true"
    CodeBehind="Default.aspx.cs"
    Inherits="Chapter13_Themes._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
    Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
  <asp:TextBox ID="TextBox1" Text="Testing" Width="95"
    runat="server" /> <br />
  <asp:ListBox ID="ListBox1" Width="100" Height="100"
    runat="server" EnableTheming="false" /> <br />
  <asp:Button ID="Button1" Width="100" Text="Test Button"
    runat="server"/>
  </div>
  </form>
</body>
</html>
```

Example 13.3-3 Skin is now coming from web.config. file

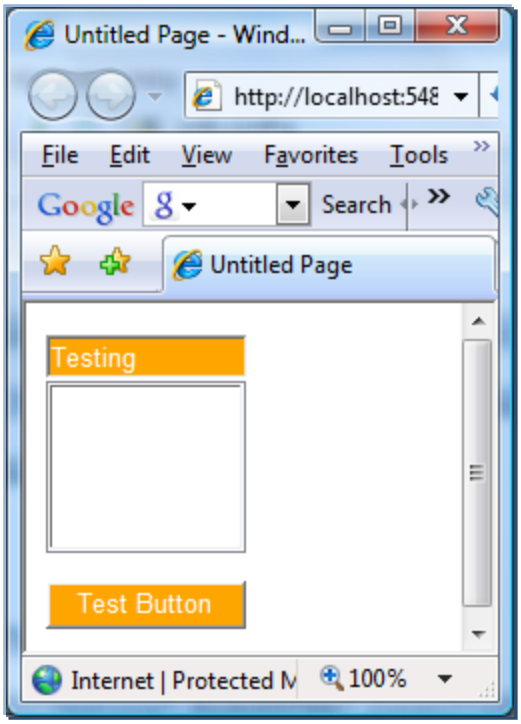


Figure 13.3-6 Controlled Skin

Figure 13.3-6 is the result. But the results not the same as in Figure 13.3-4b as this now uses the theme specified in the web.config file.

13.3.2 Using a SkinID

Ordinarily, if you create more than one theme for the same control, ASP.NET will give you a build error stating that you can have only a single default skin for each control. To get around this problem, you need to create a named skin by supplying a **SkinID** attribute. As an example there are multiple TextBox attibutes defined in the web.config file. But the "DardOrange" one has a **SkinID="Dramatic".**

The catch is that named skins aren't applied automatically like default skins. To use a named skin, you need to set the SkinID of the control on your web page to match. You can choose this value from a drop-down list that Visual Studio creates based on all your defined skin names, or you can type it in by hand:

<asp:Button ID="Button1" runat="server" SkinID="Dramatic" />

If you don't like the opt-in model for themes, you can make all your skins named. That way, they'll never be applied unless you set the control's SkinID.

ASP.NET is intelligent enough to catch it if you try to use a skin name that doesn't exist, in which case you'll get a build warning. The control will then behave as though you set EnableTheming to false, which means it will ignore the corresponding default skin. Here is the code example of the presentation.(Example 13.3-4). The result is shown on Figure 13.3-8

```
<body>
  <form id="form1" runat="server">
  <div>
  <asp:TextBox ID="TextBox1"
    Text="Testing" Width="95"
    runat="server" /> <br />
  <asp:ListBox ID="ListBox1"
    Width="100" Height="100"
    runat="server"
    EnableTheming="false" /> <br />
  <asp:Button ID="Button1"
    Width="100" Text="Test Button"
    SkinID="Dramatic"
    runat="server"/>
  </div>
  </form>
</body>
```

Example 13.3-4 Setting a SkinID

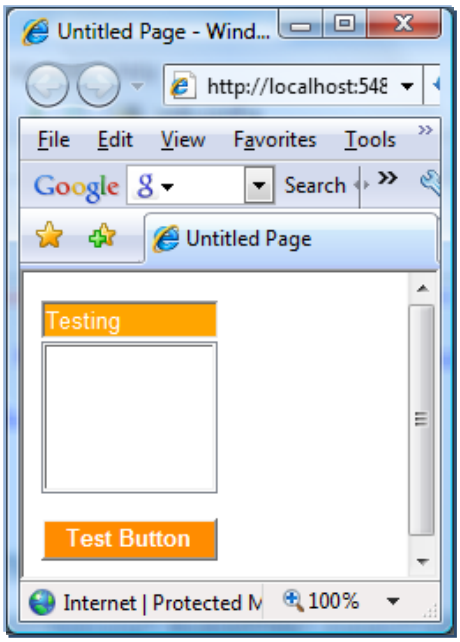


Figure 13.3-7 Controlled Skin with SkinID

➢13.0 Styles, Themes and Master Pages

13.4 Master Pages

ASP.NET master pages allow you to create a consistent layout for the pages in your application. A single master page defines the look and feel and standard behavior that you want for all of the pages (or a group of pages) in your application. You can then create individual content pages that contain the content you want to display. When users request the content pages, they merge with the master page to produce output that combines the layout of the master page with the content from the content page.

Student is assigned to read this chapter.

[Supplement] Messaging System - e-mailing

➢Supplement- Messaging System (E-Mailing)

Messaging Services:

Electronic Mailing (e-mail) is a component of the ASP.NET messaging system. Exploring the messaging system is not within the scope of this scope. But we will look at the API provided by ASP.NET to send and receive common e-mails on the web.

Following is a small e-mail client application.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="SendMail.aspx.cs" Inherits="SendMail" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    Message to:
    <asp:TextBox ID="txtTo" runat="server"></asp:TextBox>
    <br />
    Message from:
    <asp:TextBox ID="txtFrom" runat="server"></asp:TextBox>
    <br />
    Subject:
    <asp:TextBox ID="txtSubject" runat="server"></asp:TextBox>
    <br />
    Message Body:
    <br />
    <asp:TextBox ID="txtBody" runat="server" Height="171px" TextMode="MultiLine"
      Width="270px"></asp:TextBox>
    <br />
    <asp:Button ID="Btn_SendMail" runat="server" onclick="Btn_SendMail_Click"
      Text="Send Email" />
    <br />
    <br />
    <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
  </div>
  </form>
</body>
```

Example supplement 1-a: Creating an e-mail client

➢Supplement- Messaging System (E-Mailing)

Following is the code behind used to trigger when the "Send Email" button is pressed.

```
using System;
using System.Web.UI.WebControls;
using System.Net.Mail;

public partial class SendMail : System.Web.UI.Page
{
   protected void Btn_SendMail_Click(object sender, EventArgs e)
   {
      MailMessage mailObj = new MailMessage(txtFrom.Text, txtTo.Text, txtSubject.Text, txtBody.Text);
      SmtpClient SMTPServer = new SmtpClient("localhost");
      try
      {  SMTPServer.Send(mailObj); }
      catch (Exception ex)
      {  Label1.Text = ex.ToString(); }
   }
}
```

Example supplement 1-b: The code behind for the e-mail client

Note however that, if you are using the UHCL email server, you might want to set the 'host' and the 'port' as follows.

```
protected void Btn_SendMail_Click(object sender, EventArgs e)
 {
   String msgTo = "Reciever's email";
   String msgSubject = "Message Subject";
   String msgBody = "Message Body";

   MailMessage mailObj = new MailMessage();
   mailObj.Body = msgBody;
   mailObj.From = new MailAddress("Sender's Email", "Name to displayed for sender ex. Outdoor Equipment Team");
   mailObj.To.Add(new MailAddress(msgTo));
   mailObj.Subject = msgSubject;
   mailObj.IsBodyHtml = true;

   SmtpClient SMTPClient = new System.Net.Mail.SmtpClient();
   SMTPClient.Host = "smtp.gmail.com";
   SMTPClient.Credentials = new NetworkCredential("Sender's Email", "Sender's Email Password");
   SMTPClient.EnableSsl = true;
      try
      { SMTPClient.Send(mailObj); }
      catch (Exception)
      { Label1.Text = ex.ToString(); }
   }
```

Example supplement 1-c: Probable code behind for a e-mail client at UHCL

14.0 Web Navigation (Text Book Chapter 13)

➢14.0 Web Navigation

14.0 Web Navigation

You've already learned simple ways to send a website visitor from one page to another. For example, you can add **HTML links** (or HyperLink controls) to your page to let users surf through your site. If you want to perform page navigation in response to another action, you can call the **Response.Redirect()** method or the **Server.Transfer()** method in your code. But in professional web applications, the navigation requirements are more intensive. These applications need a system that allows users to surf through a hierarchy of pages, without forcing you to write the same tedious navigation code in every page.

Fortunately, ASP.NET includes a navigation model that makes it easy to let users surf through your web applications. Before you can use this model, you need to determine the hierarchy of your website—in other words, how pages are logically organized into groups. You then define that structure in a dedicated file and bind that information to specialized navigation controls. Best of all, these navigation controls include nifty widgets such as the TreeView and Menu.

The book chapter(13) teaches you everything you need to know about the ASP.NET's site map model and the navigation controls that work with it. However, exploring this section outside the scope of this course, but encouraged to read this chapter if you wish to know how to implement professional ways of Web Navigation.

Working with Data

15.0 SQL & ADO.NET Fundamentals (Text Book Chapter 14)

15.0 SQL & ADO.NET Fundamentals

15.1 What is a Database

A database is an integrated collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation of data. Mechanisms for such operations **- storing and organizing data in a manageable and consistent format are provided by a Database Management System (DBMS).** Users need not worry about how the things are represented internally and how the data is being accessed.

15.2 What is ADO.NET

ADO.NET [Active Data Objects (ADO). The .NET] provides an API for accessing database systems according to the programs. ADO.NET was created for the .NET Framework and it can be said that they are the new generation of Active Data Objects (ADO). The .NET Framework contains several namespaces and classes, which are devoted to database accesses. Microsoft has created separate namespaces that are optimized for working with different data providers (different types of databases). Generally ADO.NET works well with Microsoft's own databases such as Windows SQL Server.

15.3 How to access a Database?

Accessing a database and communicating with the database is carried via a Database Server. For example, Windows SQL Server 2008, MySQL, Oracle are some popular database servers. Behind the database server is the actual data. The Standard language to 'talk' to a database is 'SQL' (or Standard Query Language) With a database server, the client passes SQL requests as messages to the database server. The results of each SQL command are returned over the network. The server uses its own processing power to find the request data instead of passing all the records back to the client and then getting it find its own data. The result is a much more efficient use of distributed processing power. It is also known as SQL engine.

Tips: Make sure you have installed a Database server in your machine. This will help you to try out some examples of this lecture.

15.4 Relational Databases

A relational database is a logical representation of data that allows relationships between the data to be examined without consideration of the physical structure of the data. Relational Databases are composed of tables. In computers, even a normal file of information can be considered as a Database (Rather it is a data source). But they are non-relational. That is you can store data in series of files, where one does not know the content of the other, which could lead to redundant data repetition and anomalies.

At the end, a database refers to a file that is used to store information in a format that is easily retrieved and manipulated. The most common database files are made up of tables, fields and records. In a relational database 'relations' are built between these tables, fields and records

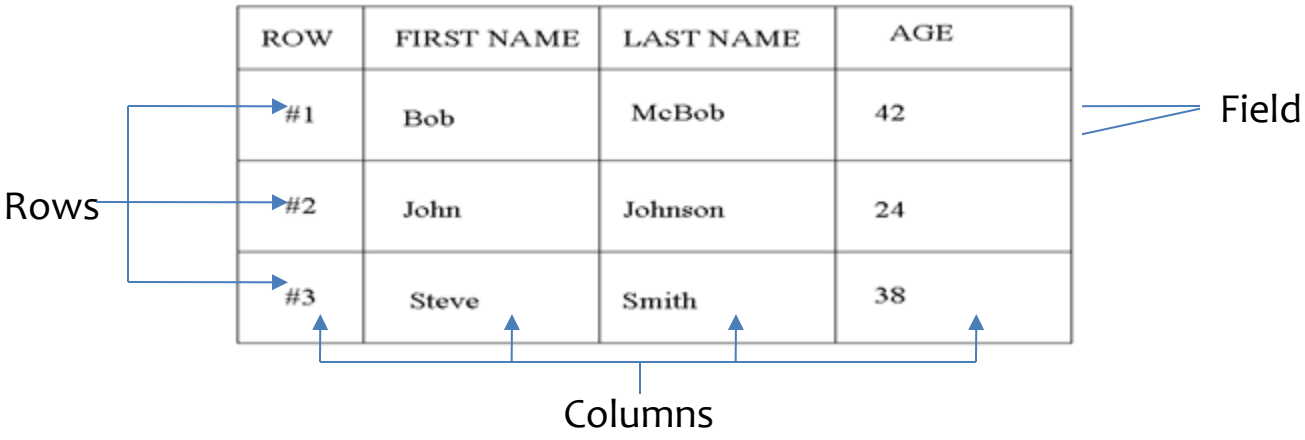


Figure 15.4-1 Example Database Table

15.4.1 Components of a Relational Database

Relational Database consists of the following components…

- A field is a single piece of data.
- A Record is a collection of fields.
- A Table is a collection of records.
- A File is the database in its entirety, i.e. all tables, records and fields. (see Figure 15.4-1)

With the data organized in this way it makes it easier to locate any one piece of information real quick.

Figure 15.4-2 shows a set of relations built amongst tables.

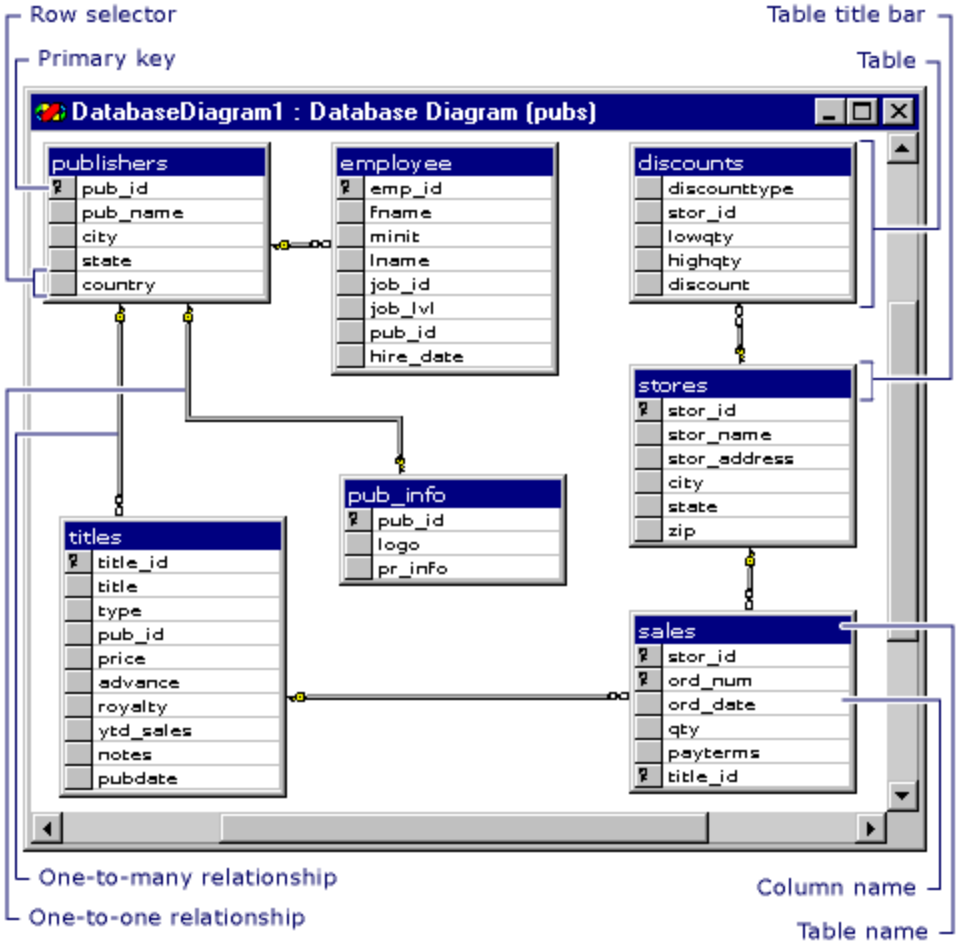


Figure 15.4-2 Relational Tables in a Database

15.5 Structured Query Language (SQL)

15.5.1 Introduction

The operations that you want to carry out on a relational database are expressed in a language that was designed specifically for this purpose, the Structured Query Language – more commonly referred to as SQL. SQL is NOT like a conventional programming language, such as C/C++, C# or Java. SQL is a declarative language – which means the SQL statements tell the SQL server what you want to do but NOT how it should be done. The how is up to the server.

15.5.2 Why SQL?

Structured Query Language (SQL) is accepted internationally as the official standard for relational database access. A major reason for the acceptance of SQL as the relational query language was the move towards client/server architectures that began in the late 1980's.

Some of the 'beauties' of the language are…..

- Easily Readable (like English)
- Syntax is easy to learn
- Construct concepts are very easy to grasp
- SQL always issue commands.

15.5.3 SQL Data Types

The types for data in a relational database are those recognized by the SQL implementation supported by the database engine, and these types will have to be mapped into any language to use the Database. The following chart (Figure 15.5-1) shows the standard SQL data types.

SQL Data Type	Description
CHAR	Fixed length string of characters
VARCHAR	Variable Length String of character
BOOLEAN	Logical Value, true or false
SMALLINT	Small integer value , from −127 to +128
INTEGER	Larger integer value, from −32767 to +32767
NUMERIC	A numeric value with a given precision, which is the number of decimal digits in the number, and a given scale, which is the number of digits after the decimal point. For instance, the value 234567.89 has a precision of 8 and a scale of 2.
FLOAT	Floating point value.
CURRENCY	Stores monetary values
DOUBLE	Higher precision floating point value
DATE	Date

Figure 15.5-1 SQL Data Types

Note : 1. **VARCHAR** is not available in all databases.
2. **NULL** represents the absence of a value of any SQL type of data, but it is not the same as the **null** we have been using in c# or Java.

15.5.4 SQL Statements

Most SQL statements, and certainly the ones we will be using, fall neatly into two categories.

- **Data Definition Language (DDL)** – Statements that are used to describe the tables and the data they contain.
- **Data Manipulation Language (DML)** – Statements that are used to operate on data in a database. DML can be further divided into two groups:

1. **SELECT** statements – statements that return a set of results
2. Everything else – statements that don't return a set of results.

Example: Assume we want to create a **"Book Database".** A **table** named **'books'** need to be crated. To make long stories short, lets assume we have only three columns in the books table names 'isbn', 'title' and 'pubcode' as shown in figure 15.5-1

Column Heading	Description
isbn	ISBN is a globally unique identifier for a book
title	Title of the book
pubcode	Code identifying the publisher of the book

Figure 15.5-1 Specification for 'isbn', 'title' and 'pubcode' columns for the 'book' table

Column Heading	Data Type
isbn	VARCHAR
title	VARCHAR
pubcode	CHAR(8)

Figure 15.5-2 Data types of 'book' table columns

In order to create the table in this example, we will use DDL, which defines syntax for commands such as CREATE TABLE an ALTER TABLE. We will use DDL statements to define the structure of the database. To carry out operations on the database, adding rows to a table or searching the date for instance, we would use DML statements. SQL 'Create' is a typical DDL statement.

15.5.4.1 The CREATE statement.

The SQL 'CREATE' statement is used below to create the database table. Here we assume that the database is created and the following statement will create the table 'books' in the database.

```
CREATE TABLE books (isbn VARCHAR  NOT NULL PRIMARY KEY,
        title  VARCHAR NOT NULL, pubcode CHAR(8));
```

After executing the above statement, we can think of a table created as shown in figure 15.5-1, but empty (no data yet).

isbn	title	pubcode

Figure 15.5-1 – Table 'books' created. But no data yet.

Note : The **NOT NULL PRIMARY KEY** for the **isbn** column tells the database two things.

1. No row of the table is allowed to contain a **NULL** value in this column. Every row in this column must always contain a valid value.
2. Because this column is the primary key field, the database should create a unique index in the **isbn** column. This ensures that there will be no more than one row with any given **isbn ID**.

This greatly assists the database when searching through and ordering records. Think how difficult would be to search for an entry in an encyclopedia without an index of unique values in one of the volumes.

Now that we have created a table, we need to put data into the table. The **SQL INSERT** statement does exactly that.

15.5.4.2 The INSERT statement.

There are three basic parts in the INSERT statement.

1. Define the target table for inserting data
2. Define the columns that will have values assigned
3. Define the values for those columns.

An insert statement begins with the keywords INSERT INTO, followed by the name of the target table , list of names of the columns between parenthesis and then the keyword VALUES and then the actual values between parenthesis.

```
INSERT INTO books (isbn, title, pubcode)
    VALUES ('12-34-543-1534Q', 'C# Programming', 'PRENHALL');
```

So we just added one row of information to the book table.

Variations of the INSERT INTO statements are available. For example we can make the above INSERT INTO a shorter one by the following statement. For larger tables it is good to have the columns names too because it acts as a "Navigator" for the builder.

INSERT INTO books
VALUES ('31-99-564-6554L', 'Beginning Linux Programming', 'WROX');

will have the same effect as

INSERT INTO books (isbn, title, pubcode)
VALUES ('31-99-564-6554L', 'Beginning Linux Programming', 'WROX');

After several more insertions, we can think of a table like the one shown in Figure 16.5. Now that we have entered data into the books table, we need to retrieve data from the Database. We use the 'SELECT" statement.

isbn	title	pubcode
12-34-543-1534Q	C# Programming	PRENHALL
31-99-564-6554L	Beginning Linux Programming	WROX
78-23-987-561-J	Java Script Professional	PRENHALL
98-32-198-377-M	Managing Risk	PEARSON

Figure 15.5-1 – A Snap of the 'books' table after data insertion

15.5.4.3 The SELECT statement

You use the SELECT statement to retrieve information from a database. There are four parts to an SQL SELECT statement:

1. Defining what you want to retrieve
2. Defining where you want to get it from
3. Defining the conditions for retrieval – joining tables, and record filtering
4. Defining the order in which you want to see the data.

For all these we use the **SELECT** statement. Unsurprisingly, the first keyword in the SELECT statement is SELECT. **Then you have several variations to retrieve data the way you want.** SELECT statement tells the database that we need to get some data back in **the form of a result set or a record set.** A result set is just a table of data with fixed numbers of columns and rows – this is some subset of data from a database table generated as a result of the SELECT statement.

The **FROM** clause is almost always the companion of the **SELECT** statement. For example

```
SELECT isbn, title FROM books
```

means to select columns **isbn** and **title** from the **books table.** With this statement **we will receive a result set like this** (Figure 15.5-2)

isbn	title
12-34-543-1534Q	C# Programming
31-99-564-6554L	Beginning Linux Programming
78-23-987-561-J	Java Script Professional
98-32-198-377-M	Managing Risk

Figure 15.5-2– Using 'SELECT' to retrieve 'isbn' & 'title' data

When a Result Set is retrieved from the database, each Result Set column has a label that, by default, is derived from the column names in the **SELECT** statement.

If you want to select all columns from a table, you can use the *. For example

```
SELECT * FROM books
```

means select all columns from the books table and put into a result set (Figure 15.5-3).

isbn	title	pubcode
12-34-543-1534Q	C# Programming	PRENHALL
31-99-564-6554L	Beginning Linux Programming	WROX
78-23-987-561-J	Java Script Professional	PRENHALL
98-32-198-377-M	Managing Risk	PEARSON

Figure 15.5-3– Result after using the wild card '*' with 'SELECT'

15.5.4.4 The WHERE clause

WHERE is optimization clause that you can use with many statements. A common occurrence is that the **WHERE** to use with the **SELECT** clause.

Consider the table given in Figure 15.5-1 and let's assume we apply the following statement.

```
SELECT * FROM books WHERE pubcode='WROX'
```

The result shall be the following (Figure 15.5-4)

isbn	title	pubcode
31-99-564-6554L	Beginning Linux Programming	WROX

Figure 15.5-4– Using 'SELECT' with WHERE caluse

There are several others variations the WHERE can be used. But we will not explore all of them in this course. Always you can look into a good SQL documentation set to see other variations of the WHERE clause.

15.5.4.5 The LIKE and NOT LIKE Search Helpers

The **LIKE** and **NOT LIKE** have two search helper symbols. The underscore _ character that looks for one character and the percentage % character that looks for zero or more characters.

```
SELECT * FROM books WHERE pubcode LIKE 'P%'
```

Will produce the table shown in Figure 15.5-5

isbn	title	pubcode
12-34-543-1534Q	C# Programming	PRENHALL
78-23-987-561-J	Java Script Professional	PRENHALL
98-32-198-377-M	Managing Risk	PEARSON

Figure 15.5-5– Result after using the LIKE 'P%' with 'SELECT'

15.5.4.6 ORDER BY clause

Another very helpful clause is the ORDER BY clause. ORDER BY – as it says ORDERS the result in ascending or in descending order if the field to order is numeric. By default it orders the result set in ascending order. Text fields are by default ordered in alphabetical order.

SELECT * FROM books ORDER BY title

Will produce the table shown in Figure 15.5-6

isbn	title	pubcode
31-99-564-6554L	Beginning Linux Programming	WROX
12-34-543-1534Q	C# Programming	PRENHALL
78-23-987-561-J	Java Script Professional	PRENHALL
98-32-198-377-M	Managing Risk	PEARSON

Figure 15.5-6– Using 'SELECT' with ORDER BY clause

15.5.4.7 The UPDATE statement

Update statements provide a way of modifying existing data in a table. Update statements are constructed in a similar way to SELECT statements. You first start with the UPDATE keyword, followed by the name of the table you wish to modify and then followed by the SET and WHERE clauses followed by the new values of columns.

For example,

UPDATE books SET pubcode = 'ADDISON'
WHERE isbn = '31-99-564-6554L'

will update the book record to reflect a change in the publisher code for the book with ISBN number 31-99-564-6554L.

UPDATE statements do not return a result set, as they merely modify data in the database.

15.5.4.8 DELETE Statement

DELETE statements provide a way of deleting a particular rows from tables in a database. Delete statements consist of the **DELETE** keyword, a **FROM** clause and a **WHERE** clause. For example,

```
DELETE FROM books WHERE isbn = '31-99-564-6554L'
```

The above statement **deletes the record in the books table with the ISBN value 31-99-564-6554L.** In the case of the books table, there can only be one row with this value, since its primary key is the ISBN.

By now, you should have a reasonably clear idea of:

- The way SQL is constructed
- How to read SQL statements
- How to construct basic SQL statements

Note however that the SQL statements shown in the above examples are the very basic ones. If you want to get some more 'real stuff', you might as well look into a Databases book or reference manual.

Now that we have seen the very basics of SQL, we will now try to understand how ADO.NET (next section) can be used to do all what we did in the previous pages via a Web Application.

15.6 ActiveX Data Objects.NET (ADO.NET)

15.6.1 Introduction

ADO(ActiveX Data Objects).NET is the data access component for the .NET Framework. ADO.NET leverages the power of XML to provide disconnected access to data. **ADO.NET is made of a set of classes that are used for connecting to a database, providing access to relational data, XML, and application data, and retrieving results.** ADO.NET data providers contain classes that represent the provider's **Connection**, **Command**, **DataAdapter** and **DataReader** objects (among others).

ADO.NET makes it possible a client to establish a connection with a data source, send queries and update statements to the data source, and process the results.

15.6.2 ADO.NET Components

ADO.NET has several key components : **Application Component**, **DataSet**, **DataReader** and **ADO.NET Data Provider**. The following Figure 15.6-1 shows a scenario how the application and the ADO.NET data provider are linked.

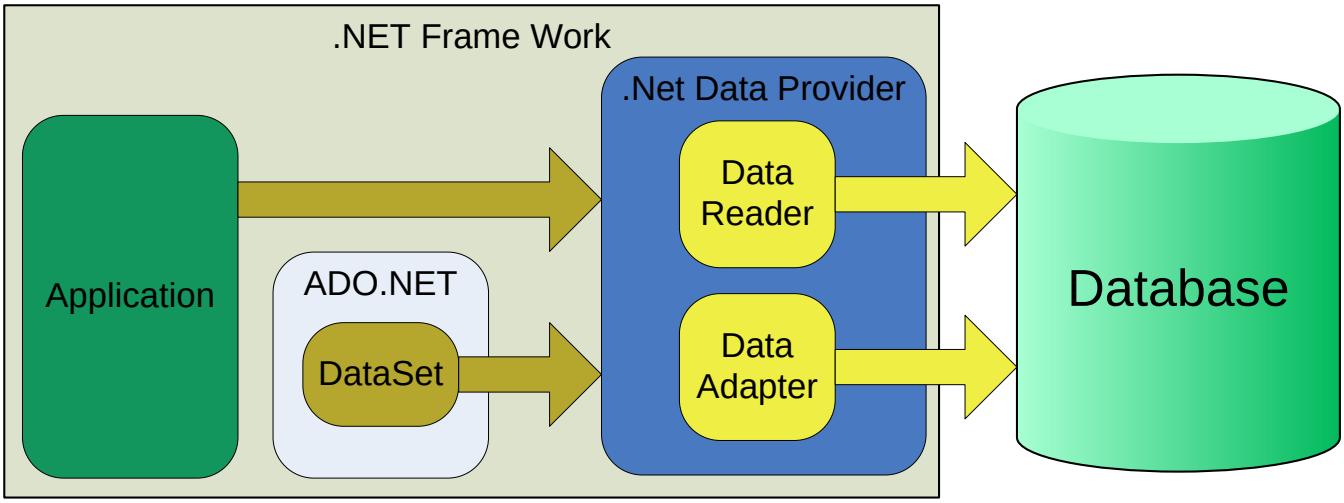


Figure 15.6-1– ADO.Net Components

- **Application component :** This component Processes and calls ADO.NET functions to submit SQL statements and retrieve results.

➢15.0 SQL and ADO.NET Fundamentals

▪**DataSet Component :** As in-memory cache of data which functions like a disconnected XML data cache. The overall functions of the DataSet closely recall those of an in-memory database. The DataSet is designed to run in the application space wherever the logic requires local data. This helps increase scalability of systems by reducing load on the major database backends and enabling local processing of data across whatever tier the application requires. For flexibility, the DataSet provides XML and relational interfaces of the data to the developer.

▪ **DataReader Component**
which provides a direct, read-only SQL interface to the backend. The DataReader is a component of the data provider.

▪ **ADO.NET Data Provider Component**
This component connects an ADO.NET application to the backend data store. The data provider comprises the Connection, Command, DataReader and DataAdapter objects. The data provider supplies connection information through the Connection object.

Data Providers : Some of the Data Providers used in the industry today are listed below:

SQL Server Provider	: Provides optimized access to a SQL Server Database.
OLE DB Provider	: Provides access to any data source that has an OLE DB driver
Oracle Provider	: Provides optimized access to an Oracle Database
ODBC Provider	: Provides access to any data source that has an ODBC (Open Database Connectivity) driver.

15.6.3 AOD.NET Namespaces

ADO.NET managed code components are scattered into several namespaces in the .NET Class library. Together the classes in these namespaces provides the functionality of the ADO.NET. The table 15.6-2 shows the namespaces.

Namespace	Purpose/Description
System.Data	The System.Data namespace provides access to classes that represent the ADO.NET architecture. ADO.NET lets you build components that efficiently manage data from multiple data sources.
System.Data.Common	The System.Data.Common namespace contains classes shared by the .NET Framework data providers.
System.Data.OleDb	The System.Data.OleDb namespace is the .NET Framework Data Provider for OLE DB. The .NET Framework Data Provider for OLE DB describes a collection of classes used to access an OLE DB data source in the managed space. Using the OleDbDataAdapter, you can fill a memory- resident DataSet, which you can use to query and update the data source.
System.Data.SqlClient	The System.Data.SqlClient namespace is the.NET Framework Data Provider for SQL Server. The.NET Framework Data Provider for SQL Server describes a collection of classes used to access a SQL Server database in the managed space. Using the SqlDataAdapter, you can fill a memory-resident DataSet that you can use to query and update the database.
System.Data.SqlTypes	The System.Data.SqlTypes namespace provides classes for native data types in SQL Server. These classes provide a safer, faster alternative to the data types provided by the .NET Framework common language runtime (CLR). Using the classes in this namespace helps prevent type conversion errors caused by loss of precision. Because other data types are converted to and from SqlTypes behind the scenes, explicitly creating and using objects within this namespace also yields faster code.

Figure 15.6-2– ADO.NET Namespaces and their Descriptions

15.6.4 AOD.NET Data Provider Classes

A data provider in the .NET Framework serves as a bridge between an application and a data source. A data provider is used to retrieve data from a data source and to reconcile changes to that data back to the data source. The following table (Figure 15.6-3) lists the .NET Framework data providers that are included in the .NET Framework.

.NET Framework data provider	Description
.NET Framework Data Provider for SQL Server	For Microsoft® SQL Server™ version 7.0 or later.
.NET Framework Data Provider for OLE DB	For data sources exposed using OLE DB.
.NET Framework Data Provider for ODBC	For data sources exposed using ODBC. Note The .NET Framework Data Provider for ODBC is not included in the .NET Framework version 1.0. If you require the .NET Framework Data Provider for ODBC and are using the .NET Framework version 1.0, you can download the .NET Framework Data Provider for ODBC at http://msdn.microsoft.com/downloads. The namespace for the downloaded .NET Framework Data Provider for ODBC is Microsoft.Data.Odbc.
.NET Framework Data Provider for Oracle	For Oracle data sources. The .NET Framework Data Provider for Oracle supports Oracle client software version 8.1.7 and later. Note The .NET Framework Data Provider for Oracle is not included in the .NET Framework version 1.0. If you require the .NET Framework Data Provider for Oracle and are using the .NET Framework version 1.0, you can download the .NET Framework Data Provider for Oracle at http://msdn.microsoft.com/downloads.

Figure 15.6-3– ADO.NET Data Providers

The **Connection, Command, DataReader, and DataAdapter** objects represent the core elements of the .NET Framework's data provider model. The following table (Figure 15.6-4) describes these objects.

Object	Description
Connection	Establishes a connection to a specific data source.
Command	Executes a command against a data source.
DataReader	Reads a forward-only, read-only stream of data from a data source.
DataAdapter	Populates a DataSet and resolves updates with the data source.

Figure 15.6-4– ADO.NET Data Provider Classes

15.6.5 ADO.NET Data Provider Classes per Provider

NET framework has defined a set of classes for these services (Connection, Command, DataReader, and DataAdapter) for each provider. The table in Figure 15.6-7 lists these classes.

	SQL Server Data Provider	OLE DB Data Provider	Oracle Data Provider	ODBC Data Provider
Connection	SqlConnection	OleDbConnection	OracleConnection	OleDbConnection
Command	SqlCommand	OleDbCommand	OracleCommand	OleDbCommand
DataReader	SqlDataReader	OleDbDataReader	OracleDataReader	OleDbDataReader
DataAdapter	SqlDataAdapter	OleDbDataAdapter	OracleDataAdapter	OleDbDataAdapter

Figure 15.6-5– ADO.NET Data Provider Classes per Provider

15.6.6 Direct Data Access Vs. Caching

A primary choice is whether you want to cache records in a dataset or whether you want to access the database directly and read records using a data reader. For some database operations, such as creating and editing database structures, you cannot use a dataset. For example, if you want to create a new database table from within your application, you cannot do so using a dataset; instead, you would execute a data command. For general data-access scenarios, however, you can often choose between storing records in a disconnected dataset and working directly with the database using data commands.

15.6.6.1Cashing

If you decide to save a dataset between round trips, you must decide where to keep it. This issue is the standard one of state maintenance in Web Forms pages — where do you store information you want to preserve between round trips? For information about saving values, see Web Forms State Management. You have two options:

1. **On the server:** save the dataset **in Session state, Application state**, or **using a cache**.
2. **On the client:** that is, in the page — save the dataset using **view state** or by putting the data into your own **hidden field.** (View state is also implemented using a hidden field.)

15.6.6.2 Direct Data Access Steps

To query information with simple data access, follow these steps.

1. **Create Connection**, Command and DataReader objects.
2. **Use the DataReader** to retrieve information from the database and display it in a control on a web worm.
3. **Close your connection**.
4. **Send the page to the user**. At this point, the information your user sees and the information in the database no longer has not connection and all the ADO.NET objects have been destroyed.

To add or update information, follow these steps

1.Create **new Connection** and Command objects.

2.Execute the command (with appropriate SQL statement).

15.6.6.3 Creating a Connection

The first thing you will need to do when interacting with a data base is to create a connection. The connection tells the rest of the ADO.NET code which database it is talking to. It manages all of the low level logic associated with the specific database protocols. This makes it easy for you because the most work you will have to do in code is instantiate the connection object, open the connection, and then close the connection when you are done. Because of the way that other classes in ADO.NET are built, sometimes you don't even have to do that much work.

Although working with connections is very easy in ADO.NET, you need to understand connections in order to make the right decisions when coding your data access routines. Understand that a connection is a valuable resource. Sure, if you have a stand-alone client application that works on a single database one one machine, you probably don't care about this. However, think about an enterprise application where hundreds of users throughout a Company are accessing the same database. Each connection represents overhead and there can only be a finite amount of them. To look at a more extreme case, consider a Web site that is being hit with hundreds of thousands of hits a day. Applications that grab connections and don't let them go can have seriously negative impacts on performance and scalability.

15.6.6.3 Creating a SQL Connection

Creating a SqlConnection Object

A SqlConnection is an object, just like any other C# object. Most of the time, you just declare and instantiate the SqlConnection all at the same time, as shown below:

```
SqlConnection conn = new SqlConnection(
   "Data Source=(local);Initial Catalog=Northwind;Integrated
   Security=SSPI");
```

The **SqlConnection** object instantiated above uses a constructor with a single argument of type string. This argument is called a connection string. Table in Figure 15.6.6 describes common parts of a connection string.

ADO.NET Connection Strings contain certain key/value pairs for specifying how to make a database connection. They include the location, name of the database, and security credentials

Connection String Parameter Name	Description
Data Source	Identifies the server. Could be local machine, machine domain name, or IP Address.
Initial Catalog	Database name.
Integrated Security	Set to SSPI to make connection with user's Windows login. [*SSPI stands for the SecuritySupport Provider Interface, which helps a client and server establish and maintain a secure channel, providing confidentiality,integrity, and authentication*]
User ID	Name of user conFigured in SQL Server.
Password	Password matching SQL Server User ID.

Figure 15.6-6– Connection String Components

Integrated Security is secure when you are on a single machine doing development. However, you will often want to specify security based on a SQL Server User ID with permissions set specifically for the application you are using. The following shows a connection string, using the User ID and Password parameters:

```
SqlConnection conn = new SqlConnection(
"Data Source=DatabaseServer;Initial Catalog=Northwind;User
   ID=YourUserID;Password=YourPassword");
```

➢15.0 SQL and ADO.NET Fundamentals

UHCL Specific Connection String Components:

UHCL Specific Variables

Data Source =dcm.uhcl.edu (DB Server Name)
Initial Catalog = In your information Slip (Database Name)
User Id = In your information Slip (Your User ID)
Password = your password (Your Password)
Asynchronous Processing=true

[Note : Your personal parameters are were sent via email to you in an information slip early in the semester.]

Example:

```
SqlConnection conn =
new SqlConnection(Data Source=dcm.uhcl.edu;Initial
Catalog=c432013fa01patir;User id= c432013fl;
Password= 9999999;Asynchronous Processing=true);
```

Set the connection setting in web.config file like this using the Information Slip You Received at the start of the semester.

```
<connectionStrings>
   <add name="made up name for this database"
   connectionString="User id= c432013fl;
   password= 9999999;
   Data Source =dcm.uhcl.edu;
   Initial Catalog =c432013fl;
   providerName="System.Data.SqlClient";
   connection timeout=30"/>
</connectionStrings>
```

Notice how the Data Source is set to DatabaseServer to indicate that you can identify a database located on a different machine, over a LAN, or over the Internet. Additionally, User ID and Password replace the Integrated security parameter.

15.6.6.4 Using a SqlConnection:

The purpose of creating a SqlConnection object is so you can enable other ADO.NET code to work with a database. Other ADO.NET objects, such as a SqlCommand and a SqlDataAdapter take a connection object as a parameter. The sequence ofoperations occurring in the lifetime of a SqlConnection are as follows:

-Instantiate the SqlConnection.
-Open the connection.
-Pass the connection to other ADO.NET objects.
-Perform database operations with the other ADO.NET objects.
-Close the connection.

We've already seen how to instantiate a SqlConnection. The rest of the steps, opening, passing, using, and closing are shown in Example 15.6.1

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace CreatingADBTable
{
  public class CreateATable
  {
    public static void Main(string[] args)
    {
      SqlConnection connectionToServer =
                new SqlConnection(
                  "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");

      string theStatement = "CREATE TABLE Books" +
                          "(isbn CHAR(15), " +
                          "title CHAR(50), " +
                          "pubcode CHAR(1))";

      SqlCommand sqlCommand = new SqlCommand(theStatement, connectionToServer);
      try
      {
       connectionToServer.Open();

       sqlCommand.BeginExecuteNonQuery();

       String insertString = "INSERT INTO Books" +
                           "(isbn, title, pubcode" +
                           "VALUES ('12-34-543-1534Q', 'Java Programming', 'PRENHALL')";
       sqlCommand.CommandText = insertString;
       sqlCommand.BeginExecuteNonQuery();

       insertString = "INSERT INTO Books" +
                    "(isbn, title,pubcode" +
                    "VALUES ('31-99-564-6554L', 'Beginning Linux Programming', 'WROX')";
       sqlCommand.CommandText = insertString;
       sqlCommand.BeginExecuteNonQuery();

       insertString = "INSERT INTO Books" +
                    "(isbn, title,pubcode" +
                    "VALUES ('78-23-987-561-J','Java Script Professional', 'PRENHALL')";
       sqlCommand.CommandText = insertString;
       sqlCommand.BeginExecuteNonQuery();

       insertString = "INSERT INTO Books" +
                    "(isbn, title,pubcode" +
                    "VALUES ('98-32-198-377-M', 'Managing Risk', 'PEARSON')";
       sqlCommand.CommandText = insertString;
       sqlCommand.BeginExecuteNonQuery();

      }
      finally
      {
       connectionToServer.Close();
      }
    }
  }
}
```

Figure 15.6-1– Creating a Table and Inserting Data

15.6.6.5 Using DataReader

A **SqlDataReader** is a type that is good for reading data in the most efficient manner possible. You can *not* use it for writing data. **SqlDataReaders** are often described as fast-forward firehose-like streams of data.

You can read from **SqlDataReader** objects in a forward-only sequential manner. Once you've read some data, you must save it because you will not be able to go back and read it again.

The forward only design of the **SqlDataReader** is what enables it to be fast. It doesn't have overhead associated with traversing the data or writing it back to the data source. Therefore, if your only requirement for a group of data is for reading one time and you want the fastest method possible, the **SqlDataReader** is the best choice. Also, if the amount of data you need to read is larger than what you would prefer to hold in memory beyond a single call, then the streaming behavior of the **SqlDataReader** would be a good choice.

15.6.6.5 Creating a SqlDataReader Object

Getting an instance of a **SqlDataReader** is a little different than the way you instantiate other ADO.NET objects. You must call **ExecuteReader** on a command object, like this:

```
SqlDataReader rdr = cmd.ExecuteReader();
```

The **ExecuteReader** method of the **SqlCommand** object, cmd , returns a **SqlDataReader** instance. Creating a **SqlDataReader** with the new operator doesn't do anything for you. As you learned in previous lessons, the **SqlCommand** object references the connection and the SQL statement necessary for the **SqlDataReader** to obtain data.

15.6.6.6 Reading Data

previous lessons contained code that used a **SqlDataReader**, but the discussion was delayed so we could focus on the specific subject of that particular lesson. This lesson builds from what you've seen and explains how to use the **SqlDataReader**.

➢15.0 SQL and ADO.NET Fundamentals

As explained earlier, the **SqlDataReader** returns data via a sequential stream. To read this data, you must pull data from a table row-by-row. Once a row has been read, the previous row is no longer available. To read that row again, you would have to create a new instance of the **SqlDataReader** and read through the data stream again.

The typical method of reading from the data stream returned by the **SqlDataReader** is to iterate through each row with a while loop. The following code in example 15.6-2a & 15.6-2b shows how to accomplish this.

```
while (rdr.Read())
{
  // get the results of each column
  string isbn = (string)rdr["isbn"];
  string title = (string)rdr["title"];
  string pubcode    = (string)rdr["pubcode"];

  // print out the results
  Console.WriteLine(isbn);
  Console.WriteLine(title);
  Console.WriteLine(pubcode);
  Console.WriteLine();
}
```

Example 15.6-1a– Iterating Data

```
while (rdr.Read())
{
 Book aBook = new Book();
  // get the results of each column
  aBook.setIsbn((string)rdr["isbn"]);
  aBook.SetTitle((string)rdr["title"]);
  aBook.SetPublicationCode((string)rdr["pubcode"]);
}
```

Example15.6-1b– Iterating Data

If you had already created a class named say Book with setXXXX methods such as SetIsbn(). SetTitle and SetPublicationCode(), then you can simple 'fill' and object of Book type. See snippet in Figure 15.6.2b.

See a full example next page.

In example 15.6-3a a Book class is first created. Then a database table is accessed, read and fills a Book object.

```
public class Book {
  string isbn, title, pubcode;
  public void SetIsbn(string isbn) { this.isbn = isbn; }
  public void SetTitle(string title) { this.title = title; }
  public void SetPublicationCode(string pubcode) { this.pubcode = pubcode;  }
}
```

Example 15.6-2a– A Book class

```
using System;
using System.Data;
using System.Data.SqlClient;

public class ReadingBookTable {
  public static void Main(string[] args) {
    SqlConnection connectionToServer = new SqlConnection( "Data Source=(local);Initial
            Catalog=Northwind;Integrated Security=SSPI");

    string theStatement = "SELECT * from Books";

    SqlCommand sqlCommand = new SqlCommand(theStatement, connectionToServer);
    try {
          connectionToServer.Open();

          SqlDataReader sqlReader = sqlCommand.ExecuteReader();
          while (sqlReader.Read()) {
                                        Book aBook = new Book();

                                        // get the results of each column
                                        aBook.SetIsbn((string)sqlReader["isbn"]);
                                        aBook.SetTitle((string)sqlReader["title"]);
                                        aBook.SetPublicationCode((string)sqlReader["pubcode"]);
                                      }
       }
    finally {   connectionToServer.Close();     }
  }
}
```

Example15.6-2b– Iterate table and 'fill' Book objects

15.6.7 Updating a Record in a Table

The **UPDATE** command indicates the records to be modified and the modification to be made. The return value of the **ExecuteNonQuery()** indicates the number of records changes. See the example below.

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace UpdatingATable
{
  public class UpdateATable
  {
    public static void Main(string[] args)
    {
      SqlConnection connectionToServer =
                new SqlConnection(
                  "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");

      string theStatement = "UPDATE books SET pubcode =
                                              'ADDISON' WHERE isbn = '31-99-564-6554L'";

      SqlCommand sqlCommand = new SqlCommand( theStatement, connectionToServer);
      try
      {
       connectionToServer.Open();

       sqlCommand.ExecuteNonQuery();
      }
      finally
      {
       connectionToServer.Close();
      }
    }
  }
}
```

Example 15.6-3 Updating a 'books' record

15.6.8 Adding a Record to a Table

The **INSERT INTO** command indicates the records to be added to the table. **ExecuteNonQuery()** indicates the number of records changes. See the example below.

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace InsertIntoATable
{
  public class InsertoToATable
  {
    public static void Main(string[] args)
    {
      SqlConnection connectionToServer =
                new SqlConnection(
                  "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");

      string theStatement = "INSERT INTO books (isbn, title, " +
                          "pubcode)VALUES ('12-34-543-1534Q', 'C# Programming', 'PRENHALL')";

      SqlCommand sqlCommand = new SqlCommand( theStatement, connectionToServer);
      try
      {
       connectionToServer.Open();

       sqlCommand.ExecuteNonQuery();
      }
      finally
      {
       connectionToServer.Close();
      }
    }
  }
}
```

Example 15.6-4 Adding records to 'books' table

16.0 C# I/O System
Files and Streams
(Text Book Chapter 15)

16.0 C# I/O System – Files and Streams

16.1 Introduction

The C# input/output system framework is designed to be flexible and easily configurable. All C# I/O classes are defined in the **System.IO** namespace. The classes and interfaces defined in the **System.IO** namespace is shown below.

Directory	FileAccess	FileShare	Path	StreamReader	TextReader
DirectoryNotFoundException	FileLoadException	FileStream	PathTooLongException	StreamWriter	TextWriter
DirectoryInfo	FileMode	IOException	SeekOrigin	StringReader	
EndOfStreamException	File	FileNotFoundException	MemoryStream	Stream	StringWriter

Figure 16.1-1 C# I/O System Namespaces

16.2 C# Stream Types

C# supports two types of input/output (I/O) – Stream-IO and Radom Access-IO.

Stream I/O : A stream is a sequence of bytes. Stream-based I/O supports reading or writing data sequentially, that is, reading or writing data successively in one direction. A stream may be opened for reading or writing, but not reading and writing.

Random Access I/O : Random access I/O supports reading and writing data at any position of a stream. A random access stream may be opened for both reading and writing.

16.2.1 Stream IO

There are **two kinds of streams**. **Byte streams** and **character streams**. Byte streams support reading data and writing data of any type, including strings, in the binary format. Character streams support reading and writing of text, using locale-dependent character encodings.

16.2.1.1 Byte Streams

The most primitive I/O capabilities are declared in two classes: **StreamReader** and **StreamWriter**. They support reading and writing of a single byte and characters.

StreamReader : The **StreamReader** class is designed for character input in a particular Encoding, whereas subclasses of Stream are designed for byte input and output. [Note: StreamReader defaults to UTF-8 encoding unless specified otherwise, instead of defaulting to the ANSI code page for the current system. UTF-8 handles Unicode characters correctly and provides consistent results on localized versions of the operating system.

When reading from a Stream, it is more efficient to use a buffer that is the same size as the internal buffer of the stream. StreamReader hierarchy is shown in Figure 16.2-1

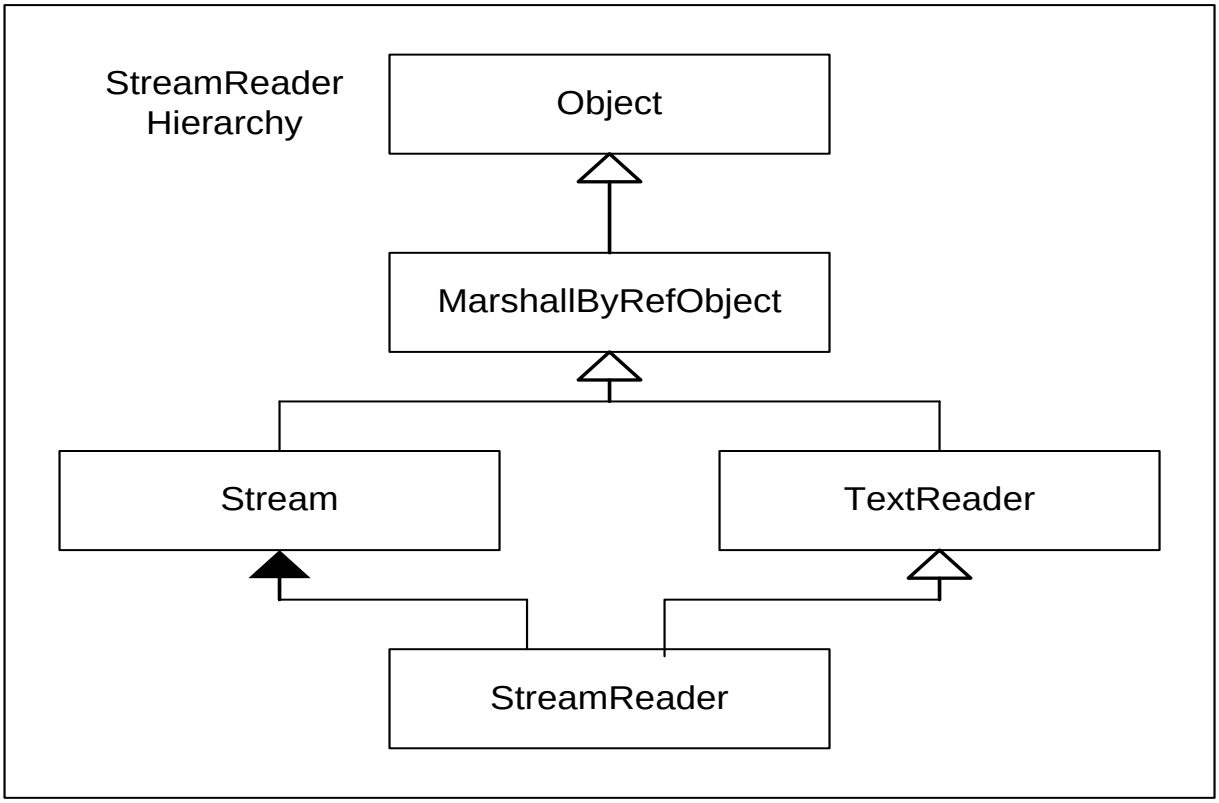


Figure 16.2-2 aStreamReader class hierarchy

Some of the common methods in the **StreamReader** Class is class is shown in Figure 16.2-2. The rest can be viewed here (StreamReader class).

➢16.0 C# I/O System (Files and Streams)

Method Name	Description
Close	Overridden. Closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
DiscardBufferedData	Allows a StreamReader object to discard its current data.
Dispose	Overloaded.
Equals	Overloaded. Determines whether two Object instances are equal. (Inherited from Object.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
Peek	Overridden. Returns the next available character but does not consume it.
Read	Overloaded. Overridden. Reads the next character or next set of characters from the input stream.
ReadBlock	Reads a maximum of count characters from the current stream and writes the data to buffer, beginning at index. (Inherited from TextReader.)
ReadLine	Overridden. Reads a line of characters from the current stream and returns the data as a string.
ReadToEnd	Overridden. Reads the stream from the current position to the end of the stream.
Synchronized	Creates a thread-safe wrapper around the specified TextReader. (Inherited from TextReader.)
ToString	Returns a String that represents the current Object. (Inherited from Object.)

Figure 16.2-2b StreamReader class methods

StreamWriter: The StreamWriter class is designed for character output in a particular Encoding, whereas subclasses of Stream are designed for byte input and output. StreamWriter defaults to using an instance of UTF8Encoding unless specified otherwise. This instance of UTF8Encoding is constructed such that the System.Text.Encoding.GetPreamble method returns the Unicode byte order mark written in UTF-8. The preamble of the encoding is added to a stream when you are not appending to an existing stream. This means any text file you create with StreamWriter has three byte order marks at its beginning. UTF-8 handles all Unicode characters correctly and gives consistent results on localized versions of the operating system. StreamWriter hierarchy is shown in Figure 16.2-3a

➢16.0 C# I/O System (Files and Streams)

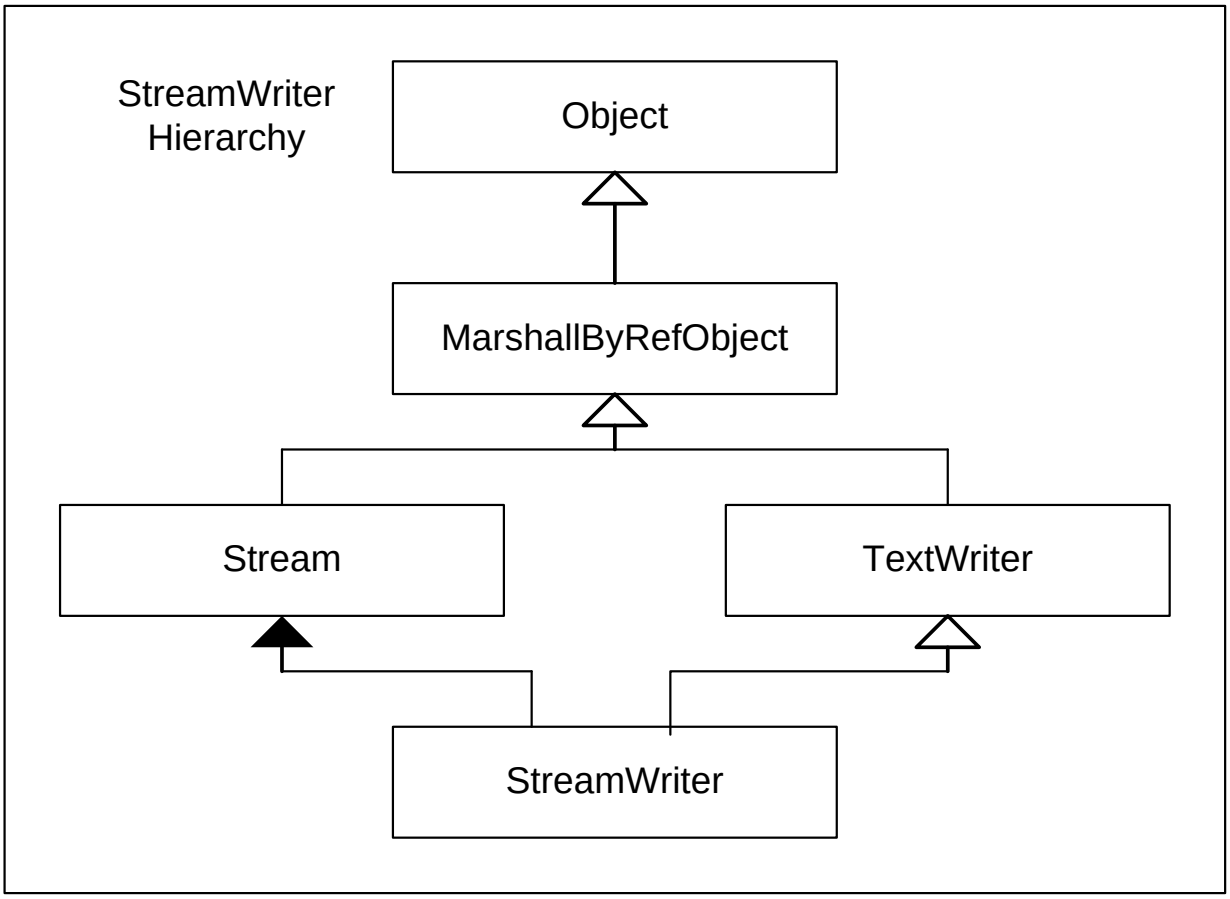


Figure 16.2-3aStreamWriter class Hierarchy

StreamWriter is a large class. It is not worth writing down the method in this class note. Some of the public method of the StreamWriter class is listed in next page Figure 16.2-3b. However, class's constructors, methods etc. can be viewed here (StreamWriter Class) .

The example 16.2.1 shows how both StreamWriter and StreamReader classes can be used along with the classes available for Directory information access. The result of the program is shown on Figure 16.2.4

Method Name	Description
Close	Overridden. Closes the current StreamWriter and the underlying stream.
CreateObjRef(inherited from MarshalByRefObject)	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object.
Equals (inherited from Object)	Supported by the .NET Compact Framework. Overloaded. Determines whether two Object instances are equal.
Flush	Supported by the .NET Compact Framework. Overridden. Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
GetHashCode (inherited from Object)	Supported by the .NET Compact Framework. Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetLifetimeService(inherited from MarshalByRefObject)	Retrieves the current lifetime service object that controls the lifetime policy for this instance.
GetType (inherited from Object)	Supported by the .NET Compact Framework. Gets the Type of the current instance.
InitializeLifetimeService (inherited from MarshalByRefObject)	Obtains a lifetime service object to control the lifetime policy for this instance.
ToString (inherited from Object)	Supported by the .NET Compact Framework. Returns a String that represents the current Object.
Write	Supported by the .NET Compact Framework. Overloaded. Overridden. Writes to the stream.
WriteLine(inherited from TextWriter)	Supported by the .NET Compact Framework. Overloaded. Writes some data as specified by the overloaded parameters, followed by a line terminator.

Figure 16.2-3b Public method of the StreamWriter class

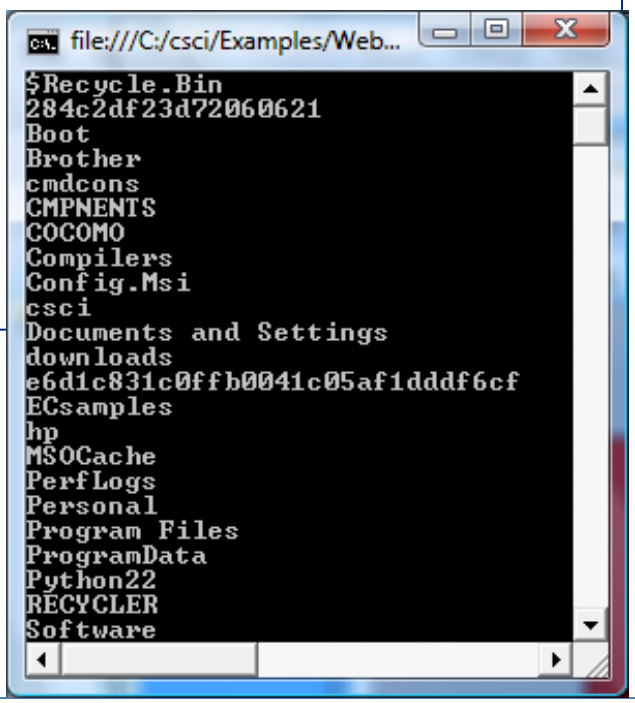
The example 16.2.1 shows how both StreamWriter and StreamReader classes can be used along with the classes available for Directory information access. The result of the program is shown on Figure 16.2.4

➢16.0 C# I/O System (Files and Streams)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace StreamReadWrite
{
 class Program
 {
  static void Main(string[] args)
  {// Get the directories currently on the C drive.
   DirectoryInfo[] cDirs = new DirectoryInfo(@"c:\").GetDirectories();
   string fileName =
     @"C:\csci\Examples\WebApplicationDevelopment\CDriveDirs.txt";

   // Write each directory name to a file.
   using (StreamWriter sw = new StreamWriter(fileName ))
   {
    foreach (DirectoryInfo dir in cDirs)
    { sw.WriteLine(dir.Name);   }
   }

    // Read and show each line from the file.
   string line = "";
   using (StreamReader sr = new StreamReader("CDriveDirs.txt"))
   {
    while ((line = sr.ReadLine()) != null)
    {  Console.WriteLine(line); }
   }
   Console.ReadLine();
  }
 }
}
```

Example 16.2-1StreamReader and StreamWriter usage

Figure 16.2-4



16.2.1.1.1 The File Class

File Class' methods can be used for shortcuts for Reading and Writing.

The File class provides several nice and quick methods to make life much easier. The File class provides two methods WriteAllLines and ReadAllLines that let you write an entire array of strings into a file and then read the entire file back into a string array. The Example 16.2-2 below shows the previous example (Example 16.2-1) now written in a different way but resulting in same. The result of the program is shown on Figure 16.2-2

```
namespace TestFileClass
{
 class Program
 {
  static void Main(string[] args)
  {// Get the directories currently on the C drive.
   string fileName =
     @"C:\csci\Examples\WebApplicationDevelopment\CDriveDirs.txt";

   DirectoryInfo[] cDirs = new DirectoryInfo(@"c:\").GetDirectories();
   string[] dirs = new string[cDirs.Length];
   for (int nextDir = 0; nextDir < dirs.Length; nextDir++)
    {  dirs[nextDir] = cDirs[nextDir].ToString();  }

   // Now write the whole array in one shot.
   File.WriteAllLines(fileName, dirs);

   // Now get written dir list back from file in one shot.
   string [] getBackDirs = File.ReadAllLines(fileName);

   foreach(string dir in getBackDirs)
         Console.WriteLine(dir);

   Console.ReadLine();
  }
}
```

Example 16.2-2StreamReader and StreamWriter usage with File class

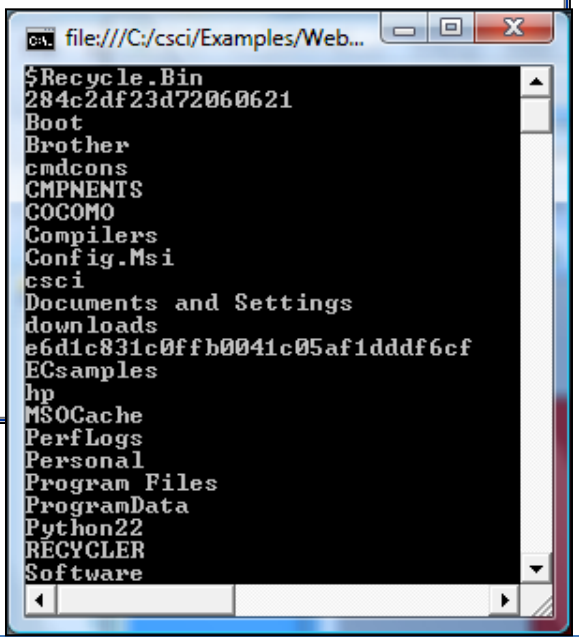


Figure 16.2-5

➢16.0 C# I/O System (Files and Streams)

Some of the other useful methods of the File class is shown in the chart below.

Method Name	Description
ReadAllText(String)	Opens a text file, reads all lines of the file, and then closes the file.
ReadAllText(String, Encoding)	Opens a file, reads all lines of the file with the specified encoding, and then closes the file.
WriteAllLines(String, array<String>[]()[])	Creates a new file, write the specified string array to the file, and then closes the file. If the target file already exists, it is overwritten.
WriteAllLines(String, array<String>[]()[], Encoding)	Creates a new file, writes the specified string array to the file using the specified encoding, and then closes the file. If the target file already exists, it is overwritten.
ReadAllLines(String)	Opens a text file, reads all lines of the file, and then closes the file.
ReadAllLines(String, Encoding)	Opens a file, reads all lines of the file with the specified encoding, and then closes the file.
static byte[] ReadAllBytes(string path)	Opens a binary file, reads the contents of the file into a byte array, and then closes the file.

Figure 16.2-6 File class methods

16.2.1.2 Binary Streams (Binary Encoded Streams)

The most primitive I/O capabilities are declared in two classes: **StreamReader** and **StreamWriter**. They support reading and writing of a single byte and characters.

For files with known internal structures, for example, Binary Files, the BinaryReader and BinaryWriter classes offer streaming functionality that's oriented towards particular data types. The good thing about writing binary files are you cannot read the files by just opening them in the notepad also binary files can be of very large size.

Binary Writing:

Binary writing is provided by the **BinaryWriter** class. The list of methods and their descriptions found in this class is listed in Figure 16.2-7.

Method Name	Description
Close	Closes the current BinaryWriter and the underlying stream.
Dispose	Releases the unmanaged resources used by the BinaryWriter and optionally releases the managed resources.
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
Flush	Clears all buffers for the current writer and causes any buffered data to be written to the underlying device.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
Seek	Sets the position within the current stream.
ToString	Returns a String that represents the current Object. (Inherited from Object.)
Write	Overloaded. Writes a value to the current stream.
Write7BitEncodedInt	Writes a 32-bit integer in a compressed format.

Figure 16.2-7 BinaryWriter class methods

Binary Reading:

Binary reading is provided by the **BinaryReader** class. The list of methods and their descriptions found in this class is listed in Figure 16.2-8

Method	Description
Close	Closes the current reader and the underlying stream.
Equals	Determines whether two Object instances are equal.
GetHashCode	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType	Gets the Type of the current instance.
PeekChar	Returns the next available character and does not advance the byte or character position.
Read	Reads characters from the underlying stream and advances the current position of the stream.
ReadBoolean	Reads a Boolean value from the current stream and advances the current position of the stream by one byte.
ReadByte	Reads the next byte from the current stream and advances the current position of the stream by one byte.
ReadBytes	Reads count bytes from the current stream into a byte array and advances the current position by count bytes.
ReadChar	Reads the next character from the current stream and advances the current position of the stream in accordance with the Encoding used and the specific character being read from the stream.
ReadChars	Reads count characters from the current stream, returns the data in a character array, and advances the current position in accordance with the Encoding used and the specific character being read from the stream.
ReadDecimal	Reads a decimal value from the current stream and advances the current position of the stream by sixteen bytes.
ReadDouble	Reads an 8-byte floating point value from the current stream and advances the current position of the stream by eight bytes.
ReadInt16	Reads a 2-byte signed integer from the current stream and advances the current position of the stream by two bytes.
ReadInt32	Reads a 4-byte signed integer from the current stream and advances the current position of the stream by four bytes.
ReadInt64	Reads an 8-byte signed integer from the current stream and advances the current position of the stream by four bytes.
ReadSByte	Reads a signed byte from this stream and advances the current position of the stream by one byte.
ReadSingle	Reads a 4-byte floating point value from the current stream and advances the current position of the stream by four bytes.
ReadString	Reads a string from the current stream. The string is prefixed with the length, encoded as an integer seven bits at a time.
ReadUInt16	Reads a 2-byte unsigned integer from the current stream using little endian encoding and advances the position of the stream by two bytes.
ReadUInt32	Reads a 4-byte unsigned integer from the current stream and advances the position of the stream by four bytes.
ReadUInt64	Reads an 8-byte unsigned integer from the current stream and advances the position of the stream by eight bytes.
ToString	Returns a String that represents the current Object.

Figure 16.2-8 BinaryReader class methods

➢16.0 C# I/O System (Files and Streams)

In the Example 16.2-3, a table consisting of 2 rows and 3 columns is created, Then it is written into a binary file called matrix_data.dat. The Figure 16.2-9 shows the debug statements emits from the program. Snap shot of the directory is show in Figure 16.2-10

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace MatrixWriter
{
 public class WriteMatrix
 {
  static void Main(String [] args)
  {
   string fileName = @"C:\csci\Examples\WebApplicationDevelopment\matrix_data.dat";
   double [,] data = {{Math.Exp(2.0), Math.Exp(3.0),Math.Exp(3.0)},
                                        {Math.Exp(-2.0), Math.Exp(-3.0),Math.Exp(-3.0)}};

   int row = data.GetUpperBound(0)+1;
   int col = data.GetUpperBound(1)+1;

   Console.WriteLine("row :  " + row);
   Console.WriteLine("col : " + col);

   int i, j;

   for (i = 0; i < row; i++)
    for (j = 0; j < col; j++)
     Console.WriteLine("data[" + i + "," + j + "] = " + data[i, j]);


   if (fileName.Length > 0)
   try {
      FileStream fileStream = File.Create(fileName);
      BinaryWriter binaryWriter = new BinaryWriter(fileStream);

       binaryWriter .Write(row);
       binaryWriter .Write(col);

       for (i = 0; i < row; i++)
         for (j = 0; j < col; j++)
            binaryWriter .Write(data[i, j]);

       binaryWriter .Close();
   }
   catch (IOException e){
    Console.WriteLine("Exception caught..");
   }
  }
 }
}
```

Example 16.2-3

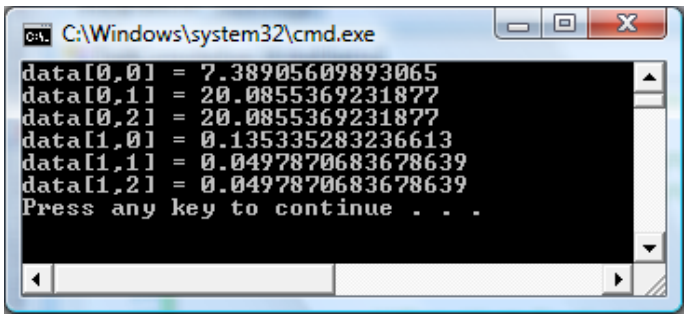


Figure 16.2-9 Output

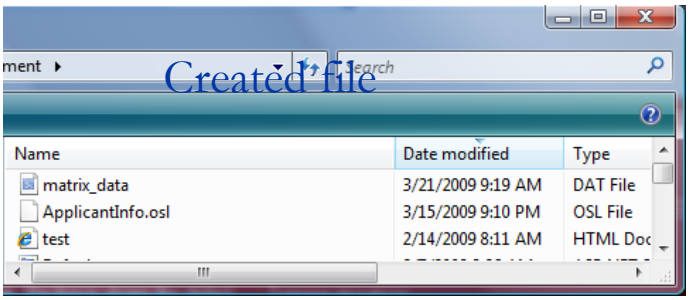


Figure 16.2-10 Created File

➢16.0 C# I/O System (Files and Streams)

In this example you will see the BinaryReaader class is used to read back the table written in Example 16.2-4 and print it back. The program opens the matrix_data.dat, first reads the first two 32-bit locations to read the number of rows and number of columns . These two values are used to re-create the table in the memory. Then reads continuously the next 6 64-bit locations and fills them into the table. At the end of the program, the table is printed on the console. Output is shown in Figure 16.2-11

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace MatrixReader
{
 public class ReadMatrix
 {
   static void Main(String[] args)
  {
   string fileName = @"C:\csci\Examples\WebApplicationDevelopment\matrix_data.dat";
   double[,] data;

   int row = 0;
   int col = 0;
   int i, j;

   if (fileName.Length > 0)
   try
   { FileStream fileStream = File.OpenRead(fileName);
    BinaryReader binaryReader = new BinaryReader(fileStream);

    row = binaryReader.ReadInt32();
    col = binaryReader.ReadInt32();

    data = new double[row, col];

    for (i = 0; i < row; i++)
      for (j = 0; j < col; j++)
         data[i, j] =  binaryReader.ReadDouble();

    binaryReader.Close();

    for (i = 0; i < row; i++)
      for (j = 0; j < col; j++)
        Console.WriteLine("data[" + i + "," + j + "] = " + data[i, j]);
   }
   catch (IOException e)
   {
    Console.WriteLine("Exception caught..");
   } } } }
```

Example 16.2-4

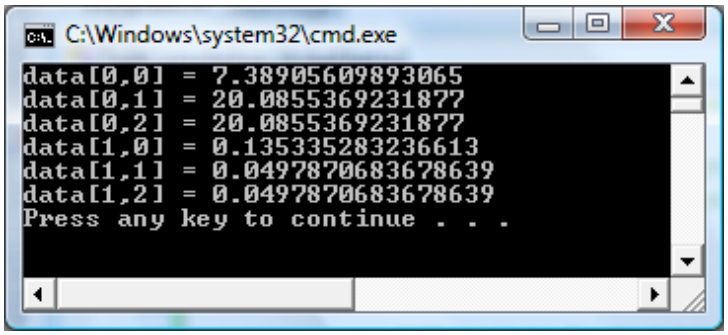


Figure 16.2-11 Output

16.3 File Usage in Web Applications

Most web applications use databases to store and retrieve data. But file handling can be used in any web application just as they are used in a non web application. Sometimes people build browser based application to be used locally within an enterprise. This may be a good example where you could use file systems and the backend to store data instead of a database. However the application cannot use the power of databases while reading and writing data. In file systems, the developer must write the code to achieve the specific needs for file I/O. The example listed below does the same you read in Example 16.2-1. But the result is displayed on a web browser instead of a console.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ShowDirListing.aspx.cs"
Inherits="UsageOfListsAndFiles._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:Label ID="aLabel" runat="server" Text="Click the button to see Directories"></asp:Label> <br/><br/>
    <asp:ListBox ID="FileList" runat="server" Width="200"></asp:ListBox><br/><br/>
    <asp:Button ID="PushButton" runat="server" Text="Show Files" OnClick="ShowDirectories"></asp:Button>
    </div>
    </form>
</body>
</html>
```

Example 16.3-1 a

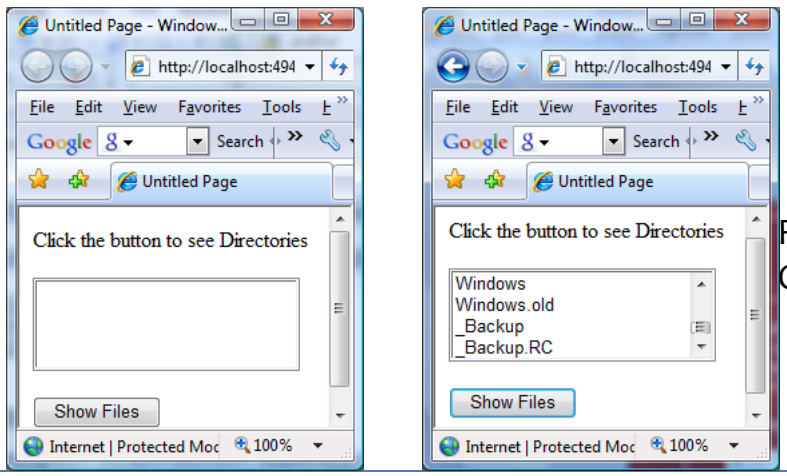


Figure 16.3-1 Output On browser.

➢16.0 C# I/O System (Files and Streams)

The code base for the example is shown in Example 16.3-1b. The output is the same as the example 16.2-1 but displayed on a browser (Figure 16.3-1)

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.IO;

namespace UsageOfListsAndFiles
{  public partial class _Default : System.Web.UI.Page
   {
    protected void Page_Load(object sender, EventArgs e) {
      }
    protected void ShowDirectories(object sender, EventArgs e)
    {
      string fileName =
        @"C:\csci\Examples\WebApplicationDevelopment\CDriveDirs.txt";

      /** Now get written dir list back from file in one shot. **/
      string[] getBackDirs = File.ReadAllLines(fileName);

      /** Fill the ListBox component with the data **/
      foreach (string nextDir in getBackDirs)
          FileList.Items.Add(nextDir);
    }
  }
}
```

Example 16.3-1b

Take a look at the example on pages 590-594 in your test book.

16.4 Other Useful Classes

16.4.1 Directory class

Use the Directory class for typical operations such as copying, moving, renaming, creating, and deleting directories. You can also use the Directory class to get and set **DateTime** information related to the creation, access, and writing of a directory.

Because all Directory methods are static, it might be more efficient to use a Directory method rather than a corresponding **DirectoryInfo** instance method if you want to perform only one action. Most Directory methods require the path to the directory that you are manipulating.

The static methods of the Directory class Perform security checks on all methods. If you are going to reuse an object several times, consider using the corresponding instance method of **DirectoryInfo** instead, because the security check will not always be necessary. Methods and descriptions of the Directory class is shown in Figure 16.4-1.

Name	Description
ChangeExtension	Changes the extension of a path string.
Combine	Combines two path strings.
GetDirectoryName	Returns the directory information for the specified path string.
GetExtension	Returns the extension of the specified path string.
GetFileName	Returns the file name and extension of the specified path string.
GetFileNameWithoutExtension	Returns the file name of the specified path string without the extension.
GetFullPath	Returns the absolute path for the specified path string.
GetInvalidFileNameChars	Gets an array containing the characters that are not allowed in file names.
GetInvalidPathChars	Gets an array containing the characters that are not allowed in path names.
GetPathRoot	Gets the root directory information of the specified path.
GetRandomFileName	Returns a random folder name or file name.
GetTempFileName	Creates a uniquely named, zero-byte temporary file on disk and returns the full path of that file.
GetTempPath	Returns the path of the current system's temporary folder.
HasExtension	Determines whether a path includes a file name extension.
IsPathRooted	Gets a value indicating whether the specified path string contains absolute or relative path information.

Figure 16.4-1 Directory class methods

16.4.2 Path class

A path is a string that provides the location of a file or directory. A path does not necessarily point to a location on disk; for example, a path might map to a location in memory or on a device. The exact format of a path is determined by the current platform. For example, on some systems, a path can start with a drive or volume letter, while this element is not present in other systems. On some systems, file paths can contain extensions, which indicate the type of information stored in the file. The format of a file name extension is platform-dependent; for example, some systems limit extensions to three characters, and others do not. The current platform also determines the set of characters used to separate the elements of a path, and the set of characters that cannot be used when specifying paths. Because of these differences, the fields of the Path class as well as the exact behavior of some members of the Path class are platform-dependent. Methods and descriptions of the Path class is shown in Figure 16.4-2.

Name	Description
ChangeExtension	Changes the extension of a path string.
Combine	Combines two path strings.
GetDirectoryName	Returns the directory information for the specified path string.
GetExtension	Returns the extension of the specified path string.
GetFileName	Returns the file name and extension of the specified path string.
GetFileNameWithoutExtension	Returns the file name of the specified path string without the extension.
GetFullPath	Returns the absolute path for the specified path string.
GetInvalidFileNameChars	Gets an array containing the characters that are not allowed in file names.
GetInvalidPathChars	Gets an array containing the characters that are not allowed in path names.
GetPathRoot	Gets the root directory information of the specified path.
GetRandomFileName	Returns a random folder name or file name.
GetTempFileName	Creates a uniquely named, zero-byte temporary file on disk and returns the full path of that file.
GetTempPath	Returns the path of the current system's temporary folder.
HasExtension	Determines whether a path includes a file name extension.
IsPathRooted	Gets a value indicating whether the specified path string contains absolute or relative path information.

Figure 16.4-2 Path class methods

16.4.3 DirectoryInfo class

Use the **DirectoryInfo** class for typical operations such as copying, moving, renaming, creating, and deleting directories. If you are going to reuse an object several times, consider using the instance method of **DirectoryInfo** instead of the corresponding static methods of the Directory class, because a security check will not always be necessary. Methods and descriptions of the **DirectoryInfor** class is shown in Figure 16.4.3.

Name	Description
Create	Overloaded. Creates a directory.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
CreateSubdirectory	Overloaded. Creates a subdirectory or subdirectories on the specified path. The specified path can be relative to this instance of the DirectoryInfo class.
Delete	Overloaded. Deletes a DirectoryInfo and its contents from a path.
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetAccessControl	Overloaded. Gets the access control list (ACL) entries for the current directory.
GetDirectories	Overloaded. Returns the subdirectories of the current directory.
GetFiles	Overloaded. Returns a file list from the current directory.
GetFileSystemInfos	Overloaded. Retrieves an array of strongly typed FileSystemInfo objects representing files and subdirectories of the current directory.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetLifetimeService	Retrieves the current lifetime service object that controls the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
GetObjectData	Sets the SerializationInfo object with the file name and additional exception information. (Inherited from FileSystemInfo.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
InitializeLifetimeService	Obtains a lifetime service object to control the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
MemberwiseClone	Overloaded.
MoveTo	Moves a DirectoryInfo instance and its contents to a new path.
Refresh	Refreshes the state of the object. (Inherited from FileSystemInfo.)
SetAccessControl	Applies access control list (ACL) entries described by a DirectorySecurity object to the directory described by the current DirectoryInfo object.
ToString	Returns the original path that was passed by the user. (Overrides Object..::.ToString()()().)

Figure 16.4-3 DirectoryInfor class methods

16.4.4 FileInfo class

Use the **FileInfo** class for typical operations such as copying, moving, renaming, creating, opening, deleting, and appending to files.

Many of the **FileInfo** methods return other I/O types when you create or open files. You can use these other types to further manipulate a file. For more information, see specific **FileInfo** members such as Open, OpenRead, OpenText, CreateText, or Create.

If you are going to reuse an object several times, consider using the instance method of FileInfo instead of the corresponding static methods of the File class, because a security check will not always be necessary.

By default, full read/write access to new files is granted to all users.

The following table describes the enumerations that are used to customize the behavior of various FileInfo methods.

FileInfor class is shown in Figure 16.4.4.

Name	Description
AppendText	Creates a StreamWriter that appends text to the file represented by this instance of the FileInfo.
CopyTo	Overloaded. Copies an existing file to a new file.
Create	Creates a file.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
CreateText	Creates a StreamWriter that writes a new text file.
Decrypt	Decrypts a file that was encrypted by the current account using the Encrypt method.
Delete	Permanently deletes a file. (Overrides FileSystemInfo..::.Delete()()().)
Encrypt	Encrypts a file so that only the account used to encrypt the file can decrypt it.
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetAccessControl	Overloaded. Gets a FileSecurity object that encapsulates the access control list (ACL) entries for the file described by the current FileInfo object.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetLifetimeService	Retrieves the current lifetime service object that controls the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
GetObjectData	Sets the SerializationInfo object with the file name and additional exception information. (Inherited from FileSystemInfo.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
InitializeLifetimeService	Obtains a lifetime service object to control the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
MemberwiseClone	Overloaded.
MoveTo	Moves a specified file to a new location, providing the option to specify a new file name.
Open	Overloaded. Opens a file with various read/write and sharing privileges.
OpenRead	Creates a read-only FileStream.
OpenText	Creates a StreamReader with UTF8 encoding that reads from an existing text file.
OpenWrite	Creates a write-only FileStream.
Refresh	Refreshes the state of the object. (Inherited from FileSystemInfo.)
Replace	Overloaded. Replaces the contents of a specified file with the file described by the current FileInfo object, deleting the original file, and creating a backup of the replaced file.
SetAccessControl	Applies access control list (ACL) entries described by a FileSecurity object to the file described by the current FileInfo object.
ToString	Returns the path as a string. (Overrides Object..::.ToString()()().)

Figure 16.4-4 FileInfor class methods

16.4.5 DriveInfo class

This class models a drive and provides methods and properties to query for drive information. Use **DriveInfo** to determine what drives are available, and what type of drives they are. You can also query to determine the capacity and available free space on the drive. **DriveInfor** class is shown in Figure 16.4.5.

Method	Description
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetDrives	Retrieves the drive names of all logical drives on a computer.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
ToString	Returns a drive name as a string. (Overrides Object..::.ToString()()().)

Figure 16.4-5 DriveInfor class methods

16.5 Object Serialization

Serialization is the process of writing the state of an object to a Byte Stream OR it is the process of reducing the objects instance into a format that can either be stored to disk or transported over a Network.

Serialization is useful when you want to save the state of your application to a persistence storage area. That storage area can be a file. At a later time, you may restore these objects by using the process of **deserialization**. In this section we will learn about serialization and why you need serialization in NET.

16.5.1 Serialization and Streams in .NET

.NET objects are serialized to a stream. Developers must use a .NET formatter class to control the serialization of the object to and from the stream. In addition to the serialized data, the serialization stream carries information about the object's type, including its assembly name, culture, and version.

16.5.2 The Binary Formatter

The Binary formatter provides binary encoding for compact serialization either for storage or for socket-based network streams. The **BinaryFormatter** class is generally not appropriate when data is meant to be passed through a firewall.

16.5.3 The SOAP Formatter

The **SOAP** (**Simple Object Access Protocol**) formatter provides formatting that can be used to enable objects to be serialized using the **SOAP** protocol. The Soap Formatter class is primarily used for serialization through firewalls or among diverse systems. SOAP is used to serialize object to XML files.

16.5.4 Other Formatters

The .NET framework also includes the abstract FORMATTERS class that can be used as a base class for custom formatters. This class inherits from the **IFormatter** interface.

16.5.4 Requirements for Object Serialization

Serialization is done so that the object can be recreated with its current state at a later point in time or at a different location. The following are required to Serialize an object:

- The object that is to serialized itself
- A stream to contain the serialized object
- A Formatter used to serialize the object

System.Runtime.Serialization.Formatters.Binary is the namespace that contains the classes that are required to serialize an object.

To serialize using the **BinaryFormatter**, you use the Serialize() method of The **BinaryFormatter** while **Deserialize()** is used to **deserialize** an object from the **BinaryFormatter** class.

16.5.5 Making a Class Serializable

16.5.5.1 Using [Serializable] tag.

Every modern language contains a feature to allow you to serialize your objects. The good news is that with C# and the .NET Framework you can do this smart and easily. But an object cannot be serialized if teh class of that object is not made to create serialized objects. So the first thing is to make a class **serializable**. This is very simple in c#. All you have to do is to add the **[Serializable]** attribute to the class.

Here is an example…

```
[Serializable]
public class Store
{

}
```

➢16.0 C# I/O System (Files and Streams)

Example 16.5-1a shows the same Store class with more information.

The **[Serializable]** attribute 'stamps' the class ready To be serialized. Note that the Serialization applied ONLY to the data in the class and NOT for the methods. Example 16.5-1b shows the code you use to serialize an object of the **Store** class.

```
[Serializable]
public class Store
{
  private int sCount;

  public int stockCount
  {
    get
    {
      return sCount;
    }
    set
    {
      sCount = value;
    }
  }

  private int temp;
  private string storeName = "My local store";

  public Store()
  {
     stockCount = 0;
  }
   // Here may follow some code for this
   //class to function
}
```

Example 16.5-1a Serializable Class

```
static void Main(string[] args)
{
Store myStore = new Store();
   myStore.stockCount = 50;

   FileStream flStream = new FileStream("MyStore.dat",
   FileMode.OpenOrCreate, FileAccess.Write);
   try
   {
      BinaryFormatter binFormatter = new BinaryFormatter();
      binFormatter.Serialize(flStream, myStore);
   }
   finally
   {
      flStream.Close();
      Console.WriteLine(Environment.NewLine + "Object
     'myStore' serialized!";
      Console.WriteLine(Environment.NewLine + "stockCount :
     =  " + myStore.stockCount ;
   }
}
```

Example 16.5-1b Serializing Process

16.5.5.2 Using the [NonSerializable] attribute

At times there are non important or temporary data members you define in a class that are not really worth serializing. In fact sometimes there are even sensitive information that may not want to serialize. We can 'mark' those data members that we do not want to serialize in a class by adding the Marker **[NonSerializable] for each** data member.

```
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class Store
{
  private int sCount;

  public int stockCount
  {
    get
    {
      return sCount;
    }
    set
    {
      sCount = value;
    }
  }
 [NonSerialized] private int temp;

  private string storeName = "My local store";

  public Store()
  {
    stockCount = 0;
  }
   // Here may follow some code for this
   //class to function
}
```

Example 16.5-2a Serializable Class

```
static void Main(string[] args)
{
   Store myStore = new Store();
   myStore.stockCount = 50;

   FileStream flStream = new FileStream("MyStore.dat",
   FileMode.OpenOrCreate, FileAccess.Write);
   try
   {
      BinaryFormatter binFormatter = new BinaryFormatter();
      binFormatter.Serialize(flStream, myStore);
   }
   finally
   {
      flStream.Close();
      Console.WriteLine(Environment.NewLine + "Object
     'myStore' serialized!";
      Console.WriteLine(Environment.NewLine + "stockCount
     = 50";
   }
}
```

Example 16.5-2b Serializing Process

In this example, the temp data member will not be serialized. Instead of the actual value of the data member, it will just store the default value of the data type. In this case for temp, it will store **0** as temp is an int type.

16.5.6 DeSerializing

16.5.6.1 Using BinaryFormatter

Deserializing is re-constructing the object back into the memory. You can use the **Deserialize()** method in the **BinaryFormatter** class. In the Example 16.5-3 you see a object reference Store is first created. Really the object is not created at tht time. But the method **Deserialize()** brings a Object type object which is then casted to a Store type and linked to the readStore reference.

```
using System.Runtime.Serialization.Formatters.Binary
static void Main(string[] args)
{
 Store readStore = null;

  flStream = new FileStream("MyStore.dat", FileMode.Open, FileAccess.Read);
  try {
        BinaryFormatter binFormatter = new BinaryFormatter();
         readStore = (Store)binFormatter.Deserialize(flStream);
      }
  finally{
            flStream.Close();
            Console.WriteLine(Environment.NewLine + "Object 'readStore' deserialized!");
            Console.WriteLine(Environment.NewLine + "stockCount = " +
                    System.Convert.ToString(readStore.stockCount));
          }
}
```

Example 16.5-3 DeSerializing using BinaryFormatter

We create an instance of **FileStream** to read the file. We create a **BinaryFormatter** object again. This time we use its 'Deserialize' method. It returns the deserialized object which we cast to our own class 'Store'. and in the end we close the stream and output the 'stockCount' value to check if everything is OK.

16.5.6.2 Using SOAP (Simple Object Access Protocol) Formatter

We have seen how binary formatter is used in Serialization. The fact that it is separated in a detached namespace may be already gave you a hint that there are other types of formatters.

The SOAP formatter serializes your object to an XML file obeying special rules. The SOAP is the standardized format which all the web services use. It is a simple XML-based protocol to let applications exchange information over HTTP.

You can replace the **BinaryFormatter** with the **SoapFormatter** and it will work with no other changes. The only thing you have to do is to add a reference to the **System.Runtime.Serialization.Formatters.Soa**p (not only a uses clause but a reference too!).

The SOAP format being an XML is readable by humans unlike the binary one. See Example 16.5-4.

```
static void Main(string [] args)
{
Store myStore = new Store();

MemoryStream memStream = new MemoryStream();
   try {
      myStore.stockCount = 50;
      SoapFormatter soapFormatter = new SoapFormatter();
      soapFormatter.Serialize(memStream, myStore);

      byte[] buff = memStream.GetBuffer();

      string soapOutput = "";

      foreach(byte b in buff)
         soapOutput += (char)b;

      tbOutput.Text += Environment.NewLine +  "Object 'myStore' serialized to SOAP!";
      tbOutput.Text += soapOutput;
   }
   finally {
      memStream.Close();
   }
}
```

Example 16.5-4a DeSerializing using SOAP

Here the **MemoryStream** class is used instead of the **FileStream** we used previously. This way we save ourselves the use of file.

After we store the data to the stream we get its contents with **GetBuffer().** Note that the **MemoryStream** is not closed right after the write as did with the **FileStream**. This is because we loose the stored data once we close it.

Look at the code:

```
foreach(byte b in buff)
    soapOutput += (char)b;
```

It is used to convert the byte array to string. Do you notice something wrong? If not, you have to study more about the strings. In C# the strings can not be changed once their value is set. This way we create a new string every time we add a character. More appropriate is the use of **StringBuffer**. In our case the buffer is small and the very serious speed losses are not visible.

The output we get by the SOAP formatter is shown in Figure 16.5-5

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <a1:Store id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/
SerializeTest/SerializeTest%2C%20Version%3D1.0.938.186%2C
%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
         <sCount>50</sCount>
         <storeName id="ref-3">My local store</storeName>
      </a1:Store>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 16.5-4b Output using SOAP Formatter