

# Beginning ASP.NET 2.0 in C# 2005

From Novice to Professional



Matthew MacDonald

## **Beginning ASP.NET 2.0 in C# 2005: From Novice to Professional**

**Copyright © 2006 by Matthew MacDonald**

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-572-5

ISBN-10 (pbk): 1-59059-572-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Contributor of Chapter 27: Julian Templeman

Lead Editor: Jonathan Hassell

Technical Reviewer: Ronald Landers

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager and Production Director: Grace Wong

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Assistant Production Director: Kari Brooks-Copony

Production Editor: Kelly Winkquist

Compositor: Pat Christenson

Proofreader: Nancy Riddiough

Indexer: Michael Brinkman

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.



# Validation and Rich Controls

**T**his chapter looks at some of the real promise of ASP.NET and the web control model. First, you'll learn about ASP.NET's validation controls. These controls take a previously time-consuming and complicated task—verifying user input and reporting errors—and automate it with an elegant, easy-to-use collection of validators. You'll learn how to add these controls to an existing page and use regular expressions, custom validation functions, and manual validation. And as usual, you'll peer under the hood to see how ASP.NET implements these features.

Next, you'll consider two controls that have no equivalent in the ordinary HTML world: the Calendar and AdRotator controls. These controls demonstrate how the web control model can invent new types of web page user interfaces without breaking browser compatibility. The Calendar and AdRotator controls are only two of several rich controls included with ASP.NET; you'll explore the others throughout this book.

Finally, you'll consider how you can create more sophisticated pages with multiple views using advanced container controls such as the MultiView and Wizard controls. 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.

## Validation

As a seasoned developer, you probably realize users will make mistakes. What's particularly daunting is the range of possible mistakes that users can make, such as the following:

- 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, such as invalid e-mail addresses that cause problems for your automatic mailing programs.

- 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).

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 is commonly used to create text boxes that accept 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 special table, list, or window that describes them all. By the time you have perfected your validation routines, a considerable amount of fine-tuned effort has gone into writing validation 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 DHTML and JavaScript to provide a more dynamic and responsive interface while still providing ordinary validation for older browsers (often referred to as *down-level* browsers).

## The Validation Controls

ASP.NET provides five validator controls, which are described in Table 8-1. Four are targeted at specific types of validation, while the fifth allows you to apply custom validation routines.

**Table 8-1.** *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, specified input control.

Control Class	Description
RegularExpressionValidator	Validation succeeds if the value in an input control matches a specified regular expression.
CustomValidator	Validation is performed by a user-defined function.

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 add a `RequiredFieldValidator` to the control. This ensures that two types of validation will be performed, effectively restricting blank values.

Like all other web controls, you add a validator as a tag in the form `<asp:ControlClassName />`. The other validation control, `ValidationSummary`, doesn't perform any actual control checking. Instead, you can use it to provide a list of all the validation errors for the entire page.

## The Validation Process

You can use the validator 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 buttonlike controls that can be used to submit the page also provide the `CausesValidation` property. Examples include the `LinkButton`, `ImageButton`, and `BulletedList`.

---

## Client-Side Validation

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 the application.

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.

## The Validator Classes

The validation control classes are found in the `System.Web.UI.WebControls` namespace and inherit from the `BaseValidator` class. This class defines the basic functionality for a validation control. Table 8-2 describes its properties.

**Table 8-2.** *Properties of the BaseValidator Class*

Property	Description
<code>ControlToValidate</code>	Identifies the control that this validator will check. Each validator can verify the value in one input control.
<code>ErrorMessage</code> , <code>ForeColor</code> , and <code>Display</code>	If validation fails, the validator control can display a text message (set by the <code>ErrorMessage</code> property). The <code>Display</code> property allows you to configure whether this error message will be added dynamically as needed (Dynamic) or whether an appropriate space will be reserved for the message (Static). 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.

Property	Description
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.
EnableClientSideScript	If set to true, ASP.NET will add JavaScript and DHTML code to allow client-side validation on browsers that support it.

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 8-3 outlines these properties.

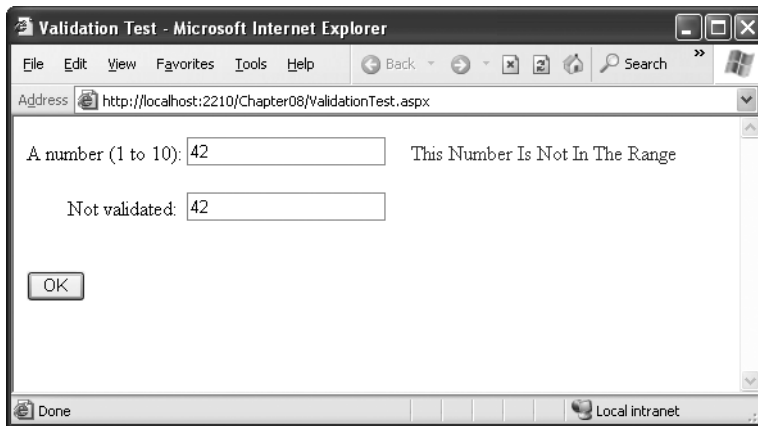
**Table 8-3.** *Validator-Specific Properties*

Validator Control	Added Members
RequiredFieldValidator	None required
RangeValidator	MaximumValue, MinimumValue, Type
CompareValidator	ControlToCompare, Operator, Type, ValueToCompare
RegularExpressionValidator	ValidationExpression
CustomValidator	ClientValidationFunction, ServerValidate event

Later in this chapter (in the “A Validated Customer Form” section), you’ll see a customer form example that demonstrates each type of validation.

## 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 RangeValidation control that validates the first text box. If validation fails, the RangeValidation control displays an error message, so you should place this control immediately next to the TextBox it’s validating. Figure 8-1 shows the appearance of the page after a failed validation attempt.



**Figure 8-1.** *Failed validation*

In addition, place a Label control at the bottom of the form. This label will report when the page has been successfully 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 layout code 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.

```
<html><body>
  <form method="post" runat="server">
    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" /><br /><br />
    <asp:Label id="lblMessage" runat="server"
      EnableViewState="false" />
  </form>
</body></html>
```



Finally, 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.";
}
```

If you're testing this web page in a modern browser (such as Internet Explorer 5 or later), 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. This code uses the special `WebUIValidation.js` script library file that is installed on your server with the .NET Framework (in the `c:\Inetpub\wwwroot\aspnet_client\system_web\[Version]` directory) and is somewhat complicated. However, ASP.NET handles all the details for you automatically. 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.

These features are relatively high-level, because they combine DHTML and JavaScript. Clearly, not all browsers will support this 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.";
}
```

This code 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 or if validation was not performed (for example, if the validation controls are disabled or if the button has `CausesValidation` set to false).

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

Remember, client-side validation is just nice frosting on top of your application. Server-side validation will always be performed, ensuring that crafty users can't "spoof" pages.

## Other Display Options

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.

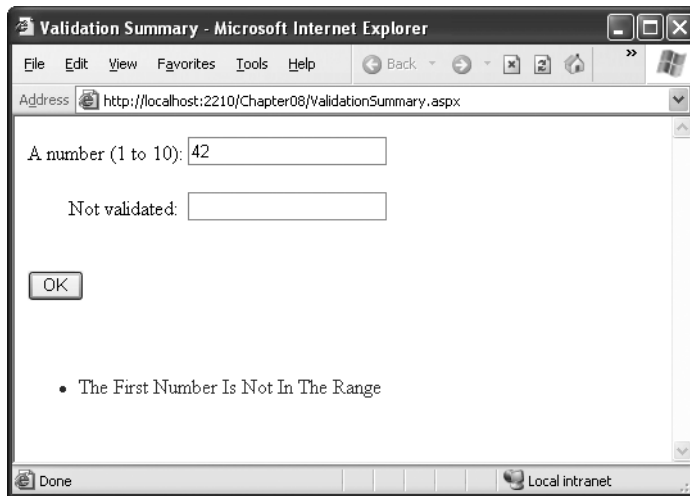
To try this, set the `Display` property of the `RangeValidator` control to `None`. This ensures the error message will never be displayed. However, validation will still be performed and the user will still be prevented from successfully clicking the OK button if some invalid information exists on the page.

Next, add the `ValidationSummary` in a suitable location (such as the bottom of the page):

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

When you run the page, you won't see any dynamic messages as you enter invalid information and tab to a new field. However, when you click the OK button, the `ValidationSummary` will appear with a list of all error messages, as shown in Figure 8-2. In this case, it retrieves one error message (from the `RangeValidator` control). However, if you had a dozen validators, it would retrieve all their error messages and create a list.

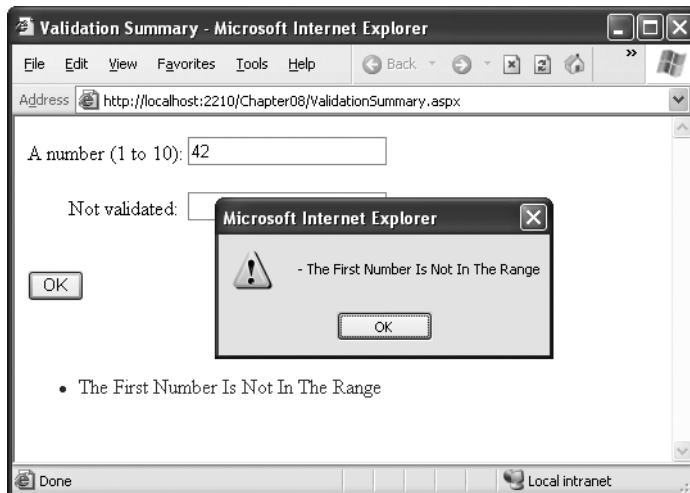
The `ValidationSummary` control also provides some useful properties you can use to fine-tune the error display. You can set the `HeaderText` property to display a special title at the top of the list (such as *Your page contains the following errors:*). You can also change the `ForeColor` and choose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



**Figure 8-2.** *The validation summary*

Finally, you choose to have the validation summary displayed in a pop-up dialog box instead of on the page (see Figure 8-3). 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 `ValidationSummary.ShowMessageBox` property to true.



**Figure 8-3.** *A message box summary*

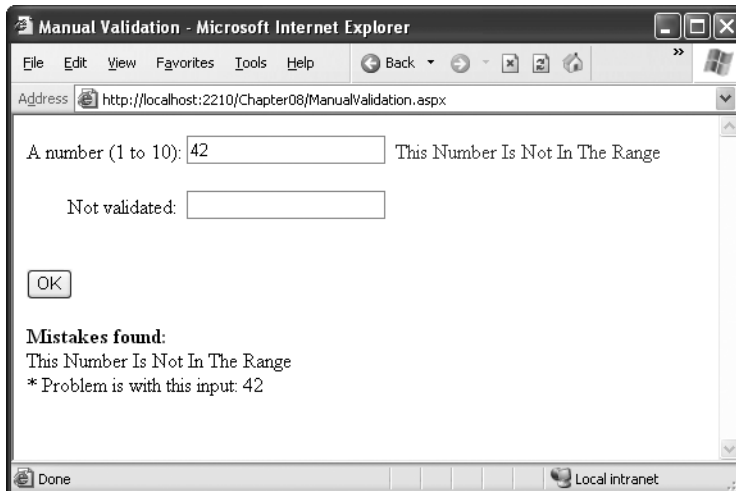
## Manual Validation

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.
- 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 8-4.



**Figure 8-4.** *Manual validation*

This technique adds a feature that wouldn't be available with automatic validation, which uses the static `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 />";

    // Create a variable to represent the input control.
    TextBox ctrlInput;

    // 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.
            ctrlInput = (TextBox)this.FindControl(ctrl.ControlToValidate);
            errorMessage += " * Problem is with this input: ";
            errorMessage += ctrlInput.Text + "<br />";
        }
    }
    lblMessage.Text = errorMessage;
}
```

This example uses an advanced technique: the `Page.FindControl()` method. It's required because the `ControlToValidate` property is just 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).

## Understanding Regular Expressions

Regular expressions are an advanced tool for matching patterns. They 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 (which may represent a new height of computer geekdom).

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.

### Literals 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 with *ing*—possible matches include *acting* and *developing*.

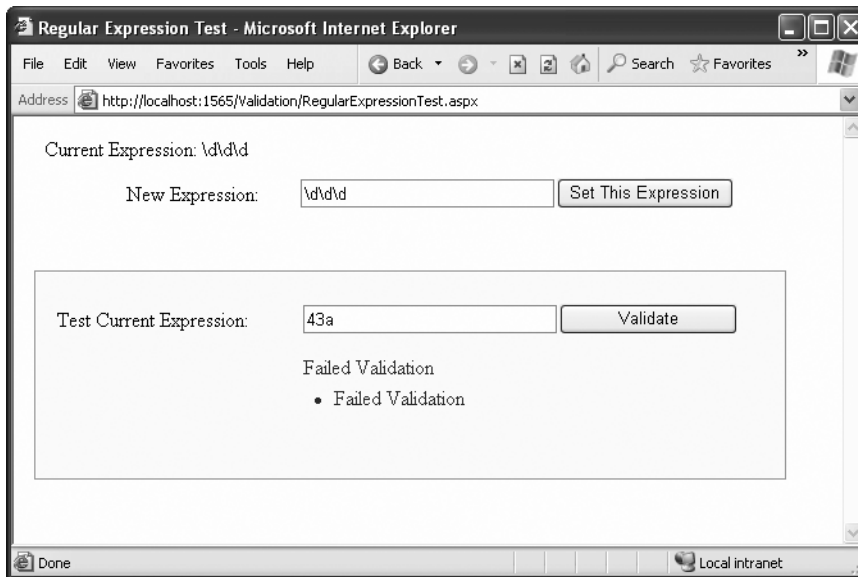
```
[a-f]\w+ing
```

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.

```
.+@.+
```

## 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 8-5. 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 validator succeeds or fails.



**Figure 8-5.** A regular expression test page

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 Validation 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 : Page
{
    protected void cmdSetExpression_Click(Object sender, EventArgs e)
    {
        TestValidator.ValidationExpression = txtExpression.Text;
        lblExpression.Text = "Current Expression: ";
        lblExpression.Text += txtExpression.Text;
    }
}
```

Table 8-4 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 8-4.** *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.
	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).

Table 8-5 shows a few of common (and useful) regular expressions.

**Table 8-5.** *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 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).

*Continued*

**Table 8-5.** *Continued*

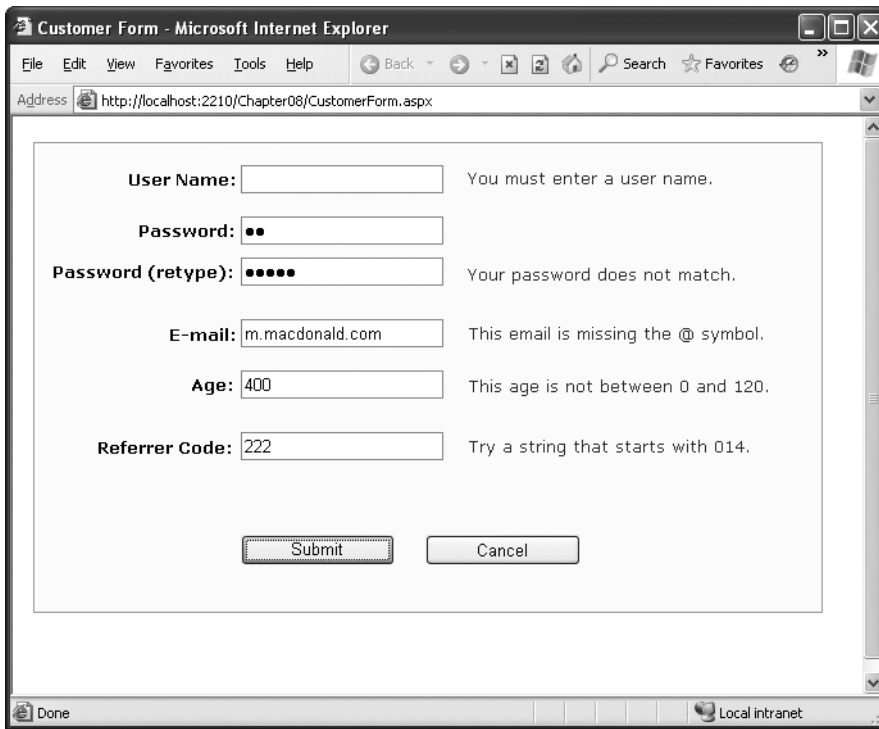
Content	Regular Expression	Description
Another advanced password	<code>[a-zA-Z]\w*\d+\w*</code>	This password starts with a letter character, followed by zero or more word characters, a digit, and then zero or more word characters. In short, it forces a password to contain a number somewhere inside it. You could use a similar pattern to require two numbers or any other special character.
Limited-length field	<code>\S{4,10}</code>	Like the password example, this allows four to ten characters, but it allows special characters (asterisks, ampersands, and so on).
Social Security number	<code>\d{3}-\d{2}-\d{4}</code>	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.

\* 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](http://en.wikipedia.org/wiki/Luhn_formula).)

## 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 8-6 shows this form.



The screenshot shows a Microsoft Internet Explorer window titled "Customer Form - Microsoft Internet Explorer". The address bar displays "http://localhost:2210/Chapter08/CustomerForm.aspx". The form contains several input fields with associated validation messages:

- User Name:** An empty text box with the message "You must enter a user name."
- Password:** A masked text box (two dots) with no message.
- Password (retype):** A masked text box (six dots) with the message "Your password does not match."
- E-mail:** A text box containing "m.macdonald.com" with the message "This email is missing the @ symbol."
- Age:** A text box containing "400" with the message "This age is not between 0 and 120."
- Referrer Code:** A text box containing "222" with the message "Try a string that starts with 014."

At the bottom of the form are two buttons: "Submit" and "Cancel". The browser's status bar at the bottom shows "Done" and "Local intranet".

**Figure 8-6.** *A sample customer form*

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

- Two `RequiredFieldValidator` controls make sure the user enters a user name and a password.
- 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.

The tags for the validator controls are as follows:

```
<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: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"
    MaximumValue="120" MinimumValue="0"
    ControlToValidate="txtAge" />

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

The form provides two validation buttons—one that requires validation and one that allows the user to cancel the task gracefully. Here's the event handling code:

```
protected void cmdSubmit_Click(Object sender, EventArgs e)
{
    if (!this.IsValid) return;
    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.";
}
```

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
    {
        // An error occurred in the conversion.
        // The value is not valid.
        e.IsValid = false;
    }
}
```

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.)

---

**Tip** In some cases, you might be able to replace custom validation with a particularly ingenious use of a regular expression. However, you can use custom validation to ensure that validation code is executed only at the server. That prevents users from seeing your regular expression template (in the rendered JavaScript code) and using it to determine how they can outwit your validation routine. For example, a user may not have a valid credit card number, but if they know the algorithm you use to test credit card numbers, they can create a false one more easily.

---

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 .NET 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 see 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. You'll notice that the JavaScript code resembles C# superficially.

```
<script language="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>
```

Once you've added the function, set the `ClientValidationFunction` property of the `CustomValidator` control to the name of the function. You can add this information manually or by using the Properties window in Visual Studio.

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

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

---

**Tip** Even when you use client-side validation, you must still include the `ServerValidate` event handler, both to provide server-side validation for clients that don't support the required JavaScript and DHTML features and to prevent clients from circumventing your validation by modifying the HTML page they receive.

---

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.

## YOU CAN VALIDATE LIST CONTROLS

The examples in this chapter have concentrated exclusively on validating text entry, which is the most common requirement in a web application. While you can't validate `RadioButton` or `CheckBox` controls, you can validate most single-select list controls.

When validating a list control, the value that is being validated is the `Value` property of the selected `ListItem` object. Remember, the `Value` property is the special hidden information attribute that can be added to every list item. If you don't use it, you can't validate the control (validating the text of the selection isn't a supported option).

## Validation Groups

In more complex pages, you might have several distinct groups of pages, 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.

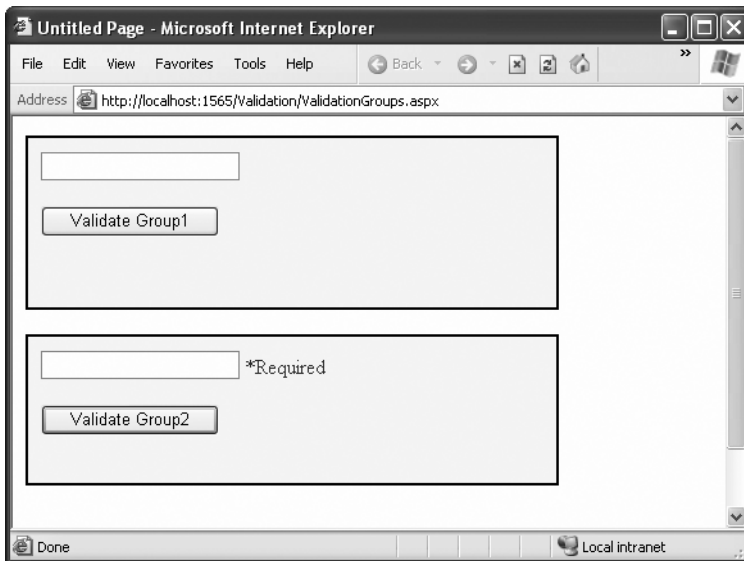
This scenario is possible thanks to a feature called *validation groups*. To create a validation group, you need to put the input controls and the CausesValidation button controls into the same logical group. You do this by setting the ValidationGroup property of every control with the same descriptive string (such as "Form1" or "Login"). 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 8-7).





**Figure 8-7.** *Grouping controls for validation*

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). Alternatively, you might choose to manage complex scenarios like these using server-side code, as shown in the following example.

You can use an overloaded version of the `Page.Validate()` method to validate just a specific group. You specify the name of the group you want to validate. For example, using the previous page, you could create a button that has no validation group assigned and respond to the `Button.Click` event with this code:

```
protected void cmdValidateAll_Click(object sender, EventArgs e)
{
    Label1.Text = "Valid: " + Page.IsValid.ToString();
    Page.Validate("Group1");
    Label1.Text += "<br />Group1 Valid: " + Page.IsValid.ToString();
    Page.Validate("Group2");
    Label1.Text += "<br />Group2 Valid: " + Page.IsValid.ToString();
}
```

Because this button isn't in any validation group, the two text boxes won't be validated automatically, and the first `Page.IsValid` check will always return true. However, when you call `Page.Validate()`, this all changes. After validating the first group, the `Page.IsValid` property will return true or false, depending on whether there is text in `TextBox1`. When you call `Page.Validate()` again to check the second group, the page becomes valid as long as the second group is valid (regardless of whether the first group is).

## Rich Controls

*Rich controls* are web controls that model complex user interface elements. Although no strict definition exists for what is and isn't a rich control, the term commonly describes web controls that provide an object model that is distinctly separate from the HTML it generates. A typical rich control can often be programmed as a single object (and defined with a single control tag) but renders itself with a complex sequence of HTML elements and may even use client-side JavaScript.

ASP.NET includes numerous rich controls that are discussed elsewhere in this book, including data-based list controls, security controls, and controls tailored for web portals. The following list identifies the rich controls that don't fall into any specialized category. The rich controls in this list all appear in the Standard tab of the Visual Studio Toolbox.

***AdRotator:*** A banner ad that displays one of a set of images based on a predefined schedule that's saved in an XML file.

***Calendar:*** A calendar that displays and allows you to move through months and days and to select a date or a range of days.

***MultiView, View, and Wizard:*** You can think of these controls as more advanced panels that let you switch between groups of controls on a page. These controls are described later in this chapter (in the section "Pages with Multiple Views").

***TreeView and Menu:*** These are two of the most impressive rich controls. Both allow you to show multilayered data, such as a menu with multiple levels or a hierarchical tree. They're often used for website navigation (see Chapter 11).

***Xml:*** This takes an XML file and an XSLT style sheet file as input and displays the resulting HTML in a browser. You'll learn about the `Xml` control in Chapter 17.

One of the best features of ASP.NET's control model is that other developers can create their own rich controls, which can then be incorporated into any ASP.NET application. You'll get a taste of this in Chapter 25, when you learn to create your own controls. Even without this knowledge, however, you can already start to use some of these advanced third-party controls. These controls provide features unlike any HTML element—including advanced grids, charting tools, and menus.

ASP.NET custom controls act like web controls in every sense. Your web page interacts with the appropriate control object, and the final output is rendered automatically every time the page is sent to the client as HTML. That means the controls need to be installed only on your server, and any client can benefit from them.

---

**Tip** The Internet contains many hubs for control sharing. One such location is Microsoft's own <http://www.asp.net>, which provides a control gallery where developers can submit their own ASP.NET web controls. Some of these controls are free (at least in a limited version), and others require a purchase.

---

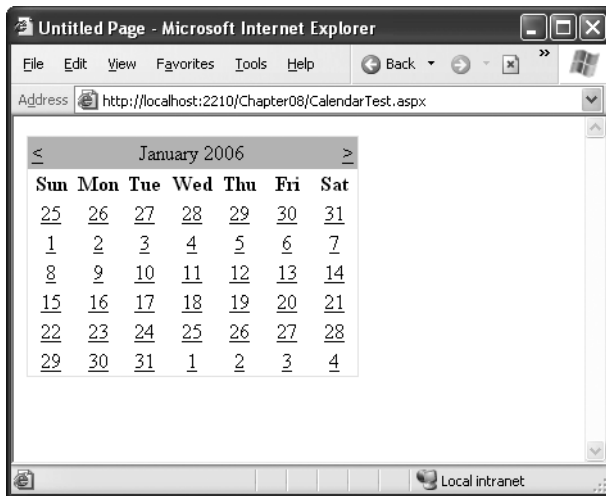
In the following sections, you'll learn about two ASP.NET rich controls: the Calendar and the AdRotator.

## The Calendar Control

The Calendar control is one of the most impressive web controls. It's commonly called a rich control because it can be programmed as a single object (and defined in a single simple tag) but rendered in dozens of lines of HTML output.

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

The Calendar control presents a single-month view, as shown in Figure 8-8. 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. You can retrieve the selected day in your code as a `DateTime` object from the `Calendar.SelectedDate` property.



**Figure 8-8.** *The default Calendar*

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.

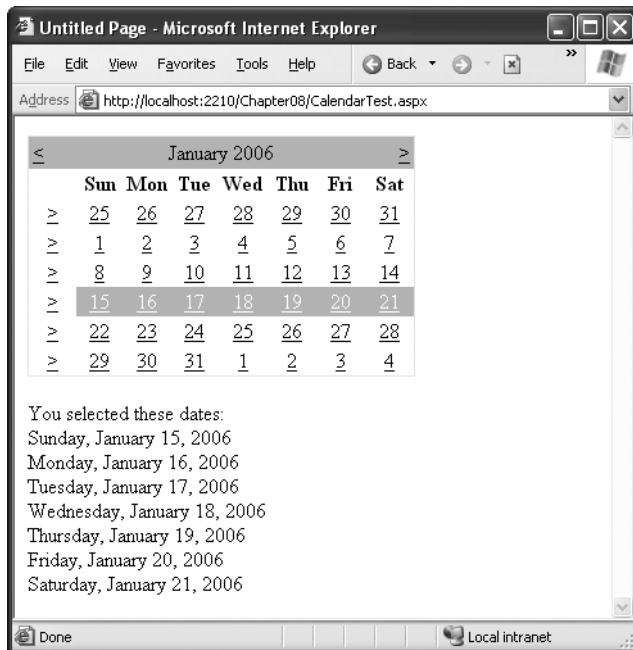
You set the type of selection through the `Calendar.SelectionMode` property. You may also need to set the `Calendar.FirstDayOfWeek` property to configure how a week is selected. (For example, set `FirstDayOfWeek` to the enumerated value `Monday`, and weeks will be selected from Monday to Sunday.)

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:

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

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

Figure 8-9 shows the resulting page after this code has been executed.



**Figure 8-9.** *Selecting multiple dates*

## Formatting the Calendar

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 8-6 lists the style properties that the Calendar control provides.

**Table 8-6.** *Properties for Calendar Styles*

Member	Description
<code>DayHeaderStyle</code>	The style for the section of the Calendar that displays the days of the week (as column headers).
<code>DayStyle</code>	The default style for the dates in the current month.
<code>NextPrevStyle</code>	The style for the navigation controls in the title section that move from month to month.

*Continued*

Table 8-6. *Continued*

Member	Description
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 TodayDate property of the Calendar).
WeekendDayStyle	The style for dates that fall on the weekend.

You can adjust each style using the Properties window. For a quick shortcut, you can set an entire related color scheme using the Calendar designer. Simply right-click the Calendar control on your design page, and select Auto Format. You'll be presented with a list of pre-defined formats that set the style properties, as shown in Figure 8-10.



Figure 8-10. *Calendar styles*

You can also use additional properties to hide some elements or configure the text they display.

## Restricting Dates

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 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 2010:

```
protected void 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;
    }
}
```

The `e.Day` object is an instance of the `CalendarDay` class, which provides various useful properties, as described in Table 8-7.

**Table 8-7.** *CalendarDay Properties*

Property	Description
Date	The <code>DateTime</code> object that represents this date.
IsWeekend	True if this date falls on a Saturday or Sunday.
IsToday	True if this value matches the <code>Calendar.TodaysDate</code> 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.

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 really a sophisticated 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:

```
protected void 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);
    }
}
```

Figure 8-11 shows the resulting calendar display.

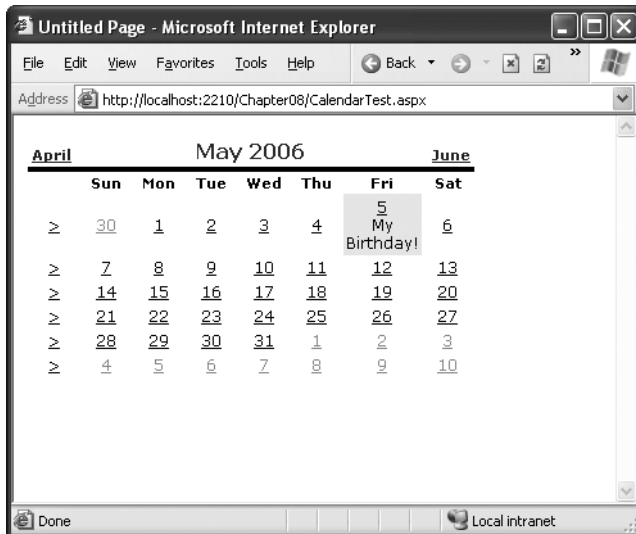


Figure 8-11. Highlighting a day



The Calendar control provides two other useful events: `SelectionChanged` and `VisibleMonthChanged`. These occur immediately after a change but before the page is returned to the user. You can react to this event and update other portions of the web page to correspond to the current calendar month. For example, you might want to set a corresponding list of times in a list control. The following code demonstrates this approach, using a different set of time values if a Monday is selected in the Calendar:

```
protected Sub SelectionChanged(Object source, EventArgs e)
{
    lstTimes.Items.Clear();

    switch (MyCalendar.SelectedDate.DayOfWeek)
    {
        case DayOfWeek.Monday:
            // Apply special Monday schedule.
            lstTimes.Items.Add("10:00");
            lstTimes.Items.Add("10:30");
            lstTimes.Items.Add("11:00");
            break;
        default:
            lstTimes.Items.Add("10:00");
            lstTimes.Items.Add("10:30");
            lstTimes.Items.Add("11:00");
            lstTimes.Items.Add("11:30");
            lstTimes.Items.Add("12:00");
            lstTimes.Items.Add("12:30");
            break;
    }
}
```

To try these features of the Calendar control, run the `Appointment.aspx` page from the online samples. This page provides a formatted Calendar control that restricts some dates, formats others specially, and updates a corresponding list control when the selection changes.

Table 8-8 gives you an at-a-glance look at almost all the members of the Calendar control class.

**Table 8-8.** *Calendar Members*

Member	Description
Caption and CaptionAlign	Gives you an easy way to add a title to the Calendar. By default, the caption appears at the top of the title area, just above the month heading. However, you can control this to some extent with the CaptionAlign property. Use Left or Right to keep the caption at the top but move it to one side or the other, and use Bottom to place the caption under the Calendar.
CellPadding	ASP.NET creates a date in a separate cell of an invisible table. CellPadding is the space, in pixels, between the border of each cell and its contents.
CellSpacing	The space, in pixels, between cells in the same table.
DayNameFormat	Determines how days are displayed in the Calendar header. Valid values are Full (as in Sunday), FirstLetter (S), FirstTwoLetters (Su), and Short (Sun), which is the default.
FirstDayOfWeek	Determines which day is displayed in the first column of the calendar. The values are any day name from the FirstDayOfWeek enumeration (such as Sunday).
NextMonthText and PrevMonthText	Sets the text that the user clicks to move to the next or previous month. These navigation links appear at the top of the Calendar and are the greater-than (>) and less-than (<) signs by default. This setting is applied only if NextPrevFormat is set to Custom.
NextPrevFormat	Sets the text that the user clicks to move to the next or previous month. This can be FullMonth (for example, December), ShortMonth (Dec), or Custom, in which case the NextMonthText and PrevMonthText properties are used. Custom is the default.
SelectedDate and SelectedDates	Sets or gets the currently selected date as a DateTime object. You can specify this in the control tag in a format like this: "12:00:00 AM, 12/31/2005" (depending on your computer's regional settings). If you allow multiple date selection, the SelectedDates property will return a collection of DateTime objects, one for each selected date. You can use collection methods such as Add, Remove, and Clear to change the selection.
SelectionMode	Determines how many dates can be selected at once. The default is Day, which allows one date to be selected. Other options include DayWeek (a single date or an entire week) or DayWeekMonth (a single date, entire week, or entire month). You have no way to allow the user to select multiple noncontiguous dates. You also have no way to allow larger selections without also including smaller selections. (For example, if you allow full months to be selected, you must also allow week selection and individual day selection.)

Member	Description
SelectMonthText and SelectWeekText	The text shown for the link that allows the user to select an entire month or week. These properties don't apply if the SelectionMode is Day.
ShowDayHeader, ShowGridLines, ShowNextPrevMonth, and ShowTitle	These Boolean properties allow you to configure whether various parts of the calendar are shown, including the day titles, gridlines between every day, the previous/next month navigation links, and the title section. Note that hiding the title section also hides the next and previous month navigation controls.
TitleFormat	Configures how the month is displayed in the title area. Valid values include Month and MonthYear (the default).
TodaysDate	Sets which day should be recognized as the current date and formatted with the TodayDayStyle. This defaults to the current day on the web server.
VisibleDate	Gets or sets the date that specifies what month will be displayed in the Calendar. This allows you to change the Calendar display without modifying the current date selection.
DayRender event	Occurs once for each day that is created and added to the currently visible month before the page is rendered. This event gives you the opportunity to apply special formatting, add content, or restrict selection for an individual date cell.
SelectionChanged event	Occurs when the user selects a day, a week, or an entire month by clicking the date selector controls.
VisibleMonthChanged event	Occurs when the user clicks the next or previous month navigation controls to move to another month.

## The AdRotator

The basic purpose of the AdRotator is to provide a banner-type graphic on a page (often used as an advertisement link to another site) that is chosen randomly from a group of possible banners. In other words, every time the page is requested, a different banner could be chosen and displayed, which is the “rotation” indicated by the name AdRotator.

In 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 banner, you would have to either 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 a special 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>
```

This example shows a single possible advertisement. 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 Table 8-9.

**Table 8-9.** *Advertisement File Elements*

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.
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.

Element	Description
Impressions	A number that sets how often an advertisement will appear. This number is relative to the numbers specified for other ads. For example, a banner with the value 10 will be shown twice as often 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 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.

## 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 8-10.

**Table 8-10.** *Special Frame Targets*

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).

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" />
```

---

**Tip** In Visual Studio, you can't link to an advertisement file unless you have added it to the current project.

---

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 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 8-12.



**Figure 8-12.** *An AdRotator with synchronized content*

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

```
protected void Ads_AdCreated(Object sender, AdCreatedEventArgs e)
{
    // Synchronize the Hyperlink control.
    lnkBanner.NavigateUrl = e.NavigateUrl;

    // Synchronhize 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.

## Pages with Multiple Views

In a typical website, you'll surf through many separate pages. For example, if you want to add an item to your shopping cart and take it to the checkout in an e-commerce site, you'll need to jump from one page to another. This design has its advantages—namely, it lets you carefully separate different tasks into different code files. It also presents some challenges; for example, you need to come up with a way to transfer information from one page to another (a topic that's covered in detail in Chapter 9).

However, in some cases it makes more sense to create a single page that can handle several different tasks. For example, you might want to provide several views of the same data (such as a grid-based view and a chart-based view) and allow the user to switch from one view to the other without leaving the page. Or, you might want to handle a small multistep task in one place (such as supplying user information for an account sign-up process). In these examples, you need a way to create dynamic pages that provide more than one possible view. Essentially, the page hides and shows different controls depending on which view you want to present.

The simplest way to understand this technique is to create a page with several Panel controls. Each panel can hold a group of ASP.NET controls. For example, imagine you're creating a simple three-step wizard. You'll start by adding three panels to your page, one for each step—say, `panelStep1`, `panelStep2`, and `panelStep3`. Then, you'll place the appropriate controls inside each panel. To start, the `Visible` property of each panel should be false, except for `panelStep1`, which appears the first time the user request the page.

---

**Note** When you set the `Visible` property of a control to false, the control won't appear in the page at runtime—in fact, no HTML will be generated for it. Any controls inside an invisible panel are also hidden from sight. However, the control will still appear in the Visual Studio design surface so that you can still select it and configure it.

---

Finally, you'll add one or more navigation buttons. For example, the following code handles the click of a Next button. It checks which step the user is currently on, hides the current panel, and shows the following panel. This way the user is moved to the next step.

```
protected void cmdNext_Clicked(object sender, EventArgs e)
{
    if (panelStep1.Visible)
    {
        // Move to step 2.
        panelStep1.Visible = false;
        panelStep2.Visible = true;
    }
    else if (panelStep2.Visible)
    {
        // Move to step 3.
        panelStep2.Visible = false;
        panelStep3.Visible = true;
    }
}
```

```

        // Change text of button from Next to Finish.
        cmdNext.Text = "Finish";
    }
    else if (panelStep3.Visible)
    {
        // The wizard is finished.
        panelStep3.Visible = false;

        // Add code here to perform the appropriate task
        // with the information you've collected.
        lblInfo.Text = "Wizard Finished.";
    }
}

```

This approach works relatively well. Even when the panels are hidden, you can still interact with all the controls on each panel and retrieve the information they contain. The problem is that you need to write all the code for controlling which panel is visible. If you make your wizard much more complex—for example, you want to add a button for returning to a previous step—it becomes more difficult to keep track of what’s happening. At best, this approach clutters your page with the code for managing the panels. At worst, you’ll make a minor mistake and end up with two panels showing at the same time.

Fortunately, ASP.NET gives you a more robust option. You can use two controls that are designed for the job—the `MultiView` and the `Wizard`. In the following sections, you’ll see how you can use both of these controls with the `GreetingCardMaker` example developed in Chapter 6.

## The MultiView Control

The `MultiView` is the simpler of the two multiple-view controls. Essentially, the `MultiView` gives you a way to declare multiple views and show only one at a time. It has no default user interface—you get only whatever HTML and controls you add. The `MultiView` is equivalent to the custom panel approach explained earlier.

Creating a `MultiView` is suitably straightforward. You add the `<asp:MultiView>` tag to your .aspx page file and then add one `<asp:View>` tag inside it for each separate view:

```

<asp:MultiView ID="MultiView1" runat="server">
    <asp:View ID="View1" runat="server">...</asp:View>
    <asp:View ID="View2" runat="server">...</asp:View>
    <asp:View ID="View3" runat="server">...</asp:View>
</asp:MultiView>

```



In Visual Studio, you create these tags by first dropping a MultiView control onto your form and then using the Toolbox to add as many View controls inside it as you want. The View control plays the same role as the Panel control in the previous example, and the MultiView takes care of coordinating all the views so that only one is visible at a time.

Inside each view, you can add HTML or web controls. For example, consider the GreetingCardMaker example demonstrated in Chapter 6, which allows the user to create a greeting card by supplying some text and choosing colors, a font, and a background. As the GreetingCardMaker grows more complex, it requires more controls, and it becomes increasingly difficult to fit all those controls on the same page. One possible solution is to divide these controls into logical groups and place each group in a separate view.

## Creating Views

Here's the full markup for a MultiView that splits the greeting card controls into three views named View1, View2, and View3:

```
<asp:MultiView id="MultiView1" runat="server" >

    <asp:View ID="View1" runat="server">
        Choose a foreground (text) color:<br />
        <asp:DropDownList ID="lstForeColor" runat="server" AutoPostBack="True"
            OnSelectedIndexChanged="ControlChanged" />
        <br /><br />
        Choose a background color:<br />
        <asp:DropDownList ID="lstBackColor" runat="server" AutoPostBack="True"
            OnSelectedIndexChanged="ControlChanged" />
    </asp:View>

    <asp:View ID="View2" runat="server">
        Choose a border style:<br />
        <asp:RadioButtonList ID="lstBorder" runat="server" AutoPostBack="True"
            OnSelectedIndexChanged="ControlChanged" RepeatColumns="2" />
        <br />
        <asp:CheckBox ID="chkPicture" runat="server" AutoPostBack="True"
            OnCheckedChanged="ControlChanged" Text="Add the Default Picture" />
    </asp:View>

    <asp:View ID="View3" runat="server">
        Choose a font name:<br />
        <asp:DropDownList ID="lstFontName" runat="server" AutoPostBack="True"
            OnSelectedIndexChanged="ControlChanged" />
        <br /><br />
        Specify a font size:<br />
```

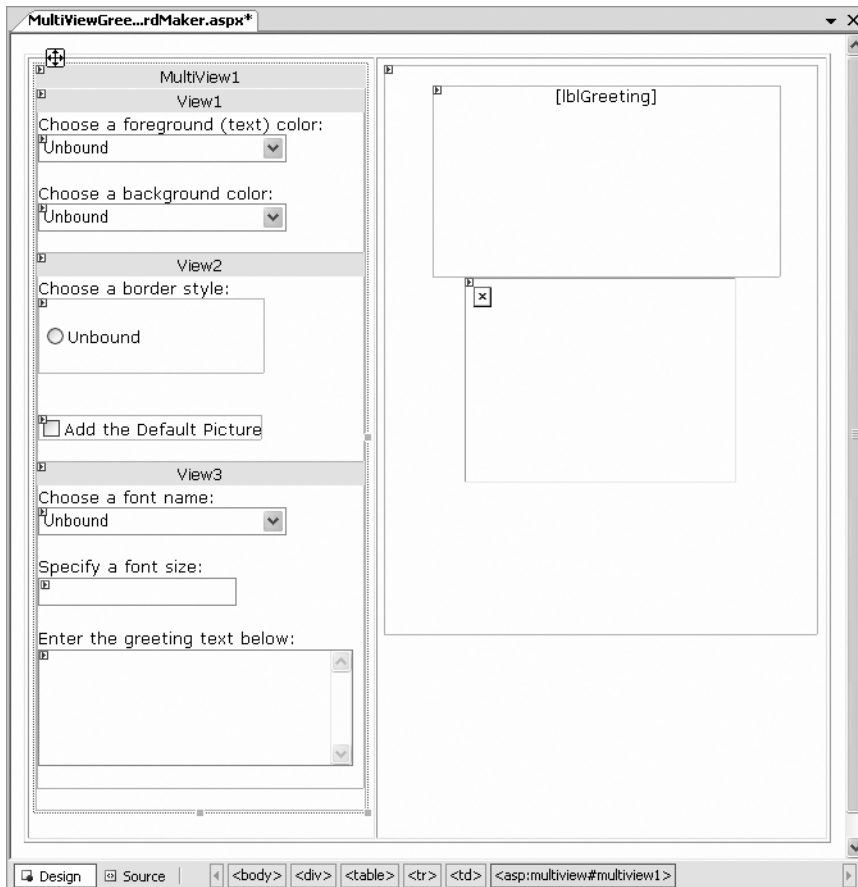
```

<asp:TextBox ID="txtFontSize" runat="server" AutoPostBack="True"
  OnTextChanged="ControlChanged" />
<br /><br />
Enter the greeting text below:<br />
<asp:TextBox ID="txtGreeting" runat="server" AutoPostBack="True"
  OnTextChanged="ControlChanged" TextMode="MultiLine" />
</asp:View>

</asp:MultiView>

```

Visual Studio shows all your views at design time, one after the other (see Figure 8-13). You can edit these regions in the same way you design any other part of the page.



**Figure 8-13.** *Designing multiple views*

## Showing a View

If you run this example, you won't see what you expect. The `MultiView` will appear empty on the page, and all the controls in all your views will be hidden.

The reason this happens is because the `MultiView.ActiveViewIndex` property is, by default, set to `-1`. The `ActiveViewIndex` property determines which view will be shown. If you set the `ActiveViewIndex` to `0`, however, you'll see the first view. Similarly, you can set it to `1` to show the second view, and so on. You can set this property using the Properties window or using code:

```
// Show the first view.  
MultiView1.ActiveViewIndex = 0;
```

This example shows the first view (`View1`) and hides whatever view is currently being displayed, if any.

---

**Tip** To make more readable code, you can create an enumeration that defines a name for each view. That way, you can set the `ActiveViewIndex` using the descriptive name from the enumeration rather than an ordinary number. Refer to Chapter 3 for a refresher on enumerations.

---

You can also use the `SetActiveView()` method, which accepts any one of the view objects you've created. This may result in more readable code (if you've chosen descriptive IDs for your view controls), and it ensures that any errors are caught earlier (at compile time instead of runtime).

```
MultiView.SetActiveView(View1);
```

This gives you enough functionality that you can create previous and next navigation buttons. However, it's still up to you to write the code that checks which view is visible and changes the view. This code is a little simpler, because you don't need to worry about hiding views any longer, but it's still less than ideal.

Fortunately, the `MultiView` includes some built-in smarts that can save you a lot of trouble. Here's how it works: The `MultiView` recognizes buttons 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 8-11 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 8-11.** *Recognized Command Names for the MultiView*

Command Name	MultiView Field	Description
PrevView	PrevViewCommandName	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.

To try this, add this button to the first view:

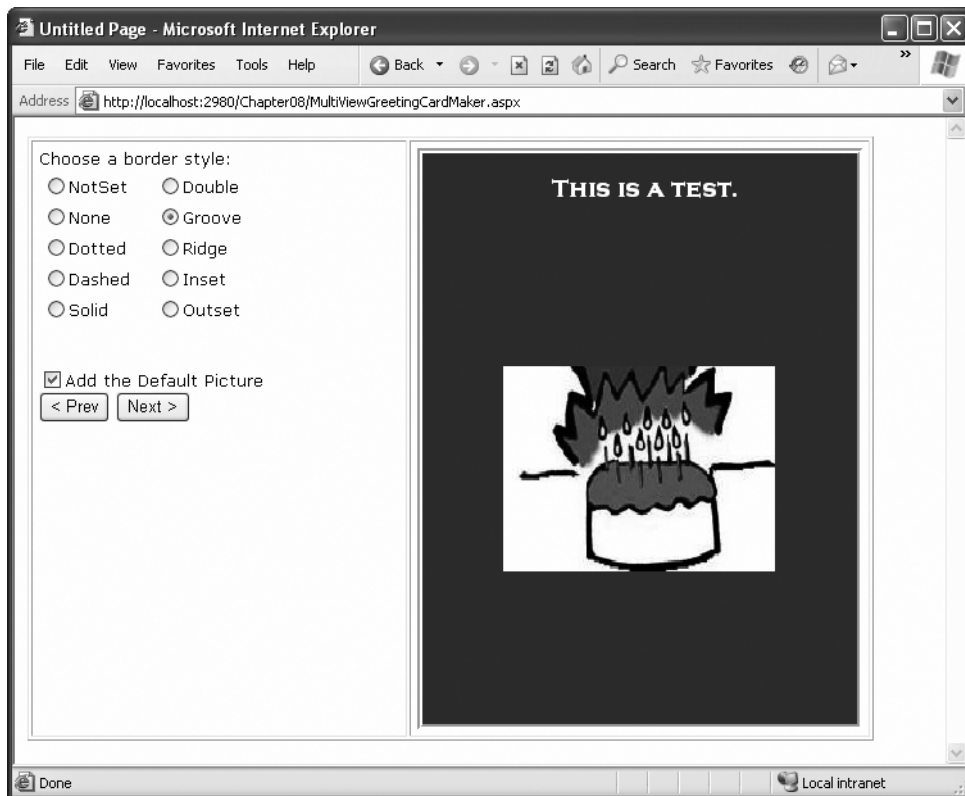
```
<asp:Button ID="Button1" runat="server" CommandArgument="View2"  
CommandName="SwitchViewByID" Text="Go to View2" />
```

When clicked, this button sets the MultiView to show the view specified by the CommandArgument (View2).

Rather than create buttons that take the user to a specific view, you might want a button that moves forward or backward one view. To do this, you use the PrevView and NextView command names. Here's an example that defines previous and next buttons:

```
<asp:Button ID="Button1" runat="server" Text="< Prev" CommandName="PrevView" />  
<asp:Button ID="Button2" runat="server" Text="Next >" CommandName="NextView" />
```

Once you add these buttons to your view, you can move from view to view easily. Figure 8-14 shows the previous example with the second view currently visible.



**Figure 8-14.** *Moving from one view to another*

---

**Tip** Be careful how many views you cram into a single page. When you use the MultiView control, the entire control model—including the controls from every view—is created on every postback and persisted to view state. For the most part, this won't be a significant factor. However, it increases the overall page size, especially if you're tweaking controls programmatically (which increases the amount of information they need to store in view state).

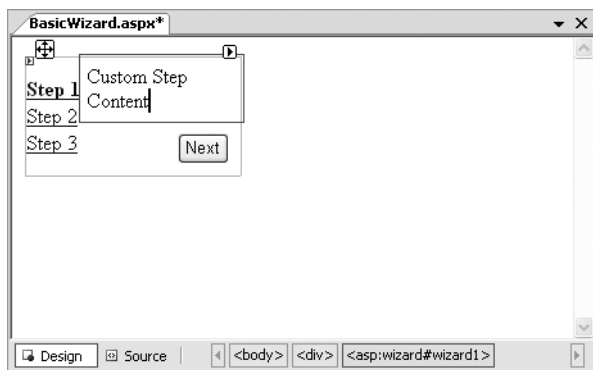
---

## The Wizard Control

The Wizard control is a more glamorous version of the MultiView control. It also supports showing one of several views at a time, but it includes a fair bit of built-in yet customizable behavior, including navigation buttons, a sidebar with step links, styles, and templates.

Usually, wizards represent a single task, and the user moves linearly through them, moving from the current step to the one immediately following it (or the one immediately preceding it in the case of a correction). The ASP.NET Wizard control also supports non-linear navigation, which means it allows you to decide to ignore a step based on the information the user supplies.

By default, the Wizard control supplies navigation buttons and a sidebar with links for each step on the left. You can hide the sidebar by setting the `Wizard.DisplaySideBar` property to false. Usually, you'll take this step if you want to enforce strict step-by-step navigation and prevent the user from jumping out of sequence. You supply the content for each step using any HTML or ASP.NET controls. Figure 8-15 shows the region where you can add content to an out-of-the-box Wizard instance.



**Figure 8-15.** *The region for step content*

### Wizard Steps

To create a wizard in ASP.NET, you simply define the steps and their content using `<asp:WizardStep>` tags. Each step takes a few basic pieces of information, as listed in Table 8-12.

**Table 8-12.** *WizardStep Properties*

Property	Description
Title	The descriptive name of the step. This name is used for the text of the links in the sidebar.
StepType	The type of step, as a value from the WizardStepType enumeration. This value determines the type of navigation buttons that will be shown for this step. Choices include Start (shows a Next button), Step (shows Next and Previous buttons), Finish (shows a Finish and Previous button), Complete (show no buttons and hides the sidebar, if it's enabled), and Auto (the step type is inferred from the position in the collection). The default is Auto, which means the first step is Start, the last step is Finish, and all other steps are Step.
AllowReturn	Indicates whether the user can return to this step. If false, once the user has passed this step, the user will not be able to return. The sidebar link for this step will have no effect, and the Previous button of the following step will either skip this step or be hidden completely (depending on the AllowReturn value of the preceding steps).

To see how this works, consider a wizard that again uses the GreetingCardMaker example. It guides the user through four steps. The first three steps allow the user to configure the greeting card, and the final step shows the generated card.

```
<asp:Wizard ID="Wizard1" runat="server" ActiveStepIndex="0"
  BackColor="LemonChiffon" BorderStyle="Groove" BorderWidth="2px"
  CellPadding="10">

  <WizardSteps>
    <asp:WizardStep runat="server" Title="Step 1 - Colors">
      Choose a foreground (text) color:<br />
      <asp:DropDownList ID="lstForeColor" runat="server" />
      <br />
      Choose a background color:<br />
      <asp:DropDownList ID="lstBackColor" runat="server" />
    </asp:WizardStep>

    <asp:WizardStep runat="server" Title="Step 2 - Background">
      Choose a border style:<br />
      <asp:RadioButtonList ID="lstBorder" runat="server" RepeatColumns="2" />
      <br /><br />
      <asp:CheckBox ID="chkPicture" runat="server"
        Text="Add the Default Picture" />
    </asp:WizardStep>
```





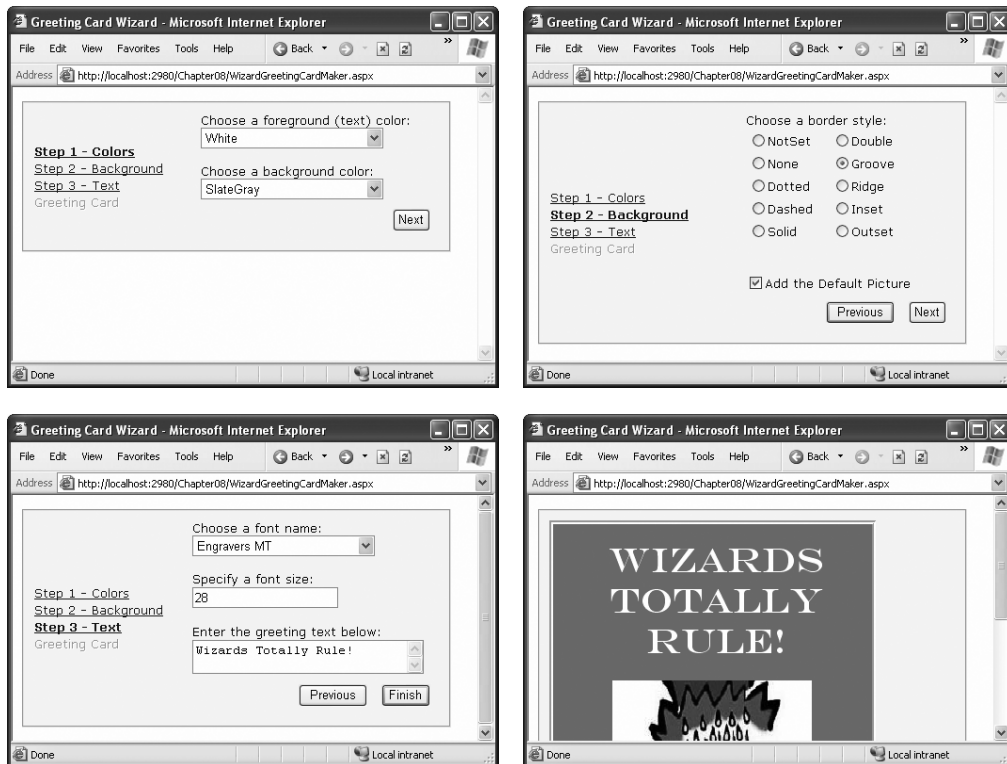


Figure 8-16. A wizard with four steps

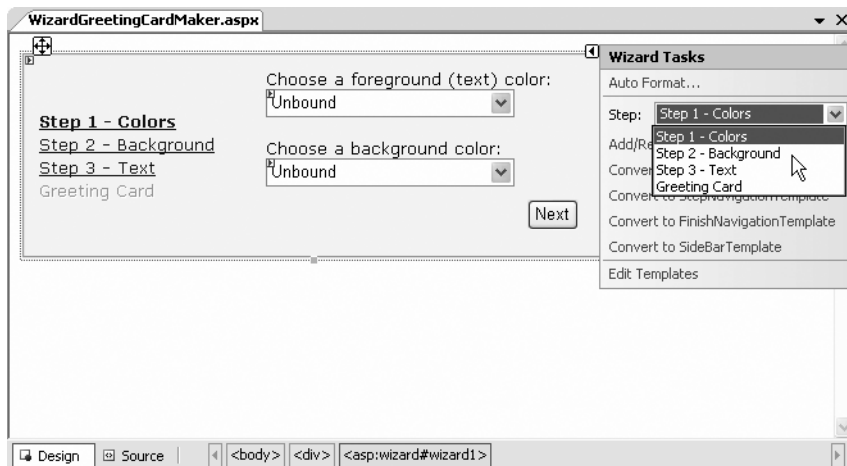


Figure 8-17. Designing a step

---

**Note** Remember, when you add controls to separate steps on a wizard, the controls are all instantiated and persisted in view state, regardless of which step is currently shown. If you need to slim down a complex wizard, you'll need to split it into separate pages, use the `Server.Transfer()` method to move from one page to the next, and tolerate a less elegant programming model.

---

## Wizard Events

You can write the code that underpins your wizard by responding to several events (as listed in Table 8-13).

**Table 8-13.** *Wizard Events*

Event	Description
ActiveStepChanged	Occurs when the control switches to a new step (either because the user has clicked a navigation button or your code has changed the <code>ActiveStepIndex</code> property).
CancelButtonClick	Occurs when the Cancel button is clicked. The Cancel button is not shown by default, but you can add it to every step by setting the <code>Wizard.DisplayCancelButton</code> property. Usually, a Cancel button exits the wizard. If you don't have any cleanup code to perform, just set the <code>CancelDestinationPageUrl</code> property, and the wizard will take care of the redirection automatically.
FinishButtonClick	Occurs when the Finish button is clicked.
NextButtonClick and PreviousButtonClick	Occurs when the Next or Previous button is clicked on any step. However, because there is more than one way to move from one step to the next, it's better to handle the <code>ActiveStepChanged</code> event.
SideBarButtonClick	Occurs when a button in the sidebar area is clicked.

On the whole, two wizard programming models exist:

*Commit-as-you-go:* This makes sense if each wizard step wraps an atomic operation that can't be reversed. For example, if you're processing an order that involves a credit card authorization followed by a final purchase, you can't allow the user to step back and edit the credit card number. To support this model, you set the `AllowReturn` property to false on some or all steps. You may also want to respond to the `ActiveStepChanged` event to commit changes for each step.

*Commit-at-the-end:* This makes sense if each wizard step is collecting information for an operation that's performed only at the end. For example, if you're collecting user information and plan to generate a new account once you have all the information,

you'll probably allow a user to make changes midway through the process. You execute your code for generating the new account when the wizard ends by reacting to the `FinishButtonClick` event.

To implement commit-at-the-end with the current example, just respond to the `FinishButtonClick` event. For example, to implement the greeting card wizard, you simply need to respond to this event to call `Update()`, the private method that refreshes the greeting card:

```
protected void Wizard1_FinishButtonClick(object sender,
    WizardNavigationEventArgs e)
{
    Update();
}
```

For the complete code, refer to Chapter 6 (or check out the downloadable sample code). If you decide to use the commit-as-you go model, you would respond to the `ActiveStepChanged` event and call `Update()` at that point to refresh the card every time the user moves from one step to another. This assumes the greeting card is always visible. (In other words, it's not contained in the final step of the wizard.) The commit-as-you-go model is similar to the previous example that used the `MultiView`.

## Formatting the Wizard

Without a doubt, the Wizard control's greatest strength is the way it lets you customize its appearance. This means if you want the basic model (a multistep process with navigation buttons and various events), you aren't locked into the default user interface.

Depending on how radically you want to change the wizard, you have several options. For less dramatic modifications, you can set various top-level properties. For example, you can control the colors, fonts, spacing, and border style, as you can with any ASP.NET control. You can also tweak the appearance of every button. For example, to change the Next button, you can use the following properties: `StepNextButtonType` (use a button, link, or clickable image), `StepNextButtonText` (customize the text for a button or link), `StepNextButtonImageUrl` (set the image for an image button), and `StepNextButtonStyle` (use a style from a style sheet). You can also add a header using the `HeaderText` property.

More control is available through styles. You can use styles to apply formatting options to various portions of the Wizard control just as you can use styles to format parts of rich data controls such as the `GridView`. Table 8-14 lists all the styles you can use. As with other style-based controls, more specific style settings (such as `SideBarStyle`) override more general style settings (such as `ControlStyle`) when they conflict. Similarly, `StartNextButtonStyle` overrides `NavigationButtonStyle` on the first step.

**Table 8-14.** *Wizard Styles*

Style	Description
ControlStyle	Applies to all sections of the Wizard control
HeaderStyle	Applies to the header section of the wizard, which is visible only if you set some text in the HeaderText property
SideBarStyle	Applies to the sidebar area of the wizard
SideBarButtonStyle	Applies to just the buttons in the sidebar
StepStyle	Applies to the section of the control where you define the step content
NavigationStyle	Applies to the bottom area of the control where the navigation buttons are displayed
NavigationButtonStyle	Applies to just the navigation buttons in the navigation area
StartNextButtonStyle	Applies to the Next navigation button on the first step (when StepType is Start)
StepNextButtonStyle	Applies to the Next navigation button on intermediate steps (when StepType is Step)
StepPreviousButtonStyle	Applies to the Previous navigation button on intermediate steps (when StepType is Step)
FinishPreviousButtonStyle	Applies to the Previous navigation button on the last step (when StepType is Finish)
CancelButtonStyle	Applies to the Cancel button, if you have Wizard. DisplayCancelButton set to true

---

**Note** The Wizard control also supports templates, which gives you a more radical approach to formatting. If you can't get the level of customization you want through properties and styles, you can use templates to completely define the appearance of each section of the Wizard control, including the headers and navigation links. Templates require data binding expressions and are discussed in Chapter 14 and Chapter 15.

---

## The Last Word

This chapter showed you how validation controls, the rich Calendar and AdRotator controls, and the MultiView and Wizard controls can go far beyond the limitations of ordinary HTML elements.

Throughout this book, you'll consider some more examples of rich controls and learn how to use them to create rich web applications that are a world apart from HTML basics. Some of the most exciting rich controls that are still ahead include the navigation controls (Chapter 11) and the data controls (Chapter 15).