# Data Entry and Validation with C# and VB .NET Windows Forms

NICK SYMMONDS

Apress™

Data Entry and Validation with C# and VB .NET Windows Forms



ISBN (pbk): 1-59059-108-9

Printed and bound in the United States of America 12345678910



Technical Reviewer: Adriano Baglioni

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wright, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Beth Christmas

Copy Editor: Nicole LeClerc

Production Manager: Kari Brooks

Proofreader: Linda Siefert

Compositor: Susan Glinert Stevens

Indexer: Rebecca Plunkett

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

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, email info@apress.com, or visit http://www.apress.com.



The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 8

# Advanced Validation and Custom Data Validation Controls

**I TOOK A BREAK** in Chapter 7 to discuss error handling. Although it may not be thought of traditionally as part of the world of data entry, error handling is even more important than normal in this arena.

This chapter covers some of the more advanced topics of data validation. You will see the Masked Edit control that was popular in VB 6.0. You will also see how to use the regular expression capability of .NET to generate short and sweet expressions for complicated data validation. Finally, you will learn how to make your own data validation control similar to the one that comes with ASP.NET.

In the next section you'll explore the most advanced, and the most flexible, of the data validation techniques: regular expressions.

## Regular Expressions in .NET

To my mind there is nothing regular about regular expressions. Consider this regular expression: `href\s*=\s*(?:""(?<1>[^""]*)""|(?<1>\S+)`.[1] Now what is so regular about this? I think that the expression "Find all the href="…" values and their locations in a string" is a lot more regular.

Although my English language expression may seem more regular and is certainly more humanly readable, the .NET Framework interprets the actual regular expression very nicely. As you know, being a programmer forces you to talk to the computer in its own language.[2] Make no mistake, "Regular Expression" may not be a language, but it has a syntax just like any another computer language such as C# or VB.

So, what are regular expressions used for and how do you use them in .NET? Regular expressions are used to parse strings. However, this is a bit simplistic. Parsing

1. I extracted this example from the online help
2. This is not *Star Trek* . . . yet.

strings brings to mind reading a line of text and extracting a substring from that text. Parsing also brings to mind extracting information from a comma-delimited file.

Regular expressions are much more than just parsing, though. Regular expressions in .NET can extract, insert, change, and delete any pattern in any string either forward or backward. This is powerful stuff.

Were regular expressions invented for .NET? No, they have been around longer than I have been programming, which is a very long time. For those real oldies who used the code editor Brief, regular expressions were a part of daily programming life. Those of you from the UNIX world dreamed in regular expression syntax.

## *The RegularExpressions Namespace*

A whole namespace is devoted to regular expressions and their use: System.Text.RegularExpressions. In here you will find eight classes and one enumeration devoted to regular expressions. You will even find an event that you can hook to for custom validation during a matching operation. I go over these classes lightly in this section just to give you an idea of what they are used for. After this I charge headlong into the geeky world of regular expression usage.

The Capture class provides a result from a regular expression's subexpression capture. This means that if you use regular expressions to extract a substring, this is where you would find the answer. This class is *immutable,* meaning its properties cannot be changed. You will not be able to instantiate this class, and it does not appear by magic. Instead, you get an instance of this class from the CaptureCollection class.

The CaptureCollection class is a collection of Capture classes.[3] So, as you probably guessed, this is a collection of the entire set of substrings returned by a particular regular expression search.

If you have a complicated regular expression that includes more than one substring search, how do you get the instances of the Capture class? You can't get them all from a CaptureCollection, because this gives you only the Capture instances for a single subexpression. What you need is a group.

The Group class is used to hold a collection of CaptureCollections. At the very least it will hold a collection of one Capture object. At most it will hold as many CaptureCollections as are needed by the expression.

**NOTE** You definitely should know that collections could contain collections ad infinitum. If you don't, then study the collection classes for some more examples.

3. Did you guess that?

Of course, you cannot have a Group object without a collection of Group objects to hold it. This is the GroupCollection class. It contains a set of groups resulting from a regular expression match.

If you were to supply a regular expression to the framework to evaluate, you would naturally need an object as a result. This object is the Match class. The Match class is derived from the Group class, which is in turn derived from the Capture class. Therefore, the Match class holds all the results from a single regular expression call. It is in the returned Match object that you start digging for your results.

The last biggie in this list of classes is the Regex class. This class holds the regular expression that you need evaluated. If you call Regex.Match with a regular expression, you will get back a Match object. Check the Success property of this object for any hits.

## The Regular Expression Syntax

Before I go on, you need to know a little about the regular expression syntax. First in the list are the escape characters. An *escape character* is a backslash followed by a special character or set of characters. For instance, in C-derived languages the escape character \n means newline, which most of the time gets converted to a carriage return/linefeed pair. Table 8-1 contains a list of regular expression escape characters.

*Table 8-1. Regular Expression Escape Characters*

| Character Sequence | Meaning |
|---|---|
| \a | Bell character. |
| \b | Backspace or word boundary. |
| \t | Tab. |
| \r | Carriage return. |
| \v | Vertical tab. |
| \f | Form feed. |
| \n | Newline. |
| \e | Escape. |
| \0nnn | *nn* represents an octal number. The whole expression represents an octal number. |
| \0xnn | *nn* represents a hex number. The hex number is an ASCII character. |
| \cA | ASCII control character. This is Ctrl-A. |

There are a few other characters, including back references, that are beyond the scope of this book. This next table is your first foray into the simple use of regular expressions. Table 8-2 contains a list of character matching commands.

*Table 8-2. Character Matching Commands*

| Command | Meaning |
|---|---|
| [abcd] | Matches anything in the brackets. |
| [^abcd] | Matches anything *not* in the brackets. |
| [0-9] | The dash is used as an extender; same as [0123456789]. |
| . | The period matches any character. |
| \p{name} | Matches any character in the named character class. |
| \P{name} | Matches any character *not* in the named character class. |
| \w | Matches any word or character that follows. |
| \W | Matches any word or character that *does not* follow. |
| \s | Matches any white space character. |
| \S | Matches any non-white-space character. |
| \d | Matches any decimal digit; same as [0-9]. |
| \D | Matches any nondecimal character; same as [^0-9]. |

Is this all you need to know about the grammar? No. There is a set of commands called *quantifiers* that you can use to add additional information to the search pattern. The quantifiers apply only to that group or character class that precedes them. So, for example, I could have the expression `[abcd]?`. This means that the question mark quantifier acts on the bracket pattern. In this case, instead of finding all matches of abcd, it only finds only zero or one match. Table 8-3 shows the list of quantifiers.

*Table 8-3. Regular Expression Quantifiers*

| Character(s) | Meaning |
|---|---|
| * | Zero or more matches |
| + | One or more matches |
| ? | Zero or one match |
| {n} | Exactly *n* matches |
| {n,} | At least *n* matches |
| {n,m} | At least *n* but no more than *m* matches |
| *? | Gets first match that consumes the fewest repeats |
| +? | Specifies as few repeats as possible but at least one repeat |
| ?? | Gets match using zero repeats if possible |
| {n}? | Same as {n} |
| {n,}? | Gets at least *n* matches with as few repeats as possible |
| {n,m}? | Gets at least *n* to *m* matches with as few repeats as possible |

Notice the overuse of the question mark? This is called the *lazy quantifier*. Usually the regular expression engine is greedy; it tries to find as many matches as possible with the constraints you gave it. The lazy quantifier tells the engine to match only what is necessary to achieve a match and nothing more. In essence, a quantifier token tells the parser how many times the previous expression should be matched. Once you start working with regular expressions, you will see that without quantifiers you can get major performance hits when running the regular expression engine. The trick is to be as specific as possible in your expression.

Here are some examples of regular expressions and their results. The text I am searching for in these examples is the same. The text being searched is the same between examples also. The only thing that changes is the quantifiers. If you are unfamiliar with regular expressions, hopefully this will clarify things for you.

Make a small console application in either VB or C#. The code for this application is shown here.

**C#**

```
using System;
using System.Text.RegularExpressions;

namespace RegX_c
{
  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      Regex r  = new Regex("Sp[ace] [1-9]*");

      for (Match m = r.Match("Space 1999 Spac 1999 Spa 1999 Sp 1999");
m.Success; m = m.NextMatch())
        Console.WriteLine(m.Value);

      Console.ReadLine();
    }
  }
}
```

**VB**

```
Option Strict On

Imports System
Imports System.Text.RegularExpressions

Module Module1

  Sub Main()
    Dim m As Match
    Dim text As String = "Space 1999 Spac 1999 Spa 1999 Sp 1999"

    Dim r As Regex = New Regex("Sp[ace] [1-9]*")

    m = r.Match(text)
    While m.Success
      Console.WriteLine(m.Value)
      m = m.NextMatch()
    End While
```

```
    Console.ReadLine()
  End Sub

End Module
```

I have here a text string that consists of several variations of the title to an old TV show called *Space 1999*. I have also instantiated a Regex object with the regular expression that determines the strings I am looking for. Here is the result of running this example:

```
Spa 1999
```

Exciting, isn't it? The regular expression told the parser to look for any strings that matched the *s,* followed by the *p,* followed by the *a,* followed by a space, followed by zero or more matches of the digits 1–9. Now, you're probably wondering, why didn't the parse spit back the actual text "Space 1999"?

When you are looking for something in brackets, it means match anything in there starting with the first character in the brackets. Because I had no quantifiers for the brackets, the default is to find only one match. The first character in the brackets is *a,* and this is what it started with. Note that the brackets represent one character position in the string. So, what happened is that the parser looked for "Spa 1999" as a first try, got a hit, and bailed out.

Change the regular expression to this:

```
"Sp[ace]? [1-9]*"
```

Now run the program and you should get the following results:

```
Spa 1999
Sp 1999
```

Why the two results? I had you add a ? quantifier to the bracketed text. Table 8-3 states that ? means find zero or one matches. So the parser first found zero matches in the form of "Sp 1999" and then it found one match in the form of "Spa 1999". Again, it spit these matches out and bailed. The parser did no more than what you told it to do.

Change the regular expression to this:

```
"Sp[ace]+ [1-9]*"
```

All you are doing is swapping out the ? quantifier with a + quantifier. Here are the results:

```
Space 1999
Spac 1999
Spa 1999
```

The + quantifier tells the parser to find one or more matches. It found all three. Remember that this quantifier has to find at least one match to succeed. Now change the quantifier from + to * like this:

```
"Sp[ace]* [1-9]*"
```

This means find zero or more matches. Here are the results:

```
Space 1999
Spac 1999
Spa 1999
Sp 1999
```

The parser found everything. The "Sp 1999" answer is the result of finding zero matches. The other answers are the result of finding the "or more" matches.

The three quantifiers ?, *, and + are like wild cards. You need to be careful with them because they could take up more time than you think and force your computer to come to a grinding halt. The only one that is really safe here is ?, the lazy quantifier. However, even this quantifier can return results you may not be looking for.

### *Exact Matching*

If you have a situation where you are looking for an exact number of hits, use exact matching. Change the regular expression again to this:

```
Sp[ace]{2} [1-9]*"
```

What you are doing here is telling the parser to find an exact match of two characters in the brackets in any order. Here is what it found:

```
Spac 1999
```

This does not tell you much. How about changing your search string to this:

```
"Space 1999 Spca 1999 Spac 1999 Spa 1999 Sp 1999"
```

You added an alternate spelling of "Spac" with "Spca". It uses the same two characters but swapped. Here is the result:

```
Spca 1999
Spac 1999
```

As you can see, the numerical quantifier does not care about the order of the characters in the brackets. It will ruthlessly hunt down any variation and tell you about it.

So, is this all there is to regular expressions? Not by a long shot. In fact, many articles and books are dedicated to the subject. I know a few people who pride themselves on inventing the most complicated-looking regular expressions you could imagine. I have a plan in mind for you regarding regular expressions, however, so I will show you only a little more.

### *Text Replacement*

There are two more features of regular expressions you need to know about: search-and-replace and search-and-delete. They are actually the same thing, but you can treat them as different here. Search and delete is actually search and replace with an empty string.

The Regex class has several static functions. These static functions allow you to input a regular expression and get an answer without having to compile the regular expression first. In this case, you will be using the overloaded function called Regex.Replace.

Essentially, this static function creates a one-time use of a Regex class, uses it for the intended purpose, and then throws it away. Here is a simple replacement function using one of the overloaded versions of the Regex.Replace method.

**C#**

```
private static void Replace()
{
  //Replace all instances of the word could with the word should
  string OrgString = "This could be done. It could be accomplished now. " +
                     "I couldn't get it done in time";
  string SearchPattern = "could ";
  string ReplacePattern = "should ";

  Console.WriteLine(OrgString + "\n\n");
  Console.WriteLine(Regex.Replace(OrgString, SearchPattern, ReplacePattern));

  Console.ReadLine();
}
```

**VB**

```
Sub Replace()
  'Replace all instances of the word could with the word should
  Dim OrgString As String = "This could be done. " + _
                            "It could be accomplished now. " + _
                            "I couldn't get it done in time"
  Dim SearchPattern As String = "could "
  Dim ReplacePattern As String = "should "

  Console.WriteLine(OrgString + vbCrLf + vbCrLf)
  Console.WriteLine(Regex.Replace(OrgString, SearchPattern, ReplacePattern))

  Console.ReadLine()
End Sub
```

This is about as simple as a replacement can get. I search for any instance of the string "could" and replace it with the string "should". I include the space in the search string to avoid the hit on the word "couldn't".

Suppose the first word of a sentence was capitalized? This replace expression would miss it. An easy way to fix this problem is to use the replace function twice.

**C#**

```
  private static void Replace2()
  {
    //Replace all instances of the word could with the word should
    string OrgString      = "Could it be done? It could be done now.";

    Console.WriteLine(OrgString + "\n");
    OrgString = Regex.Replace(OrgString, "Could", "Should");
    Console.WriteLine(Regex.Replace(OrgString, "could", "should"));

    Console.ReadLine();
  }
```

**VB**

```
Sub Replace2()
  'Replace all instances of the word could with the word should
  Dim OrgString As String = "Could it be done? It could be done now."
```

```
    Console.WriteLine(OrgString + "\n")
    OrgString = Regex.Replace(OrgString, "Could", "Should")
    Console.WriteLine(Regex.Replace(OrgString, "could", "should"))

    Console.ReadLine()
  End Sub
```

The output of this function is as follows:

```
Could it be done? It could be done now.
Should it be done? It should be done now.
```

This is simple text replacement. You can get quite a bit more complicated. If you want to know more about text replacement, I suggest the reams of information available on the Internet or in the online help.

So, why am I covering regular expressions? Validation.

## *Regular Expression Validation*

Now you know a little about regular expressions. Take my word for it, this introduction only scratches the surface. Now what?

Remember the validation routines for TextBox input? A few of them that you have seen look for patterns of characters using conditional statements in code. What about replacing those statements with regular expressions? Here are some common things to validate for in text box input:

- Accept only nonnumeric characters.
- Accept only numeric characters.
- Accept characters in a certain order.
- Accept dates based on the culture setting.
- Match a registration key that is entered in a specific format.
- Allow US-style ZIP codes.
- Allow only US-style phone numbers.
- Allow international phone numbers.

- Validate a URI.
- Validate an IP address.
- Accept passwords that must have at least six characters of which two are numbers.

Some of this stuff can be quite lengthy to validate using code. Much of it can be boiled down to a single line of code using a regular expression. Table 8-4 shows some common regular expressions and what they do.

*Table 8-4. Common Data Validation Expressions*

| Expression | Meaning |
|---|---|
| [0-9] | Matches any single number within a string |
| \d | Matches any single number within a string |
| [^0-9] | Matches any single nonnumeric character within a string |
| \D | Matches any single nonnumeric character within a string |
| [A-Za-z] | Matches any uppercase or lowercase letter in a string |
| \d{5}(-\d{4})? | Matches U.S. 5-digit ZIP code or 5+4-digit ZIP code |
| 1-[2-9]{1}\d{2}-\d{3}-\d{4} | Matches U.S.-style phone number (i.e., n-nnn-nnn-nnnn) |
| \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3} | Matches IP address format (not actual addresses) |
| ([\w-]+)@([\w-]+\.)+[A-Za-z]{2,3} | Matches common e-mail addresses |

These are only a few of the things you can do with regular expressions. If you want to find a period, then you need to escape it like this: `\.`. Unless it is inside a set of brackets, just use the period by itself. The \w construct is the same as using [A-Za-z0-9_]. Notice that I used [\w-]—this allows me to trap on any word character, including the dash.

In case you were wondering about the phone number expression, area codes cannot start with a 0 or 1. Therefore, I allow only 2 through 9 at the start of an area code. Note also that the phone number expression allows only one format. You

can lengthen this regular expression considerably by allowing more formats such as a dash or a slash between numbers, or perhaps by making the area code optional.

There is one other thing to note here. If you are testing a whole string, it is best to anchor the regular expression at the beginning and at the end. Use a caret (^) as the first character in the expression and use a dollar sign ($) as the last. This allows you to test the string inclusive.

So, how do you use these regular expressions to validate something? Try these methods.

**C#**

```
//Matches string of consecutive numbers
private static bool IsInteger(string number)
{
  return(Regex.IsMatch(number, "^[+-]?[0-9]+$"));
}

//Matches string of consecutive letters
private static bool IsAlpha(string str)
{
  return(Regex.IsMatch(str, "^[A-Za-z]+$"));
}

//Checks for format of 5 or 5+4 zip code
private static bool IsValidZip(string code)
{
  return(Regex.IsMatch(code, "^\\d{5}(-\\d{4})?$"));
}

//Checks for format of most all email addresses
private static bool IsValidEmail(string email)
{
  return(Regex.IsMatch(email, "^([\\w-]+)@([\\w-]+\\.)+[A-Za-z]{2,3}$"));
}

//Checks for format of USA phone number
private static bool IsValidPhone(string phone)
{
  return(Regex.IsMatch(phone, "^([\\w-]+)@([\\w-]+\\.)+[A-Za-z]{2,3}$"));
}
```

```
//Checks for format of USA date
//separators = /-.
//format = xx/xx/xxxx or xx/xx/xx
//Month and day must be within correct calendar range
//Year can be anything either 2 or 4 digits
private static bool IsValidUSAdate(string dt)
{
  return(Regex.IsMatch(dt, "^(0[1-9]|1[0-2])[./-]" +
                           "(0[1-9]|1[0-9]|2[0-9]|3[0-1])" +
                           "[./-](\\d{2}|\\d{4})$"));
}

//Checks for format of military time
private static bool IsValidMilitaryTime(string tm)
{
  return(Regex.IsMatch(tm, "^([0-1][0-9]|2[0-3]):[0-5][0-9]$"));
  // ([0-1][0-9]|2[0-3]) Check for 00-19 OR 20-23 as hours
  // [0-5][0-9]          Check for 00-59 as minutes
}

//Checks for format of password
//format = 6-15 characters
//         Must include 2 consecutive digits
//         Must include at least one lowercase letter
//         Must include at least one uppercase letter
private static bool IsPasswordFormatValid(string Pword)
{
  return(Regex.IsMatch(Pword,"^(?=.*\\d{2})(?=.*[a-z])(?=.*[A-Z]).{6,15}$"));
  // ?= means look ahead in the string for what follows
  // (?=.*\\d{2}) Starting at the beginning find zero or more of any
  //              character and at 2 consecutive digits in the string.
  // (?=.*[a-z])  Starting at the beginning find zero or more of any
  //              character and a lowercase letter somewhere in the string.
  // (?=.*[A-Z])  Starting at the beginning find zero or more of any
  //              character and an uppercase letter somewhere in the string.
  // .{6,15}      With all else being equal, There must be between 6 and 15
  //              characters in the string
}
```

**VB**

```
'Matches string of consecutive numbers
Function IsInteger(ByVal number As String) As Boolean
  Return (Regex.IsMatch(number, "^[+-]?[0-9]+$"))
End Function

'Matches string of consecutive letters
Function IsAlpha(ByVal str As String) As Boolean
  Return (Regex.IsMatch(str, "^[A-Za-z]+$"))
End Function

'Checks for format of 5 or 5+4 zip code
Function IsValidZip(ByVal code As String) As Boolean
  Return (Regex.IsMatch(code, "^\\d{5}(-\\d{4})?$"))
End Function

'Checks for format of most all email addresses
Function IsValidEmail(ByVal email As String) As Boolean
  Return (Regex.IsMatch(email, "^([\\w-]+)@([\\w-]+\\.)+[A-Za-z]{2,3}$"))
End Function

'Checks for format of USA phone number
Function IsValidPhone(ByVal phone As String) As Boolean
  Return (Regex.IsMatch(phone, "^([\\w-]+)@([\\w-]+\\.)+[A-Za-z]{2,3}$"))
End Function

'Checks for format of USA date
'separators = /-.
'format = xx/xx/xxxx or xx/xx/xx
'Month and day must be within correct calendar range
'Year can be anything either 2 or 4 digits
Function IsValidUSAdate(ByVal dt As String) As Boolean
  Return (Regex.IsMatch(dt, "^(0[1-9]|1[0-2])[./-]" + _
                            "(0[1-9]|1[0-9]|2[0-9]|3[0-1])" _
                            "[./-](\\d{2}|\\d{4})$"))
End Function

'Checks for format of military time
Function IsValidMilitaryTime(ByVal tm As String) As Boolean
  Return (Regex.IsMatch(tm, "^([0-1][0-9]|2[0-3]):[0-5][0-9]$"))
  ' ([0-1][0-9]|2[0-3]) Check for 00-19 OR 20-23 as hours
  ' [0-5][0-9]          Check for 00-59 as minutes
End Function
```

```
'Checks for format of password
'format = 6-15 characters
'         Must include 2 consecutive digits
'         Must include at least one lowercase letter
'         Must include at least one uppercase letter
Function IsPasswordFormatValid(ByVal Pword As String) As Boolean
  Return (Regex.IsMatch(Pword, "^(?=.*\\d{2})(?=.*[a-z])(?=.*[A-Z]).{6,15}$"))
  ' ?= means look ahead in the string for what follows
  ' (?=.*\\d{2}) Starting at the beginning find zero or more of any
  '              character and at 2 consecutive digits in the string.
  ' (?=.*[a-z])  Starting at the beginning find zero or more of any
  '              character and a lowercase letter somewhere in the string.
  ' (?=.*[A-Z])  Starting at the beginning find zero or more of any
  '              character and an uppercase letter somewhere in the string.
  ' .{6,15}      With all else being equal, There must be between 6 and 15
  '              characters in the string
End Function
```

Every single one of these methods works. Check them out. Try coding some of these regular expressions and see how much space they take up.

Probably the most obscure one here is the password checker. It uses a construct I have not explicitly covered. The regular expression parser has the capability to look forward in a string from its cursor point or look behind it. I am using the positive look-ahead search character set (?=pattern). The comments explain what the construct is doing. For further details on other subexpression patterns like this, I suggest you consult the online help.

Some of these expressions can be tricky, but they have so much power to get you what you want. As a contrast, look at the following code. This is an alternate way of validating the password as opposed to the regular expression.

**C#**

```
private static bool LongWayPassword(string Pword)
{
  //Check length first
  if(Pword.Length < 6 || Pword.Length > 15)
    return false;

  string upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  string lower = upper.ToLower();
  bool FoundUpper = false;
  bool FoundLower = false;
  int  NumsFound  = 0;
```

```
    char[] chars = Pword.ToCharArray();
    foreach(char c in chars)
    {
      //look for at least one uppercase letter
      if(Char.IsUpper(c))
        FoundUpper = true;
      //look for at least one lowercase letter
      if(Char.IsLower(c))
        FoundLower = true;
      if(Char.IsNumber(c))
        NumsFound++;
    }
    if(FoundUpper && FoundLower && NumsFound > 1)
      return true;
    else
      return false;
  }
```

**VB**

```
  Function LongWayPassword(ByVal Pword As String) As Boolean
    'Check length first
    If Pword.Length < 6 Or Pword.Length > 15 Then
      Return False
    End If

    Dim upper As String = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    Dim lower As String = upper.ToLower()
    Dim FoundUpper As Boolean = False
    Dim FoundLower As Boolean = False
    Dim NumsFound As Int32 = 0

    Dim chars() As Char = Pword.ToCharArray()
    Dim c As Char
    For Each c In chars
      'look for at least one uppercase letter
      If Char.IsUpper(c) Then
        FoundUpper = True
      End If
      'look for at least one lowercase letter
      If Char.IsLower(c) Then
        FoundLower = True
      End If
```

```
      If Char.IsNumber(c) Then
        NumsFound += 1
      End If
    Next
    If FoundUpper And FoundLower And NumsFound > 1 Then
      Return True
    Else
      Return False
    End If
  End Function
```

So, for the C# code I saved some 20 lines of code by using the regular expression. For the VB code I saved about 25 lines of code. I saved not only code but also bugs. As you know, every line of code entered is a potential bug.

Regular expressions are not only a powerful but also an efficient way to perform search missions, as you saw with the password example. Wouldn't it be nice to have a TextBox that you could enter a regular expression as a property to be run against during validation? That's coming up toward the end of the chapter. For now, let's look at a very powerful control called the Masked Edit control.

## The Masked Edit Control

The Masked Edit control is not native to .NET. Normally, I am reluctant to discuss third-party controls because most programmers will not have access to them. This one is different, however.

The Masked Edit control comes with Visual Studio 6.0. I would imagine that nearly everyone who is using .NET and is reading this book has Visual Studio 6.0 or 5.0 on his or her machine.

The fact that this control comes from Visual Studio 6.0, specifically from VB 6.0, should tell you something. It is an OCX; therefore, it is a COM control. Because it is not native to .NET, you will not see it on your Toolbox. You can get it easily enough, though, by following these steps:

1. Open up a Windows Forms project.
2. Bring the Toolbox into view.
3. Right-click the Toolbox and choose Customize Toolbox.
4. Choose the COM Components tab.
5. Select Microsoft Masked Edit Control, version 6.0.

Figure 8-1 shows the Customize Toolbox window.

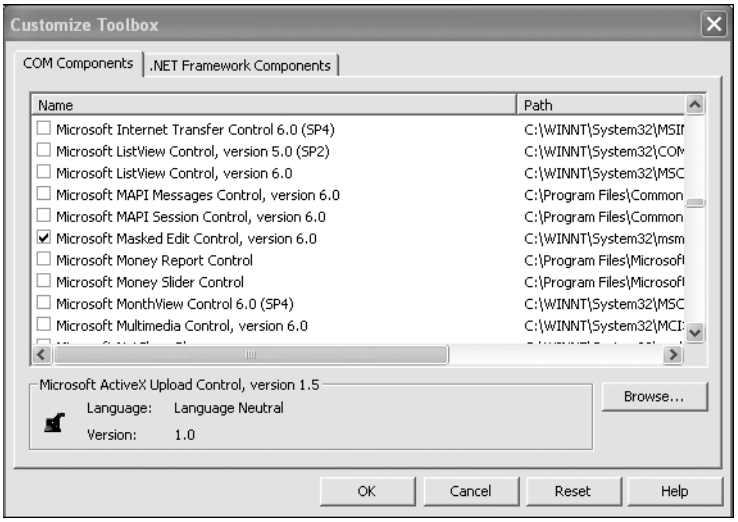


*Figure 8-1. Choosing the Masked Edit control*

Click OK and you should now have the Masked Edit control in your Toolbox. Figure 8-2 shows what it looks like.

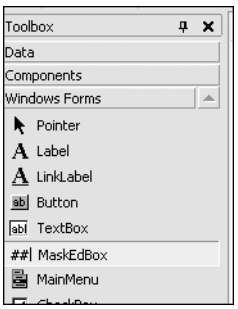


*Figure 8-2. The Masked Edit control in the Toolbox*

You should also notice that two references were also added: AxInterop.MSMask and Interop.MSMask. Now, you already know that .NET talks to VB programs and OCXs via the Interop services, right? If not, you should. You can easily go along and code a perfectly good program without knowing how .NET is talking to this control,

but to truly excel at this game you should know what is going on. Coverage of Interop is beyond the scope of this book, but I do urge you to spend some time learning at least the basics of how the COM Interop services work. OK, enough of that.

This control looks like a normal TextBox when dropped on a form. It has some enhancements, though. You can set a masked property to validate input and also to format output. You can even add some visual cues in the form of literal characters. For instance, you can make the box display the dashes for a normal phone number. I could add a mask in this form: ##/##/##. This is a typical date mask. The front slashes appear on the control as visual cues. Figure 8-3 shows this.

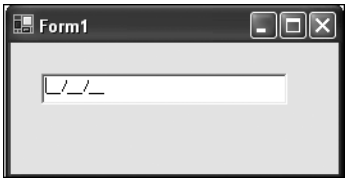


*Figure 8-3. The Masked Edit control with a date mask*

As you type in numbers, the cursor will skip over the slashes and put the numbers where they belong. Isn't this so much easier than hooking into the Paint and KeyPress events? The mask is easy to change as well.

You can mask just about any set of characters you like. Be aware, though, that the masking you can do here is not even close to the validation you can do using regular expressions. Table 8-5 shows the masking characters available for this control.

*Table 8-5. Masking Characters for the Masked Edit Control*

| Character | Meaning |
|---|---|
| # | Digit placeholder |
| . | Decimal placeholder based on culture settings |
| , | Thousand separator based on culture settings |
| : | Time separator based on culture settings |
| / | Date separator based on culture settings |
| \ | Escapes the next character as a literal |
| & | Character placeholder |

*Table 8-5. Masking Characters for the Masked Edit Control (Continued)*

| Character | Meaning |
|---|---|
| < | Converts all characters to lowercase |
| > | Converts all characters to uppercase |
| A | Required alphanumeric character |
| a | Optional alphanumeric character |
| 9 | Optional digit placeholder |
| C | Works like the & character |
| ? | Letter placeholder (upper- or lowercase) |

If you enter any character that is not valid according to the mask, it rejects that character. This is just like setting the Handled property of a TextBox to true during the KeyPress event. If you want to trap this rejection, you can hook into the ValidationError event.

All this is really nice, don't you think? Well, guess what?

It does not work properly in .NET!

Now, I know that quite a few of you VB programmers are probably sitting there in disbelief, so I will prove it to you. First, I will show you the sequence of events (literally) and the resulting Text property when you use this control in a VB 6.0 program. You can try this out if you want to see how it works, but suffice it to say that this is exceedingly easy VB 6.0 code.

**VB 6.0**

```
Private Sub me1_GotFocus()
  L.AddItem "m1 got focus event"
End Sub

Private Sub me1_LostFocus()
  L.AddItem "m1 lost focus event"
End Sub

Private Sub me1_ValidationError(InvalidText As String, StartPosition As Integer)
  L.AddItem "m1 validation error even " & me1.Text
End Sub
```

```
'=======================================================
Private Sub me2_GotFocus()
  L.AddItem "m2 got focus event"
End Sub

Private Sub me2_LostFocus()
  L.AddItem "m2 lost focus event"
End Sub

Private Sub me2_ValidationError(InvalidText As String, StartPosition As Integer)
  L.AddItem "m2 validation error event " & me2.Text
End Sub
```

Figure 8-4 shows the result of running this code.

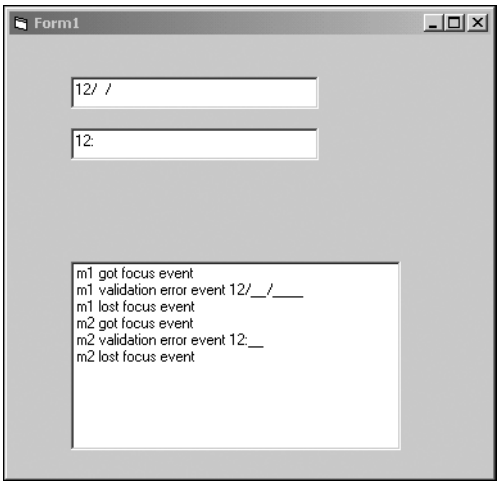


*Figure 8-4. VB 6.0 using Masked Edit control*

You can see that the events are in the following correct order:

1. The control got focus.
2. A validation error event fired because I left the control too early.
3. The second control got focus.
4. A validation error event fired because I left the control too early.
5. The first control got focus.

This is what you would expect, and it is what you get. Notice that when I display the Text property of the control, I get the text entered as well as the visual cues. I went from control to control by using the Tab key.

Now let's do the same thing in C# and in VB. Believe it or not, this control behaves slightly differently in the two languages.

Start a new project in either C# or in VB. Mine is called "MaskedEdit." Add the following controls and properties:

1. Add a Label whose text reads **Enter Date**.

2. Below this Label add a Masked Edit control called **meDate**. Change its TabIndex to 0. Change its mask to 9#/9#/####.

3. Add a Label whose text reads **Enter Military Time**.

4. Below this Label add a Masked Edit control called **meTime**. Change its TabIndex to 1. Change its mask to ##:##.

5. Add a ListBox called **L**. Change its TabStop property to false.

Figure 8-5 shows what this looks like.

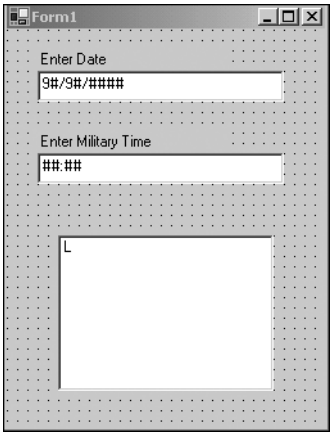


*Figure 8-5. .NET Masked Edit control test*

The code for this project is similar to the code for the VB 6.0 project. Listings 8-1a and 8-1b show the code for the Form_Load event handler and also for the delegates that handle the enter, leave, and validation error events.

*Listing 8-1a. C# Code for the Masked Edit Control Test Program*

```
private void Form1_Load(object sender, System.EventArgs e)
{
  meDate.ValidationError +=
              new MaskEdBoxEvents_ValidationErrorEventHandler(DateErr);
  meDate.Enter += new EventHandler(DateEnter);
  meDate.Leave += new EventHandler(DateLeave);

  meTime.ValidationError +=
              new MaskEdBoxEvents_ValidationErrorEventHandler(TimeErr);
  meTime.Enter += new EventHandler(TimeEnter);
  meTime.Leave += new EventHandler(TimeLeave);

}

#region Masked Edit events

private void DateEnter(object sender, EventArgs e)
{
  L.Items.Add("Date got focus");
}

private void DateLeave(object sender, EventArgs e)
{
  L.Items.Add("Date left");
  L.Items.Add("Date Text = " + meDate.Text);
}

private void DateErr(object sender, MaskEdBoxEvents_ValidationErrorEvent e)
{
  L.Items.Add("Date validation error");
}

private void TimeEnter(object sender, EventArgs e)
{
  L.Items.Add("Time got focus");
}

private void TimeLeave(object sender, EventArgs e)
{
  L.Items.Add("Time left");
  L.Items.Add("Time Text = " + meTime.Text);
}
```

```
    private void TimeErr(object sender, MaskEdBoxEvents_ValidationErrorEvent e)
    {
      L.Items.Add("Time validation error");
    }

    #endregion
```

*Listing 8-1b. VB Code for the Masked Edit Control Test Program*

```
  Private Sub Form1_Load(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) Handles MyBase.Load

    AddHandler meDate.ValidationError, _
              New MaskEdBoxEvents_ValidationErrorEventHandler(AddressOf DateErr)
    AddHandler meDate.Enter, New EventHandler(AddressOf DateEnter)
    AddHandler meDate.Leave, New EventHandler(AddressOf DateLeave)

    AddHandler meTime.ValidationError, _
              New MaskEdBoxEvents_ValidationErrorEventHandler(AddressOf TimeErr)
    AddHandler meTime.Enter, New EventHandler(AddressOf TimeEnter)
    AddHandler meTime.Leave, New EventHandler(AddressOf TimeLeave)

  End Sub

#Region "Masked Edit events"

  Private Sub DateEnter(ByVal sender As Object, ByVal e As EventArgs)
    L.Items.Add("Date got focus")
  End Sub

  Private Sub DateLeave(ByVal sender As Object, ByVal e As EventArgs)
    L.Items.Add("Date left")
    L.Items.Add("Date Text = " + meDate.Text)
  End Sub

  Private Sub DateErr(ByVal sender As Object, _
                      ByVal e As MaskEdBoxEvents_ValidationErrorEvent)
    L.Items.Add("Date validation error")
  End Sub

  Private Sub TimeEnter(ByVal sender As Object, ByVal e As EventArgs)
    L.Items.Add("Time got focus")
  End Sub
```

```
  Private Sub TimeLeave(ByVal sender As Object, ByVal e As EventArgs)
    L.Items.Add("Time left")
    L.Items.Add("Time Text = " + meTime.Text)
  End Sub

  Private Sub TimeErr(ByVal sender As Object, _
                      ByVal e As MaskEdBoxEvents_ValidationErrorEvent)
    L.Items.Add("Time validation error")
  End Sub

#End Region
```

Compile and run the program. Type in a few valid characters in the date field and tab over to the time field. Type a few valid characters in the time field and tab back over to the date field. Figure 8-6 shows the result of this.

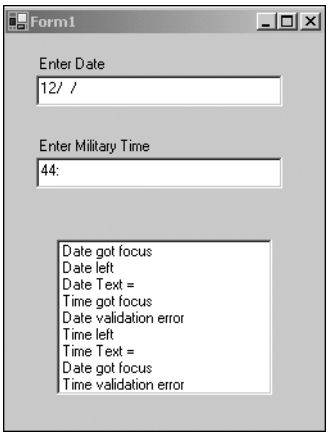


*Figure 8-6. .NET rendition of the Masked Edit control test*

As you can see, the events are in the following order:

1. The date field got focus.
2. The date field lost focus.
3. The time field got focus.
4. The date field fired a validation error.

5. The time field lost focus.

6. The date field got focus.

7. The time field fired a validation error.

Talk about a time lag! The validation error event is not fired until the next control already has focus. The reason I would use this event is to bring focus back to the offending control if not enough characters were entered. The control itself takes care of any error while you are still in the control, but this event is the way for you to know if the control has enough characters.

Imagine this. What do you think would happen if I had the following code for these two controls?

**C#**

```
private void DateErr(object sender, MaskEdBoxEvents_ValidationErrorEvent e)
{
  L.Items.Add("Date validation error");
  meDate.Focus();
}
private void TimeErr(object sender, MaskEdBoxEvents_ValidationErrorEvent e)
{
  L.Items.Add("Time validation error");
  meTime.Focus();
}
```

**VB**

```
Private Sub DateEnter(ByVal sender As Object, ByVal e As EventArgs)
  L.Items.Add("Date got focus")
  meDate.Focus()
End Sub
Private Sub TimeErr(ByVal sender As Object, _
                    ByVal e As MaskEdBoxEvents_ValidationErrorEvent)
  L.Items.Add("Time validation error")
  meTime.Focus()
End Sub
```

All I did was set the focus back to the offending control. Let me tell you what happens here: complete lockup. Because the events come out of order, the validation code for the date control sets focus back to the date control when focus is already in the time control. This makes the time control validation error code

return focus back to the time control. You have set up a game of high-speed ping-pong. You can spend days trying to overcome this, but it can't be done without lots of code. Eliminating lots of code is the whole point of using this control.

Notice in Figure 8-6 that when I printed out the Text property of each control, all I got was an empty string. There is no way to get at the Text property of this control in .NET.

Now for the difference in using this control between C# and VB. The Text property does not even show up in the C# IntelliSense. It does in VB .NET.

So, why didn't I just tell you that this control does not work? Because you needed to feel the pain I went through trying to get it to work. Really, I show you this because I don't want you to think that this control can save you hours of programming and that it's the answer to most of your data entry and validation problems. It isn't.

Does it end here? Is there any hope? No and yes. How about making your own Masked Edit control?

## Rolling Your Own Masked Edit Control

Because you can't use the VB 6.0–supplied Masked Edit control, how about making your own? This is such a useful control that it is well worth taking the time to cover it here.

I liked the idea of the Masked Edit control and its features. Here is what you will be programming as features in this control:

- Ability to have predefined mask formats
- Ability to have user-defined mask formats (limited)
- Ability to have no mask and accept anything
- Ability to have a validation error event that fires before the control loses focus
- Ability to suppress the validation error event (which fires only during validation) by setting the TextBox.CausesValidation property to false
- Ability to have visual cues so the user knows how many characters to input
- Ability to allow literals that cannot be deleted

All this can be yours for only $19.95! And if you call now, I will throw in another control for free!

I must warn you about writing this control and its associated test program. It is not easy, and you have to take care to do everything perfectly or your control will crash. Here we go.

Start a new C# or VB Windows control library. This is not a normal Windows project—it is a user control that you will modify heavily. Mine is called "CustomMask."

**NOTE** In the code for this chapter (you can download the code for this book from the Downloads section of the Apress Web site at `http://www.apress.com`), you will see that the VB project is called CustomMask-vb and the C# project is called CustomMask-c. I tell you this because you may see two of the same controls in the .NET control library after you write the test program.

There is no user interface to this control. What you see on the screen is a small borderless window. I show you how to get rid of it shortly. For now, look at the code generated for you.

**CAUTION** Whatever you do, don't double-click the form to get to the code. You will get the Form_Load event delegate definition and the delegate code as well. You will be making some changes to the code, which will cause some errors in the wizard-generated code, and you will then need to get rid of this code by hand.

This class derives from System.Windows.Forms.UserControl. This is fine for most user controls, but what I want to do here is derive from the TextBox. This allows me to extend the TextBox properties. Here are the before and after class definitions. (Be sure to add System.Text and System.Text.RegularExpressions namespace references.)

**C# Class Definition Before the Change**

```
  public class UserControl1 : System.Windows.Forms.UserControl
  {
…
  }
```

**C# Class Definition After the Change**

```
public class MaskedTextBox_C : System.Windows.Forms.TextBox
{
…
}
```

**VB Class Definition Before the Change**

```
Public Class UserControl1
    Inherits System.Windows.Forms.UserControl
…
End Class
```

**VB Class Definition After the Change**

```
Public Class MaskedTextBox_VB
  Inherits System.Windows.Forms.TextBox
…
End Class
```

As you can see, the code now inherits from the TextBox class. If you try to view the form, you will not be able to because it does not exist anymore.

Now it is time to add some class local variables. Some of these will be public so you will be able to see them in your test form. Here they are.

**C#**

```
    #region local variables

    public event ValidationErrorEventHandler ValidationError;

    public enum FormatType
    {
      None,
      Date,
      Numbers,
      Alpha
    };
    string        mUserMask;
    FormatType    mFmt           = FormatType.None;
    char          mNumberPlace   = '#';
    char          mCue           = '_';
    string        mRegNum        = "[0-9]";
```

```
    char          mAlphaPlace    = '?';
    string        mRegAlpha      = "[A-Za-z]";
    string        mAnything      = ".*";
    string        mRegExpression = string.Empty;
    StringBuilder mText          = new StringBuilder();
    int           mValidationErrors;

    #endregion
```

**VB**

```
#Region "local variables"

  Public Event ValidationError As ValidationErrorEventHandler

  Public Enum FormatType
    None
    DateFormat
    Numbers
    Alpha
  End Enum

  Dim mUserMask As String
  Dim mFmt As FormatType = FormatType.None
  Dim mNumberPlace As Char = "#"c
  Dim mCue As Char = "_"c
  Dim mRegNum As String = "[0-9]"
  Dim mAlphaPlace As Char = "?"c
  Dim mRegAlpha As String = "[A-Za-z]"
  Dim mAnything As String = ".*"
  Dim mRegExpression As String = String.Empty
  Dim mText As StringBuilder = New StringBuilder()
  Dim mValidationErrors As Int32

#End Region
```

The Format enumeration is used for the Format property you will code next. This enumeration allows anyone using this control to choose an easy-to-see format rather than entering in a number. You will see this more clearly when you get to the client portion of this project.

Remember the regular expressions? As you can see from this variable list, you will use them extensively in this control.

Now it is time to add two new properties, Format and Mask. Here is the code.

**C#**

```
#region New properties

/// <summary>
/// Sets one of three formats
/// Date must be ##/##/####
/// Numbers must be 0-9 exactly 8 digits.
/// Alpha must be A-Z or a-z, exactly 8 digits
/// </summary>
public FormatType Format
{
  get{return mFmt;}
  set
  {
    mFmt = value;
    mText = new StringBuilder();

    if(mFmt == FormatType.None)
    {
      mUserMask = "";
      mRegExpression = mAnything;
    }
    else if(mFmt == FormatType.Date)
    {
      mUserMask = "##/##/####";
      mText.Append("__/__/____");
      mRegExpression = "\\d{2}/\\d{2}/\\d{4}";
    }
    else if(mFmt == FormatType.Alpha)
    {
      mUserMask = "????????";
      mRegExpression = mRegAlpha + "{8}";
      mText.Append("________");
    }
    else if(mFmt == FormatType.Numbers)
    {
      mUserMask = "########";
      mRegExpression = mRegNum + "{8}";
      mText.Append("________");
    }
    Mask = mUserMask;
  }
}
```

```
/// <summary>
/// If the Format property is set then
/// this property is invalid
/// Mask properties recognized are:
///   # for number
///   ? for alpha (upper or lowercase)
/// all other characters are literals
/// </summary>
public string Mask
{
  get{return mUserMask;}
  set
  {
    if(mFmt == FormatType.None)
    {
      mRegExpression = string.Empty;
      mText = new StringBuilder();
      mUserMask = value;
      char[] chars = mUserMask.ToCharArray();
      foreach(char c in chars)
      {
        if(c == mNumberPlace)
        {
          mRegExpression += mRegNum;
          mText.Append(mCue);
        }
        else if(c == mAlphaPlace)
        {
          mRegExpression += mRegAlpha;
          mText.Append(mCue);
        }
        else
        {
          mRegExpression += c.ToString();
          mText.Append(c);
        }
      }
    }
    this.Text = mText.ToString();
  }
}

#endregion
```

**VB**

```
#Region "New properties"

  Public Property Format() As FormatType
    Get
      Return mFmt
    End Get
    Set(ByVal Value As FormatType)
      mFmt = Value
      mText = New StringBuilder()

      If mFmt = FormatType.None Then
        mUserMask = ""
        mRegExpression = mAnything
      ElseIf mFmt = FormatType.DateFormat Then
        mUserMask = "##/##/####"
        mText.Append("__/__/____")
        mRegExpression = "\\d{2}/\\d{2}/\\d{4}"
      ElseIf mFmt = FormatType.Alpha Then
        mUserMask = "????????"
        mRegExpression = mRegAlpha + "{8}"
        mText.Append("________")
      ElseIf mFmt = FormatType.Numbers Then
        mUserMask = "########"
        mRegExpression = mRegNum + "{8}"
        mText.Append("________")
      End If
      Mask = mUserMask
    End Set
  End Property

  Public Property Mask() As String
    Get
      Return mUserMask
    End Get
    Set(ByVal Value As String)
      If mFmt = FormatType.None Then
        mRegExpression = String.Empty
        mText = New StringBuilder()
        mUserMask = Value
        Dim chars() As Char = mUserMask.ToCharArray()
        Dim c As Char
        For Each c In chars
```

```
          If c = mNumberPlace Then
            mRegExpression += mRegNum
            mText.Append(mCue)
          ElseIf c = mAlphaPlace Then
            mRegExpression += mRegAlpha
            mText.Append(mCue)
          Else
            mRegExpression += c.ToString()
            mText.Append(c)
          End If
        Next
        Me.Text = mText.ToString()
      End If
    End Set
  End Property

#End Region
```

Let's look at these properties in some detail. The Format property builds a regular expression based on the type of data allowed in. At the same time, it builds the string to be shown in the TextBox. I am using the underscore (_) character as a visual cue for the user. If you want, you can add another property to allow the user to choose his or her cue.

In the interest of limiting the code I need, I limited the alpha and numeric formats to only eight characters. Also note that I am not allowing any optional numbers or optional alpha characters.

The Mask property is dependent upon what the user chooses for the Format property. If the user chooses a format of None, then this property accepts the user format. I allow only # and ? as replacement characters. All other characters entered are treated as literal and cannot be changed by the user during runtime. A literal is defined as a character that appears at its correct position in the control, and it cannot be deleted or overwritten.

Notice that this Mask property builds the correct regular expression string, as it looks at each character in the input mask. At the same time, I am also building the text that is to be displayed in the control during runtime.

The next bit of code you will need takes care of the user entering in characters during runtime. I also have a delete routine in here.

**C#**

```
#region Helper stuff

public string EnterLetter(ref int position, char c)
{
  //User pressed delete key
  if((int)c == 8)
  {
    position--;
    return DeleteLetter(ref position);
  }

  //User trying to go beyond bounds
  if(position >= mText.Length)
    return mText.ToString();

  //If we have hit a literal then advance one
  //Do this in a loop in case there are more literals.
  if(mText[position] != mCue)
    position++;

  //check to see if the character is ok
  if(mUserMask[position] == mNumberPlace)
  {
    if(Regex.IsMatch(c.ToString(), mRegNum))
    {
      mText[position] = c;
      position ++;
    }
  }
  else if(mUserMask[position] == mAlphaPlace)
  {
    if(Regex.IsMatch(c.ToString(), mRegAlpha))
    {
      mText[position] = c;
      position ++;
    }
  }

  return mText.ToString();
}
```

```
    public string DeleteLetter(ref int pos)
    {
      //If the character to be deleted is a cue then bail out
      if(mText[pos] != mCue)
      {
        //If the character to be deleted is valid then change it back to a cue
        if(mUserMask[pos] == mNumberPlace || mUserMask[pos] == mAlphaPlace)
          mText[pos] = mCue;
      }
      return mText.ToString();
    }

    #endregion
```

**VB**

```
#Region "Helper stuff"

  Public Function EnterLetter(ByRef position As Int32, _
                              ByVal c As Char) As String
    'User pressed delete key
    If Val(c) = 8 Then
      position -= 1
      Return DeleteLetter(position)
    End If

    'User trying to go beyond bounds
    If position >= mText.Length Then
      Return mText.ToString()
    End If

    'If we have hit a literal then advance one
    'Do this in a loop in case there are more literals.
    If mText.Chars(position) <> mCue Then
      position += 1
    End If

    'check to see if the character is ok
    If mUserMask.Chars(position) = mNumberPlace Then
      If Regex.IsMatch(c.ToString(), mRegNum) Then
        mText.Chars(position) = c
        position += 1
      End If
```

```
    ElseIf mUserMask.Chars(position) = mAlphaPlace Then
      If Regex.IsMatch(c.ToString(), mRegAlpha) Then
        mText.Chars(position) = c
        position += 1
      End If
    End If

    Return mText.ToString()
  End Function

  Public Function DeleteLetter(ByRef pos As Int32) As String
    'If the character to be deleted is a cue then bail out
    If mText.Chars(pos) <> mCue Then
      'If the character to be deleted is valid then change it back to a cue
      If mUserMask.Chars(pos) = mNumberPlace Or _
         mUserMask.Chars(pos) = mAlphaPlace Then
        mText.Chars(pos) = mCue
      End If
    End If
    Return mText.ToString()
  End Function

#End Region
```

Let's look at the EnterLetter routine first. The first thing I do here is check for a backspace. If I find it, I decrement the position pointer and call the Delete function, which returns the corrected string.

If the character puts me over the allowed length, I disallow the character and return the original string. If the character is about to overwrite a cue character, I advance the pointer and then add the character to the string. If the character is in a spot where I have a regular expression character, I use the Regex object to decide if it is a match. If so, I replace the character in the internal string and return that string. If this character does not match, I ignore it and return the original string.

This EnterLetter routine is called from the KeyPress event handler. The only way for me to ignore any character is to set the KeyPressEventArgs.Handled property to true. As you will see, I do this for every character that comes in.

The DeleteLetter routine replaces the deleted character with a cue character. It will not replace a literal with a cue character. This prevents the control from deleting literals.

**NOTE** This control can handle only one literal character in a row. There is no checking for multiple literals in a row, and there is no looping to allow this. If you enter two literals in a row while testing this control, it will explode. The point of this example is to show you how it is done, not to do it all for you. You have the knowledge to enhance this control and make it robust if you so desire.

The next region of code to be added contains the various event handlers for the TextBox. You will be adding handlers for the following events:

- TextBox.Enter
- TextBox.Leave
- TextBox.KeyDown
- TextBox.KeyPress

Here is the code for these delegates.

**C#**

```
#region hooked events

private void MaskBoxEnter(object sender, EventArgs e)
{
  this.Text = mText.ToString();
  this.SelectionLength=0;
}

private void MaskBoxLeave(object sender, EventArgs e)
{
  if(mUserMask == string.Empty)
  {
    mText = new StringBuilder(this.Text);
  }
}
```

```
    private void MaskBoxKeyDown(object sender, KeyEventArgs e)
    {
      //No mask so let in any character
      if(mUserMask == string.Empty)
        return;

      int pos = this.SelectionStart;
      if(e.KeyData == Keys.Delete)
      {
        this.Text = DeleteLetter(ref pos);
        this.SelectionStart = pos;
      }
      e.Handled = true;
    }

    private void MaskBoxKeyPress(object sender, KeyPressEventArgs e)
    {
      //No mask so let in any character
      if(mUserMask == string.Empty)
        return;

      int pos = this.SelectionStart;
      this.Text = EnterLetter(ref pos, e.KeyChar);
      e.Handled = true;
      this.SelectionStart = pos;
    }

    #endregion
```

**VB**

```
#Region "hooked events"

  Private Sub MaskBoxEnter(ByVal sender As Object, ByVal e As EventArgs)
    Me.Text = mText.ToString()
    Me.SelectionLength = 0
  End Sub

  Private Sub MaskBoxLeave(ByVal sender As Object, ByVal e As EventArgs)
    If mUserMask = String.Empty Then
      mText = New StringBuilder(Me.Text)
    End If
  End Sub
```

```
  Private Sub MaskBoxKeyDown(ByVal sender As Object, ByVal e As KeyEventArgs)
    'No mask so let in any character
    If mUserMask = String.Empty Then
      Return
    End If

    Dim pos As Int32 = Me.SelectionStart
    If e.KeyData = Keys.Delete Then
      Me.Text = DeleteLetter(pos)
      Me.SelectionStart = pos
    End If
    e.Handled = True
  End Sub

  Private Sub MaskBoxKeyPress(ByVal sender As Object, _
                              ByVal e As KeyPressEventArgs)
    'No mask so let in any character
    If mUserMask = String.Empty Then
      Return
    End If

    Dim pos As Int32 = Me.SelectionStart
    Me.Text = EnterLetter(pos, e.KeyChar)
    e.Handled = True
    Me.SelectionStart = pos
  End Sub

#End Region
```

I hook into the KeyDown event for the sole purpose of trapping the Delete key. I use the SelectionStart property to determine where the cursor is in the control. I set the text of the control to the string that gets returned from the Delete function. I then set the Handled property to true to disallow the multiple keystrokes bug.

The Enter function sets the selection length to zero to prevent the whole string from being selected. It also sets the text of the box to the current version of text held in memory.

The KeyPress handler lets in any character if no mask or format is provided. This lets the control act as a normal TextBox. The rest of this function gets the new string from the EnterLetter function and resets the position of the cursor in the control.

**TIP** There is no explicit property to retrieve the cursor position in a TextBox control. Instead, use the SelectionStart property, which gives you the point at which the next character gets entered.

So, now you have all the new properties and event handlers necessary for a Masked Edit control. Just wire up the delegates in the class constructor.

**C#**

```
public MaskedTextBox_C()
{
  InitializeComponent();

  mValidationErrors     = 0;
  this.Enter            += new EventHandler(MaskBoxEnter);
  this.Leave            += new EventHandler(MaskBoxLeave);
  this.KeyDown          += new KeyEventHandler(MaskBoxKeyDown);
  this.KeyPress         += new KeyPressEventHandler(MaskBoxKeyPress);
}
```

**VB**

```
Public Sub New()
  MyBase.New()

  InitializeComponent()

  mValidationErrors = 0
  AddHandler Me.Enter, New EventHandler(AddressOf MaskBoxEnter)
  AddHandler Me.Leave, New EventHandler(AddressOf MaskBoxLeave)
  AddHandler Me.KeyDown, New KeyEventHandler(AddressOf MaskBoxKeyDown)
  AddHandler Me.KeyPress, New KeyPressEventHandler(AddressOf MaskBoxKeyPress)

End Sub
```

There it is. You now have a control that extends the properties of the TextBox with two extra ones allowing for masking. If you enter a mask, this control will validate each character entered against that mask. In a little while, you will be testing this control. There is one thing missing, however.

What happens when the user tabs out of this control before everything is entered? You get no warning that a problem has occurred. After all, there are missing characters and you need to know this. The solution is to add an event.

The event that you will add is called ValidationErrorEvent. This event will get fired if two conditions are met: if the user leaves a control and if that controls text does not match the supplied mask. I will be firing this event from within the Validated event that comes with the TextBox control. I do this for two reasons:

- The Validated event can be suppressed by setting the CausesValidation property of the TextBox to false. This also suppresses my new event.

- The Validated event is called before the Leave event to allow the user to stop focus from transferring to another control if he or she so wishes.

Before I have you enter the new event code, I suggest that you look over the numerous examples in the online help. Like I keep saying, you can program away happily without knowing what really goes on, but to understand things like events and delegates you need to dig in. The effort is well worth it.

I could have written this event using the .NET-supplied EventHandler and the accompanying EventArgs. This is boring, though, not to mention uninformative. So in order to make some new type of event arguments, you will need to generate a new class that is derived from the class EventArgs. Put this class code above the class code for your control. Along with this class, you will need to define a delegate for a client to hook into.

**C#**

```
public class ValidationErrorEventArgs : EventArgs
{

  private string mMask;
  private string mText;

  public ValidationErrorEventArgs(string mask, string OffendingText)
  {
    mMask = mask;
    mText = OffendingText;
  }

  public string Mask{get{return mMask;}}
  public string Text{get{return mText;}}
}
public delegate void ValidationErrorEventHandler(object sender,
                                                  ValidationErrorEventArgs e);
```

**VB**

```
Public Class ValidationErrorEventArgs
  Inherits EventArgs

  Private mMask As String
  Private mText As String

  Public Sub New(ByVal mask As String, ByVal OffendingText As String)
    mMask = mask
    mText = OffendingText
  End Sub

  Public ReadOnly Property Mask() As String
    Get
      Return mMask
    End Get
  End Property

  Public ReadOnly Property Text() As String
    Get
      Return mText
    End Get
  End Property

End Class
Public Delegate Sub ValidationErrorEventHandler(ByVal sender As Object, _
                                                ByVal e As _
                                                ValidationErrorEventArgs)
```

Now that you have this class, you will need to provide an Onxxx method that goes with the event. Enter the following region of code.

**C#**

```
    #region OnXXX event stuff

    public void RaiseError()
    {
      ValidationErrorEventArgs e = new ValidationErrorEventArgs(mUserMask,
        mText.ToString());
      OnValidationError(e);
    }
```

```
    protected virtual void OnValidationError(ValidationErrorEventArgs e)
    {
      ValidationError(this, e);
    }

    #endregion
```

**VB**

```
#Region "OnXXX event stuff"

  Public Sub RaiseError()
    Dim e As ValidationErrorEventArgs = _
          New ValidationErrorEventArgs(mUserMask, mText.ToString())
    OnValidationError(e)
  End Sub

  Protected Sub OnValidationError(ByVal e As ValidationErrorEventArgs)
    RaiseEvent ValidationError(Me, e)
  End Sub

#End Region
```

You must call the OnValidationError method to raise the event. This is required and you will be flogged if you do not do it this way (see the online help).

I include the RaiseEvent method for ease of use. Notice that I pass in the current mask and text so the client can determine the problem from ValidationErrorEventArgs.

Because I need to call this from inside the Validation event, I need to wire this event up. At the same time, I also wire up a delegate to my new event so I can keep track of how many times matching failed in this control.

Add these two delegates to your "hooked events" region.

**C#**

```
    private void MaskBoxValidated(object sender, EventArgs e)
    {
      if(!Regex.IsMatch(mText.ToString(), mRegExpression))
        RaiseError();
    }
```

```
  //Special event to handle case of no one connecting to my delegate
  //avoids a null reference
  private void DefaultHandler(object sender, ValidationErrorEventArgs e)
  {
    mValidationErrors++;
  }
```

**VB**

```
Private Sub MaskBoxValidated(ByVal sender As Object, ByVal e As EventArgs)
  If Not Regex.IsMatch(mText.ToString(), mRegExpression) Then
    RaiseError()
  End If
End Sub

'Special event to handle case of no one connecting to my delegate
'avoids a null reference
Private Sub DefaultHandler(ByVal sender As Object, _
                           ByVal e As ValidationErrorEventArgs)
  mValidationErrors += 1
End Sub
```

By the way, connecting to my own event avoids a nasty null reference bug in case the client decides not to use this event.

The MaskBoxValidated event handler uses the regular expression engine to evaluate the whole text to the mask. If this match fails, then I start the ball rolling and fire the event.

Of course, in order to catch the Validated event you need to connect to it. Add the following lines to your constructor.

**C#**

```
    this.Validated        += new EventHandler(MaskBoxValidated);
    this.ValidationError  += new ValidationErrorEventHandler(DefaultHandler);
```

**VB**

```
  AddHandler Me.Validated, New EventHandler(AddressOf MaskBoxValidated)
  AddHandler Me.ValidationError, _
             New ValidationErrorEventHandler(AddressOf DefaultHandler)
```

There you are. A shiny new Masked Edit control. Compile it and make sure you have no syntax bugs. Because this is part of a control library, you will end up with a DLL after compilation. You will bring this DLL into your Toolbox when you make the client.

## Testing the Masked Edit Control

Testing the new control is simple. Start a new Windows Forms project in C# or VB. Mine is called "MaskClient." Note here that I can use the VB control in the C# project or vice versa. This is the CLR at work.

You will need to add the control to your Toolbox. You can make a new Toolbox tab or use one of the existing ones. I use the General tab. Bring up the General tab on the Toolbox. Right-click the tab and choose Customize Toolbox. Choose the .NET Framework Components tab in the Customize Toolbox dialog box.

If you are writing in VB, click Browse and look for your DLL in the BIN subdirectory of your project. Select it and click OK on the dialog box. If you are writing in C# you will find the DLL in the BIN\Debug subdirectory of your project. Figure 8-7 shows the dialog box after you have chosen the control.

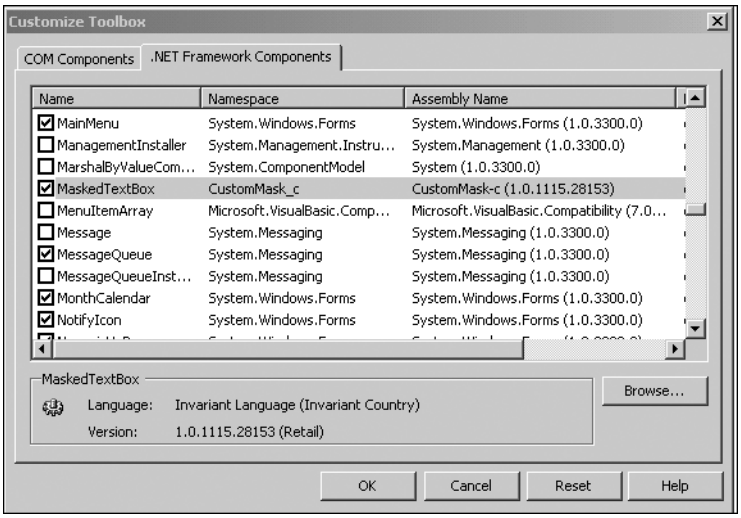


*Figure 8-7. Choosing the new control*

Figure 8-8 shows what your Toolbox should now look like.

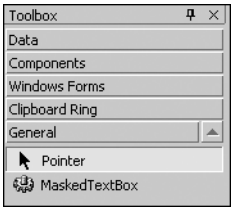


*Figure 8-8. The new control in the Toolbox*

You have a single gear to work with. You can change this icon if you want. See the online help for assistance.

Drag this control over to the form and call it **me1**. Drag another control such as a TextBox over to the form as well. This other control allows you to leave the Masked Edit control and catch the Validate event. Your form should look like the one shown in Figure 8-9.

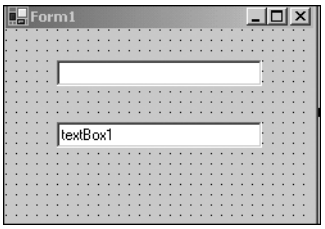


*Figure 8-9. The test form*

You can see that this new control looks the same as the TextBox. Click the Masked Edit control and look at the properties. You should see the two new properties at the bottom of the list. Click the Format property and the enumeration will appear (see Figure 8-10).

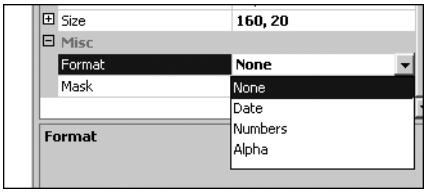


*Figure 8-10. The new properties*

If you choose one of the formats and click in the mask, you will see the correct mask appear for that format. You will also see the correct text appear in the box. This is pretty cool!

Choose None for a format and enter the following mask property: **##?/#?/##**. This means allow two numbers, one alpha, a literal, a number, an alpha, a literal, and two numbers. Run the program and your form should look like Figure 8-11.

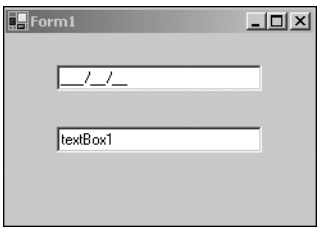


*Figure 8-11. The running form*

Enter in some characters in the edit box and you will see that only those corresponding to the mask are allowed. You will also see the control skip over the literals. Try deleting and backspacing over characters. Pretty neat, huh? Now tab out to the next control. You will not know that this control failed.

Double-click the form and add the following code.

**C#**

```
private void Form1_Load(object sender, System.EventArgs e)
{
  me1.ValidationError += new ValidationErrorEventHandler(MaskValid);
}

private void MaskValid(object sender, ValidationErrorEventArgs e)
{
  MessageBox.Show(e.Mask + "  " + e.Text);
}
```

**VB**

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) Handles MyBase.Load
  AddHandler me1.ValidationError, _
             New ValidationErrorEventHandler(AddressOf Valid)
End Sub
```

```
Private Sub Valid(ByVal sender As Object, ByVal e As ValidationErrorEventArgs)
  MessageBox.Show(e.Mask + "  " + e.Text)
End Sub
```

Now run the form again, enter some values in the mask control, and tab over to the next control. You will need to make sure that you reference the namespace of the control you created. Figure 8-12 shows the results.

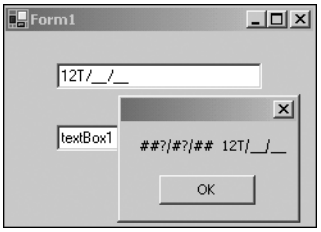


*Figure 8-12. Mask results*

When you tab out of the control without the correct values in it, ValidateErrorEvent gets fired. If the control was valid, you would not get this event. Now this event gives you the opportunity to restore focus to this control. It allows you to be strict and lock up the whole form unless the control has the correct values.

That's it for this Masked Edit control and test program. What do you think? Pretty neat, isn't it? I can see doing this for quite a few different controls. I can see making controls that are target specific, such as a control that handles only e-mail addresses. Perhaps you want a control that's used only for login purposes. You could make the mask accept some obtuse set of characters in a certain order. You could make this login control write to a special file whenever someone tries to hack into your program. You could make this login control shut down and reformat the hard drive if three unsuccessful attempts at logging in were detected. You could . . .

There is also another way to do this kind of thing. You could go the same route as the ErrorProvider control. You could make a control that extends properties of any control you want.

## Extending Control Properties

You can do two things about making your own validation controls in .NET. One of them is to make a completely new control by building on an existing control. This

is what you did in the last example. The other thing you can do is add a few new properties to a control without making a new one altogether. This is called extending the control.

The neat thing about this extension is that it can work with any control you like. Want a for-instance? How about the ToolTip control? I have not discussed this control in any detail, so how about the ErrorProvider control?

As you know from previous chapters, the ErrorProvider control is something that you plop down on your form and it appears in the tray below the form. It does not appear on the form itself. In case you have not been paying attention, as soon as you put the ErrorProvider control on the form, every visible control on the form has several new properties that are related to the ErrorProvider control. Try a new project with several controls on it and look at the Properties page. Now add an ErrorProvider to your form. Your several controls now have a few extra properties related to the ErrorProvider.

The ErrorProvider is a class that inherits from Component and also implements the IExtenderProvider interface. It is set up to extend the properties of any control on the current form. The ToolTip control does the same thing.

Extending the properties of a control offers you a big advantage over making your own control. The code works the same (within limits) for any control. It also allows you to turn any control on your form into . . . SuperControl!

This next example contains two classes that extend a TextBox control. The first class provides properties that make the control accept only numbers or only numbers within a certain range. The second class provides a regular expression property for validation before focus leaves the control.

By the way, both of these classes allow the TextBox to work as normal by using a Required property. This allows you to have 50 TextBoxes on the screen and use the extensions on only a few of them.

I start the next section by introducing you to my buddy Vlad the validator.

## *The Number Validation Extender*

Start a new project in either C# or VB. Make sure it is a class library that you are starting. You want to end up with a DLL that you can put in your Toolbox as a new control. The name of my project is “Vlad.”

This class has no user interface (obviously), so I will get right down to the code. The first thing to do is add a reference to the System.Windows.Forms DLL. You do this in the project window. Figure 8-13 shows the project after the reference has been added.

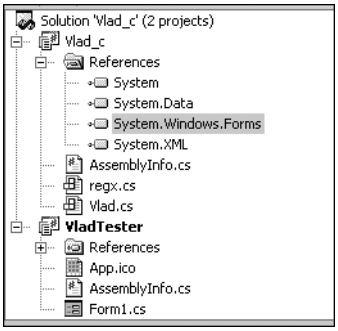


*Figure 8-13. A reference to the form's DLL*

You will notice that my solution has two projects. You will add the test project later.

The next thing to add is some namespace references to your code.

**C#**

```
using System;
using System.Collections;
using System.Windows.Forms;
using System.ComponentModel;
using System.Text.RegularExpressions;
```

**VB**

```
Option Strict On

Imports System
Imports System.Collections
Imports System.Windows.Forms
Imports System.ComponentModel
Imports System.Text.RegularExpressions
```

As you can see, you will be using regular expressions again.

**TIP** The first part of this chapter covers regular expressions. Reading the first part of this chapter makes this part so much more understandable.

Next is the class definition. I use some specialized attributes for these classes. I also use some for the properties within. I suggest you read up on attributes before you go any further. (At least read up a little so you know what attributes are.) Here is the class definition.

**C#**

```
  [ProvideProperty("Required",            typeof(Control))]
  [ProvideProperty("RangeValueRequired",  typeof(Control))]
  [ProvideProperty("MinValue",            typeof(Control))]
  [ProvideProperty("MaxValue",            typeof(Control))]
  public class NumberValidate : Component, IExtenderProvider
  {
…
}
```

**VB**

```
<ProvideProperty("Required", GetType(Control)), _
 ProvideProperty("RangeValueRequired", GetType(Control)), _
 ProvideProperty("MinValue", GetType(Control)), _
 ProvideProperty("MaxValue", GetType(Control))> _
Public Class NumberValidate
  Inherits Component
  Implements IExtenderProvider
…
End Class
```

I am telling any control that looks in here (using reflection) that I am providing four properties:

- *Required:* This tells that control to accept only numbers.
- *RangeValueRequired:* This tells the control to accept numbers in the supplied range.
- *MinValue:* This is the minimum value for the range.
- *MaxValue:* This is the maximum value for the range.

Whenever you implement the IExtenderProvider interface, you need to override the CanExtend property. Add the following code inside the class block.

**C#**

```
bool IExtenderProvider.CanExtend(object target)
{
  if (target is TextBox)
  {
    TextBox t = (TextBox)target;
    tp.Active = true;
    return true;
  }
  else
    return false;
}
```

**VB**

```
Function CanExtend(ByVal target As Object) As Boolean _
                  Implements IExtenderProvider.CanExtend
  If TypeOf target Is TextBox Then
    Dim t As TextBox = CType(target, TextBox)
    tp.Active = True
    Return True
  Else
    Return False
  End If
End Function
```

I have set the control type I will be extending to a TextBox. If you like, you can extend any control you want. How about extending the RichTextBox with a class that does spell checking on the fly? You can do a lot with extenders.

OK, you have the properties defined that appear in the Properties page of the extended control. You also have narrowed the set of extended control to all TextBoxes. It is time to add the guts to this class. Be sure to add the following locals region.

**C#**

```
#region locals

// holds Key - value pair for efficient retrieval
//Holds all properties of all controls that use this extender.
private System.Collections.Hashtable CustomProps  =
                    new System.Collections.Hashtable();
```

```
    ErrorProvider er = new ErrorProvider();
    ToolTip       tp = new ToolTip();

    //Holds all the custom properties of a control
    private class Props
    {
      public bool Required      = false;
      public bool RangeRequired = false;
      public Decimal  MinVal    = 0;
      public Decimal  MaxVal    = 999;
    }

    #endregion
```

**VB**

```
#Region "locals"

  ' holds Key - value pair for efficient retrieval
  'Holds all properties of all controls that use this extender.
  Private CustomProps As System.Collections.Hashtable = _
                        New System.Collections.Hashtable()

  Private er As ErrorProvider = New ErrorProvider()
  Private tp As ToolTip = New ToolTip()

  'Holds all the custom properties of a control
  Private Class Props
    Public Required As Boolean = False
    Public RangeRequired As Boolean = False
    Public MinVal As Decimal = 0
    Public MaxVal As Decimal = 999
  End Class

#End Region
```

Next, I add the delegates for the events I will use. Vlad will hook into the KeyPress event and the Validating event. Here is the delegate region of code.

**C#**

```
#region events

private void KeyPress(object sender, KeyPressEventArgs e)
{
  //Allow backspace
  if(e.KeyChar == (char)8)
    return;

  //Allow 0-9
  if(!Regex.IsMatch(e.KeyChar.ToString(), "[0-9]"))
    e.Handled = true;
}

private void ValidateProp(object sender, CancelEventArgs e)
{
  Control ctl = (Control)sender;

  //Reset the error
  er.SetError(ctl, "");

  if(GetRequired(ctl))
  {
    if(ctl.Text == string.Empty)
    {
      er.SetError(ctl, "No value was entered when one was required");
      return;
    }

    if(GetRangeValueRequired(ctl))
    {
      Props p = (Props)CustomProps[ctl];
      try
      {
        if(Decimal.Parse(ctl.Text) < p.MinVal)
          er.SetError(ctl, "Value entered is less than minimum value");
      }
      catch
      {
        er.SetError(ctl, "Value is non-numeric");
      }
```

```
        try
        {
          if(Decimal.Parse(ctl.Text) > p.MaxVal)
            er.SetError(ctl, "Value entered is greater than minimum value");
        }
        catch
        {
          er.SetError(ctl, "Value is non-numeric");
        }
      }
    }
  }

  #endregion
```

**VB**

```
#Region "events"

  Private Sub KeyPress(ByVal sender As Object, ByVal e As KeyPressEventArgs)
    'Allow backspace
    If e.KeyChar = Chr(8) Then
      Return
    End If

    'Allow 0-9
    If Not Regex.IsMatch(e.KeyChar.ToString(), "[0-9]") Then
      e.Handled = True
    End If
  End Sub

  Private Sub ValidateProp(ByVal sender As Object, ByVal e As CancelEventArgs)
    Dim ctl As Control = CType(sender, Control)

    'Reset the error
    er.SetError(ctl, "")

    If GetRequired(ctl) Then
      If ctl.Text = String.Empty Then
        er.SetError(ctl, "No value was entered when one was required")
        Return
      End If
```

```
      If GetRangeValueRequired(ctl) Then
        Dim p As Props = CType(CustomProps(ctl), Props)
      If GetRangeValueRequired(ctl) Then
        Dim p As Props = CType(CustomProps(ctl), Props)
        If IsNumeric(ctl.Text) Then
          If Decimal.Parse(ctl.Text) < p.MinVal Then
            er.SetError(ctl, "Value entered is less than minimum value")
          End If
          If Decimal.Parse(ctl.Text) > p.MaxVal Then
            er.SetError(ctl, "Value entered is greater than minimum value")
          End If
        Else
          er.SetError(ctl, "Value is non-numeric")
        End If
      End If
      End If
    End If
  End Sub

#End Region
```

The KeyPress delegate uses the regular expression engine to validate only numbers. I allow a backspace in here as well. The Validating delegate checks to see if validating is required and if the validation should be range sensitive.

Notice here that I choose to instantiate a new ErrorProvider within my extender. If something is wrong, I set the ErrorProvider text and flash the icon next to the offending control.

If you want, you can prevent the control from losing focus in the case of an error. This is problematic, though, because if you try to exit the program while the control is invalid, you will not be able to. One solution I have seen to this "exit" problem is to walk the stack and find the WM_CLOSE message as the reason you are leaving the control. I think that setting an error is just fine.

The error text says nothing about how the control should be used. In fact, how is the user supposed to know that this control needs only numbers within a certain range? The answer is to use the ToolTip control. You saw in the locals region that I included a ToolTip control. Enter the following code, which fills the text.

**C#**

```
private void SetToolTip(Control ctl)
{
  string tip = string.Empty;
  Props p = (Props)CustomProps[ctl];
  if(p.Required)
    tip = "Validation of numbers required";
  if(p.RangeRequired)
  {
    tip += " / Number range required";
    tip += " / Min value = " + p.MinVal.ToString();
    tip += " / Max value = " + p.MaxVal.ToString();
  }
  tp.SetToolTip(ctl, tip);
}
```

**VB**

```
Private Sub SetToolTip(ByVal ctl As Control)
  Dim tip As String = String.Empty
  Dim p As Props = CType(CustomProps(ctl), Props)
  If p.Required Then
    tip = "Validation of numbers required"
  End If
  If p.RangeRequired Then
    tip += " / Number range required"
    tip += " / Min value = " + p.MinVal.ToString()
    tip += " / Max value = " + p.MaxVal.ToString()
  End If
  tp.SetToolTip(ctl, tip)
End Sub
```

As you can see, I change the text to reflect the properties set for this control. Anytime one of the properties is changed, I call this method.

When you define a property using attributes, you need a property get and set function to go with it. The name of the function must be the name of the property preceded by the word "Get" or "Set." Here is the code region for the Required property.

**C#**

```
#region Required property

public bool GetRequired(Control ctl)
{
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    SetToolTip(ctl);
    return p.Required;
  }
  else
    return false;
}

public void SetRequired(Control ctl, bool val)
{
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    if(val == p.Required)
      return;

    p.Required = val;
    CustomProps[ctl] = p;
    SetToolTip(ctl);
    if(val)
    {
      ctl.KeyPress   += new KeyPressEventHandler(KeyPress);
      ctl.Validating += new CancelEventHandler(ValidateProp);
    }
    else
    {
      ctl.KeyPress   -= new KeyPressEventHandler(KeyPress);
      ctl.Validating -= new CancelEventHandler(ValidateProp);
    }
  }
  else
  {
    Props p = new Props();
    p.Required = val;
```

```
        CustomProps.Add(ctl, p);
        SetToolTip(ctl);
        ctl.Validating += new CancelEventHandler(ValidateProp);
      }
    }

    #endregion
```

**VB**

```
#Region "Required property"

  Public Function GetRequired(ByVal ctl As Control) As Boolean
    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      SetToolTip(ctl)
      Return p.Required
    End If
    Return False
  End Function

  Public Sub SetRequired(ByVal ctl As Control, ByVal val As Boolean)
    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      If val = p.Required Then
        Return
      End If

      p.Required = val
      CustomProps(ctl) = p
      SetToolTip(ctl)
      If val Then
        AddHandler ctl.KeyPress, _
                  New KeyPressEventHandler(AddressOf KeyPress)
        AddHandler ctl.Validating, _
                  New CancelEventHandler(AddressOf ValidateProp)
      Else
        RemoveHandler ctl.KeyPress, _
                  New KeyPressEventHandler(AddressOf KeyPress)
        RemoveHandler ctl.Validating, _
                  New CancelEventHandler(AddressOf ValidateProp)
      End If
```

```
    Else
      Dim p As Props = New Props()
      p.Required = val

      CustomProps.Add(ctl, p)
      SetToolTip(ctl)
      AddHandler ctl.Validating, New CancelEventHandler(AddressOf ValidateProp)
    End If
  End Sub

#End Region
```

Interesting code, this is. Note that I instantiated a hash table in the locals region. I also made a small class that holds the properties. I use the hash table because it accepts a key/value pair. The key in this case is the Control object and the value is the property object instated from the Props class. Hash tables are also extremely fast data structures.

The Get function looks to see if the hash table contains a set of properties for this control. If not, I return a default value of false.[4] The Set function again looks to see if this control already contains a set of properties. If so, I get these properties, make the change, and then store this property back in the hash table. If this Set function does not find the Set properties in the hash table, I instantiate a new set of properties, change the individual property I need, and store it in the hash table. Note that I set the ToolTip property in each of these functions.

If the user sets the Required property to true, I wire up the delegates to the correct events. If not, I unwire them. Doing this obviates the need for my delegates to test whether or not they should be validating. This is the preferred method.

Next on the list are the RangeValueRequired properties.

**C#**

```
    #region RangeValue required

    //Get method for range required
    public bool GetRangeValueRequired(Control ctl)
    {
      //This makes best use of a hashtable for quick retrieval
      if(CustomProps.Contains(ctl))
```

4. I would not hard-code values like this in a real program.

```
  {
    Props p = (Props)CustomProps[ctl];
    SetToolTip(ctl);
    return p.RangeRequired;
  }
  else
    return false;
}

//Set method for Range required
public void SetRangeValueRequired(Control ctl, bool val)
{
  if(CustomProps.Contains(ctl))
  {
    //See if this property is already correctly set
    Props p = (Props)CustomProps[ctl];
    if(val == p.RangeRequired)
      return;

    //Set this property and add it back to the list
    p.RangeRequired = val;
    CustomProps[ctl] = p;
    if(val)
    {
      ctl.KeyPress   += new KeyPressEventHandler(KeyPress);
      ctl.Validating += new CancelEventHandler(ValidateProp);
    }
    else
    {
      ctl.KeyPress   -= new KeyPressEventHandler(KeyPress);
      ctl.Validating -= new CancelEventHandler(ValidateProp);
    }
    SetToolTip(ctl);
  }
  else
  {
    //Set this property and add it to the list
    Props p = new Props();
    p.RangeRequired = val;
    CustomProps.Add(ctl, p);
    if(val)
      ctl.KeyPress += new KeyPressEventHandler(KeyPress);
```

```
        SetToolTip(ctl);
      }
    }

    #endregion
```

**VB**

```
#Region "RangeValue required"

  'Get method for range required
  Public Function GetRangeValueRequired(ByVal ctl As Control) As Boolean
    'This makes best use of a hashtable for quick retrieval
    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      SetToolTip(ctl)
      Return p.RangeRequired
    End If
    Return False
  End Function

  'Set method for Range required
  Public Sub SetRangeValueRequired(ByVal ctl As Control, ByVal val As Boolean)
    If CustomProps.Contains(ctl) Then
      'See if this property is already correctly set
      Dim p As Props = CType(CustomProps(ctl), Props)
      If val = p.RangeRequired Then
        Return
      End If

      'Set this property and add it back to the list
      p.RangeRequired = val
      CustomProps(ctl) = p
      If val Then
        AddHandler ctl.KeyPress, _
                  New KeyPressEventHandler(AddressOf KeyPress)
        AddHandler ctl.Validating, _
                  New CancelEventHandler(AddressOf ValidateProp)
      Else
        RemoveHandler ctl.KeyPress, _
                  New KeyPressEventHandler(AddressOf KeyPress)
        RemoveHandler ctl.Validating, _
                  New CancelEventHandler(AddressOf ValidateProp)
      End If
```

```
      SetToolTip(ctl)
    Else
      'Set this property and add it to the list
      Dim p As Props = New Props()
      p.RangeRequired = val
      CustomProps.Add(ctl, p)
      If val Then
        AddHandler ctl.KeyPress, New KeyPressEventHandler(AddressOf KeyPress)
      End If
      SetToolTip(ctl)
    End If
  End Sub

#End Region
```

This code is much the same as the code for the Required property. One last piece of code is needed here. The following is the code for the minimum and maximum value properties.

**C#**

```
    #region Min and Max Values

    public Decimal GetMinValue(Control ctl)
    {
      if(CustomProps.Contains(ctl))
      {
        Props p = (Props)CustomProps[ctl];
        SetToolTip(ctl);
        return p.MinVal;
      }
      else
        return 0;
    }

    [DefaultValue(0)]
    public void SetMinValue(Control ctl, Decimal val)
    {
      if(val < 0)
        val = 0;
```

```
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    p.MinVal = val;
    CustomProps[ctl] = p;
    SetToolTip(ctl);
  }
  else
  {
    Props p = new Props();
    p.MinVal = val;
    CustomProps.Add(ctl, p);
    SetToolTip(ctl);
  }
}

public Decimal GetMaxValue(Control ctl)
{
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    SetToolTip(ctl);
    return p.MaxVal;
  }
  else
    return 999;
}

[DefaultValue(999)]
public void SetMaxValue(Control ctl, Decimal val)
{
  if(val > 999)
    val = 999;

  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    p.MaxVal = val;
    CustomProps[ctl] = p;
    SetToolTip(ctl);
  }
```

```
      else
      {
        Props p = new Props();
        p.MaxVal = val;
        CustomProps.Add(ctl, p);
        SetToolTip(ctl);
      }
    }

    #endregion
```

**VB**

```
#Region "Min and Max Values"

  Public Function GetMinValue(ByVal ctl As Control) As Decimal
    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      SetToolTip(ctl)
      Return p.MinVal
    End If
    Return 0
  End Function

  '   <DefaultValue(0)>_
  Public Sub SetMinValue(ByVal ctl As Control, ByVal val As Decimal)
    If val < 0 Then val = 0

    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      p.MinVal = val
      CustomProps(ctl) = p
      SetToolTip(ctl)
    Else
      Dim p As Props = New Props()
      p.MinVal = val
      CustomProps.Add(ctl, p)
      SetToolTip(ctl)
    End If
  End Sub
```

```
  Public Function GetMaxValue(ByVal ctl As Control) As Decimal
    If (CustomProps.Contains(ctl)) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      SetToolTip(ctl)
      Return p.MaxVal
    End If
    Return 999
  End Function

  ' [DefaultValue(999)]
  Public Sub SetMaxValue(ByVal ctl As Control, ByVal val As Decimal)
    If val > 999 Then val = 999

    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      p.MaxVal = val
      CustomProps(ctl) = p
      SetToolTip(ctl)
    Else
      Dim p As Props = New Props()
      p.MaxVal = val
      CustomProps.Add(ctl, p)
      SetToolTip(ctl)
    End If
  End Sub

#End Region
```

Note that in this region I use attributes on the Set properties to define a default value.

OK, that's it for this extender. What you have is a control that you can load into your Toolbox and drag onto a form. If this control sees a TextBox on the form, the TextBox will get these four new properties.

If the new properties are set correctly, you will not be able to enter any letters into the TextBox. You can also set the range of numbers allowed. This control provides a custom ToolTip for the TextBox and also provides a custom ErrorProvider control. You have everything you need to get going.

Now for the testing part. Actually, do you want to test this before adding the next extender class? Yes? OK.

## Testing the NumberExtender Control

Add another project to your Vlad solution space. Make this project a Windows Forms project. Also make this the start-up project. Then add two controls to this form. Make sure that one of the controls is a TextBox. If you look at the properties of the TextBox, you will see this is your normal, everyday TextBox.

Now for the magic. While you are in the form, open up the Toolbox. Right-click the Toolbox and choose the .NET Framework Components tab, as shown in Figure 8-14. This is just like choosing the custom TextBox you made in the last example.

Browse to the BIN directory under the VB project (BIN\Debug under the C# project) and double-click Vlad.dll, as shown in Figure 8-14.[5]

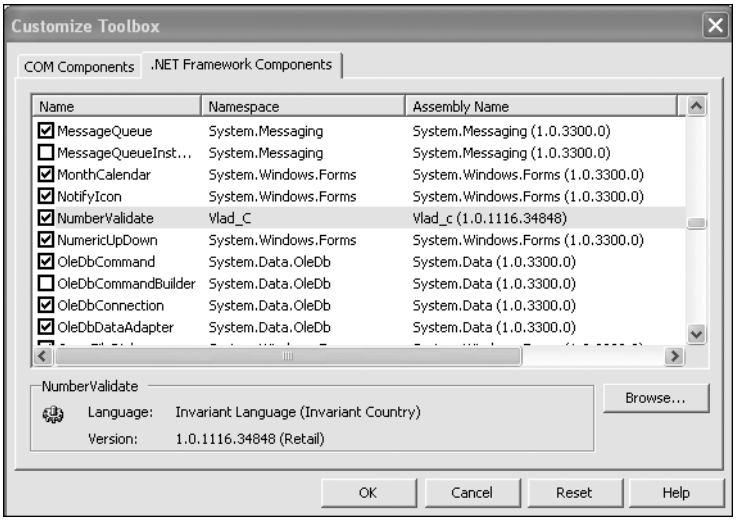


*Figure 8-14. Choosing the new validation control*

Click OK and you should have this control in your Toolbox. Now drag this control over to your form. Make sure that you clear the default text from this control. Figure 8-15 shows the Properties screen for the TextBox after adding the control to the form.

5. Because I have both VB and C# controls, I differentiated them with -C and -VB.

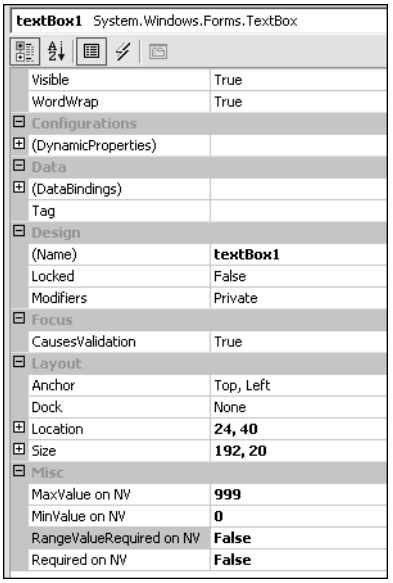


*Figure 8-15. New properties added to control*

See that the default properties came up correctly?

Enable the two required properties and change the MinValue to 50 and the MaxValue to 55. Now run the program. Figure 8-16 shows my form after I exited the TextBox with an invalid entry.

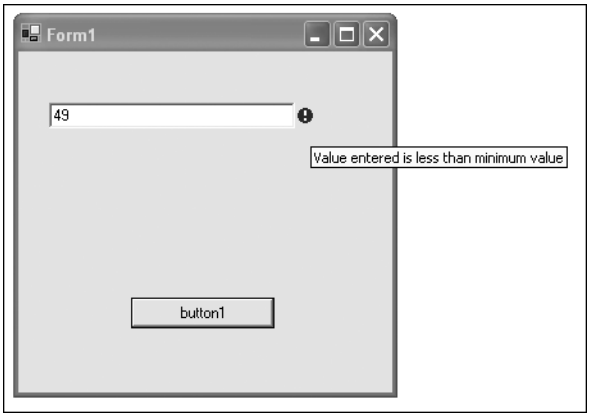


*Figure 8-16. An invalid entry*

Notice that the ErrorProvider has appeared next to the offending TextBox. I moused over the ErrorProvider icon and I got the message you see here.

How is the user supposed to know what is required in the TextBox? This is where the ToolTip text comes in. Hover your mouse over the TextBox and your screen should look like Figure 8-17.

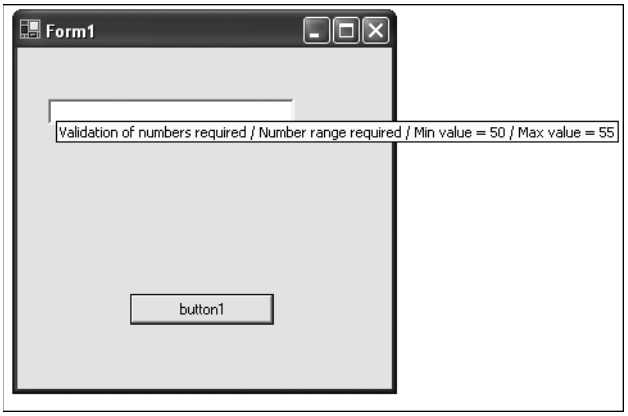


*Figure 8-17. ToolTip help*

This text changes based on what the values were for validating this control. Pretty cool, isn't it? Well, I think so anyway.

Now back to the second validation control.

## *The Regular Expression Validator*

Here is a cool control that I decided to make separate from the NumberValidator. Add a new class to the Vlad project called **RegxValidate**. This class also extends the TextBox control, but with two properties this time. The first property is a Required property and the second is a Mask property. The code for this is much the same as for the NumberValidator. There are a few enhancements, though, that I explain shortly. First, enter the code in Listings 8-2a and 8-2b.

*Listing 8-2a. C# Code for the Regx Class*

```
using System;
using System.Collections;
using System.Windows.Forms;
using System.ComponentModel;
using System.Text.RegularExpressions;
using System.Diagnostics;

namespace Vlad_c
{
  /// <summary>
  /// Allow user to enter a regular expression for validation purposes
  /// </summary>

  [ProvideProperty("Required",          typeof(Control))]
  [ProvideProperty("RegularExpression", typeof(Control))]
  public class RegxValidate: Component, IExtenderProvider
  {

    #region locals

    // holds Key - value pair for efficient retrieval
    //Holds all properties of all controls that use this extender.
    private System.Collections.Hashtable CustomProps  =
                            new System.Collections.Hashtable();

    ErrorProvider er = new ErrorProvider();
    ToolTip       tp = new ToolTip();

    //Holds all the custom properties of a control
    private class Props
    {
      public bool Required  = false;
      public string Regx    = ".*";
    }

    #endregion

    public RegxValidate()
    {
    }
```

```
bool IExtenderProvider.CanExtend(object target)
{
  //Target can be anything; like a ComboBox if needed
  //Target can be multiple things also
  if (target is TextBox)
  {
    return true;
  }
  else
    return false;
}

#region Required property

public bool GetRequired(Control ctl)
{
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    SetToolTip(ctl);
    return p.Required;
  }
  else
    return false;
}

public void SetRequired(Control ctl, bool val)
{
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    if(val == p.Required)
      return;

    p.Required = val;
    CustomProps[ctl] = p;
    SetToolTip(ctl);
    if(val)
      ctl.Validating += new CancelEventHandler(ValidateProp);
    else
      ctl.Validating -= new CancelEventHandler(ValidateProp);
  }
```

```
    else
    {
      Props p = new Props();
      p.Required = val;

      CustomProps.Add(ctl, p);
      SetToolTip(ctl);
      ctl.Validating += new CancelEventHandler(ValidateProp);
    }
  }

  #endregion

  #region Regular Expression property

  public string GetRegularExpression(Control ctl)
  {
    if(CustomProps.Contains(ctl))
    {
      Props p = (Props)CustomProps[ctl];
      SetToolTip(ctl);
      return p.Regx;
    }
    else
      return ".*";
  }

  [DefaultValue("")]
  public void SetRegularExpression(Control ctl, string val)
  {
    //Put something here to verify that regular expression is valid
    try
    {
      Regex.IsMatch("abcdefg", val);
    }
    catch(Exception e)
    {
      string err = "Invalid Regular Expression on:\n";
      err += ctl.Name + "\n\n" + e.Message;
      MessageBox.Show(err);
      er.SetError(ctl, "Invalid Regular Expression!!");
      return;
    }
```

```
  if(CustomProps.Contains(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    p.Regx = val;
    CustomProps[ctl] = p;
  }
  else
  {
    Props p = new Props();
    p.Regx = val;
    CustomProps.Add(ctl, p);
  }
  SetToolTip(ctl);
}

#endregion

#region events

private void ValidateProp(object sender, CancelEventArgs e)
{
  Control ctl = (Control)sender;

  //Reset the error
  er.SetError(ctl, "");

  if(GetRequired(ctl))
  {
    Props p = (Props)CustomProps[ctl];
    if(!Regex.IsMatch(ctl.Text, p.Regx))
      er.SetError(ctl, "Value did not match input restrictions");
    return;
  }
}

#endregion

#region other stuff
```

```
    private void SetToolTip(Control ctl)
    {
      string tip = string.Empty;
      tp.Active = false;
      SetTooltipActive();
      Props p = (Props)CustomProps[ctl];
      if(p.Required)
        tip = "Regular Expression validation: " + p.Regx ;
      tp.SetToolTip(ctl, tip);
    }

    [Conditional("DEBUG")]
    private void SetTooltipActive()
    {
      tp.Active = true;
    }

    #endregion
  }
}
```

*Listing 8-2b. VB Code for the Regx Class*

```
Option Strict On

Imports System
Imports System.Collections
Imports System.Windows.Forms
Imports System.ComponentModel
Imports System.Text.RegularExpressions

<ProvideProperty("Required", GetType(Control)), _
 ProvideProperty("RegularExpression", GetType(Control))> _
Public Class RegxValidate
  Inherits Component
  Implements IExtenderProvider

#Region "locals"

  ' holds Key - value pair for efficient retrieval
  'Holds all properties of all controls that use this extender.
  Private CustomProps As System.Collections.Hashtable = _
                                New System.Collections.Hashtable()
```

```
  Private er As ErrorProvider = New ErrorProvider()
  Private tp As ToolTip = New ToolTip()

  'Holds all the custom properties of a control
  Private Class Props
    Public Required As Boolean = False
    Public Regx As String = ".*"
  End Class

#End Region

  Public Sub New()

  End Sub

  Function CanExtend(ByVal target As Object) As _
                     Boolean Implements IExtenderProvider.CanExtend
    'Target can be anything; like a ComboBox if needed
    'Target can be multiple things also
    If TypeOf target Is TextBox Then
      Return True
    Else
      Return False
    End If
  End Function

#Region "Required property"

  Public Function GetRequired(ByVal ctl As Control) As Boolean
    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      SetToolTip(ctl)
      Return p.Required
    End If
    Return False
  End Function

  Public Sub SetRequired(ByVal ctl As Control, ByVal val As Boolean)
    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      If val = p.Required Then
        Return
      End If
```

```
            p.Required = val
            CustomProps(ctl) = p
            SetToolTip(ctl)
            If val Then
              AddHandler ctl.Validating, _
                        New CancelEventHandler(AddressOf ValidateProp)
            Else
              RemoveHandler ctl.Validating, _
                        New CancelEventHandler(AddressOf ValidateProp)
            End If
          Else
            Dim p As Props = New Props()
            p.Required = val

            CustomProps.Add(ctl, p)
            SetToolTip(ctl)
            AddHandler ctl.Validating, New CancelEventHandler(AddressOf ValidateProp)
          End If
        End Sub

    #End Region

    #Region "Regular Expression property"

        Public Function GetRegularExpression(ByVal ctl As Control) As String
          If CustomProps.Contains(ctl) Then
            Dim p As Props = CType(CustomProps(ctl), Props)
            SetToolTip(ctl)
            Return p.Regx
          Else
            Return ".*"
          End If
        End Function

        <DefaultValue(""")> _
        Public Sub SetRegularExpression(ByVal ctl As Control, ByVal val As String)
          'Put something here to verify that regular expression is valid
          Try
            Regex.IsMatch("abcdefg", val)
          Catch e As Exception
            Dim err As String = "Invalid Regular Expression on:" + vbCrLf
            err += ctl.Name + vbCrLf + vbCrLf + e.Message
            MessageBox.Show(err)
            er.SetError(ctl, "Invalid Regular Expression!!")
```

```
      Return
    End Try

    If CustomProps.Contains(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      p.Regx = val
      CustomProps(ctl) = p
    Else
      Dim p As Props = New Props()
      p.Regx = val
      CustomProps.Add(ctl, p)
    End If
    SetToolTip(ctl)
  End Sub

#End Region

#Region "events"

  Private Sub ValidateProp(ByVal sender As Object, ByVal e As CancelEventArgs)
    Dim ctl As Control = CType(sender, Control)

    'Reset the error
    er.SetError(ctl, "")

    If GetRequired(ctl) Then
      Dim p As Props = CType(CustomProps(ctl), Props)
      If Not Regex.IsMatch(ctl.Text, p.Regx) Then
        er.SetError(ctl, "Value did not match input restrictions")
      End If
      Return
    End If
  End Sub

#End Region

#Region "other stuff"

  Private Sub SetToolTip(ByVal ctl As Control)
    Dim tip As String = String.Empty
    tp.Active = False
    SetTooltipActive()
    Dim p As Props = CType(CustomProps(ctl), Props)
```

```
    If p.Required Then
      tip = "Regular Expression validation: " + p.Regx
    End If
    tp.SetToolTip(ctl, tip)
  End Sub

  <Conditional("DEBUG")> _
  Private Sub SetTooltipActive()
    tp.Active = True
  End Sub

#End Region
End Class
```

Let's look at some interesting code. I decided that I do not want the designer of the form that this component goes on to have a problem with regular expressions. After all, not everyone has read the first part of this chapter and is now an expert.

It is all too easy to enter in a regular expression that cannot be parsed. What do you do then? I have code in the SetRegularExpression method to trap this error. Here is the C# code.

**C#**

```
…
      //Put something here to verify that regular expression is valid
      try
      {
        Regex.IsMatch("abcdefg", val);
      }
      catch(Exception e)
      {
        string err = "Invalid Regular Expression on:\n";
        err += ctl.Name + "\n\n" + e.Message;
        MessageBox.Show(err);
        er.SetError(ctl, "Invalid Regular Expression!!");
        return;
      }
…
```

Because I have no way of parsing a regular expression, I figured I would let the regular expression engine take a look at it. As you can see, I wrap a simple call to the parser in a Try-Catch block. If this call fails, the regular expression passed in was in error. I then pop up a message box with a descriptive error and at the same time I add an error icon next to the control.

Now test this new control. Compile this code and make sure you get no errors. Go into the tester project you have in this solution and remove the number validation control from the form. Also go into the Toolbox and delete this control from the Toolbox. Exit out of the IDE as a whole.

**NOTE** I tested this on a few different machines. On some I needed to reload the IDE and on some I did not. Just to be safe, reload the IDE before you reload the control.

Customize the Toolbox with the same Vlad class you did before. You should now have two new controls. Figure 8-18 shows this.

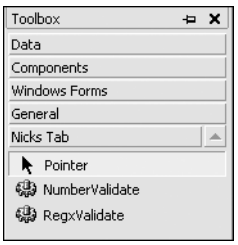


*Figure 8-18. Two new controls*

Drag the RegxValidate control over to your form and you are ready to go.

The neat thing about all this new error code is that it happens at design time. Figure 8-19 shows my form at design time when I enter in the following regular expression: [**0-9+**. Enter this in and click another property or on the form itself.

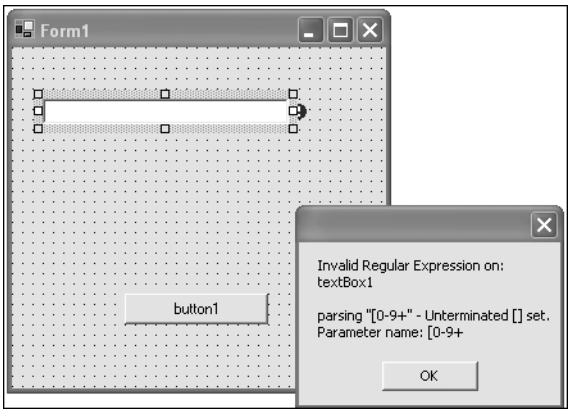


*Figure 8-19. Design-time errors*

See the nice error box? If the expression was good, I would not get this problem. Now just in case you click OK and then get an important phone call and forget what you were doing, I added the error icon to this control. See it in the back there? Figure 8-20 shows the text for this error during design time.

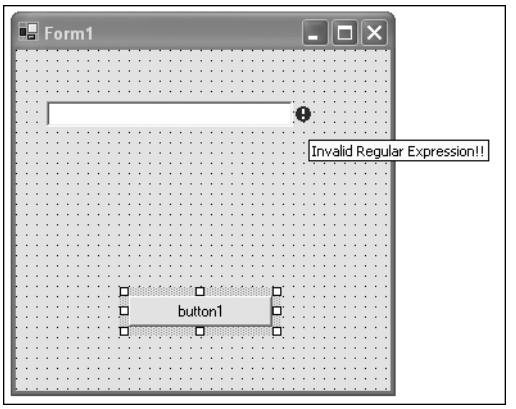


*Figure 8-20. The error icon at design time*

Now there is no way you cannot tell me that this is really cool. It is quite helpful as well. When you come back from your Mountain Dew break, you have a visual cue that you may have forgotten to correct a regular expression problem.

## Special Debug Code

Look at the code for this class carefully and you will see that I have added some special debug code in the form of a ToolTip.

When most users use this control, they will have no clue about regular expressions. The designer of the form will (hopefully) have included some training and a help file. This help file will tell the user what data is valid for this control. As you know, you can enter some pretty hairy regular expressions that may require some weird data pattern to be entered. Showing the user the regular expression that his or her data will be checked against is at best useless. It is also rather unfriendly.

Wouldn't it be nice to offer some ToolTip text for the test engineers? They should, after all, know something about regular expressions. They can at least call you over to see what could be wrong.

Here is the C# code that allows this to happen. The VB code is much the same.

**C#**

```
#region other stuff

private void SetToolTip(Control ctl)
{
  string tip = string.Empty;
  tp.Active = false;
  SetTooltipActive();
  Props p = (Props)CustomProps[ctl];
  if(p.Required)
    tip = "Regular Expression validation: " + p.Regx ;
  tp.SetToolTip(ctl, tip);
}

[Conditional("DEBUG")]
private void SetTooltipActive()
{
  tp.Active = true;
}

#endregion
```

I used the Conditional attribute, which tests for the DEBUG compiler directive. If the debug version of this code is run, the SetToolTipActive method will get called. Otherwise, the line of code that calls this function never gets compiled. Now I think this is a neat feature.

Notice that in the SetToolTip method I set the ToolTip to be inactive. The next line of code is the call to the method that sets it to be active. If you made a release version of this code, you would never see this ToolTip. Figure 8-21 shows this ToolTip in action.

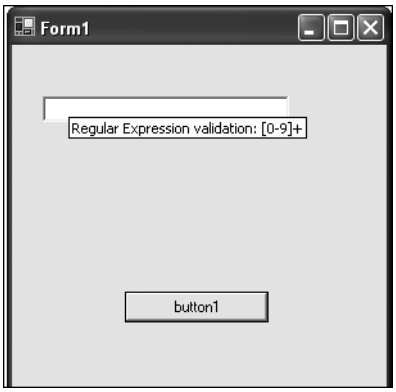


*Figure 8-21. Debug code showing the regular expression*

Both you and your test engineers will love this feature.

Before you leave this section, you will need to test the functionality of this control. I leave it up to you to enter in any regular expression you like and prove to yourself that it works. Try the regular expression shown in Figure 8-21. Remember that this control does not do on-the-fly validation. It validates only when you leave the control.

## Summary

This was a fun chapter. It provided you with an introduction to regular expressions. The regular expression syntax allows you to match or replace anything in a string. This is a powerful tool and I presented only the basics to you here.

I also showed you how you could use the Masked Edit control that comes with VB 6.0. You could use it if it worked, that is. For some reason, it does not work properly in .NET and it can give you some nasty ping-pong bugs.

The Masked Edit control is a nice idea, so I showed you how to make your own. At the same time, I showed you how to put the regular expressions to good use as a validation technique.

I finished the chapter with an explanation of how to write control extenders. These extenders allow you to add properties to any existing control you choose.

The next chapter deals with the subject of XML data and how to validate it.