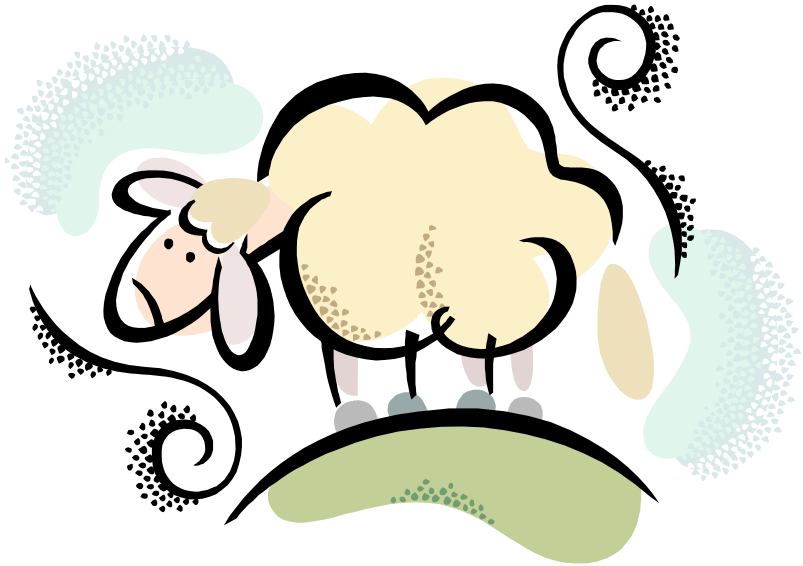

C# Programming IV-8:

System.Text and *System.Text.RegularExpressions Namespaces*

Legal Stuff

This presentation as well as other presentations and documents found on the sheepsqueezers.com website may contain quoted material from outside sources such as books, articles and websites. It is our intention to diligently reference all outside sources. Occasionally, though, a reference may be missed. No copyright infringement whatsoever is intended, and all outside source materials are copyright of their respective author(s).

.NET Lecture Series

sheepsqueezers.com

	C# Programming I: Concepts of OOP	C# Programming II: Beginning C#	C# Programming III: Advanced C#	
C# Programming IV-1: System Namespace	C# Programming IV-2: System.Collections Namespace	C# Programming IV-3: System.Collections. Generic Namespace	C# Programming IV-4A: System.Data Namespace	C# Programming IV-4B: System.Data.Odbc Namespace
C# Programming IV-4C: System.Data.OleDb Namespace	C# Programming IV-4D: Oracle.DataAccess.Client Namespace	C# Programming IV-4E: System.Data.SqlClient Namespace	C# Programming IV-4F: System.Data.SqlTypes Namespace	C# Programming IV-5: System.Drawing/(2D) Namespace
C# Programming IV-6: System.IO Namespace	C# Programming IV-7: System.Numerics	**C# Programming IV-8: System.Text and System.Text. RegularExpressions Namespaces**	C# Programming V: Introduction to LINQ	

C# Self-Inflicted Project #1 Address Cleaning

C# Self-Inflicted Project #2 Large Intersection Problem

Charting Our Course

- The `System.Text` and `System.Text.RegularExpressions` Namespace
- What Next?

The `System.Text`/`System.Text.RegularExpressions` Namespaces

The `System.Text` namespace is defined by Microsoft as follows:

The System.Text namespace contains classes that represent ASCII and Unicode character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes; and a helper class that manipulates and formats String objects without creating intermediate instances of String.

Note that most of the classes available in `System.Text` are esoteric (such as converting blocks of characters to and from blocks of bytes), so we will skip over those classes. One class in particular, the `StringBuilder` class, behaves a lot like the `String` class, but whereas the `String` class creates immutable strings, the `StringBuilder` does not. Depending on the amount of string manipulation you are doing, the `StringBuilder` class may be the more efficient choice.

The System.Text.RegularExpressions namespace is defined by Microsoft as follows:

The System.Text.RegularExpressions namespace contains classes that provide access to the .NET Framework regular expression engine. The namespace provides regular expression functionality that may be used from any platform or language that runs within the Microsoft .NET Framework. In addition to the types contained in this namespace, the System.Configuration.RegexStringValidator class enables you to determine whether a particular string conforms to a regular expression pattern.

The `System.Text`/`System.Text.RegularExpressions` Namespaces

When writing code using this namespace, include the following line at the top of your C# program:

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

The `System.Text` Namespace

- → Classes
  - → `StringBuilder`

StringBuilder

The `StringBuilder` class represents a mutable string of characters. According to Microsoft's website, specifically referring to performance considerations: *The Concat and AppendFormat methods both concatenate new data to an existing String or StringBuilder object. A String object concatenation operation always creates a new object from the existing string and the new data. A StringBuilder object maintains a buffer to accommodate the concatenation of new data. New data is appended to the end of the buffer if room is available; otherwise, a new, larger buffer is allocated, data from the original buffer is copied to the new buffer, then the new data is appended to the new buffer. The performance of a concatenation operation for a String or StringBuilder object depends on how often a memory allocation occurs. A String concatenation operation always allocates memory, whereas a StringBuilder concatenation operation only allocates memory if the StringBuilder object buffer is too small to accommodate the new data. Consequently, the String class is preferable for a concatenation operation if a fixed number of String objects are concatenated. In that case, the individual concatenation operations might even be combined into a single operation by the compiler. A StringBuilder object is preferable for a concatenation operation if an arbitrary number of strings are concatenated; for example, if a loop concatenates a random number of strings of user input.*

StringBuilder

Constructors

- StringBuilder() - Initializes a new instance of the StringBuilder class
- StringBuilder(Int32) - Initializes a new instance of the StringBuilder class using the specified capacity
- StringBuilder(String) - Initializes a new instance of the StringBuilder class using the specified string
- StringBuilder(Int32, Int32) - Initializes a new instance of the StringBuilder class that starts with a specified capacity and can grow to a specified maximum
- StringBuilder(String, Int32) - Initializes a new instance of the StringBuilder class using the specified string and capacity
- StringBuilder(String, Int32, Int32, Int32) - Initializes a new instance of the StringBuilder class from the specified substring and capacity

Properties

- Capacity - Gets or sets the maximum number of characters that can be contained in the memory allocated by the current instance
- Chars - Gets or sets the character at the specified character position in this instance
- Length - Gets or sets the length of the current StringBuilder object
- MaxCapacity - Gets the maximum capacity of this instance

Methods

- Append(Boolean) - Appends the string representation of a specified Boolean value to this instance
- Append(Byte) - Appends the string representation of a specified 8-bit unsigned integer to this instance
- Append(Char) - Appends the string representation of a specified Unicode character to this instance
- Append(Char[]) - Appends the string representation of the Unicode characters in a specified array to this instance
- Append(Decimal) - Appends the string representation of a specified decimal number to this instance
- Append(Double) - Appends the string representation of a specified double-precision floating-point number to this instance
- Append(Int16) - Appends the string representation of a specified 16-bit signed integer to this instance
- Append(Int32) - Appends the string representation of a specified 32-bit signed integer to this instance
- Append(Int64) - Appends the string representation of a specified 64-bit signed integer to this instance
- Append(Object) - Appends the string representation of a specified object to this instance
- Append(SByte) - Appends the string representation of a specified 8-bit signed integer to this instance
- Append(Single) - Appends the string representation of a specified single-precision floating-point number to this instance
- Append(String) - Appends a copy of the specified string to this instance
- Append(UInt16) - Appends the string representation of a specified 16-bit unsigned integer to this instance
- Append(UInt32) - Appends the string representation of a specified 32-bit unsigned integer to this instance

Methods (continued)

- Append(UInt64) - Appends the string representation of a specified 64-bit unsigned integer to this instance
- Append(Char, Int32) - Appends a specified number of copies of the string representation of a Unicode character to this instance
- Append(Char[], Int32, Int32) - Appends the string representation of a specified subarray of Unicode characters to this instance
- Append(String, Int32, Int32) - Appends a copy of a specified substring to this instance
- AppendFormat(String, Object) - Appends the string returned by processing a composite format string, which contains zero or more format items, to this instance. Each format item is replaced by the string representation of a single argument
- AppendFormat(String, Object[]) - Appends the string returned by processing a composite format string, which contains zero or more format items, to this instance. Each format item is replaced by the string representation of a corresponding argument in a parameter array
- AppendFormat(IFormatProvider, String, Object[]) - Appends the string returned by processing a composite format string, which contains zero or more format items, to this instance. Each format item is replaced by the string representation of a corresponding argument in a parameter array using a specified format provider
- AppendFormat(String, Object, Object) - Appends the string returned by processing a composite format string, which contains zero or more format items, to this instance. Each format item is replaced by the string representation of either of two arguments
- AppendFormat(String, Object, Object, Object) - Appends the string returned by processing a composite format string, which contains zero or more format items, to this instance. Each format item is replaced by the string representation of either of three arguments
- AppendLine() - Appends the default line terminator to the end of the current StringBuilder object
- AppendLine(String) - Appends a copy of the specified string followed by the default line terminator to the end of the current StringBuilder object
- Clear - Removes all characters from the current StringBuilder instance
- CopyTo - Copies the characters from a specified segment of this instance to a specified segment of a destination Char array
- EnsureCapacity - Ensures that the capacity of this instance of StringBuilder is at least the specified value
- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Equals(StringBuilder) - Returns a value indicating whether this instance is equal to a specified object
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- Insert(Int32, Boolean) - Inserts the string representation of a Boolean value into this instance at the specified character position

Methods (continued)

- Insert(Int32, Byte) - Inserts the string representation of a specified 8-bit unsigned integer into this instance at the specified character position
- Insert(Int32, Char) - Inserts the string representation of a specified Unicode character into this instance at the specified character position
- Insert(Int32, Char[]) - Inserts the string representation of a specified array of Unicode characters into this instance at the specified character position
- Insert(Int32, Decimal) - Inserts the string representation of a decimal number into this instance at the specified character position
- Insert(Int32, Double) - Inserts the string representation of a double-precision floating-point number into this instance at the specified character position
- Insert(Int32, Int16) - Inserts the string representation of a specified 16-bit signed integer into this instance at the specified character position
- Insert(Int32, Int32) - Inserts the string representation of a specified 32-bit signed integer into this instance at the specified character position
- Insert(Int32, Int64) - Inserts the string representation of a 64-bit signed integer into this instance at the specified character position
- Insert(Int32, Object) - Inserts the string representation of an object into this instance at the specified character position
- Insert(Int32, SByte) - Inserts the string representation of a specified 8-bit signed integer into this instance at the specified character position
- Insert(Int32, Single) - Inserts the string representation of a single-precision floating point number into this instance at the specified character position
- Insert(Int32, String) - Inserts a string into this instance at the specified character position
- Insert(Int32, UInt16) - Inserts the string representation of a 16-bit unsigned integer into this instance at the specified character position
- Insert(Int32, UInt32) - Inserts the string representation of a 32-bit unsigned integer into this instance at the specified character position
- Insert(Int32, UInt64) - Inserts the string representation of a 64-bit unsigned integer into this instance at the specified character position
- Insert(Int32, String, Int32) - Inserts one or more copies of a specified string into this instance at the specified character position
- Insert(Int32, Char[], Int32, Int32) - Inserts the string representation of a specified subarray of Unicode characters into this instance at the specified character position
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- Remove - Removes the specified range of characters from this instance

Methods (continued)

- Replace(Char, Char) - Replaces all occurrences of a specified character in this instance with another specified character
- Replace(String, String) - Replaces all occurrences of a specified string in this instance with another specified string
- Replace(Char, Char, Int32, Int32) - Replaces, within a substring of this instance, all occurrences of a specified character with another specified character
- Replace(String, String, Int32, Int32) - Replaces, within a substring of this instance, all occurrences of a specified string with another specified string
- ToString() - Converts the value of this instance to a String. (Overrides Object.ToString().)
- ToString(Int32, Int32) - Converts the value of a substring of this instance to a String

For example,

```
using System;
using System.Text;

class MainProgram {

 public static void Main() {

  StringBuilder oSB_NAME = new StringBuilder("John Doe");
  StringBuilder oSB_ADDRESS = new StringBuilder("123 Main Street, Anytown, PA 12345-6789");
  StringBuilder oSB_FULLINFO = new StringBuilder();
  oSB_FULLINFO.AppendFormat("{0} lives at {1}.",oSB_NAME,oSB_ADDRESS);
  Console.WriteLine(oSB_FULLINFO.ToString());

 }

}
```

results in: `John Doe lives at 123 Main Street, Anytown, PA 12345-6789.`

Introduction to Regular Expressions

Introduction to Regular Expressions

Unlike searching for text within a string, you can use regular expressions to perform matches based on complex matching rules. Now, before you dive into regular expressions, you need to know the *Regular Expressions Language Elements*, how to use *Character Classes* as well as how to use *Alternation*. This information is available on Microsoft's website at http://msdn.microsoft.com/en-us/library/az24scfc.aspx. Please note that character classes such as `[:digit:]` and `[:alpha:]` are not available. You will have to use the backslashed forms such as `\d` for digits and `\w` for alphabetic characters.

Now, at the very simplest, you can use the `IsMatch` static method of the `Regex` class within the `System.Text.RegularExpressions` namespace. For example,

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

class MainProgram {

 public static void Main() {

 String sADDRESS_1 = "123 Main Street";
  String sRegex_1 = @"^\d+\s+\w+\s+(Street|Str|St)$";
  Boolean bIsMatch_1 = Regex.IsMatch(sADDRESS_1,sRegex_1);
  Console.WriteLine("Match? {0}",bIsMatch_1); //Match? True

  String sADDRESS_2 = "12345 John F. Kennedy Blvd";
  String sRegex_2 = @"^\d+\s+\w+\s+(Blvd|Street|Str|St)$";
  Boolean bIsMatch_2 = Regex.IsMatch(sADDRESS_2,sRegex_2);
  Console.WriteLine("Match? {0}",bIsMatch_2); //Match? False

 }

}
```

Introduction to Regular Expressions

The first regular expression `^\d+\s+\w+\s+(Street|Str|St)$` indicates that we are looking for our string to match this pattern. This pattern means the following:

- ^ → indicates, along with the $, that the search must match the entire string provided to it and not just a substring within the full string.
- \d+ → indicates what must follow next is a series of *one or more* (+) digits (\d).
- \s+ → indicates that what must now follow is a series of one or more (+) whitespace characters (\s) such as blanks, line feeds, tabs, etc.
- \w+ → indicates that one or more (+) alphabetic characters (\w) must follow.
- \s+ → indicates that what must now follow is a series of one or more (+) whitespace characters (\s) such as blanks, line feeds, tabs, etc.
- (Street|Str|St) → indicates that one of the provided words must match. The vertical bar along with the parentheses indicate that an alternation is occurring.
- $ → indicates, along with the ^, that the search must match the entire string provided to it and not just a substring within the full string.

Now, you don't have to use the ^ and $, and can search within your string to determine if a match occurred. For example,

Introduction to Regular Expressions


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

class MainProgram {

 public static void Main() {

  String sADDRESS = "123 Main Street";
  String sRegex = @"\d+";
  Boolean bIsMatch = Regex.IsMatch(sADDRESS,sRegex);
  Console.WriteLine("Match? {0}",bIsMatch); //Match? True

 }

}
```

Here we are just looking for one or more digits within the string sADDRESS. Naturally, the 123 will be found and the result returned is a True.

But, what if you wanted to know what the matched number was? In this case, you can use the static `Match` method of the `Regex` class. Be aware that the `Match` method returns a `Match` object and not a string!! Here is an example (see next slide).

Introduction to Regular Expressions


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

class MainProgram {

 public static void Main() {

  String sADDRESS = "123 Main Street";
  String sRegex = @"\d+";
  Match oDigitsMatch = Regex.Match(sADDRESS,sRegex);
  Console.WriteLine("The match is: {0}.",oDigitsMatch.Value); //The match is: 123.

 }

}
```

In the code above we get back the number 123 as our matched value. But, what happens if there are several numbers within our string? Here we can iterate through the matches using the `NextMatch` method:

```
  String sADDRESS = "123 456 789 012";
  String sRegex = @"\d+";
  Match oDigitsMatch = Regex.Match(sADDRESS,sRegex);
  while(oDigitsMatch.Success) {

   Console.WriteLine("This match is: {0}.",oDigitsMatch.Value);
   oDigitsMatch = oDigitsMatch.NextMatch();

  }
```

```
This match is: 123.
This match is: 456.
This match is: 789.
This match is: 012.
```

Introduction to Regular Expressions

Now, you can collect *all of your matches at once*, rather than performing the while loop as shown on the previous slide, by using the `Matches` method. This method returns a `MatchCollection` object rather than a `Method` object. For example,

```
String sADDRESS = "123 456 789 012";
String sRegex = @"\d+";
MatchCollection oDigitsMatchColl = Regex.Matches(sADDRESS,sRegex);
foreach(Match oMatch in oDigitsMatchColl) {
 Console.WriteLine("This match is: {0}.",oMatch.Value);
}
```

Be aware that if you want to copy the resulting collection to an array, say, you can use the `CopyTo` method:

```
String sADDRESS = "123 456 789 012";
String sRegex = @"\d+";
MatchCollection oDigitsMatchColl = Regex.Matches(sADDRESS,sRegex);
Object[] asMatches = new Object[4];
oDigitsMatchColl.CopyTo(asMatches,0);
for(Int32 indx=0;indx<asMatches.Length;indx++) {
 Console.WriteLine(asMatches[indx]);
}
```

Note that, for some reason, I can only get the Object datatype to work; the String datatype does not work in the code above!!

Introduction to Regular Expressions

So far, we've seen how to determine if a regular expression has a match within a text string. But, what happens if you want to know what that match was. In this case, we have to modify the regular expression slightly so that we can "capture" that information. For example, the regular expression to capture the each number in the text string "123 456 789 012" would look like this:

```
String sADDRESS = "123 456 789 012";
String sRegex = @"(\d+)";
MatchCollection oMatches = Regex.Matches(sADDRESS,sRegex);
Console.WriteLine(oMatches[0].Groups[1].Captures[0].Value);
```

The result of this is the output "123". Notice that the regular expression is NOT constrained with the ^ and $ symbols, so our regular expression will match each number (123, 456, 789 and 012) in turn if we used a foreach loop or the NextMatch method. In this case, though, we gather the first match (oMatches[0]), the first group (Groups[1]) and the first capture (Capture[0]). Note that the group corresponds to the parenthesized part of the regular expression. Note, also, that we are starting at the second element ([1]) rather than the traditional first element ([0]). This is because Groups[0] returns the complete match…we will see more about this in a moment.

Here is another example:

Introduction to Regular Expressions


```
String sADDRESS = "123 456 789 012";
String sRegex = @"(\d+)";
MatchCollection oMatches = Regex.Matches(sADDRESS,sRegex);
foreach(Match oMatch in oMatches) {
 Console.WriteLine(oMatch.Groups[1].Captures[0].Value);
}
```

In this case, the output we receive is:

```
123
456
789
012
```

Note that the foreach loop is giving us back each individual match based on our regular expression. In this case, we get back 123 first, then 456, and so on. And it is because of this that there is only one group, (\d+), and only one capture. Another way to approach this is to change the regular expression to match more than one thing at a time, as in this example:

```
String sADDRESS = "123 456 789 012";
String sRegex = @"(\d+\s*)+";
Match oMatch = Regex.Match(sADDRESS,sRegex);
GroupCollection oGrpColl = oMatch.Groups;
Group oGrp = oGrpColl[1];
Console.WriteLine("Captured ==>{0}<==",oGrp.Captures[0].Value);
Console.WriteLine("Captured ==>{0}<==",oGrp.Captures[1].Value);
Console.WriteLine("Captured ==>{0}<==",oGrp.Captures[2].Value);
Console.WriteLine("Captured ==>{0}<==",oGrp.Captures[3].Value);
```

Introduction to Regular Expressions

The output is:

```
Captured ==>123 <==
Captured ==>456 <==
Captured ==>789 <==
Captured ==>012<==
```

Notice that we are using `Regex.Match` and not `Regex.Matches`. This will allow us to gather all of the groups and captures in the group collection.

Another way to approach this is to add additional groups to our regular expression:

```
String sADDRESS = "123 456 789 012";
String sRegex = @"(\d+)\s+(\d+)\s+(\d+)\s+(\d+)";
Match oMatch = Regex.Match(sADDRESS,sRegex);
Console.WriteLine(oMatch.Groups[0].Captures[0].Value); //Full match when Groups[0] at 0-index.
Console.WriteLine(oMatch.Groups[1].Captures[0].Value); //[1] ==> first set of parens (i.e., group)
Console.WriteLine(oMatch.Groups[2].Captures[0].Value); //[2] ==> second set of parens (i.e., group)
Console.WriteLine(oMatch.Groups[3].Captures[0].Value); //[3] ==> third set of parens (i.e., group)
Console.WriteLine(oMatch.Groups[4].Captures[0].Value); //[4] ==> fourth set of parens (i.e., group)
```

The output is:

```
123 456 789 012
123
456
789
012
```

Introduction to Regular Expressions

In this case, we have *four groups*, one for each of the corresponding sets of numbers. Now, each group will only capture one thing instead of multiple things as shown on Slide #20. Notice, again, that Group[0] is a special case and can be ignored; Groups should start counting at 1, not 0.

Here is another example using an address. In this case, we have three groups, one for house number, one for street name, and one for street type:

```
String sADDRESS = "123 Main Street";
String sRegex = @"^(\d+)\s+(\w+)\s+(\w+)$";
Match oMatch = Regex.Match(sADDRESS,sRegex);
Console.WriteLine(oMatch.Groups[0].Captures[0].Value);
Console.WriteLine(oMatch.Groups[1].Captures[0].Value);
Console.WriteLine(oMatch.Groups[2].Captures[0].Value);
Console.WriteLine(oMatch.Groups[3].Captures[0].Value);
```

The results are:

```
123 Main Street
123
Main
Street
```

As you can see, Groups[0] is just the full match and can be ignored. Note that each group will have only one capture which is why we used Captures[0].Value throughout this example.

Introduction to Regular Expressions

Now, up to this point we have been using static methods of the `Regex` class instead of instantiating an object using one of the `Regex` constructors. According to Microsoft's website: *Static regular expression methods are recommended as an alternative to repeatedly instantiating a regular expression object with the same regular expression. You should replace…inefficient code with a call to static `Regex` methods. This eliminates the need to instantiate a `Regex` object each time you want to call a pattern-matching method, and enables the regular expression engine to retrieve a compiled version of the regular expression from its cache. By default, the last 15 most recently used static regular expression patterns are cached. For applications that require a larger number of cached static regular expressions, the size of the cache can be adjusted by setting the `Regex.CacheSize` property.*

Now, when you use an instantiated `Regex` object, you can specify the option `RegexOptions.Compiled` in the constructor to force your regular expression compiled. According to Microsoft's website: *Regular expression patterns that are not bound to the regular expression engine through the specification of the Compiled option are interpreted. When a regular expression object is instantiated, the regular expression engine converts the regular expression to a set of operation codes. When an instance method is called, the operation codes are converted to MSIL and executed by the JIT compiler. Similarly, when a static regular expression method is called and the regular expression cannot be found in the cache, the regular expression engine converts the regular expression to a set of operation codes and stores them in the cache. It then converts these operation codes to MSIL so that the JIT compiler can execute them.*

Introduction to Regular Expressions

Interpreted regular expressions reduce startup time at the cost of slower execution time. Because of this, they are best used when the regular expression is used in a small number of method calls, or if the exact number of calls to regular expression methods is unknown but is expected to be small. As the number of method calls increases, the performance gain from reduced startup time is outstripped by the slower execution speed.

Regular expression patterns that are bound to the regular expression engine through the specification of the Compiled option are compiled. This means that, when a regular expression object is instantiated, or when a static regular expression method is called and the regular expression cannot be found in the cache, the regular expression engine converts the regular expression to an intermediary set of operation codes, which it then converts to MSIL. When a method is called, the JIT compiler executes the MSIL. In contrast to interpreted regular expressions, compiled regular expressions increase startup time but execute individual pattern-matching methods faster. As a result, the performance benefit that results from compiling the regular expression increases in proportion to the number of regular expression methods called.

To summarize, we recommend that you use interpreted regular expressions when you call regular expression methods with a specific regular expression relatively infrequently. You should use compiled regular expressions when you call regular expression methods with a specific regular expression relatively frequently.

Introduction to Regular Expressions

The exact threshold at which the slower execution speeds of interpreted regular expressions outweigh gains from their reduced startup time, or the threshold at which the slower startup times of compiled regular expressions outweigh gains from their faster execution speeds, is difficult to determine. It depends on a variety of factors, including the complexity of the regular expression and the specific data that it processes. To determine whether interpreted or compiled regular expressions offer the best performance for your particular application scenario, you can use the Stopwatch class to compare their execution times.

Now, you can get further performance increases by compiling your regular expressions to an assembly. According to Microsoft's website: *The .NET Framework also enables you to create an assembly that contains compiled regular expressions. This moves the performance hit of regular expression compilation from run time to design time. However, it also involves some additional work: You must define the regular expressions in advance and compile them to an assembly. The compiler can then reference this assembly when compiling source code that uses the assembly's regular expressions. Each compiled regular expression in the assembly is represented by a class that derives from Regex. To compile regular expressions to an assembly, you call the Regex.CompileToAssembly(RegexCompilationInfo[], AssemblyName) method and pass it an array of RegexCompilationInfo objects that represent the regular expressions to be compiled, and an AssemblyName object that contains information about the assembly to be created.*

Introduction to Regular Expressions

We recommend that you compile regular expressions to an assembly in the following situations:

- *If you are a component developer who wants to create a library of reusable regular expressions.*
- *If you expect your regular expression's pattern-matching methods to be called an indeterminate number of times -- anywhere from once or twice to thousands or tens of thousands of times. Unlike compiled or interpreted regular expressions, regular expressions that are compiled to separate assemblies offer performance that is consistent regardless of the number of method calls.*

If you are using compiled regular expressions to optimize performance, you should not use reflection to create the assembly, load the regular expression engine, and execute its pattern-matching methods. This requires that you avoid building regular expression patterns dynamically, and that you specify any pattern-matching options (such as case-insensitive pattern matching) at the time the assembly is created. It also requires that you separate the code that creates the assembly from the code that uses the regular expression.

Now, to increase the cache size, do this:

```
Regex.CacheSize=100; //up the cache to hold 100 regexes.
```

Introduction to Regular Expressions

To use the `RegexOptions.Compiled` option, instantiate a `Regex` using the constructor that allows for `RegexOptions`:

```
String sADDRESS = "123 456 789 012";
String sRegex = @"(\d+)\s+(\d+)\s+(\d+)\s+(\d+)";
Regex oRE = new Regex(sRegex,RegexOptions.Compiled);
Match oMatch = oRE.Match(sADDRESS);
Console.WriteLine(oMatch.Groups[0].Captures[0].Value); //Full match when Groups[0] at 0-index.
Console.WriteLine(oMatch.Groups[1].Captures[0].Value); //[1] ==> first set of parens (i.e., group)
Console.WriteLine(oMatch.Groups[2].Captures[0].Value); //[2] ==> second set of parens (i.e., group)
Console.WriteLine(oMatch.Groups[3].Captures[0].Value); //[3] ==> third set of parens (i.e., group)
Console.WriteLine(oMatch.Groups[4].Captures[0].Value); //[4] ==> fourth set of parens (i.e., group)
```

NOTE: YOU MAY WANT TO TRY YOUR CODE WITH AND WITHOUT THE `RegexOptions.Compiled` SWITCH! I FOUND THAT WHEN THE SWITCH IS TURNED ON, THE FIRST CALL TO THE MATCH METHOD TAKES A WHILE TO COMPILE AND RETURN RESULTS WHEREAS WHEN THE SWITCH IS OFF THAT FIRST CALL TO MATCH TAKES NO TIME AT ALL!! IN EITHER CASE, THE SECOND CALL TO MATCH TAKES NO TIME AT ALL!!

Now, if you are dealing with a few regular expressions, the code above is probably sufficiently fast. Please read this article for more information: http://msdn.microsoft.com/en-us/library/gg578045.aspx.

Introduction to Regular Expressions

We seen several times how to access capture groups in a regular expression using the parentheses. But, what happens if you need to search for one of several alternate choices? This is where *alternation* comes in. No, alternation has nothing to do with tailoring clothes. Alternation allows you to specify a pipe-delimited series of characters as alternatives within your regular expression. For example, in the example below, I am going to try to match the text `fred1wilma` and `fred2wilma` using a single regular expression with alternation:

```
String sTEXT="fred2wilma";
Regex oRE=new Regex(@"^fred(1|2)wilma$");

Match oMatch = oRE.Match(sTEXT);
Console.WriteLine(oMatch.Success);
if (oMatch.Success) {
 Console.WriteLine("Match ==>{0}<==",oMatch.Value);
}

sTEXT="fred1wilma";
oMatch = oRE.Match(sTEXT);
Console.WriteLine(oMatch.Success);
if (oMatch.Success) {
 Console.WriteLine("Match ==>{0}<==",oMatch.Value);
}
```

As you see, the regular expression contains the alternation `(1|2)` which is a pipe-delimited series of numbers, in this case 1 and 2. This is equivalent to the two regular expressions "`^fred1wilma$`" and "`^fred2wilma$`". Now, alternation is nice because you can specify words instead of just characters or numbers.

Introduction to Regular Expressions

Below, we're going to break apart the address "123 Main Street" into the house number, the street name and the street type. Note that we are using alternation as well as capture groups to find all of this information.

Note that the parentheses for alternation also act to indicate a capture group.

```
String sADDRESS="123 MAIN STREET";
Regex oRE=new Regex(@"^(\d+) +(\w+) +(STREET|STR|ST|PLACE|PL|DRIVE|DR){1} *$");

Match oMatch = oRE.Match(sADDRESS);
if (oMatch.Success) {
 GroupCollection oGrpColl = oMatch.Groups;
 Console.WriteLine("Captured ==>{0}<==",oGrpColl[1].Captures[0].Value);
 Console.WriteLine("Captured ==>{0}<==",oGrpColl[2].Captures[0].Value);
 Console.WriteLine("Captured ==>{0}<==",oGrpColl[3].Captures[0].Value);
}
```

The results are:

```
Captured ==>123<==
Captured ==>MAIN<==
Captured ==>STREET<==
```

Take note that in the alternation above, we are specifying several alternatives to the word STREET as well as PLACE and DRIVE. We are also specifying {1} to indicate that only one of the alternatives should occur.

Now, there may be times where you don't want the alternation parentheses to signify a capture group. You can turn off the capture group for alternation…

Introduction to Regular Expressions

…by specifying a `?:` just after the left alternation parenthesis. For example,

```
Regex oRE=new Regex(@"^(\d+) +(\w+) +(?:STREET|STR|ST|PLACE|PL|DRIVE|DR){1} *$");
```

This will indicate two capture groups instead of three in the previous example since the last one has been turned off.

Now, alternation has a nice feature called *conditional matching*. This allows you to provide an expression and if that expression is matched, the first conditional regular expression is executed; otherwise, the second conditional regular expression is executed. This is coded as follows:

```
(?(conditional_expression) found_regex | notfound_regex)
```

Take note of the very important vertical bar as well as ending parenthesis!

For example, suppose you have a file containing either the United States Social Security Number (NNN-NN-NNNN) or the United Kingdom National Insurance Number (LL-NNNNNNL), where L is a letter and N is a number. We can use a conditional regular expression to return either the US Social Security Number or the UK National Insurance Number. See the example on the next slide.

Introduction to Regular Expressions


```
Regex oRE=new Regex(@"^(?(\d{3}-\d{2}-\d{4})\d{3}-\d{2}-\d{4}|\w{2}\d{6}\w{1})$");

String sID1="111-22-3333";
Match oMatch = oRE.Match(sID1);
if (oMatch.Success) {
 Console.WriteLine("Found ==>{0}<==",oMatch.Value);
}

String sID2="AB123456C";
oMatch = oRE.Match(sID2);
if (oMatch.Success) {
 Console.WriteLine("Found ==>{0}<==",oMatch.Value);
}
```

As you can see, the regex code is rather confusing. Here is the regex in living color:

```
Regex oRE=new Regex(@"^(?(\d{3}-\d{2}-\d{4})\d{3}-\d{2}-\d{4}|\w{2}\d{6}\w{1})$");
```

The purple code is the conditional expression; the green code is the regular expression that is executed if the conditional expression is true; the red code is the regular expression that is executed if the conditional expression is false. The results are:

```
Found ==>111-22-3333<==
Found ==>AB123456C<==
```

Introduction to Regular Expressions

For the last few slides we've been talking about grouping constructs using one or more pairs of regular expressions contained within a pair of parentheses. Now, by default, each pair of grouping construct parentheses is assigned an internal number assigned for each left parenthesis starting at the beginning of the regular expression. Rather than just using C# code such as `GrpColl[1].Captures[0].Value`, you can refer to each capture as \1, \2, and so on within the same regular expression; or, $1, $2, and so on such when using the `Replace` method.

For example, say we are given a string containing "111-222". We can use regular expressions to easily "flip" the string into "222-111" by using the following code:

```
String sTEXT="111-222";
Regex oRE = new Regex(@"^(\d{1,})-(\d{1,})$");
String sFlipped = oRE.Replace(sTEXT,@"$2-$1"); // $1 is first capture, $2 is second capture
Console.WriteLine(sFlipped); //222-111
```

Note that in this case, we are using $1 to refer to the first capture (the 111 in the example string) and $2 to refer to the second capture (the 222 in the example string).

When you need to refer to a capture within the same regular expression, you use the \1, etc. construct instead of the $1, etc. construct. For example…

Introduction to Regular Expressions


```
String sTEXT="111-222-111";
Regex oRE = new Regex(@"^(\d{1,})-(\d{1,})-\1$");
String sFlipped = oRE.Replace(sTEXT,@"$2-$1($1)");
Console.WriteLine(sFlipped); //222-111(111)
```

Note that, in this case, I have place a \1 in the regular expression. This \1 refers to the first capture.

Now, instead of allowing .NET to use the automatic numbering for captures, you can give each capture a name. Within the capture group, place a ? followed by a less than symbol (<), followed by your desired name, followed by a greater than symbol (>): `?<NAME>`. Now, to refer to this named capture in the Replace method, use a $, followed by a left brace ({), followed by the name, followed by a right brace (}): `${NAME}`. The code starts to get more cluttered and complicated, but this option is there for the emotionally scarred. Here is an example:

```
String sTEXT="111-222";
Regex oRE = new Regex(@"^(?<NBR1>\d{1,})-(?<NBR2>\d{1,})$");
String sFlipped = oRE.Replace(sTEXT,@"${NBR2}-${NBR1}");
Console.WriteLine(sFlipped); //222-111
```

You can specify a series of options within a capture group to enable or disable certain functionality. For example, to ignore case, use the ?i: option within your capture group: `(?i:`*`regex`*`)`. See the next slide for an example.

Introduction to Regular Expressions


```
String sTEXT="AaA-BbB";
Regex oRE = new Regex(@"^((?i:[A-Z]+))-((?i:[A-Z]+))$");
String sFlipped = oRE.Replace(sTEXT,@"$2-$1");
Console.WriteLine(sFlipped); //BbB-AaA
```

Note the I surrounded each capture group by a second set of parentheses. This is because when I tried this example without the surrounding set of parentheses, there were no captures captured.

Now, at this point, you've probably surmised that as each part of a regular expression is matched, a "pointer" to the input string is moving forward to the next part of the regex to be analyzed. This is true for everything we've talked about except for *conditional matching*. You'll notice that with conditional matching, the condition tested does NOT force the "pointer" within the input string to move forward; but, when either the true or false regular expression is executed, then the pointer begins to move forward again. The condition portion of the conditional matching regular expression is called a *zero-width assertion*. There are four zero-width assertions available to you:

1. Zero-Width Positive Lookahead Assertions
2. Zero-Width Negative Lookahead Assertions
3. Zero-Width Positive Lookbehind Assertions
4. Zero-Width Negative Lookbehind Assertions

Introduction to Regular Expressions

Zero-Width Positive Lookahead Assertions

A *zero-width positive lookahead assertion* is coded in a regular expression as follows: `(?=subexpression)`. According to Microsoft's website: *For a regular expression match to be successful, the input string must match the regular expression pattern in subexpression, although the matched substring is not included in the match result.* Based on this information, a *conditional matching* (which we talked about a few slides back) can be coded like this:

```
(?(?=expression)yes|no)
```

as well as the original form we learned:

```
(?(expression)yes|no)
```

Either form works.

Below is an example of zero-width positive lookahead assertions.

Introduction to Regular Expressions


```
String sTEXT="AB123456C";
Regex oRE = new Regex(@"^(?=[AB]{2})(.*)$");
Match oMatch = oRE.Match(sTEXT);
if (oMatch.Success) {
 GroupCollection oGrpColl = oMatch.Groups;
 Console.WriteLine("Captured ==>{0}<==",oGrpColl[1].Captures[0].Value);
}
```

The code in the red above is the zero-width assertion and it is asserting that the product number must start with exactly two letters, AA, AB, BA, or BB. Note that the first capture group is actually the `(.*)` and NOT the assertion itself! Also, note that the result is `Captured ==>AB123456C<==`. As you see, the first two letters are captured as well…this indicates that the pointer within the input string did NOT move despite there being additional regex code within parentheses before the `(.*)`.

For the rest of the assertions, please refer to the Grouping Constructs webpage at http://msdn.microsoft.com/en-us/library/bs2twtah(v=VS.100).aspx.

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `CaptureCollection`

CaptureCollection

The `CaptureCollection` class represents the set of captures made by a single capturing group. According to Microsoft's website: *The collection is immutable (read-only) and has no public constructor. The CaptureCollection object contains one or more Capture objects. Instances of the CaptureCollection class are returned by the following properties:*

- *The Group.Captures property. Each member of the collection represents a substring captured by a capturing group. If a quantifier is not applied to a capturing group, the CaptureCollection includes a single Capture object that represents the same captured substring as the Group object. If a quantifier is applied to a capturing group, the CaptureCollection includes one Capture object for each captured substring, and the Group object provides information only about the last captured substring.*
- *The Match.Captures property. In this case, the collection consists of a single Capture object that provides information about the match as a whole. That is, the CaptureCollection object provides the same information as the Match object.*

To iterate through the members of the collection, you should use the collection iteration construct provided by your language (such as foreach in C# and For Each…Next in Visual Basic) instead of retrieving the enumerator that is returned by the GetEnumerator method.

CaptureCollection

Properties

- Count - Gets the number of substrings captured by the group.
- IsReadOnly - Gets a value that indicates whether the collection is read only.
- IsSynchronized - Gets a value that indicates whether access to the collection is synchronized (thread-safe).
- Item - Gets an individual member of the collection.
- SyncRoot - Gets an object that can be used to synchronize access to the collection.

Methods

- CopyTo - Copies all the elements of the collection to the given array beginning at the given index.
- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetEnumerator - Provides an enumerator that iterates through the collection.
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- ToString - Returns a string that represents the current object. (Inherited from Object.)

Extension Methods

- AsParallel - Enables parallelization of a query. (Defined by ParallelEnumerable.)
- AsQueryable - Converts an IEnumerable to an IQueryable. (Defined by Queryable.)
- Cast<TResult> - Converts the elements of an IEnumerable to the specified type. (Defined by Enumerable.)
- OfType<TResult> - Filters the elements of an IEnumerable based on a specified type. (Defined by Enumerable.)

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `Capture`

Capture

The `Capture` class represents the represents the results from a single successful subexpression capture. According to Microsoft's website: *A Capture object is immutable and has no public constructor. Instances are returned through the CaptureCollection object, which is returned by the Match.Captures and Group.Captures properties. However, the Match.Captures property provides information about the same match as the Match object. If you do not apply a quantifier to a capturing group, the Group.Captures property returns a CaptureCollection with a single Capture object that provides information about the same capture as the Group object. If you do apply a quantifier to a capturing group, the Group.Index, Group.Length, and Group.Value properties provide information only about the last captured group, whereas the Capture objects in the CaptureCollection provide information about all subexpression captures.*

Properties

- Index - The position in the original string where the first character of the captured substring was found.
- Length - The length of the captured substring.
- Value - Gets the captured substring from the input string.

Methods

- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- ToString - Gets the captured substring from the input string. (Overrides Object.ToString().)

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `GroupCollection`

GroupCollection

The `GroupCollection` class represents returns the set of captured groups in a single match. According to Microsoft's website: *The collection is immutable (read-only) and has no public constructor. A GroupCollection object is returned by the Match.Groups property. The collection contains one or more System.Text.RegularExpressions.Group objects. If the match is successful, the first element in the collection contains the Group object that corresponds to the entire match. Each subsequent element represents a captured group, if the regular expression includes capturing groups. If the match is unsuccessful, the collection contains a single System.Text.RegularExpressions.Group object whose Success property is false and whose Value property equals String.Empty. To iterate through the members of the collection, you should use the collection iteration construct provided by your language (such as foreach in C# and For Each…Next in Visual Basic) instead of retrieving the enumerator that is returned by the GetEnumerator method.*

Properties

- Count - Returns the number of groups in the collection.
- IsReadOnly - Gets a value that indicates whether the collection is read-only.
- IsSynchronized - Gets a value that indicates whether access to the GroupCollection is synchronized (thread-safe).
- Item[Int32] - Enables access to a member of the collection by integer index.
- Item[String] - Enables access to a member of the collection by string index.
- SyncRoot - Gets an object that can be used to synchronize access to the GroupCollection.

GroupCollection

Methods

- CopyTo - Copies all the elements of the collection to the given array beginning at the given index.
- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetEnumerator - Provides an enumerator that iterates through the collection.
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- ToString - Returns a string that represents the current object. (Inherited from Object.)

Extension Methods

- AsParallel - Enables parallelization of a query. (Defined by ParallelEnumerable.)
- AsQueryable - Converts an IEnumerable to an IQueryable. (Defined by Queryable.)
- Cast<TResult> - Converts the elements of an IEnumerable to the specified type. (Defined by Enumerable.)
- OfType<TResult> - Filters the elements of an IEnumerable based on a specified type. (Defined by Enumerable.)

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `Group`

Group

The `Group` class represents the results from a single capturing group. According to Microsoft's website: *A capturing group can capture zero, one, or more strings in a single match because of quantifiers. (For more information, see Quantifiers.) All the substrings matched by a single capturing group are available from the Group.Captures property. Information about the last substring captured can be accessed directly from the Value and Index properties. (That is, the Group instance is equivalent to the last item of the collection returned by the Captures property, which reflects the last capture made by the capturing group.)*

Properties

- Captures - Gets a collection of all the captures matched by the capturing group, in innermost-leftmost-first order (or innermost-rightmost-first order if the regular expression is modified with the RegexOptions.RightToLeft option). The collection may have zero or more items.
- Index - The position in the original string where the first character of the captured substring was found. (Inherited from Capture.)
- Length - The length of the captured substring. (Inherited from Capture.)
- Success - Gets a value indicating whether the match is successful.
- Value - Gets the captured substring from the input string. (Inherited from Capture.)

Methods

- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- Synchronized - Returns a Group object equivalent to the one supplied that is safe to share between multiple threads.
- ToString - Gets the captured substring from the input string. (Inherited from Capture.)

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `MatchCollection`

MatchCollection

The `MatchCollection` class represents the set of successful matches found by iteratively applying a regular expression pattern to the input string. According to Microsoft's website: *The collection is immutable (read-only) and has no public constructor. The Regex.Matches method returns a MatchCollection object. The collection contains zero or more System.Text.RegularExpressions.Match objects. If the match is successful, the collection is populated with one System.Text.RegularExpressions.Match object for each match found in the input string. If the match is unsuccessful, the collection contains no System.Text.RegularExpressions.Match objects, and its Count property equals zero. When applying a regular expression pattern to a particular input string, the regular expression engine uses either of two techniques to build the MatchCollection object:*

- *Direct evaluation: The MatchCollection object is populated all at once, with all matches resulting from a particular call to the Regex.Matches method. This technique is used when the collection's Count property is accessed. It typically is the more expensive method of populating the collection and entails a greater performance hit.*

- *Lazy evaluation: The MatchCollection object is populated as needed on a match-by-match basis. It is equivalent to the regular expression engine calling the Regex.Match method repeatedly and adding each match to the collection. This technique is used when the collection is accessed through its GetEnumerator method, or when it is accessed using the foreach statement (in C#) or the For Each...Next statement (in Visual Basic).*

MatchCollection

To iterate through the members of the collection, you should use the collection iteration construct provided by your language (such as foreach in C# and For Each…Next in Visual Basic) instead of retrieving the enumerator that is returned by the GetEnumerator method.

Properties

- Count - Gets the number of matches.
- IsReadOnly - Gets a value that indicates whether the collection is read only.
- IsSynchronized - Gets a value indicating whether access to the collection is synchronized (thread-safe).
- Item - Gets an individual member of the collection.
- SyncRoot - Gets an object that can be used to synchronize access to the collection.

Methods

- CopyTo - Copies all the elements of the collection to the given array starting at the given index.
- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetEnumerator - Provides an enumerator that iterates through the collection.
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- ToString - Returns a string that represents the current object. (Inherited from Object.)

Extension Methods

- AsParallel - Enables parallelization of a query. (Defined by ParallelEnumerable.)
- AsQueryable - Converts an IEnumerable to an IQueryable. (Defined by Queryable.)
- Cast<TResult> - Converts the elements of an IEnumerable to the specified type. (Defined by Enumerable.)
- OfType<TResult> - Filters the elements of an IEnumerable based on a specified type. (Defined by Enumerable.)

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `Match`

Match

The `Match` class represents the results from a single regular expression match. According to Microsoft's website: *The Match object is immutable and has no public constructor. An instance of the Match class is returned by the Regex.Match method and represents the first pattern match in a string. Subsequent matches are represented by Match objects returned by the Match.NextMatch method. In addition, a MatchCollection object that consists of zero, one, or more Match objects is returned by the Regex.Matches method. If the Regex.Matches method fails to match a regular expression pattern in an input string, it returns an empty MatchCollection object. You can then use a foreach construct in C# or a For Each construct in Visual Basic to iterate the collection. If the Regex.Match method fails to match the regular expression pattern, it returns a Match object that is equal to Match.Empty. You can use the Success property to determine whether the match was successful.*

Properties

- Captures - Gets a collection of all the captures matched by the capturing group, in innermost-leftmost-first order (or innermost-rightmost-first order if the regular expression is modified with the RegexOptions.RightToLeft option). The collection may have zero or more items. (Inherited from Group.)
- Empty - Gets the empty group. All failed matches return this empty match.
- Groups - Gets a collection of groups matched by the regular expression.
- Index - The position in the original string where the first character of the captured substring was found. (Inherited from Capture.)
- Length - The length of the captured substring. (Inherited from Capture.)
- Success - Gets a value indicating whether the match is successful. (Inherited from Group.)
- Value - Gets the captured substring from the input string. (Inherited from Capture.)

Methods

- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- NextMatch - Returns a new Match object with the results for the next match, starting at the position at which the last match ended (at the character after the last matched character).
- Result - Returns the expansion of the specified replacement pattern.
- Synchronized - Returns a Match instance equivalent to the one supplied that is suitable to share between multiple threads.
- ToString - Gets the captured substring from the input string. (Inherited from Capture.)

The `System.Text.RegularExpressions` Namespace

→ Classes

→ `Regex`

Regex

The `Regex` class represents an immutable regular expression. According to Microsoft's website: *The Regex class represents the .NET Framework's regular expression engine. It can be used to quickly parse large amounts of text to find specific character patterns; to extract, edit, replace, or delete text substrings; or to add the extracted strings to a collection to generate a report.*

Note Microsoft's comment: *If your primary interest is to validate a string by determining whether it conforms to a particular pattern, you can use the System.Configuration.RegexStringValidator class.*

Note that if you instantiate a Regex object, the regular expression you provide cannot be changed at a later time. If you use the static methods, rather than instantiating a Regex object, you can change the regular expression. Note that performance is comparable between the instance and static methods. But, if you are using a lot of regular expressions, be aware that the first 15 regular expressions are cached and after that they are recompiled. To prevent this, increase the Regex.CacheSize property!

Constructors

- Regex() - Initializes a new instance of the Regex class
- Regex(String) - Initializes and compiles a new instance of the Regex class for the specified regular expression
- Regex(SerializationInfo, StreamingContext) - Initializes a new instance of the Regex class by using serialized data
- Regex(String, RegexOptions) - Initializes and compiles a new instance of the Regex class for the specified regular expression, with options that modify the pattern

Properties

- CacheSize - Gets or sets the maximum number of entries in the current static cache of compiled regular expressions
- Options - Returns the options passed into the Regex constructor
- RightToLeft - Gets a value indicating whether the regular expression searches from right to left.

Methods

- CompileToAssembly(RegexCompilationInfo[], AssemblyName) - Compiles one or more specified Regex objects to a named assembly
- CompileToAssembly(RegexCompilationInfo[], AssemblyName, CustomAttributeBuilder[]) - Compiles one or more specified Regex objects to a named assembly with the specified attributes
- CompileToAssembly(RegexCompilationInfo[], AssemblyName, CustomAttributeBuilder[], String) - Compiles one or more specified Regex objects and a specified resource file to a named assembly with the specified attributes
- Equals(Object) Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Escape - Escapes a minimal set of characters (\, *, +, ?, |, {, [, (,), ^, $,., #, and white space) by replacing them with their escape codes. This instructs the regular expression engine to interpret these characters literally rather than as metacharacters
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetGroupNames - Returns an array of capturing group names for the regular expression
- GetGroupNumbers - Returns an array of capturing group numbers that correspond to group names in an array
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- GroupNameFromNumber - Gets the group name that corresponds to the specified group number
- GroupNumberFromName - Returns the group number that corresponds to the specified group name
- InitializeReferences - Infrastructure. Used by a Regex object generated by the CompileToAssembly method
- IsMatch(String) - Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string
- IsMatch(String, Int32) - Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string
- IsMatch(String, String) - Indicates whether the specified regular expression finds a match in the specified input string
- IsMatch(String, String, RegexOptions) - Indicates whether the specified regular expression finds a match in the specified input string, using the specified matching options
- Match(String) - Searches the specified input string for the first occurrence of the regular expression specified in the Regex constructor

Methods (continued)

- Match(String, Int32) - Searches the input string for the first occurrence of a regular expression, beginning at the specified starting position in the string
- Match(String, String) - Searches the specified input string for the first occurrence of the specified regular expression
- Match(String, Int32, Int32) - Searches the input string for the first occurrence of a regular expression, beginning at the specified starting position and searching only the specified number of characters
- Match(String, String, RegexOptions) - Searches the input string for the first occurrence of the specified regular expression, using the specified matching options
- Matches(String) - Searches the specified input string for all occurrences of a regular expression
- Matches(String, Int32) - Searches the specified input string for all occurrences of a regular expression, beginning at the specified starting position in the string
- Matches(String, String) - Searches the specified input string for all occurrences of a specified regular expression
- Matches(String, String, RegexOptions) - Searches the specified input string for all occurrences of a specified regular expression, using the specified matching options
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- Replace(String, String) - Within a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string
- Replace(String, MatchEvaluator) - Within a specified input string, replaces all strings that match a specified regular expression with a string returned by a MatchEvaluator delegate
- Replace(String, String, Int32) - Within a specified input string, replaces a specified maximum number of strings that match a regular expression pattern with a specified replacement string
- Replace(String, String, String) - Within a specified input string, replaces all strings that match a specified regular expression with a specified replacement string
- Replace(String, String, MatchEvaluator) - Within a specified input string, replaces all strings that match a specified regular expression with a string returned by a MatchEvaluator delegate
- Replace(String, MatchEvaluator, Int32) - Within a specified input string, replaces a specified maximum number of strings that match a regular expression pattern with a string returned by a MatchEvaluator delegate
- Replace(String, String, Int32, Int32) - Within a specified input substring, replaces a specified maximum number of strings that match a regular expression pattern with a specified replacement string
- Replace(String, String, String, RegexOptions) - Within a specified input string, replaces all strings that match a specified regular expression with a specified replacement string. Specified options modify the matching operation
- Replace(String, String, MatchEvaluator, RegexOptions) - Within a specified input string, replaces all strings that match a specified regular expression with a string returned by a MatchEvaluator delegate. Specified options modify the matching operation
- Replace(String, MatchEvaluator, Int32, Int32) - Within a specified input substring, replaces a specified maximum number of strings that match a regular expression pattern with a string returned by a MatchEvaluator delegate

Methods (continued)

- Split(String) - Splits the specified input string at the positions defined by a regular expression pattern specified in the Regex constructor
- Split(String, Int32) - Splits the specified input string a specified maximum number of times at the positions defined by a regular expression specified in the Regex constructor
- Split(String, String) - Splits the input string at the positions defined by a regular expression pattern
- Split(String, Int32, Int32) - Splits the specified input string a specified maximum number of times at the positions defined by a regular expression specified in the Regex constructor. The search for the regular expression pattern starts at a specified character position in the input string
- Split(String, String, RegexOptions) - Splits the input string at the positions defined by a specified regular expression pattern. Specified options modify the matching operation
- ToString - Returns the regular expression pattern that was passed into the Regex constructor. (Overrides Object.ToString().)
- Unescape - Converts any escaped characters in the input string

Note that you can find a list of the regular expression language elements at the following website: http://msdn.microsoft.com/en-us/library/az24scfc.aspx.

The `System.Text.RegularExpressions` Namespace

- → Classes
  - → `RegexCompilationInfo`

RegexCompilationInfo

The `RegexCompilationInfo` class provides information about a regular expression that is used to compile a regular expression to a stand-alone assembly. According to Microsoft's website: *An array of RegexCompilationInfo objects is passed to the CompileToAssembly method to provide information about each regular expression to be included in the assembly. Each compiled regular expression that is included in the assembly is represented as a class derived from Regex. The properties of the RegexCompilationInfo type define the regular expression's class name, its fully qualified name (that is, its namespace and its type name), its regular expression pattern, and any additional options (such as whether the regular expression is case-insensitive). You can instantiate a RegexCompilationInfo object by calling its class constructor.*

Constructors

- RegexCompilationInfo - Initializes a new instance of the RegexCompilationInfo class that contains information about a regular expression to be included in an assembly.

Properties

- IsPublic - Gets or sets a value that indicates whether the compiled regular expression has public visibility.
- Name - Gets or sets the name of the type that represents the compiled regular expression.
- Namespace - Gets or sets the namespace to which the new type belongs.
- Options - Gets or sets the options to use when compiling the regular expression.
- Pattern - Gets or sets the regular expression to compile.

Methods

- Equals(Object) - Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
- Finalize - Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetHashCode - Serves as a hash function for a particular type. (Inherited from Object.)
- GetType - Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone - Creates a shallow copy of the current Object. (Inherited from Object.)
- ToString - Returns a string that represents the current object. (Inherited from Object.)

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ Attributes

Attributes

There are no attributes in these namespaces.

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ EventArgs

EventArgs

There are no `EventArgs` in these namespaces.

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ Structures

Structures

There are no structures in these namespaces.

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ Interfaces

Interfaces

There are no interfaces in these namespaces.

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ Delegates

Delegates

There is one delegate in the `System.Text.RegularExpressions` namespace.

Delegates

- MatchEvaluator - Represents the method that is called each time a regular expression match is found during a Replace method operation.

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ Enumerations

Enumerations

There is one enumeration available in the `System.Text.RegularExpressions` namespace.

Enumerations

- RegexOptions - Provides enumerated values to use to set regular expression options.

The `RegexOptions` members are listed below:

- None - Specifies that no options are set.
- IgnoreCase - Specifies case-insensitive matching.
- Multiline - Multiline mode. Changes the meaning of ^ and $ so they match at the beginning and end, respectively, of any line, and not just the beginning and end of the entire string.
- ExplicitCapture - Specifies that the only valid captures are explicitly named or numbered groups of the form (?<name>…). This allows unnamed parentheses to act as noncapturing groups without the syntactic clumsiness of the expression (?:…).
- Compiled - Specifies that the regular expression is compiled to an assembly. This yields faster execution but increases startup time. This value should not be assigned to the Options property when calling the CompileToAssembly method.
- Singleline - Specifies single-line mode. Changes the meaning of the dot (.) so it matches every character (instead of every character except \n).
- IgnorePatternWhitespace - Eliminates unescaped white space from the pattern and enables comments marked with #. However, the IgnorePatternWhitespace value does not affect or eliminate white space in character classes.
- RightToLeft - Specifies that the search will be from right to left instead of from left to right.
- ECMAScript - Enables ECMAScript-compliant behavior for the expression. This value can be used only in conjunction with the IgnoreCase, Multiline, and Compiled values. The use of this value with any other values results in an exception.
- CultureInvariant - Specifies that cultural differences in language is ignored. See Performing Culture-Insensitive Operations in the RegularExpressions Namespace for more information.

The `System.Text` and `System.Text.RegularExpressions` Namespaces

→ Exceptions

Exceptions

There are no exceptions available in these namespaces.

What Next?

In *C# Programming IV-#*, we look at specific classes within specific namespaces such as the System namespace, the System.Text namespace, etc.

References

Click the book titles below to read more about these books on Amazon.com's website.

- Introducing Microsoft LINQ, Paolo Pialorsi and Marco Russo, Microsoft Press, ISBN:9780735623910
- LINQ Pocket Reference, Joseph Albahari and Ben Albahari, O'Reilly Press, ISBN:9780596519247
- Inside C#, Tom Archer and Andrew Whitechapel, Microsoft Press, ISBN:0735616485
- C# 4.0 In a Nutshell, O'Reilly Press, Joseph Albahari and Ben Albahari, ISBN:9780596800956
- The Object Primer, Scott W. Ambler, Cambridge Press, ISBN:0521540186
- CLR via C#, Jeffrey Richter, Microsoft Press, ISBN:9780735621633

Support sheepsqueezers.com

If you found this information helpful, please consider supporting sheepsqueezers.com. There are several ways to support our site:

- ☐ Buy me a cup of coffee by clicking on the following link and donate to my PayPal account: Buy Me A Cup Of Coffee?.
- ☐ Visit my Amazon.com Wish list at the following link and purchase an item: http://amzn.com/w/3OBK1K4EIWIR6

Please let me know if this document was useful by e-mailing me at `comments@sheepsqueezers.com`.